# 

|  |  |  |
| --- | --- | --- |
| **Dates** | **Version** | **Notes/Comments** |
| 15th Dec 2008 | 0.1 | Initial Draft |
| 1st June 2009 | 1.0 | Initial Version |
| 1st July 2010 | 1.1 | Updated to include fixes for XFS2 |
|  |  |  |

**Document Properties**

Version 1.1

(Supersedes 1.0)

1st July 2010

Owner: Steven Perry ([steven.perry@citi.com](mailto:steven.perry@citi.com))

Location: [CATE Security Domain Standards](https://catecollaboration.citigroup.net/domains/security/appsec/default.aspx) (Click on Domain Standards)

Acknowledgement – I would like to acknowledge work done by the GCG-NA eBusiness Information Security Team that helped form some of the material in this Domain Standard.

**Classification: Internal**

**TABLE OF CONTENTS**

[1 Introduction 3](#_Toc265942397)

[1.1 Purpose 3](#_Toc265942398)

[1.2 Audience 3](#_Toc265942399)

[2 Coding Standards 4](#_Toc265942400)

[2.1 Data Validation 4](#_Toc265942401)

[2.1.1 Bypass Application Input Validation 4](#_Toc265942402)

[2.1.2 XSS Cross-site Scripting 6](#_Toc265942403)

[2.1.3 XFS Cross Frame Scripting 8](#_Toc265942404)

[2.1.4 SQL Injection 10](#_Toc265942405)

[2.1.5 Command Injection 12](#_Toc265942406)

[2.1.6 Log Injection 13](#_Toc265942407)

[2.1.7 HTTP Response Splitting 15](#_Toc265942408)

[2.1.8 Buffer Overflow 16](#_Toc265942409)

[2.1.9 Missing XML Validation 18](#_Toc265942410)

[2.1.10 Missing URL Validation Before Redirection 19](#_Toc265942411)

[2.1.11 Missing Validation before File Upload 20](#_Toc265942412)

[2.1.12 Missing Validation Before File Access 21](#_Toc265942413)

[2.1.13 Missing Validation of Out Of range 23](#_Toc265942414)

[2.1.14 Missing Both Client Side And Server Side Validation 23](#_Toc265942415)

[2.2 Configuration Management 24](#_Toc265942416)

[2.2.1 Use of Java APPLETS 25](#_Toc265942417)

[2.2.2 Hidden Fields 25](#_Toc265942418)

[2.2.3 Insecure Pseudo-random Number Generator 27](#_Toc265942419)

[2.2.4 Information Leakage 29](#_Toc265942420)

[2.2.5 Browser Auto-complete Enabled 30](#_Toc265942421)

[2.2.6 No-cache headers not being set 31](#_Toc265942422)

[2.2.7 HTTP POST replay 32](#_Toc265942423)

[2.2.8 Weak Encryption and Weak Ciphers Supported 34](#_Toc265942424)

[2.2.9 Hard Coded Secrets (passwords, cryptographic keys, PINs, etc.) 35](#_Toc265942425)

[2.2.10 Storing Secrets in Readable (e.g. clear text) Format 38](#_Toc265942426)

[3 Appendix 39](#_Toc265942427)

# Introduction

With the increasing complexity of business applications within Citi, there is a need to establish clear and concise requirements for secure coding practices.

## Purpose

The purpose of this document is to detail a set of requirements that should be addressed during the construction phase (e.g. coding) and verified during code reviews to assess the security posture of the application. The aim of these requirements is to prevent most threats and vulnerabilities found in web applications through the use of secure coding practices.

Whenever new threats emerge with root causes originating in code, new countermeasures (e.g. code changes) need to be implemented in order to mitigate the risk posed by such threats. For this reason, the content of this standard represents a living document that will be updated to cover more specific aspects of secure coding, coding examples as well as new web application security issues and recommended remediation.

## Audience

The primary audience of this document is developers and application managers. Moreover,if you are involved in the design, development, testing, deployment or operation of software applications you should take the [Advanced Security Coding](http://www.citigroup.net/informationsecurityservices/ace/course_description_wadasc.htm) training that is available on-line through Citi GLMS (Global Learning Management System). The Advanced Secure Coding Course consist of four modules that cover critical aspects of secure coding as well as best coding practices to remediate the risk posed by threats and vulnerabilities in web applications.

# Coding Standards

Throughout this section, the Recommendations are presented in tables. For example, the initial section covers Data Validation and information leakage.

## Data Validation

Since web applications often use data supplied from the user, issues arise when such data is not properly validated before being processed by the application. For example, when special characters are not filtered before being processed by the application, they can be used to inject commands in a query string allowing unauthorized access to data as well as data tampering and deletion (e.g. SQL Injection attacks). Because of the client/server architecture that web applications rely upon, all data that travels unprotected (e.g. through HTTP) from the client browser to the server where the application resides can be captured by a malicious user and manipulated before being processed by the application. The kind of input data that can be manipulated ranges from data stored in URL parameters and cookies, as well as other data that is processed by the server side and sent back to the user (e.g. authentication tokens, sessionID). For that reason, both data coming into the application and data coming out from the application and sent back to the client need to be validated to avoid exploitation of attacks directed toward the client (e.g. reflected Cross-Site Scripting - XSS). Typically, bypassing input validation attacks exploit weakness of the application in validating data at different layers such as front end web pages exposed to the client, middleware application and server components, and back end database functionality.

### Bypass Application Input Validation

Applications often use client side only validation to ensure the validity of the data within the application. While this technique provides convenience it does not provide any security. Proxies and other tools used by attackers can easily bypass any front end input validation controls in place, thus leaving the application wide open for data manipulation attacks.

Recommendations

|  |  |  |
| --- | --- | --- |
| **Requirements** | **Context** | **Applicability** |
| Validate user supplied input on the server side to ensure that when an attacked bypasses client side controls, malicious data does not enter the program. All input should be validated on the following criteria: -   * Data type (string, integer, real, etc…) * Allowed character set * Minimum and maximum input string length * Whether NULL value is allowed * Whether the input parameter is required or not * Whether duplicates are allowed * Numeric range * Specific legal values (enumeration) * Specific format and patterns (regular expressions) * Using white listing (e.g. validate what is allowed, deny anything else) instead of blacklisting (invalidate what is not allowed and allow everything else)   Keeping user input out of control path, do not let user input control the flow of the application. | By checking input data for expected format, value, type and range you will constrain the application to accept only validated data and to rejecting malicious inputs. You should check input data for each entry point as well as each tier (e.g. client, server, backend) of the application.  Avoid common implementation mistakes in input validation that lead to security issues.  Common implementation mistakes include:   * Relying on client-side input validation only * Not filtering input from unsafe characters * Not sanitizing (i.e. make safe) the output using proper encoding * Making functional decisions based upon user input instead to check server side parameters * Assuming that malicious data only comes from the client | All applications that accept user input. |

Insecure Coding Examples

The following code snippet shows the use of HTML JavaScript client validation to validate email input data. It exposes the application to data validation issues because no server validation is being enforced. You should avoid coding JavaScript client code for input validation and focus on implementing server side data validation instead.

<html>

<head>

<script type="text/javascript">

function validate\_email(field,alerttxt){

with (field){

apos=value.indexOf("@")

dotpos=value.lastIndexOf(".")

if (apos<1||dotpos-apos<2)

{alert(alerttxt);return false}

else {return true}

}

}

function validate\_form(thisform){

with (thisform){

if (validate\_email(email,"Not a valid e-mail address!")==false)

{email.focus();return false}

}

}

</script>

</head>

<body>

<form action="submitpage.htm" onsubmit="return validate\_form(this);" method="post">

Email: <input type="text" name="email" size="30">

<input type="submit" value="Submit">

</form>

</body>

</html>

Secure Coding Examples

The following example shows how to perform input validation on the server side by using default validators such as the ones provided by the application framework (e.g. apache struts). By using apache struts you will simply declare the field email validation property in the validate.xml configuration struts file like in the example below:

<field property="customeremail" depends= “required, mask” >

<msg name="mask" key="registrationForm.email.maskmsg"/>

<arg0 key="registrationForm.email.displayname"/>

<var>

<var-name>mask</var-name>

<var-value>^.\*@.\*$</var-value>

</var>

</field>

The validation.xml file is application specific and describes which validation rules in the validation-rules.xml should be processed on the server side to validate the data. In struts 1.1 the paths to the validator configuration files are declared in the struts-config.xml file so that no coding is required:

<plug-in className=”org.apache.struts.validator.ValidatorPlugIn”>

<set-property Property=”pathnames” Value=”/WEB-INF/validator-rules.xml,/WEB-INF/validator.xml”/>

</plug-in>

### XSS Cross-site Scripting

Cross-site scripting attacks are an instantiation of injection problems, in which malicious scripts are injected into the otherwise benign and trusted web sites. Data that has not been validated from the HTTP response can cause the execution of malicious scripts on the client browser (XSS). This vulnerability has the potential to be used to steal information stored in the client browser such as cookies and other cached information.

XSS attacks come in one of two flavors, reflective or stored. Reflective XSS shows you the results right away, such as in an alert dialog box popping up. Typically reflecting XSS attacks are unleashed through social engineering and phishing email where a user is enticed to select a link that looks like it is from a trusted web site. Consequently the link has an embedded malicious script.

Stored XSS is a little more complicated and requires a different technique. Stored XSS occurs when the attack is stored on the server; typically from a message board. Potential victims click on the URL and the script is executed on the client machine. Stored XSS live permanently on the server side and target a larger number of users than in the case of reflected XSS.

Recommendations

|  |  |  |
| --- | --- | --- |
| **Requirements** | **Context** | **Applicability** |
|  Validate all input at server side (see [bypass validation](#_Bypass_Application_Input_Validation))   Sanitize input at the server side from any malicious characters, such as (<,>, (,), #, % ‘) that can be used to execute a malicious script on the client browser. In addition to the characters above, checks for the key words “script” and “javascript” should also be done.   HTML encode to the output stream any malicious characters so it cannot be executed as script by the client browser | XSS occurs because un-trusted data enters the application. Validating all user supplied input and filtering malicious characters from the input ensure that malicious input does not enter the program but does not prevent a user to bypass such validation to enter unsafe characters. For this reason you should also check for all representations of such characters and encode the output to HTML. Output filtering is especially effective in dealing with XSS attacks since it ensures all HTML output is encoded to prevent it from being executable. Before sending back the output to the client browser you should also encode to HTML all special characters. If you are using struts, you can use tag libraries or the Velocity Tag Library, which is part of the Apache Commons project. | All applications that accept user input. |

Insecure Coding Examples

The following Java code snippet illustrates a JSP page that is vulnerable to reflective XSS. Notice that the variable myDisplayParam is coming directly from the request without any validation. A malicious attacker could insert script tags and change the functionality of the page.

String myDisplayParam = request.getParameter("myDisplayParam");

<a href="some URL">&nbsp;<%= myDisplayParam%></a>

A more complete solution would incorporate data validation on the server side before a value is displayed back to the user and escaping any malicious characters. The following example replaces the malicious characters with \*.

..

Public String sanitize(String input)

input=input.replace(‘<’,’\*’);

input=input.replace(‘>’,’\*’);

input=input.replace(‘\\’,’\*’);

input=input.replace(‘\,’,’\*’);

...

return input;

}

#### Secure Coding Examples

The following code example shows a code snippet (in C/C++) to safely remediate any reflected XSS by encoding to HTML. Since HTML encoding is used, javascript code is not reflected back to the client, thus no malicious script will be executed by the browser.

Bool HtmlEncode (char \*rawHTML, std:string &result)

size\_t iLen=0;

size\_t i=0;

if (rawHTML && (iLen=strlen(rawHTML))) {

for (i=0; I < iLen; i++){

switch (rawHTML[i]) {

case ’\0’ : break;

case ’<’ : result.append(“&lt;”);break;

case ’>’ : result.append(“&gt;”);break;

case ’(’ : result.append(“&#40;”);break;

case ’)’ : result.append(“&#41;”);break;

case ’&’ : result.append(“&amp;”);break;

case ’”’ : result.append(“&quot;”);break;

default : result.append(1, rawHTML[i]); break;

}

}

Return i == iLen? true: false;

}

### XFS Cross Frame Scripting

Cross frame scripting is a technique that allows web application developers to share scripts across frames within the application. Normally the sharing of scripts across frames is restricted to frames within the same domain, such as citi.com. However, cross frame scripting can be leveraged to exploit vulnerabilities in the client browser in conjunction with a phishing attack. All the attacker needs to do is include the real site in a frame with their malicious script. An attacker can send a forged link to a user, where the site is embedded inside a malicious frame controlled by the attacker.

#### Recommendation

|  |  |  |
| --- | --- | --- |
| **Requirements** | **Context** | **Applicability** |
| * Avoid using frames in web applications, however often frames are a convenient design and offer flexibility for the application. In these cases the use of frames should adhere to the rest of the Recommendations. * Always check to make sure your parent window is indeed the parent window * If a top portal page frames child pages on the same server, check for top.location to be the same as self.location * If a top portal page frames child pages from different federated servers check if top.location belongs to a predefined list of allowed domains | Cross frame scripting is a vulnerability affecting clients and is not an application vulnerability per-se, but a browser based vulnerability. However, applications can mitigate such vulnerability by checking the location of the frame as self location. This technique ensures that a malicious attacker cannot include pages of Citi applications in their sites and compromise confidential customer information. | All web applications. |

#### Insecure Coding Examples

Refer to XFS vulnerabilities (<http://xforce.iss.net/xforce/xfdb/10066>)

#### Secure Coding Examples

The following examples show a mitigation of XFS by applying script code. Developers should select the appropriate option based on the functionality and capabilities of their application: -

**Option A**

This option will blank out the page using style-sheets when framed: -

<style>

html { display:none; }

</style>

<script>

if( self == top )

{ document.documentElement.style.display = 'block' ;

} else

{ top.location = self.location ; }

</script>

*For additional information, visit:*

*http://seclab.stanford.edu/websec/framebusting/framebust.pdf*

**Option B**

This option will blank provide a friendly error message to the user utilizing div tags in HTML.

<html>

<head>

<script>

Function showpage() {

if (self == top)

{ document.getElementByld(‘errorpage’).style.display=’none’;

document.getElementByld(‘realpage’).style.display=’visible’;

} else {

Top.location = self.location;

}

}

</script>

</head>

<body onload=”showpage();”>

<div id=”errorpage”>

*This site requires javascript*

</div?

<div id=”realpage”style=”visibility:hidden;”>

The real page content goes here

</div>

</body>

</html>

**Option C**

This option will redirect users to a page where you can display a friendly error message to the users. It utilizes the meta-http header which is being deprecated in new browsers. To avoid this issue, Option A and Option B are preferred.

<head>

<style>

Html {visibility:hidden;}

</style>

<script>

if( self == top )

{ document.documentElement.style.display = 'visible' ;

} else

{ top.location = self.location ; }

</script>

<script>

document.write('<' + '!--');

*Important Note*

1. *If your application's pages rely on framing, the fix will need to be individually determined. Please contact the VA team with your individual situation and we can help determine the solution along with your development team.*
2. *The X-Frame-Options header in IE8 can also be utilized to solve this issue. However, this option is only implemented in IE8.*

### SQL Injection

Constructing dynamic SQL statements from user generated input may allow attackers to change the grammar of the SQL query and run additional SQL commands under user input control.

The root cause of SQL injection vulnerabilities is the ability of an attacker to change context of the SQL query, causing a value that the programmer intended to be interpreted as data to be interpreted as a command instead. A malicious user can exploit a SQL injection vulnerability to gather information that they are not authorized to see, such as query the database for unauthorized data as well as maliciously alter (e.g. insert table, drop a table) the data in the database. 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.

#### Recommendations

|  |  |  |
| --- | --- | --- |
| **Requirements** | **Context** | **Applicability** |
|  Never trust external input to SQL statements and always perform server side input validation before processing the input.   Filter SQL special characters such as **“;’-&\** that can be used to alter the SQL query and perform a SQL injection attack.   Use prepared statements, parameterized queries and stored procedures.   Avoid concatenating input parameters with SQL statements, string concatenation of input parameters should be avoided and parameters should be set independently from the query by using appropriate method calls such as SetInt(), SetString() etc. | SQL injection is exploited by changing the meaning of the SQL query. Attackers can accomplish this by bypassing input validation on the client, so server side input validation is the first line of defense in ensuring that malicious data does not enter the program. Sanitize input at the server side. In particular, script tags, special characters such as single quotes, double quotes, and semicolons. Hex characters should be treated as potentially dangerous and should be filtered. For input validation, use of white-list data validation such as regular expressions that match the only allowed characters and deny anything else (e.g. default deny).  Another line of defense for SQL injection is the use of prepared statements and parameterized input. Using prepared statements eliminates the need for concatenating query strings and provides another layer of protection. Using the setInt() and setString() functions for the input allows the query to add an addition type check on the data entering the queryRisk Analysis. | All applications that accept input. |

#### Insecure Coding Examples

String query = buildQuery(request.getParameter(“userID”),

Request.getParameter(“business”));

query = query+” where manager = “+getManager();

Statement statement=connection.getStatement();

Statement.executeQuery(query);

The above example illustrates query string concatenation with un-validated user supplied data from the request. This example is very dangerous and is susceptible to SQL injection from a code perspective.

#### Secure Coding Examples

A better approach is to use prepared statements and parameterized input that has been validated. The following example illustrates how to mitigate SQL injection attacks.

String query = buildQuery(request.getParameter(“userID”),

Request.getParameter(“business”));

String userName = ctx.getAuthenticatedUserName();

String itemName = request.getParameter("itemName");

String query ="SELECT \* FROM items WHERE itemname=? AND owner=?"; PreparedStatement stmt = conn.prepareStatement(query);

stmt.setString(1, itemName);

stmt.setString(2, userName);

ResultSet results = stmt.execute();

### Command Injection

Command injection problems stem from using user controlled input without validating the input. This allows the malicious attacker to inject their own commands and control the program. Command injection is a common problem with wrapper programs. Often, parts of the command to be run are controllable by the end user; a malicious user can inject a delimiting character such as “;”and insert another command of his/her choosing (e.g. CRLF “\r\n”).

#### Recommendations

|  |  |  |
| --- | --- | --- |
| **Requirements** | **Context** | **Applicability** |
| Validate all user input to prevent attackers to insert malicious input in resources such as log files, email and HTTP headers.  In particular to prevent CRLF injection and other vulnerabilities you should filter the following characters that can be processed by the application:   * Carriage Return (CR, ASCII 13) * Line Feed (LF, ASCII 10) * Pipe | * Semi-colon ; | Command injection requires proper data validation. Apply default deny and white list checking (e.g. define what is allowed and reject everything else). Validate input length and range by checking size and boundaries (e.g. min and max length), and validate input type and expected formats using regular expressions.  A common exploit for command injection includes incorporating a carriage return line feed in to the user supplied data. Often this is done to control the redirection URL. CR and LF characters should be escaped from input to prevent being processed by the application | All applications that accept input. |

#### Insecure Coding Examples

Refer to this reference for coding examples vulnerable to injection of CRLF to generate HTTP response splitting: <http://www.webappsec.org/projects/threat/classes/http_response_splitting.shtml>

#### Secure Coding Examples

The following code example shows how to filter CRLF (\r and \n) using regular expressions with javascript:

<html>  
 <head>  
 <title>textarea.htm</title>  
 <script type="text/javascript">  
 function crlf() {  
 var form = document.forms[0];  
 var data = form.data.value;  
 var regx = new RegExp("^(?:\r\n)+|(?:\r\n)+$|((\r\n)\2)\2+");  
 regx.global = true;  
 regx.multiline = false;  
 var temp = data.replace(regx,"$1");  
 temp = temp.replace(regx,"$1");  
 alert(data.length + " : " + temp.length);  
 form.data.value = temp;  
 }  
 </script>  
 </head>  
 <body>  
 <form>  
 <textarea name="data" cols="50" rows="10"></textarea>  
 <br><input type="button" value="OK" onclick="crlf()">  
 </form>  
 </body>  
</html>

### Log Injection

Log injection, or log forging, is the process of un-validated user input ending up in the application logs. This allows an attacker the opportunity to cover their tracks by injecting false log statements into the application and other system log files.

Log files are generated for audit purposes, statistical analysis, and debugging purposes. Log files that contain user controlled input that has not undergone validation are prone to the log injection attack. Log files that have been attacked cannot be trusted sources of audit data, nor can it be used for an effective debug session because the analysis is misdirected due to the injected statements.

#### Recommendations

|  |  |  |
| --- | --- | --- |
| **Requirements** | **Context** | **Applicability** |
| Log injection is dangerous because it helps the attacker cover up their tracks and renders the logs useless for investigative and debug purposes. Validating and user supplied input to the logs ensures that the integrity of the log files stays intact. | Validate all user supplied input entered into the log statements. | All applications that accept input. |

#### Insecure Coding Examples

The following example illustrates how log injection works. Un-validated data from the request is logged using standard application logging functionality. It is possible for attackers to insert log statements that cover their tracks.

String val = request.getParameter("val");

try {

int value = Integer.parseInt(val);

}

catch (NumberFormatException) {

log.info("Failed to parse val = " + val);

}

#### Secure Coding Examples

In the example herein is shown how to configure a facility such as log4j to log data securely. In the configuration file you can set the log level and identify the appender:

log4j.rootLogger=debug, R

log4j.appender.R=org.apache.log4j.ConsoleAppender

log4j.appender.R.layout=org.apache.log4j.SimpleLayout

The following code example shows how to configure the log level, invoke the logger and log the message.

PropertyConfigurator.configure(<config file name>);

Logger log = Logger.getLogger(<logger name>);

log.debug(<message>);

### HTTP Response Splitting

HTTP Response splitting occurs when normally only one response header would be returned, but through code the response is split into two parts. Typically this is exploited by allowing un-validated data enter the web application which can lead to XSS attacks.

HTTP Response splitting attacks take place where the application places un-validated user data in HTTP response headers. This is possible when using HTTP.sendRedirect() or by adding cookies to the headers, thereby forcing a set-cookie directive and splitting the response in two.

Typically an attacker injects malicious or unexpected characters such as CRLF in user input which is then used for a 302 redirect in the location of a set-cookie header. The original response will be maliciously split into two if the embedded user data properly terminates the original HTTP response and generates a new HTTP response, which includes malicious code to be run against the target.

This injection is possible if the application (that runs on top of the web server) places un-validated user data in a redirection, cookie setting, or any other manner that eventually causes user data to become part of the HTTP response headers.  
  
With HTTP Response Splitting, it is possible to mount various kinds of attacks: XSS, browser cache poisoning, and cookie poisoning to name a few.

#### Recommendations

|  |  |  |
| --- | --- | --- |
| **Requirements** | **Context** | **Applicability** |
| Validate all user supplied input in HTTP response headers, the application should filter invalid characters such as CR = %d = \r and LF =%0a =\n. Input validation should be performed on the server side. As general rule the client, including cookies should not be trusted without being validated on the server side. | By checking for the malicious injection of characters, the application will mitigate the risk of a malicious user gaining control of the HTTP headers and the body of the response the application intends to send. It also mitigates the risk of creating additional responses that would be under their control. | All applications that accept input. |

#### Insecure Coding Examples

The following example splits the HTTP response by creating a cookie using un-validated data and adding it to the response.

Cookie myCookie = new Cookie(MY\_COOKIE, request.getParameter(“my\_param”));

myCookie.setVersion(0);

myCookie.setSecure(true);

myCookie.setPath("/");

response.addCookie(myCookie);

#### Secure Coding Examples

Refer to the example in 6.5.5 as mitigation for CRLF injection that can cause HTTP response splitting. In the case of use un-validated cookies you should refer to the data validation examples.

### Buffer Overflow

Buffer overflow attacks result from placing a larger amount of data in memory than the allocated space. The excess data can cause the program to crash or run an unauthorized command as a result of the overflow of data in memory. Often the commands are malicious in nature such as del \*.\*

#### Recommendations

|  |  |  |
| --- | --- | --- |
| **Requirements** | **Context** | **Applicability** |
| Check the bounds access for every input that is read from the network, a file or from the command line and transferred/copied/memory allocated to internal data structures (e.g. arrays). Use only safe string handling calls (e.g. strsafe, STL strings). Compile code with stack protection options (e.g. /GS compiler switch). | Computers execute applications by storing program and data in memory. Applications written with good coding practices verify and validate inputs. If the input is not the one that is expected it should rejected by the application by throwing and error. Let’s take the login validation example. At the login prompt the user will enter username and password. Let’s assume that the application expects a six character password and receives a 27 character password. Let’s assume the password size is not verified, the program will process the data regardless of whether the password is valid or not. The extra 21 characters of the password will be processed, inserted in system memory causing arbitrary commands to execute, potentially seizing control of target machine. The key to prevent this to happen is to validate input data such as checking input data size before processing on the data itself.  The first line of defense from stack-based buffer overflows is applied by programmers. If possible, all C string manipulation functions should be marked unsafe and their usage should be deprecated. C language functions such as “fgets, gets, getws, memcpy, memove, scanf, sprintf, strcat, strcpy” and other defined in “string.h” headers could potentially generate buffer overflows if not handled correctly. The same applies for Microsoft libraries such as “wcscpy, \_tcscpy, \_mbscpy, wcscat, \_tcscat, \_mbscat CopyMemory, etc”. Some Microsoft string manipulation function such as “MultiByteToWideChar” are prone to errors: the function requires a maximum size as the number of characters, but programmers often give the size as bytes (the more common requirement), resulting in a buffer overflow vulnerability. Even if performing validation of inputs and bound checking can prevent buffer overflows to happen, finding such functions in the application is like finding code vulnerabilities: it could tell an exploit creator where to look for buffer overflow exploitation.  As a general rule it is better for a C and C++ programmer to take advantage of safe string manipulation functions such as the String class of the Standard Template Library (STL) and CString of Microsoft Foundation Classes (MFC).  Applications written in Java, Perl, C# (in safe mode) and other high level languages that do run time checking of array boundaries and have safe string types (i.e string objects) are less prone to buffer overflows. Unfortunately most operative systems are still written in C and C++ and therefore potentially subject to buffer overflows.  For applications developed for Windows with the Visual C++ .NET compiler a default defense to buffer overflow is enabled by compiling with the /GS option. The /GS option is similar to StackGuard, created by Cryspin Cowan (available at <http://www.immunix.org>) for the gcc compiler. It uses a test value known as a canary after the miner’s practice of taking a canary into a coal mine, to make static stack-based buffer overflows more difficult to exploit. | All applications that accept input. |

#### Insecure Coding Examples

The following C++ code example shows an input string (cBuffSource) being copied in a destination buffer (cBuffDestination) without checking the input string maximum length of 32 chars so that the buffer cannot be overwritten.

void DoSomething(char \*cBuffSource, DWORD cbBuffSource) {

char cBuffDestination[32];

memcpy(cBuffDestination,cBuffSource,cbBuffSource);

}

In the following example, the developer is trusting the user input that szData length is iLen but szData length is not checked to be iLen. This may result in a buffer overflow since this parameter is not validated:

void foo(char\* szData, int iLen){

char szDest[80] = {0};

if(iLen < sizeof(szDest)) {

strcpy(szDest, szData);

}

}

#### Secure Coding Examples

This code example shows how to use strcpy as safe as possible, that is making sure that the input string is not longer than the destination buffer (szDest). Notice that in this code implementation there is no user input string length parameter iLen since such parameter cannot be trusted.

void foo(char\* szData)

{

char szDest[32] = {0};

if(szData == NULL){

assert (false):

return;

}

if(strlen(szData) < sizeof(szDest)){

strcpy(szDest, szData);

}

}

### Missing XML Validation

Failure to validate the XML document against a Document Type Definition (DTD) or schema allows attackers to enter malicious XML format. DTD or schema validation does not validate the data in the XML document, but will ensure the structure of the documents is correct.

#### Recommendations

|  |  |  |
| --- | --- | --- |
| **Requirements** | **Context** | **Applicability** |
| Validate all XML documents with a DTD or schema | Validating XML documents with a schema or DTD is analogous to data input validation. Ensuring that the data is the correct type and follows the correct schema allows one less exploit by an attacker. | All applications that accept XML. |

#### Insecure Coding Examples

Errors in XML validation that can be exploited for vulnerabilities are illustrated at: -

<http://www.webappsec.org/lists/websecurity/archive/2006-04/msg00001.html>

#### Secure Coding Examples

Refer to <http://java.sun.com/j2se/1.5.0/docs/api/javax/xml/validation/package-summary.html>. As an example on how to validate an XML document with the Validation API in JAVA

### Missing URL Validation Before Redirection

Validation of redirection URLs is imperative to preserve the confidentiality of information. Header parameters are not tamper proof, thus a malicious attacker could sit in the middle and modify the HTTP headers which effectively redirect a user to a malicious site.

#### Recommendations

|  |  |  |
| --- | --- | --- |
| **Requirements** | **Context** | **Applicability** |
| The first rule of redirection is using relative URLs. Using relative URLs will ensure that a malicious attacker does not allow the user to get redirected outside of the original website. As a second precaution or if relative URLs are not an option, validate redirection URLs using a white list of known good redirection URLs. | The redirection URL as part of the request is fully validated to ensure that it is to an acceptable site to prevent an attacker from redirecting a user to their own malicious site. | All web applications. |

#### Insecure Coding Examples

The following function redirects to the file in the request form without validation that it is an acceptable (non-malicious) URL.

function foo() {

var fileName = Request.Form(“Filename”);

Response.write(fileName);

}

#### Secure Coding Examples

The following code shows how to validate the request by enumeration of only specific sites for which the redirection is allowed (white list of allowed sites).

function foo() {

var fileID = Request.Form(“FileID”);

If(fileID == ‘A’)

Response.write(“fileA.html”);

else if(fileID == ‘B’)

Response.write(“fileB.html”);

else

Response.write(“Error! Invalid File”);

}

### Missing Validation before File Upload

Accepting Uploaded files without any validation could lead to malicious scripts executing on the server. Improper validation of uploading files can cause server outages and data loss. A malicious user could upload files of various types, including executables, zipped files, and HTML files in the attachment portion of the application process. These files could then be downloaded and run by an unsuspecting user that had access to the attacker’s profile. This could potentially allow the attacker to compromise an internal system, if the attacker uploaded a malicious program and that file was executed by an internal employee.

#### Recommendations

|  |  |  |
| --- | --- | --- |
| **Requirements** | **Context** | **Applicability** |
| Before uploading a file into the application:   Validate the file type.   Do not let the user dictate the file extension   Do not place the file in the webroot directory   Do not allow execute right in the upload directory   Do not derive the name of the file from the user name   Scan the file before storing into the server   Display content back to the user. Before displaying the data back to the user, remember to encode the file contents in order to mitigate the execution of any malicious scripts. | Uploaded files should be validated for the expected type and format so that a malicious user cannot upload malicious content that can be processed by the application. For example binary executable files should not be uploaded at all.  Improperly formatted files such as XML for example can overload the process parsing the file for content and cause a denial of service. Viruses can be uploaded through the file upload function of the application therefore every file should be scanned for viruses before being stored on the server. Uploaded files containing text should be displayed back to the user in a safe format for display (i.e. encode the output in order to mitigate malicious scripts). | All applications that allow users to upload files |

#### Insecure Coding Examples

The following code example shows a file upload vulnerability. An attacker can exploit this vulnerability to upload arbitrary code and execute it in the context of the web server process. This may facilitate unauthorized access or privilege escalation; other attacks are also possible.

<form action="http://www.badexample.com/upload.php?Command=FileUpload&Type=File&CurrentFolder=/" method="POST" enctype="multipart/form-data">  
 File Upload<br>  
 <input id="txtFileUpload" type="file" name="NewFile">  
 <br>  
 <input type="submit" value="Upload">  
</form>

#### Secure Coding Examples

The following code example shows how to check for valid extensions in the file before being allowed for upload. In the example only safe data type are allowed (return true), and that includes text files (.txt), Rich Text Format files (.rtf), and some graphic formats. All others are denied by default (white list filtering)

bool AllowFileUploadByFileExtension(char \*szFilename) {  
    bool allowed = false;  
    if (szFilename) {  
        size\_t cFilename = strlen(szFilename);  
        if (cFilename >= 3) {  
            char \*szOKExt[] =   
                {".txt", ".rtf", ".gif", ".jpg", ".bmp"};  
   
            char \*szLCase =   
                \_strlwr(\_strdup(szFilename));  
   
            for (int i=0;   
                i < sizeof(szOKExt) / sizeof(szOKExt[0]);   
                i++)  
                if (szLCase[cFilename-1] == szOKExt[i][3] &&   
                    szLCase[cFilename-2] == szOKExt[i][2] &&   
                    szLCase[cFilename-3] == szOKExt[i][1] &&   
                    szLCase[cFilename-4] == szOKExt[i][0])  
                    allowed = true;  
        }  
    }  
    return allowed;  
}

### Missing Validation Before File Access

Time-of-check, time-of-use (TOCTOU) conditions exploit race conditions that occur when a resource is checked for a particular value, the value changes, then the resource is used, based on the assumption that the value is still the same as it was at check time. A race condition occurs when a program has an unexpected ordering of events that produces contention over a resource. This might occur because the application has not set exclusive access to the resource and could interfere with another process accessing the same resource. Many security problems occur if an outside attacker exploits race conditions to interfere with application that handle files without locking them. For example, in the case of a process opening and deleting a file without locking it, it is possible for an attacker to exploit the time gap between the time the file is checked for existence and the time is actually used to have another process setting a link to the resource (i.e. hijacking the resource handle) and use it to delete a file in another directory.

#### Recommendations

|  |  |  |
| --- | --- | --- |
| **Requirements** | **Context** | **Applicability** |
| To avoid TOCTOU attacks you should:   * Not modify global resources without locking. * Use the class “FileLock “to set and acquire a lock on a file resource * Not place files in directories with shared access such as directories for temporary files * Use atomic (reentrant) methods to check for resources. For example, use the “CreateNewFile” method to check for the existence of a file in a single atomic operation.    Handle locks with multiple processes correctly. Make sure to avoid deadlocks. For example, if the code throws an exception while holding the lock you’ll deadlock any other code that requires the lock. Always acquire multiple locks in the same order and release them in the opposite order acquired   Block signaling during non-atomic operations when the application uses signaling of events to synchronize thread access to resources   Ensure the file actually exists   Lock the resource before validation starts | Locking the resource before validation ensures that assumptions made during validation hold true while action is taken on the file. Additionally, file permissions should be checked in order to ensure that the user has appropriate access to the file. | All applications. |

#### Insecure Coding Examples

This psuedo code shows the faulty logic that might expose the application to TOCTOU attacks:

char\* fileName = “/tmp/tempFile”;

check\_access(fileName); // After the access check is done attacker changes /tmp/tempFile // to a symbolic link pointing at /etc/shadow

read\_file(fileName); // File being read is /etc/shadow rather than // the tempFile expected

#### Secure Coding Examples

Refer to TOCTOU examples of incorrect and correct code <https://www.buildsecurityin.uscert.gov/daisy/bsi/743.html?layoutType=plain>

Refer to TOCTOU examples in <http://www-cse.ucsd.edu/classes/sp07/cse127/127sp07Lec9.pdf>

### Missing Validation of Out Of range

This vulnerability applies to languages, such as C and C++ that do not have automatic range checking built in. Applications using these languages should check all range values before they are used. The issue of not performing bound checks can be exploited for buffer overflows.

#### Recommendations

|  |  |  |
| --- | --- | --- |
| **Requirements** | **Context** | **Applicability** |
| All upper and lower bounds should be checked before being processed by Application Interfaces (APIs). When using programming languages such as C and C++ array bounds when initializing arrays and string lengths before storing them in memory through string manipulation functions should be properly checked. Safe string manipulation functions should be used instead of unsafe string manipulation APIs (e.g. strcpy, strncpy) and bound checked STL containers (e.g. maps, vectors) instead of array types. | Validating the range on arrays is necessary to stop buffer overflow, null pointer exceptions, and a host of other issues. Failure to check the range before using the data could result in significant application failures. | All applications that accept user input. |

#### Insecure Coding Examples

See [Insecure Coding Examples](#_Insecure_Coding_Examples).

#### Secure Coding Examples

See [Secure Coding Examples](#_Secure_Coding_Examples).

### Missing Both Client Side And Server Side Validation

Client side validation is a convenient technique used to ensure that all required fields have data before making a round trip to the server. This type of validation is not required and does not enhance the security of an application. Not using server side validation is extremely dangerous because malicious data could enter the application and cause it to fail.

#### Recommendations

|  |  |  |
| --- | --- | --- |
| **Requirements** | **Context** | **Applicability** |
| Perform input validation on all user supplied inputs | Validation of user supplied input on the server side ensures that when an attacker bypasses client side controls, malicious data does not enter the program | All applications that accept user input. |

#### Insecure Coding Examples

In this example, the validation is missing by default both on the client and the server side. In this particular page a price item can be entered as input parameter and is not validated before processing an order. Since the price value exposed to the client as form field is trusted for valid, a malicious user can provide his own price in the form field and the order will be processed without price validation.

Public void doPost(HttpServletRequest req,..) {

String customerId = req.getParameter(“customerID”);

String sku = req.getParameter(“sku”);

//get price from FORM parameter

String stringPrice = req.getParameter(“price”);

Integer price = Integer.valueOf(stringPrice);

orderManager.submitOrder(sku,customerId,price);

}

#### Secure Coding Examples

This shows the same example with server validation of price value based upon a validated price queried parameter on the database.

Public void doPost(HttpServletRequest req,..) {

String customerId = req.getParameter(“customerID”);

String sku = req.getParameter(“sku”);

//Get price parameter from DB

Integer price = skuManager.getPrice(sku);

orderManager.submitOrder(sku,customerId,price);

}

## Configuration Management

Reusing application components and simplifying application complexity and deployment are typical goals of development teams. To achieve this goal, application modules are made configurable. Configuration management issues arise from improperly securing the application’s configurable parts, such as deployment descriptors and other external configuration files.

### Use of Java APPLETS

Java applets present a way of providing a rich user interface experience. However, Java applets are restricted technology at Citi.

#### Recommendations

|  |  |  |
| --- | --- | --- |
| **Requirements** | **Context** | **Applicability** |
| Use of applets is considered restricted technology by Citi and can be approved only when meeting specific requirements for usage. | Applets are discouraged because poorly written Java code can severely diminish the performance of the application and reflect poorly on Citi. Other security concerns include the ability to decompile and reverse engineer the code and could potentially provide confidential information to the attacker. | All web applications that use JAVA Applets. |

#### Insecure Coding Examples

Any applet tag in the HTML source is considered an issue. An example of HTML code that calls applet code is provided herein:

<APPLET CODE="MyApplet.class" WIDTH=200 HEIGHT=50>

<PARAM NAME=TEXT VALUE="Hello world!">

<P>Hello world!><P>

</APPLET>

#### Secure Coding Examples

When the use of an applet is permitted, risk mitigations (e.g. countermeasures) should be in place to prevent tampering of the applet such as signing as well as to prevent reverse engineering through code obfuscation. A design and secure code review should also be performed on the applet to make sure the applet is safe such as cleaned from any hard coded secrets.

<http://java.sun.com/developer/onlineTraining/Programming/JDCBook/signed.html>

<http://www.wutka.com/hackingjava/ch12.htm>

### Hidden Fields

Hidden form fields can be used as a mechanism to pass values to the server. As the hidden fields are sent from the client side they can be easily manipulated by malicious clients. Hidden fields that contain sensitive and confidential information could lead to disclosure of such information since hidden fields provide security by obscurity, and therefore are an ineffective security control for protecting the confidentiality of the data.

#### Recommendations

|  |  |  |
| --- | --- | --- |
| **Requirements** | **Context** | **Applicability** |
| HTML hidden fields are considered restricted technology and should be used with caution. A good rule of thumb to consider when using hidden fields is to remember that the data in the hidden field is no different that a regular text field and as such should be treated the same within the application. This means that all data coming through hidden form fields should be validated on the server side to mitigate any malicious tampering. | Hidden fields in HTML pages provide target rich environment for data tampering. On the other hand, hidden fields do not provide security since the data is just hidden to the user. Using hidden fields allows attackers to save HTML files, change the data and re-render them with the malicious data and submitting the data back to the server. When hidden parameters are used for storing sensitive information, such information can be captured by a malicious user using a web proxy. When hidden parameters hold business sensitive information such as price they can be modified to alter the business rules of the application (e.g. lower the price of an item before charge on a credit card). The use of hidden FORM fields for sensitive information such as PII exposes information to disclosure to unauthorized users. | All web applications that use hidden fields. |

#### Insecure Coding Examples

The example shows that prices for the items that are passed as hidden fields and can be manipulated during transmission to alter the price charged for the item being ordered.

<FORM method=post action="http://www.coolcart.com/cgi-bin/shop/coolcart.exe/sneaky/bastards">

<b><font size="5">Sale Price $169.95!</font></b><BR>

<INPUT TYPE="HIDDEN" NAME="ID" VALUE="PESL100">

<INPUT TYPE="HIDDEN" NAME="Describe"VALUE="Pro Series Telephone Analyzer">

<INPUT NAME="Qty" size=3 VALUE=""> Quantity <BR>

<INPUT TYPE="HIDDEN" NAME="Price" VALUE="169.95">

<INPUT TYPE="HIDDEN" NAME="Ship" VALUE="2.00">

<INPUT TYPE="HIDDEN" NAME="Multi" VALUE="N">

<INPUT TYPE="HIDDEN" NAME="Weight" VALUE="2.00">

<BR><INPUT TYPE="submit" VALUE="Add to Cart">

</FORM>

In the bad code example shown here HIDDEN fields are used to control server side credit card validation functionality. A malicious user can capture the hidden flag and change it to true to bypass credit card validation on the server side

<input value= “true” type= “hidden” bean=”thisFormHandler.verifyCreditCardNumber”/>

<input value=”true” type=”hidden” bean=”thisFormHandler.validatePrice”/>

#### Secure Coding Examples

Hidden fields can be used safely when they do not store personal and business sensitive information. For further information consult OWASP reference on examples on when and how hidden fields can be safely used <http://www.owasp.org/index.php/Data_Validation#Hidden_fields>

### Insecure Pseudo-random Number Generator

If the entropy of a random number generator is low, the ability for an attacker to guess the number will be high, making the solution Insecure.

#### Recommendations

|  |  |  |
| --- | --- | --- |
| **Requirements** | **Context** | **Applicability** |
| The application should only use a cryptographically secure pseudo-random number generator (PRNG). | Insecure randomness errors occur when a function that can produce predictable values is used as a source of randomness in a security-sensitive context. Computers are deterministic machines, and as such are unable to produce true randomness. Pseudo-Random Number Generators 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.  When unpredictability is critical, as is the case with most security-sensitive uses of randomness, use a cryptographic secure PRNG. For any application functionality that requires the use of a random number generator (*e.g.* password generation, creation of encryption keys and session tokens, etc.), it is required that a cryptographically secure PRNG, such as java.security.SecureRandom, is used instead of java.util.Random.  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.) | All applications that use random-number generation. |

#### Insecure Coding Examples

The following example uses the Random class. The Random class is pseudorandom and is deterministic.

Random random=new Random();

int pin=Random.next Int();

#### Secure Coding Examples

The next example illustrates the secure random class of java, which provides a cryptographically strong pseudo-random number generator (PRNG).

try {

// Create a secure random number generator

SecureRandom sr = SecureRandom.getInstance("SHA1PRNG");

// Get 1024 random bits

byte[] bytes = new byte[1024/8];

sr.nextBytes(bytes);

// Create a secure number generator with a seed

int seedByteCount = 10;

byte[] seed = sr.generateSeed(seedByteCount);

sr = SecureRandom.getInstance("SHA1PRNG");

sr.setSeed(seed);

sr.nextBytes(bytes);

}

catch (NoSuchAlgorithmException e) {

}

### Information Leakage

Revealing system data or debugging information helps an adversary learn about the system and form a plan of attack. An information leak occurs when system data or debugging information leaves the program through an output stream or logging function.

#### Recommendations

|  |  |  |
| --- | --- | --- |
| **Requirements** | **Context** | **Applicability** |
| The application should prevent logging any detailed system information such as server type and technology, stack traces and exception errors that reveal directory tree structure and the underlying database type to the end user. Such information should be made available only to developers and administrators of the application. | Messages, whether written to the log, to a user, or other output device, should be written with security in mind. Logging information, such as stack traces, potentially divulges information about the system, file structure, language, and platform. Always use caution when deciding what information to log, and in production environments refrain from using the stack trace unless you are in debug mode. Use brief messages that will assist in managing the application. 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, the message "Invalid password" reveals to the attacker that the user exists on the system. | All applications. |

#### Insecure Coding Examples

The following example illustrates multiple ways in which information is leaked from an application. In all cases the standard output device is utilized. The first example prints the stack trace information, the second statement prints the exception error message, and the third one gives away too much information to an attacker in the form of an error message.

catch(Exception ex){

ex.printstackTrace();

System.out.println(ex.getMessage());

System.out.println(“User password is invalid”);

}

#### Secure Coding Examples

In the following example no standard output device is used for logging application information to the user such as exception. Instead only necessary user error message information is shown to the user. In general you should be careful in choosing what you log and where you log it.

Catch (Exception ex){

logError(“caught exception while trying to connect to the database”);

//create error message for user

msg=”The userID or password you entered is incorrect”;

}

### Browser Auto-complete Enabled

Modern browsers have many convenience built into them. One convenience is the ability to auto-complete form field data, so that the user does not have to continually type such information when logging into the application. A common example of this is the user id and password fields. On the other hand, even if this provides usability and convenience (do not need to remember username, password) it represents a risk when the application can be accessed through a shared terminal such as in an internet café, kiosk etc

#### Recommendations

|  |  |  |
| --- | --- | --- |
| **Requirements** | **Context** | **Applicability** |
| While auto-complete functionality is nice it presents many security hazards, especially for folks using public computers. It is recommended to turn off the auto-complete functionality within the HTML form. | Auto-complete functionality stores information in the browser. Often user IDs and passwords use this functionality. It would be very easy for an attacker to use the victim’s PC and browse to the site where the auto-complete is enabled and the attacker would be able to gain unauthorized access to the site. Disabling the auto-complete forces all users at all times to login with their credentials. | All web applications. |

#### Insecure Coding Examples

With this coding example, the password will be saved on the client browser allowing for further compromise:

<INPUT TYPE="text" NAME="password" AUTOCOMPLETE="ON">

#### Secure Coding Examples

By setting the AUTOCOMPLETE value to OFF the value will not be saved on the client browser:

<INPUT TYPE="text” NAME=”password" AUTOCOMPLETE="off">

The AUTOCOMPLETE attribute can also be used to disable the feature for the entire form:

<FORM AUTOCOMPLETE="off">:

</FORM>

### No-cache headers not being set

Networking and application performance are a concern for application developers. Network components and browsers who serve the dynamic content utilize techniques to minimize performance hits. One technique is to cache web pages being accessed by the user’s browser. However, caching presents a security risk on pages that contain confidential and personally identifiable information, because the browser or proxy may substitute a cached page to satisfy a response. This means that unauthorized users could potentially see the data being cached from a previous web session and retrieve sensitive data such as, username password and other confidential and business sensitive data (e.g. bank accounts)

#### Recommendations

|  |  |  |
| --- | --- | --- |
| **Requirements** | **Context** | **Applicability** |
| It is required to explicitly disable caching on Citi web applications. | Cache control directives (cache control and pragma) must be set for both HTTP 1.1 and HTTP 1.0. HTTP 1.1 is the recent version of the protocol, but because not all devices on the Internet are using the 1.1 version of the protocol both cache headers must be specified.  The secure page cache directive should use three HTTP 1.1 headers for all pages containing sensitive or volatile data.  The headers are:   Cache-Control: no-cache, no-store   Pragma: no-cache   Expires: -1  META HTTP-EQUIV tags can also be used to set cache control HTTP headers from within the HTML document. However, the use of these directives is discouraged because the META tags must appear at the very top of the HTML HEAD section, and because a known bug in IE 4 allows pages to be cached even when these directives are present (for more information about that bug, refer to <http://support.microsoft.com/default.aspx?scid=KB;en-us;q222064>).  For a complete discussion of using HTTP headers and META HTTP-EQUIV tags to prevent caching, refer to: <http://support.microsoft.com/kb/q234067/>  In some cases, it is necessary to allow IE to cache a page for a very short period of time. For example, if the application allows Office documents to be downloaded, they must first be downloaded to a temporary cache before they can be opened. In order to allow this behavior, consider setting a short lifespan for the cache and implement the “must-revalidate” header.   * Cache-control: must-revalidate, max-age=30 * Expires: (current time + 30 seconds) | All web applications. |

#### Insecure Coding Examples

Whenever CACHE-CONTROL and PRAGMA directives are not set to NO-CACHE the web page contents are cached in the client browser.

#### Secure Coding Examples

Here is an example of secure cache directive settings:

CACHE\_CONTROL=no-cache

PRAGMA=no-cache

Expires:-1

By using the struts directive in struts-config.xml the same headers can be declaratively applied to every request with no coding:

<controller processorClass="MyRequestProcessor" nocache="true" />

### HTTP POST replay

HTTP POST replay attacks allow for an authorized user to use the “back” button to re-gain access to a session which timed out or was terminated by the “log out” command.

This attack was conducted by following this procedure:

 Victim logged into the application (with valid credentials).

 Victim used the application normally, and then pressed “Log Out” when done.

 Victim did not close browser when done using the site, and walked away from the console.

 Attacker walked up to the victim’s console and observed the web browser left open at the login screen.

 Attacker pressed the “back” button to return to the application. This action supplied the victim’s credentials to the server.

The attacker then had access to the application at the same level as the victim user whose connection was re-established.

An additional security concern raised by this condition was that a malicious user could recover the username and password of the victim. This would require a personal proxy server such as Achilles or Paros, or a packet capture package such as Ethereal to eavesdrop on the traffic being sent from the browser to the server. In both scenarios, successfully exploiting this attack requires physical access to a machine used by another user of the application. Furthermore, the browser window must be left open after the session has expired or after the user has logged off of the application. The medium “ease of exploitation” listed below is a compromise between a “high” rating for attackers with physical access to co-workers’ computers and “impossible” for isolated users.

#### Recommendations

|  |  |  |
| --- | --- | --- |
| **Requirements** | **Context** | **Applicability** |
| To prevent the POST replay attack you should:   Set the secure page cache directive using three HTTP 1.1 and HTTP 1.0 headers for all pages containing sensitive or volatile data.   Alter the authentication mechanism so that users’ credentials are submitted to a page which redirects the user with an HTTP 302 “Moved Temporarily” response. (HTTP 307 is technically correct, however older browsers will not interpret the code correctly, thus 302 is used instead.) | Setting the cache headers of no-cache prevents the web pages to be cached in the user browser and being replayed back to the server. Modern browsers will not cache parameters within their histories for pages which responded with HTTP 302 messages. | All web applications. |

#### Insecure Coding Examples

HTTP traffic analysis through a web proxy can show evidence of the issue such as in the example herein where the 200 OK response is returned to a HTTP POST and the POST is used to send sensitive information to the server such as during login.

**[REQUEST]**

POST /struts-examples-1/upload/upload-submit.do?queryParam=Successful HTTP/1.1

..

Connection: Keep-Alive

Cache-Control: no-cache

Cookie: JSESSIONID=0000kRnFv\_Lfx9wCzLhm0mvOFca:-1

**[RESPONSE]**

HTTP/1.1 200 OK

Content-Type: text/html;charset=ISO-8859-1

Content-Language: en-US

Content-Length: 1452

Date: Thu, 19 Jul 2007 15:01:13 GMT

#### Secure Coding Examples

The following example shows how to set redirect response to POST request by declaratively setting the redirect=true parameter in the action mappings done in the struts configuration xml file.

<action path=”/upload-submit” Type=”org.apache.struts.webapp.upload.UploadAction” Name=”uploadForm” Scope=”request” Validate=”true” Input=”input”>

<forward name=”input” path=”/upload.jsp”/>

<forward name=”display” path=”/display.jsp” redirect=true/>

</action>

### Weak Encryption and Weak Ciphers Supported

SSL encryption is commonly used to provide a server level solution for transmitting sensitive data across the Internet between a web server and a client. The SSL encryption takes place in two stages. The first stage uses a public key algorithm such as RSA to verify authenticity and to establish a symmetric session key for use in the second stage. The second stage uses a symmetric key algorithm, such as DES or RC4 for data encryption and transmittal. Public key encryption is computationally expensive, thus the reason for using a symmetric key algorithm for data exchange.

In some cases, the use of cryptography suffers from the lack of adequate security keys. All security of cryptographic operations that rely on a rather simple and predictable algorithm implementation is weak.

#### Recommendations

|  |  |  |
| --- | --- | --- |
| **Requirements** | **Context** | **Applicability** |
| The application should only implement Citi approved encryption algorithms and minimum key strengths. The minimum key strength for symmetric algorithms is 128 bit while 1024 bit for asymmetric algorithm. Symmetric algorithms such as blowfish are not allowed. Only SHA-1 hash algorithm is allowed. MD5 is part of restricted crypto standards.  While performing encryption and decryption rely on Citi approved standard strong standard ciphers that use security keys and random generation functions  Do not develop your own cryptography and always protect the keys using secure key storage. | When implementing SSL for web applications, it is not necessary to support browsers that use 40 or 56 bit encryption. By setting the server to require stronger encryption, such as 128 bit encryption you will be compliant with Citi encryption standards. When implementing hashes such as for storing passwords, you should use only approved hash algorithms such as SHA-1. Also HASH implementation should include a seed value.  Often times, the result of implementing your own cryptography translates in weak protection because is difficult to implement strong cryptography. Strong ciphers are already available and provide reliable encryption/decryption and have provisions for key management and configuration.  For the data in the key to remain secure, the key has to be protected. Use JAVA KeyStore to store the key securely, if the key is stored in the database insure that the value is kept confidential by hashing it, and require special permissions to access the table with the key.  Also make sure that the key changes over time so old keys cannot be re-used. | All web applications. |

#### Insecure Coding Examples

In this example MD5 is used inappropriately such as using weak keys (<128 bit) and hashing without a salt:

public static String digest(String password) {

MessageDigest md5 =MessageDigest.getInstance(“MD5");

byte[] hash = md5.update(password.getBytes());

return makeStringFromBytes(hash);

}

public static String makeStringFromBytes(byte[] bytes) {

String result = "";

for (int i=0; i<bytes.length; ++i) {

int n = bytes[i];

result = result + " " + Integer.toHexString(n);

}

return result;

}

#### Secure Coding Examples

The following example shows how to correctly implement a strong digest algorithm, such as SHA-1

public static String digest(String string)

throws NoSuchAlgorithmException, UnsupportedEncodingException{

MessageDigest sha1 = MessageDigest.getInstance("SHA-1");

sha1.update(string.getBytes());

String digest = makeStringFromBytes(sha1.digest());

return digest;

}

public static String makeStringFromBytes(byte[] bytes) {  
 String result = "";  
 for (int i=0; i<bytes.length; ++i) {

    int n = bytes[i];

result = result + " " + Integer.toHexString(n);

}

   return result;

}

### Hard Coded Secrets (passwords, cryptographic keys, PINs, etc.)

Using hard coded passwords, such as PINs, cryptographic keys, or any other secret increases the probability that the secret will be discovered.

#### Recommendations

|  |  |  |
| --- | --- | --- |
| **Requirements** | **Context** | **Applicability** |
| The use of hard-coded secrets has many negative implications, the most significant of these being a failure of authentication measures under certain circumstances.  Applications that authenticate to multiple back-end systems, such as a database, raise significant risk when secrets are hard coded. The vulnerability stems from the fact that the string is discoverable by decompiling the binary source. Hard coding secrets provide a false sense of security because the secret is not truly safe.  While performing encryption and decryption, do not write your own encryption algorithm but rely on modern strong standard ciphers that use security keys and random generation functions. When storing information into the keys, you should not hardcode them in files but store them securely in a secure repository and provide confidentiality by hashing the key value. Citi has two approved methods for storing IDs, password, keys, etc. For the UNIX platform the recommended solution is [PWP](https://gcateng-nj1p.nam.nsroot.net:6550/confluence/display/35164/PWP) and for windows the solution is [WiSP](http://www.emea.citigroup.net/emeaarchitecturegroup/104917/274997/157587/?a=5441). | This issue poses a risk for authentication controls: If hard-coded secrets are used, it is almost certain that malicious users will gain access through the account in question. For this reason this issue finding should be considered HIGH RISK when the ease of exploit and the impact are both HIGH (e.g. files containing sensitive information exposed to non-authorized users) | All web applications. |

#### Insecure Coding Examples

In this example the key is hard coded and can be accessed by a user by looking at the code. We should remember that even if source code is not delivered to the user, decompiled, and recovered from the executable using decompilers (e.g. JAD for Java and .NET, IDA pro for C++ and reflector/ salamander for C#).

The example also shows reliance on home-grown encryption algorithm. This result in weak protection because is difficult to implement strong cryptography. Also strong ciphers are already available and provide reliable encryption/decryption and have provisions for key management and configuration. Some standard crypto algorithms are also certified to be used for some applications such as FIPS-140 etc. For the data in the key to remain secure, the key has to be protected. Use JAVA KeyStore to store the key securely, if the key is stored in the database insure that the value is kept confidential by hashing it and be setting special permissions to access the table with the key. Also make sure that the key changes over time so old keys cannot be re-used. To minimize exposure keep unencrypted data close to the algorithm and minimize the time when data are kept un-encrypted in memory, perform just in time JIT processing while passing plaintext to an algorithm. Finally use the correct algorithm and correct key size (128 bit or more for symmetric algorithms and 1024 or more for asymmetric algorithms).

final public static byte key[] =

{(byte) 0x31, (byte) 0xAB, (byte) 0x05, (byte) 0xF7,

(byte) 0x45, (byte) 0x65, (byte) 0x98, (byte) 0xAB};

try {

encryptor.setKey(key);

plainText = new String(encryptor.decrypt(text));

}

catch (Throwable te) {

//handle the exception

}

#### Secure Coding Examples

This example shows use of standard JAVA APIs for safe key generation and storage. The function in the example opens an instance of the JAVA keystore by passing the keystore a key alias and the password for the secret key. Additionally, it shows how to get the certificate and the public key from the cert and returns a KeyPair

public KeyPair getPrivateKey(KeyStore keystore, String alias, char[] password)  {  
 try {

// Get private key

Key key = keystore.getKey(alias, password);

if (key instanceof PrivateKey) {  
        // Get certificate of public key  
          java.security.cert.Certificate cert = keystore.getCertificate(alias);

//Code to add certificate to the store  
          keystore.setCertificateEntry(alias, cert);  
      
          // Get public key

PublicKey publicKey = cert.getPublicKey();  
  
          // Return a key pair

          return new KeyPair(publicKey, (PrivateKey)key);

       }

}   
 catch (UnrecoverableKeyException e) {  
 }   
 catch (NoSuchAlgorithmException e) {  
 }  
 catch (KeyStoreException e) {  
 }

return null;  
}

### Storing Secrets in Readable (e.g. clear text) Format

The storage of secrets, passwords, PINs, and cryptographic keys, or other sensitive information, in a recoverable format subjects the system to compromise.

#### Recommendations

|  |  |  |
| --- | --- | --- |
| **Requirements** | **Context** | **Applicability** |
| Do not store secrets of sensitive data such as passwords, PINs, cryptographic keys in clear text in files and cookies. Protect the confidentiality of such data with encryption in storage and transit. Passwords should be stored using irreversible encryption (salted hashes). | Storing a plaintext password or other sensitive information in a configuration file allows anyone who can read the file to access the sensitive information from the resource. This increases the probability that the password will be used maliciously. | All applications. |

#### Insecure Coding Examples

The following properties file illustrates hard coded recoverable secrets in the properties file for an application. Without proper controls on this file it is extremely vulnerable to malicious attacks. However, it is still vulnerable to malicious insiders, see the example here below:

MyDatabaseConnection.properties

userid=admin

password=admin

server=server1

The following HTML source example shows username and password being stored in cookies

HTTP/1.1 302 Found

Date: Tue, 21 Feb 2006 19:16:08 GMT

Server: Apache/2.0.46 (Red Hat)

Accept-Ranges: bytes

X-Powered-By: PHP/4.3.2

Expires: Thu, 19 Nov 1981 08:52:00 GMT

Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0

Pragma: no-cache

**Set-Cookie: userid=jdoe**; expires=Thu, 01-Jun-2006 19:16:08 GMT; path=/

**Set-Cookie: password=dog**; expires=Thu, 01-Jun-2006 19:16:08 GMT; path=/

Set-Cookie: communityid=202; expires=Thu, 01-Jun-2006 19:16:08 GMT; path=/

#### Secure Coding Examples

A more secure approach is to use encryption to protect sensitive secrets such as passwords and database connection information. Refer to generic code examples that show how to protect the confidentiality of the data through encryption.

# Appendix

Encoding of Common Characters

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Characters** | **Decimal** | **Hexadecimal** | **HTML Character Set** | **Unicode** |
| **"** (double quotation marks) | &#34 | &#x22 | &quot; | \u0022 |
| **'** (single quotation mark) | &#39 | &#x27 | &apos; | \u0027 |
| **&** (ampersand) | &#38 | &#x26 | &amp; | \u0026 |
| < (less than) | &#60 | &#x3C | &lt; | \u003c |
| > (greater than) | &#62 | &#x3E | &gt; | \u003e |