not a MANual Tool for Risk Analysis

Risk Analysis Report [Validation & Verification Plan]

This document describes the features needed to be implemented in the verification step to avoid the most risky threats to the system

*** Report Generated by: guifre**

*** Date: Fri Oct 26 10:25:54 CEST 2012**

*** Issued by: Computer Architecture and Operating Systems (CAOS)**
Universitat Autonoma de Barcelona.

Department of Architecture and Operating Systems

Universitat Autònoma de Barcelona

This is an alpha release, if for some arbitrary reason this got to you and have any comment suggestion question idea, you can ping me at guifre.ruiz at the gmail dot com server ;-).

1. Time of Check to Time Of Use (TOCTOU)

1.1. Affected Components

* Path[0] Threat Agent[Anonymous User] Asset Element[config].
* Path[1] Threat Agent[Anonymous User] Asset Element[Logs].
* Path[2] Threat Agent[Identified User] Asset Element[config].
* Path[3] Threat Agent[Identified User] Asset Element[Logs].

1.2. Description

Summary

It is fairly common for an application to need to check some condition before undertaking an action. For example, it might check to see if a file exists before writing to it, or whether the user has access rights to read a file before opening it for reading. Because there is a time gap between the check and the use, an attacker can sometimes use that gap to mount an attack. Temporary Files: A classic example is the case where an application writes temporary files to publicly accessible directories. You can set the file permissions of the temporary file to prevent another user from altering the file. However, if the file already exists before you write to it, you could be overwriting data needed by another program, or you could be using a file prepared by an attacker, in which case it might be a hard link or symbolic link, redirecting your output to a file needed by the system or to a file controlled by the attacker.

Example

```
if (access("file", W_OK) != 0) {
  exit(1);
}
fd = open("file", O_WRONLY);
write(fd, buffer, sizeof(buffer));

If an attacker executes symlink("/etc/passwd", "file") after access() and before
open(), a succesful attack will be performed.
```

References

http://capec.mitre.org/data/definitions/29.html
http://cwe.mitre.org/data/definitions/367.html

1.3. Software Specifications (AKA countermeasures)

The code must be reviewed for statementes that validates a system resource and make sure that it is not possible to modify the resource before openning it. If there is no lock before the check or a verification after openning it, then re resource is probably vulnerable to this attack, and must be tested.

To test it you can create a symbolic link to a file which you do not have permissions.

2. Cross Site Request Forgery (CSRF)

2.1. Affected Components

* Path[0] Threat Agent[Anonymous User] Asset Element[VO Admin].
* Path[1] Threat Agent[Anonymous User] Asset Element[Identified User].
* Path[2] Threat Agent[Identified User] Asset Element[VO Admin].
* Path[3] Threat Agent[Identified User] Asset Element[Identified User].

2.2. Description

Summary

An attacker crafts malicious web links and distributes them (via web pages, email, etc.), typically in a targeted manner, hoping to induce users to click on the link and execute the malicious actionagainst some third-party application. If successful, the action embedded in the malicious link will be processed and accepted by the targeted application with the users' privilege level. This type of attack leverages the persistence and implicit trust placed in user session cookies by many web applications today. In such an architecture, once the user authenticates to an application and a session cookie is created on the user's system, all following transactions for that session are authenticated using that cookie including potential actions initiated by an attacker and simply "riding" the existing session cookie.

Example

```
The following code executes arbitrary HTTP requests in the users' broser:
<form name="badform" method="post"
  action="http://bank/Transfer">
  <input type="hidden" name="destinationAccountId" value="2" />
  <input type="hidden" name="amount" value="1000" />
 </form>
 <script type="text/javascript">
 document.badform.submit();
 </script>
•••

On the other hand, the code of the bank site is:
String id = response.getCookie(“user”);
userAcct = GetAcct(id);
If (userAcct != null)  {
```

```
   deposits.xfer(userAcct, toAcct, amount);
}
```

```
Since the credential is stored in the cookie, the victim will automatically be logged
in and the transferm carried out.
```

References

http://capec.mitre.org/data/definitions/62.html
http://cwe.mitre.org/data/definitions/352.html
https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_ Sheet
http://www.codeguru.com/forum/showthread.php?t=371569
http://en.wikipedia.org/wiki/Cross-site_request_forgery

2.3. Software Specifications (AKA countermeasures)

All HTML “form” fields should be tested against potential CSRF attacks. To do so, you have to create a malicious javascript code in a different domain that tries to force logged users to unintentionally submit requests to those forms. You can find more information regarding this in the previous summary, examples and references.

3. SQL Injection Attacks

3.1. Affected Components

* Path[0] Threat Agent[Anonymous User] Asset Element[Relational Database].
* Path[1] Threat Agent[Identified User] Asset Element[Relational Database].

3.2. Description

Summary

A SQL injection attack consists of insertion or injection of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to effect the execution of predefined SQL commands..

Example

```
sql_query= "SELECT ProductName, ProductDescription FROM Products WHERE ProductNumber =
" & Request.QueryString("ProductID")
It expects users to define the parameter ProductID with a number such as 123. However,
an attacker can execute arbitrary SQL queries by using SQL syntax to change the meaning
of the query.For example: '123 ;DROP ALL tables' would execute
SELECT ProductName, ProductDescription FROM Products WHERE ProductNumber = 123; DROP
all tables
```

References

http://cwe.mitre.org/data/definitions/89.html
https://www.owasp.org/index.php/SQL_injection
http://www.unixwiz.net/techtips/sql-injection.html

3.3. Software Specifications (AKA countermeasures)

All user supplied data used in SQL statements should be tested agains SQL Injection attacks. You should attempt the following vector attacks in the relevant fields:admin' --
admin' #
admin'/*
' or 1=1--
' or 1=1#
' or 1=1/*
') or '1'='1--
') or ('1'='1--
' HAVING 1=1 --
' GROUP BY table.columnfromerror1 HAVING 1=1 --
' GROUP BY table.columnfromerror1, columnfromerror2 HAVING 1=1 --
' GROUP BY table.columnfromerror1, columnfromerror2, columnfromerror(n) HAVING 1=1 --

You can use an automated penetration testing tool to facilitate this task. For instance, Zaproxy:
http://code.google.com/p/zaproxy/
You can find more information regarding the tests at:
http://www.softwaretestinghelp.com/sql-injection-%E2%80%93-how-to-test-application-for-sql-injection-attacks/

4. Reflected Cross Site Scripting (RXSS)

4.1. Affected Components

* Path[0] Threat Agent[Anonymous User] Asset Element[VO Admin].
* Path[1] Threat Agent[Anonymous User] Asset Element[Identified User].
* Path[2] Threat Agent[Identified User] Asset Element[VO Admin].
* Path[3] Threat Agent[Identified User] Asset Element[Identified User].

4.2. Description

Summary

Cross-Site Scripting attacks are a type of injection problem, in which malicious scripts are injected into the otherwise benign and trusted web sites. Cross-site scripting (XSS) attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user in the output it generates without validating or encoding it.

An attacker can use XSS to send a malicious script to an unsuspecting user. The end user’s browser has no way to know that the script should not be trusted, and will execute the script. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by your browser and used with that site. These scripts can even rewrite the content of the HTML page.

Example

```
<% String eid = request.getParameter("eid"); %>
  ...
Employee ID: <%= eid %>
In the previous example the eid parameter is displayed to the web browser without
escaping or validating HTML characters.
```

References

http://capec.mitre.org/data/definitions/32.html
http://cwe.mitre.org/data/definitions/79.html
https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

4.3. Software Specifications (AKA countermeasures)

One way to test for XSS vulnerabilities is to verify whether an application or web server will respond to requests containing simple scripts with an HTTP response that could be executed by a browser. For

example, Sambar Server (version 5.3) is a popular freeware web server with known XSS vulnerabilities. Sending the server a request such as the following generates a response from the server that will be executed by a web browser: http://server/cgi-bin/testcgi.exe?<SCRIPT>alert("Cookie"+document.cookie)</SCRIPT> The script is executed by the browser because the application generates an error message containing the original script, and the browser interprets the response as an executable script originating from the server. All web servers and web applications are potentially vulnerable to this type of misuse, and preventing such attacks is extremely difficult. You can test the following fuzz vectors:>"><script>alert("XSS")</script>&
">
>"
";!--"<XSS>=&{()}
<IMG SRC="javascript:alert('XSS');">
<IMGSRC=&#0000106&#0000097&<WBR>#0000118&#0000097&#0000115&<WBR>#0000099&#0000114&#0000105&<WBR>#0000112&#0000116&#0000058
<IMG SRC="jav	ascript:alert(<WBR>'XSS');">
Furthermore, you can find a set of fuzz vectors at
https://www.owasp.org/index.php/OWASP_Testing_Guide_Appendix_C:_Fuzz_Vectors

5. E-mail Headers Injection

5.1. Affected Components

* Path[0] Threat Agent[Anonymous User] Asset Element[Email server].
* Path[1] Threat Agent[Identified User] Asset Element[Email server].

5.2. Description

Summary

An attacker manipulates the headers and content of an email message by injecting data via the use of delimeter characters native to the protocol. Many applications allow users to send email messages by filling in fields. For example, a web site may have a link to "share this site with a friend" where the user provides the recipient's email address and the web application fills out all the other fields, such as the subject and body. In this pattern, an attacker adds header and body information to an email message by injecting additional content in an input field used to construct a header of the mail message. This attack takes advantage of the fact that RFC 822 requires that headers in a mail message be separated by a carriage return. As a result, an attacker can inject new headers or content simply by adding a delimiting carriage return and then supplying the new heading and body information. This attack will not work if the user can only supply the message body since a carriage return in the body is treated as a normal character.

Example

```
A common function to send emails is the following one:
<?php mail($to,$subject,$message,"From: $from
"); ?>
As we can see, the subject is user-controllable data. A maliciou user can trick the
SMTP server by providing the following string as $from header:
haxor@attack.com%0AContent-
Type:text/html%0A%0AMy%20%New%0A<u>HTML%20Anonymous%20Message.</u>%0A
The resulting e-mail would look like:
To: buddy@pal.xxx
Subject: Visit our site www.website.xxx !
From: haxor@attack.com
Content-Type:text/html

My New
<u>HTML Anonymous Message.</u>

Hello,
A friend thought you might want to see this page : www.website.xxx.
Bye Bye
```

References

http://capec.mitre.org/data/definitions/41.html
https://www.owasp.org/index.php/Testing_for_IMAP/SMTP_Injection
http://www.securephpwiki.com/index.php/Email_Injection

5.3. Software Specifications (AKA countermeasures)

The first step is to detect vulnerable parameters, the tester has to analyze the application's ability in handling input. Input validation testing requires the tester to send bogus, or malicious, requests to the server and analyse the response. In a secure application, the response should be an error with some corresponding action telling the client that something has gone wrong. In a vulnerable application, the malicious request may be processed by the back-end application that will answer with a "HTTP 200 OK" response message. After identifying all vulnerable parameters, the tester needs to determine what level of injection is possible and then design a testing plan to further exploit the application.
It is important to remember that, in order to execute an IMAP/SMTP command, the previous command must be terminated with the CRLF (%0d%0a) sequence. For more information refer to https://www.owasp.org/index.php/Testing_for_IMAP/SMTP_Injection

6. Insecure Cryptographic Storage

6.1. Affected Components

* Path[0] Threat Agent[Anonymous User] Asset Element[config].
* Path[1] Threat Agent[Anonymous User] Asset Element[Relational Database].

6.2. Description

Summary

Protecting sensitive data with cryptography has become a key part of most applications. Simply failing to encrypt sensitive data is very widespread. Applications that do encrypt frequently contain poorly designed cryptography, either using inappropriate ciphers or making serious mistakes using strong ciphers. These flaws can lead to disclosure of sensitive data and compliance violations.

Example

```
> select * from users;
id username password
1 Brett 5f4dcc3b5aa765d61d8327deb882cf99 |
2 Dan 3c3662bcb661d6de679c636744c66b62 |
The passwords in these table are 32 characters long. Could these passwords be MD5
hashes?

As with all hashing algorithms, MD5 hashes can't be reversed. However, they can be pre-
computed. Using a hash table lookup we can identify what the password is before it was
ran through the MD5 hashing algorithm.

After inserting 5f4dcc3b5aa765d61d8327deb882cf99 into the hash table lookup the
resulting password is returned. In this example, the password is "password."
```

References

http://cwe.mitre.org/data/definitions/326.html
https://www.owasp.org/index.php/Top_10_2007-Insecure_Cryptographic_Storage
http://cwe.mitre.org/data/definitions/327.html
http://bretthard.in/2009/09/insecure-cryptographic-storage/

6.3. Software Specifications (AKA countermeasures)

All sensitive data stored in the system should be encrypted using known to be strong cryptographic algorithms. You can find more information regarding code reveiw and penetration testing at https://www.owasp.org/index.php/Guide_to_Cryptography