Codemash 2014 Applied Application Security

Session Whitepaper

January 7, CodeMash hosted a four hour session of Applied Application Security presented by Bill Sempf, based on his sixteen hour Secure Coding course. This whitepaper covers the salient points, definitions, projects and products that were covered in that course.

Core Security Principles

While each individual vulnerability discussed in this course has a specific set of coding guidelines, there are seven points of Defense in Depth that can be used to craft and vet procedures.

Defense in Depth

Refers to the application of secure coding throughout the application, not just at a single 'security layer.' One example of this is input validation at the browser. Attackers don't use browsers, they use attack proxies, so the browser validation is ineffective. Instead, input and output should be validated and encoded at the client and server level.

Positive Security Model

Use a whitelist to allow in the known good, rather than a blacklist to reject the known bad. While you control the requirements for known good input, the attacker controls the list of known bad inputs.

Fail Securely

Especially in error handling, it is better to fail by disallowing an action that fail by allowing it. For example, if a search fails, show zero records rather than showing all records.

Principle of Least Privilege

Grant as little power as possible to each component of a system. For example, when creating a new user grant no privilege to start, and add them as necessary.

Separation of Duties

Coders code and testers test. Don't assume that developers will write secure code. Assign security testing tasks to testers, and enforce the creation of patches by developers.

Avoid Security by Obscurity

Hiding functionality doesn't work. Apply authorization rules to all assets, no matter where in the system they reside.

Do Not Trust the Client

Assume all data coning into the application from any source is dangerous. This applies not only to user generate input, but cookies, request headers, and service calls.

Information Disclosure

One of the key features that application security scanner, static code analyzers and web application firewalls lack is intuition. Giving users insight into the inner workings of your system will lead to a dedicated attacker using information to bypass the most sophisticated of controls.

You can reduce the total information disclosure attack surface by considering three core sources:

- What is available on the Internet
- Your application's source code
- Exception management policy

That big database called the Internet

An attacker can use Google to research your applications. To do so, use the following advanced search features:

- The **site** parameter restricts the search to a particular second level domain, e.g. `site:pastebin.com connectionstring`
- The **ext** parameter restricts searches to particular document types, e.q. `site:chase.com ext:xlsx`
- The **inurl** parameter looks for a string in the url of pages, e.g. `inurl:WS_FTP.log passwd`
- The **intitle** looks for search terms in the title only, e.g. `intitle:"Live View – AXIS"`
- The **link** parameter finds pages that link to a particular URL, e.g. `link:interestingdomain.com/file.html`
- There are **operators** that one can use as well
    - Dash - is a negative operator
    - Bar | is used for a Boolean 'or'
    - Quotes "" define a phrase
    - Parens () are for combinging Boolean searches
    - Tilde ~ finds similar words

Your source code

There are several key parts of your coding and configuration environment that require review to prevent information disclosure.

Comments

Comments provide an endless source of information for an attacker.

- Developer's names in comments provide search material for social networks like Stack Overflow
- Old code can provide insight into hidden or not-yet-implemented functionality
- Test data can appear in comments.

Headers

HTTP headers give attackers a lot of information about the underlying server environment.

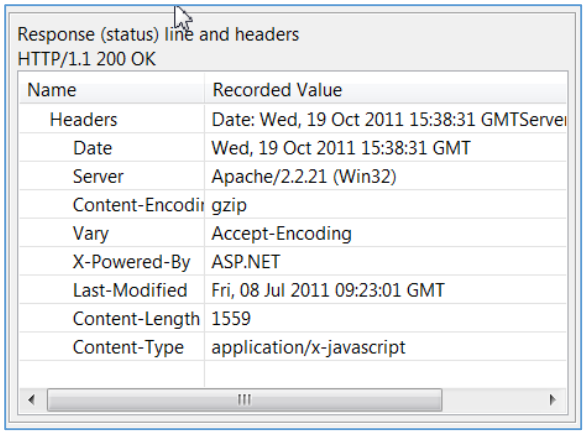
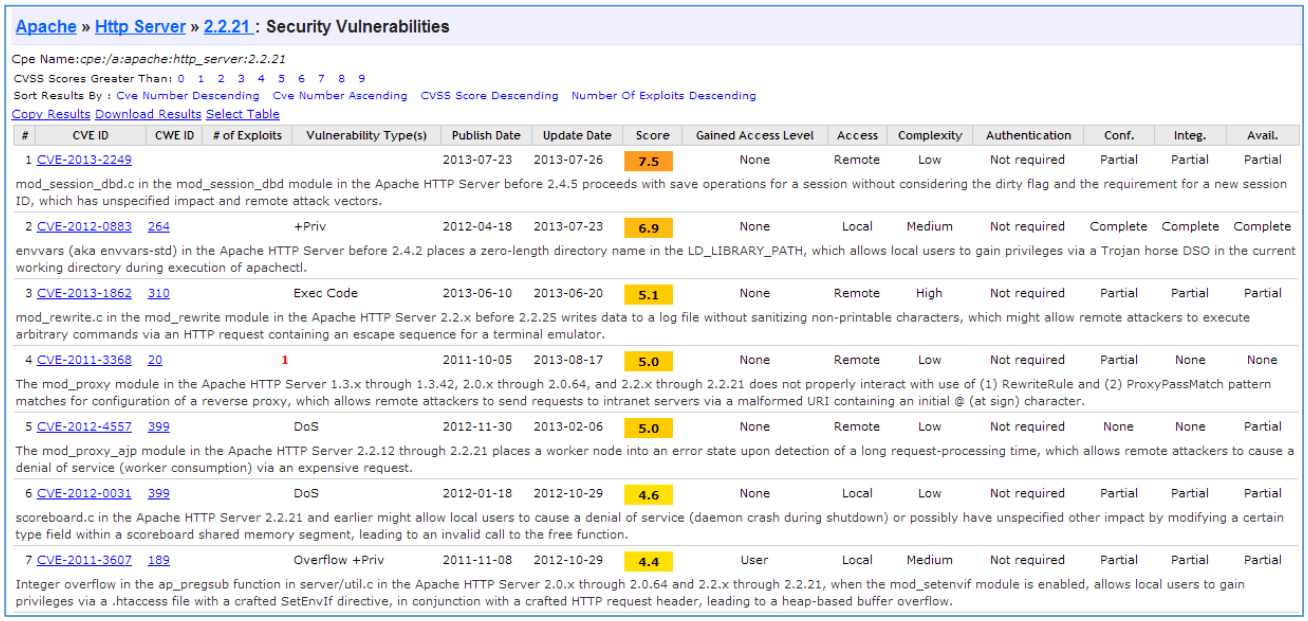


Figure 1: Server environment disclosed in header

Information, such as an outdated Apache server version, can give attackers a great place to start attacking. For instance, Apache 2.2.21 has a number of vulnerabilities, as disclosed on the CVS:



Apache » Http Server » 2.2.21 : Security Vulnerabilities

Cpe Name:*cpe:/a:apache:http_server:2.2.21*

CVSS Scores Greater Than: 0 1 2 3 4 5 6 7 8 9

Sort Results By : Cve Number Descending Cve Number Ascending CVSS Score Descending Number Of Exploits Descending

Copy Results Download Results Select Table

#	CVE ID	CWE ID	# of Exploits	Vulnerability Type(s)	Publish Date	Update Date	Score	Gained Access Level	Access	Complexity	Authentication	Conf.	Integ.	Avail.
1	CVE-2013-2249				2013-07-23	2013-07-26	7.5	None	Remote	Low	Not required	Partial	Partial	Partial
mod_session_dbd.c in the mod_session_dbd module in the Apache HTTP Server before 2.4.5 proceeds with save operations for a session without considering the dirty flag and the requirement for a new session ID, which has unspecified impact and remote attack vectors.														
2	CVE-2012-0883	264		+Priv	2012-04-18	2013-07-23	6.9	None	Local	Medium	Not required	Complete	Complete	Complete
envvars (aka envvars-std) in the Apache HTTP Server before 2.4.2 places a zero-length directory name in the LD_LIBRARY_PATH, which allows local users to gain privileges via a Trojan horse DSO in the current working directory during execution of apachectl.														
3	CVE-2013-1862	310		Exec Code	2013-06-10	2013-06-20	5.1	None	Remote	High	Not required	Partial	Partial	Partial
mod_rewrite.c in the mod_rewrite module in the Apache HTTP Server 2.2.x before 2.2.25 writes data to a log file without sanitizing non-printable characters, which might allow remote attackers to execute arbitrary commands via an HTTP request containing an escape sequence for a terminal emulator.														
4	CVE-2011-3368	20		1	2011-10-05	2013-08-17	5.0	None	Remote	Low	Not required	Partial	None	None
The mod_proxy module in the Apache HTTP Server 1.3.x through 1.3.42, 2.0.x through 2.0.64, and 2.2.x through 2.2.21 does not properly interact with use of (1) RewriteRule and (2) ProxyPassMatch pattern matches for configuration of a reverse proxy, which allows remote attackers to send requests to intranet servers via a malformed URI containing an initial @ (at sign) character.														
5	CVE-2012-4557	399		DoS	2012-11-30	2013-02-06	5.0	None	Remote	Low	Not required	None	None	Partial
The mod_proxy_ajp module in the Apache HTTP Server 2.2.12 through 2.2.21 places a worker node into an error state upon detection of a long request-processing time, which allows remote attackers to cause a denial of service (worker consumption) via an expensive request.														
6	CVE-2012-0031	399		DoS	2012-01-18	2012-10-29	4.6	None	Local	Low	Not required	Partial	Partial	Partial
scoreboard.c in the Apache HTTP Server 2.2.21 and earlier might allow local users to cause a denial of service (daemon crash during shutdown) or possibly have unspecified other impact by modifying a certain type field within a scoreboard shared memory segment, leading to an invalid call to the free function.														
7	CVE-2011-3607	189		Overflow +Priv	2011-11-08	2012-10-29	4.4	User	Local	Medium	Not required	Partial	Partial	Partial
Integer overflow in the ap_pregsub function in server/util.c in the Apache HTTP Server 2.0.x through 2.0.64 and 2.2.x through 2.2.21, when the mod_setenvif module is enabled, allows local users to gain privileges via a .htaccess file with a crafted SetEnvIf directive, in conjunction with a crafted HTTP request header, leading to a heap-based buffer overflow.														

Figure 2: Vulnerabilities associated with Apache 2.2.21

Hidden Files

Files with non-web standard extensions, such as '.bak' or '.old' are not impossible to download by an attacker. The attacker just has to guess that they are there. Fortunately (for them at least) there are a number of tools that make the collection of hidden files quite easy.

Files not directly used by the application should be removed from the web root at deploy time.

JSON and Web Services

SOAP or ReST services, JSON and XML feeds, and the like are all particularly ripe for attackers, especially if you have a mobile application environment. Servers that were previously used by the web server in a SOA

environment are now often tunneled through the firewall. These under-tested services are as ripe of a target as the website itself, and should be subjected to the same testing rigor.

URLs

URLs contain a lot of information about the application in the form of parameters and directory structure. Add to this that they are often stored in logs, referrers and bookmarks, and you end up with a fair amount of leakage.

No information should be passed in the URL that couldn't be better passed in a form variable or cookie. While these values are still subject to attack, they are not left lying about like URL strings are.

Client Side Objects

Understand that any binary can be decompiled. Java, flash, Silverlight, ActiveX: all of them have decompilers that will give the attacker insight into the functionality of the application. Treat the source code of a rich client object as though it is visible to the attacker.

Exception management

The last of the three key sources of information disclosure is exception management. Exceptions should give the user the minimum information necessary, and log the maximum information necessary. To so this, use structured exceptions, usually in the form of a try/catch/finally clock.

```
try
{
    //Code that could fail
}
catch(SpecificException se)
{
    //Handle specific exceptions when you can
}
catch(ApplicationException e)
{
    //Handle general exceptions when you can't
}
finally
{
    //clean up anything that might be left over because the try was interrupted
}
```

Authentication and Session Management

Authentication is the act of making sure the user is who they say they are. This can be accomplished by asking them for something they know, like a password, for something they have, like a smartcard, or something they are, like a thumbprint.

Authorization is the act of making sure they user is allowed to do what they ask to do in the application. This can be accomplished by storing their 'role' within the application, or checking various claims about themselves with a trusted third party.

Session Management is keeping track of all of this so the user doesn't have to provide the same information to the application over and over.

Authentication

There are a number of different forms of authentication that have become popular on the web.

- **Form based** authentication is using an HTML form to submit credentials that are then verified against a database or other backend system.
- **HTTP Basic** authentication is authentication via username and password provided by the web server.
- **HTTP Digest** authentication is also web server based, but uses a nonce rather than passing credentials over the wire.
- **Physical Factor** authentication uses something the user has, like a smart phone or token.
- **Biometric** authentication uses something the user is, such as a thumbprint, to identify the user.
- **Server** and **Client** authentication is the act of using asymmetric encryption to authentication the web server or client browser.
- **Single Signon** uses a third party authentication system to share identity with some standard such as OAuth or SAML.

Attacks

Attacks around authentication usually revolve around guessing credentials. If usernames are known, then common passwords can be tried to gain access. Recent release of password lists from large providers have given unusual insight into what users are selecting for passwords.

Other methods include watching unsecured login traffic for credentials, for login forms that aren't secured with SSL. Another popular tactic is to attack poorly secured password recovery mechanisms.

Defenses

If the attacker gets a copy of the user's credentials, they can access the application – it's as simple as that. Defenses should be focused on prevention of that activity. Sounds simple, but it gets forgotten a lot.

- **Make good usernames.** Don't use something that is obvious or guessable, like an integer.
- **Make strong passwords.** Jury is out on how string they should be, but prevent users form entering one letter passwords, or using their username, or the word password.
- **Create a secure password recovery system.** Generally, the process should look like this:
    - Click Forgot Link
    - Send out-of-band token
    - One time link

    - Ask a security question
    - Require New Password
- **Use a secure password storage system.** See the OWASP Password Storage Cheat Sheet for the latest information.
- **Transmit credentials over Secure Sockets Layer.**
- **Use Multifactor authentication.** I recommend Duo Security.
- **Handle authentication errors properly.** Don't tell the user if they got their username or password wrong, just tell them that they failed authentication.
- **Have a reasonable lockout policy.** After 6 failed logins, require a reset or use CAPTCHA.

Authorization

Authorization is the act of using roles or claims to give the user access to functionality that the user is supposed to have access to, and prevent access to functionality that the user is not supposed to have access to.

Attacks

Attacks on authorization are significantly broader than those on authentication. Authorization just consists of taking a token and seeing if it matches an expected result. Authorization can be implemented many different ways. This leads to many different attacks.

- **Cross-Site Request Forgery (CSRF)** is an attack which forces an end user to execute unwanted actions on a web application in which he/she is currently authenticated. It usually consists of tricking a user to go to a page with malicious content and then using the session information from the target application to perform an undesired function on the target system.
- **Forced Browsing** is simply requesting resources that the user is not authorized to use outside the scope of the navigation. For instance, guessing the name of the admin screen is a common forced browsing attack.
- **Parameter tampering** is the discovery and alteration of role information in the cookie, header, form or URL inputs in order to achieve elevated permissions.
- **Clickjacking** is luring an authorized user to an unrelated site that uses a transparent frame to trick the user to click on site navigation, while believing that they are performing some function on the attacker's site.

Defenses

Cross site request forgery is best defended with an anti-forgery token. This is a per-page unique value that assures that the request came from the expected form rather than from an attacker. Most web frameworks have an anti-forgery token built in, but it has to be turned on.

Forced browsing defense is just a matter of checking authorization on every resource. Don't assume that hiding the admin page from the navigation will prevent it from being found – it won't. Check the user's roles or claims provide the requested access every time the resource is requested.

Parameter tampering requires that the parameter be there in the first place. Instead, just use a session identifier that is a pseudo-random string, and use that to look up the role information every time.

Clickjacking has an easy defense – just use the x-frame-options server header. It will tell the browser how the site can be used in frames.

Session Management

Sessions are manages in order to maintain some state of the logged-in user over time using the site. Usually, this is done with some token that is passed between the browser and the server. There are a number of ways to do this, but the only right way is to use the cookies collection, and provide a pseudo-random string to the client each time that they can return to the server to resume their session.

Attacks

There are three common attacks against session management features.

- **Interception**: If the attacker can get a copy of the session token while it is in use, and this still valid, they can take over the user's session.
- **Prediction**: In the case that the session token is guessable, the attack might predict the next few that are in use, and then use them after a user logs in.
- **Fixation**: This attack involves getting a user to authenticate himself with a known session token.

Defenses

Preventing session token interception requires a few common defense-in-depth kinds of things:

- Use of SSL
- Keeping the session token in the cookies collection
- Making sure the session cookie uses HTTPOnly and Secure attributes.

Prediction is an encryption issue. The developer must make sure that the session identifier is sufficiently large, and pseudo-random, in order to prevent predictable tokens. Use the build-in session management whenever possible.

Session fixation simply requires that the token be reset after a successful login.

Content Validation

Data is evil. All user input – no matter the source – should be assumed to be malicious. Validating the content of a request is the most important structural security code that a developer can possibly write.

Cross Site Scripting

CSS, or Browser Injection, consists of making a web browser execute unwanted commends in the users content. This vulnerability is because of the power of JavaScript, and the fact that the browser is now so complex it is practically an operating system of its own.

The most damaging attack that most applications face from a cross site scripting vulnerability is the theft of session tokens. Under certain circumstances, this could give an unauthenticated attacker multiple levels of access to a system.

Stored XSS

Stored Cross Site Scripting occurs when an XSS attack is stored in a system for another user to see. For instance, if the user profile screen of an application had a cross site scripting vulnerability, an attacker could place an XSS attack in the profile and then request assistance from the admin. When the admin views the user's profile, the XSS attack silently sends the session token to the attacker. The attacker then can take over the administrator's session.

Reflected XSS

Reflected XSS is significantly less serious, but it still can be attacked. Reflected XSS occurs when the system displays values form a request on the browser without sufficient encoding. For instance, this could be a URL string that provides reflected XSS, if the `error` parameter was displayed on the screen without encoding.

```
https://www.example.com/page.aspx?error=%3Cscript%3Ealert(%27xss%27)%3C%2Fscript%3E
```

Attacks with reflected XSS are the same as with stored – the attacker sends a link to the admin requesting help, and the attack is encoded within the link.

DOM XSS

DOM XSS is exactly like reflected XSS, except the vulnerability doesn't use the server at all – it is totally within the browser. For instance, if a JavaScript function on the site reads the above error parameter, rather than the value being rendered by the server, that is DOM XSS.

Encoding is XSS defense

Preventing XSS is all about encoding, but using Server.HTMLEncode (for .NET developers) isn't enough. There is an encoding standard for every data type in the Web world, and it is tough to keep them all straight.

Your best bet is the use of an encoding library. There are a number listed on the XSS Prevention Cheat Sheet.

Command Injection

Command Injection is the execution of arbitrary commands on the backend system. This is in opposition to Browser Injection, which is the execution of arbitrary commands on the frontend system (the browser).

Database Injection

Database Injection, usually called SQL Injection or SQLi, is the execution of an arbritary command on the backend database using a carefully structured query. It is probably the most talked about vulnerability in the history of the world, because it is so damaging. SQLi is now significantly less prevalent.

While the other command vulnerabilities has a common defense, SQLi is slightly different. Input validation is still the answer, but you can have the database do the work for you by using parameterized queries. Orject Role Models, structured queries or stored procedures must be the standard for data access.

Operating System Injection

This is the act of executing arbitrary commends on the underlying operating system. For example:

```
<? system($_REQUEST['cmd']); ?>
```

Anytime you call a system process from a network capable application, you are asking for trouble.

Directory Injection

Queries to directory services like Active Directory or LDAP can cause directory injection. For example:

```
String ldapSearchQuery = "(cn=" + $userName + ")";
System.out.println(ldapSearchQuery);
```

XPATH Injection

XML is no different than any other database, and the underlying parser is just as injectable. For instance:

```
//users/user[username/text()='user' or '1' = '1' and password/text()='password']
```

File System Injection

Unrestricted file upload can lead to file system injection. Like SQLi, this is a special case that has different answers based on the kind of file uploading that is occurring. I cover several of the instances on my blog.

HTTP Header Injection

This very cool attack consists of using the details of the HTTP protocol to send in two HTTP commands under the same request, or conversely inject hostile data into the user's browser.

Input validation is command injection defense

The only effective way to prevent all injection attacks is through input validation. Using a white list for validation – comparing input to the list of possible known good values for the field – will assure that no arbitrary commands are sent to the underlying systems. OWASP has an excellent document showing input validation options.