IIG University of Freiburg

Web Security, Summer Term 2012

Injection Flows

Dr. E. Benoist

Sommer Semester

Table of Contents

Injection in PHP

```
$myvar = 'somevalue';
$x = $_GET['arg'];
eval('$myvar␣=␣' . $x . ';');
```

- **if "arg" is set to "`10; system('/bin/echo uh-oh')`"**
- **The system executes:** `/bin/echo uh-oh)`
- **The attacker receives the same rights as the user owning the http-deamon**

Use of variable variables in PHP

```
$safevar = "0";
$param1 = "";
$param2 = "";
$param3 = "";
# my own "register␣globals" for param[1,2,3]
foreach ($_GET as $key => $value) {
  $$key = $value;
}
```

- **If the attacker provides** `"safevar=bad"` **in the query string**
- **then** `$safevar` **will be set to the value** `"bad"`**.**

Shell Injection[1]

- Shell Injection is named after Unix shells,
- But it applies to most systems which allows software to programmatically execute command line.
- Typical sources of Shell Injection is calls:
  - `system()`,
  - `StartProcess()`,
  - `java.lang.Runtime.exec()`,
  - `System.Diagnostics.Process.Start()`
  - and similar APIs.
- Considere the following short program

```
<?php
passthru ( "␣/home/user/phpguru/funnytext␣"
            . $_GET['USER_INPUT'] );
?>
```

[1]Source: Wikipedia

Shell Injection (Cont.)

This program can be injected in multiple ways:

- `‘command‘` will execute command.
- `$(command)` will execute command.
- `; command` will execute command, and output result of command.
- `| command` will execute command, and output result of command.
- `&& command` will execute command, and output result of command.
- `|| command` will execute command, and output result of command.
- `> /home/user/phpguru/.bashrc` will overwrite file .bashrc.
- `< /home/user/phpguru/.bashrc` will send file .bashrc as input to funnytext.

Examples of injection

Suppose we have the following shell

```
<?php
if(isset($_GET['name'])){
  system('echo␣'.$_GET['name']);
}
?>
```

The following content will hack the system

- `‘ls ../../..‘` Executes a command, the returned value is given as a parameter to `echo`.
- Produces the following command line:

  echo ‘ls ../../..‘

- `$(cat /home/bie1/.emacs)` Displays the content of the emacs config file of user bie1.

  echo $(cat /home/bie1/.emacs)

Examples of injection (Cont.)

- `; touch /tmp/myfile.txt` Creates the following command

  echo ; touch /tmp/myfile.txt

  Makes a `echo`, then starts something new, it creates a new file `/tmp/myfile.txt` which is empty.
- `Hello World | wc` creates the following command line:

  echo Hello World | wc

  It makes a `echo` then its output is transfered to the `wc` (word count).
- `test > /tmp/test2.txt` Creates:

  echo test > /tmp/test2.txt

  It writes in the file `/tmp/test2.txt` the content that is given as output by `echo`.

Attacks using shell injection flow

- **An attacker can create any type of file**
  - A txt file
  - A PHP file
  - A shell file
- **Can see and modify config files**
  - Can visit directories
  - Can cat the content of a file
  - Can overwrite the content of an existing file
- **Attacker inherits the strength of web user**
  - If web server is run as a normal user: lot of possibilities
  - If the web user is restricted to the minimum, risk is smaller.

Defense agains Shell Injection

- **PHP offers functions to perform encoding before calling methods.**
  - `escapeshellarg()`
  - and `escapeshellcmd()`
- **However, it is not recommended to trust these methods to be secure**
- **also validate/sanitize input.**

XML-Injection[2]

- **The attacker trys to inject XML**
  - The application relies on XML (stores information in an XML DB for instance)
  - The information provided by the attacker is evaluated together with the existing one.
- **We will see a practical example**
  - A XML style communication will be defined
  - Method for inserting XML metacharacters
  - Then the attacker has information about the XML structure
  - Possibility to inject XML data and tags.

[2]Source: OWASP Testing Guide

Example

- **Let us suppose we have the following xmlDB file (information is stored in an XML)**

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
  <user>
    <username>gandalf</username>
    <password>!c3</password>
    <userid>0</userid>
    <mail>gandalf@middleearth.com</mail>
  </user>
  <user>
    <username>Stefan0</username>
    <password>w1s3c</password>
    <userid>500</userid>
    <mail>Stefan0@whysec.hmm</mail>
  </user>
</users>
```

Insertion of a new user

- **Is done with a form (with the GET method)**
  - Three fields: `username`, `password` and `email`
- **Suppose the clients sends the following values**
  - username=Emmanuel
  - password=B3n0is7
  - email= emmanuel@uni-freiburg.de
- **It produces the following GET request**

http://*www.benoist.ch/addUser.php?username=Emmanuel&*
password=B3n0is7&email=emmanuel@uni−freiburg.de

Insertion of a new user (Cont.)

- **The program will create a new XML `user`-node**

```
<user>
  <username>Emmanuel</username>
  <password>B3n0is7</password>
  <userid>500</userid>
  <mail>emmanuel@uni−freiburg.de</mail>
</user>
```

- **The new entry in entered inside the XML DataBase**

Vulnerability Testing

- **First step for XML Injection vulnerability**
  - Try to insert XML metacharacters
- **Metacharacters are:**
  - `'` (single quote)
  - `"` (double quote)
  - > and < (angular partentheses)
  - `<!-- -->` XML comment tags

Single Quote '

- **This character could throw an exception during XML parsing**
- **Suppose we have the following attribute**

  <node attrib='$inputValue'/>

- **So if: `inputValue = foo'` we obtain the following XML**

  <node attrib='foo''/>

  Which is a malformed XML expression: Exception at parsing the DB

Double Quote "

- **Has the same meaning as single quotes**
  - Can be used instead of ' if " is used in the document
- **So if we create the following XML**

  <node attrib="$inputValue"/>

  and we set `inputValue = foo"` we obtain the following XML

  <node attrib="foo""/>

  Which is also malformed

Angular parentheses < and >

- **We create an unbalanced tag**
- Suppose we use the value `username = foo<` in the user XML-DataBase
- This creates a new user:

```
<user>
  <username>foo<</username>
  <password>B3n0is7</password>
  <userid>500</userid>
  <mail>test@test.de</mail>
</user>
```

- This document is not valid anymore.

Comments tags `<!-- -->`

- **This sequence of fharacters is interpreted as the beginning and end of a comment.**
- One can inject this sequence in the username parameter: `username= foo<!--`
- The application would create such a node:

```
<user>
  <username>foo<!--</username>
  <password>Un6R34kb!e</password>
  <userid>500</userid>
  <mail>s4tan@hell.com</mail>
</user>
```

- Which is not valid

Ampersand &

- **Ampersand is used to represent XML entities**
  - Like `&symbol;`
  - Example `&lt;` for representing the character <
- **Can be used to test injection**
  - One can give `username=&foo`
  - The created node contains:

    <username>&foo</username>
  - Which is a malformed expression, `&foo` should be ended with a ;
  - but `&foo;` would also be undefined.

CDATA section delimiters

- `<![CDATA[` and `]]` are start and end delimiters of CDATA
- Inside a node a cdata section may be:

```
<node>
   <![CDATA[<foo>]]
</node>
```

- `<foo>` won't be parsed as markup is a character data.
- If a node is build in the following way

```
<username><![CDATA[<$userName]]></username>
```

- Tester will try to inject `]]` to invalidate the page.
  - if `username=]]>`
  - Then the node contains `<username><![CDATA[]]>]]></username>` which is not a valid XML fragment.

Result of the Test

- Once having tested all the possiblities,
  - Insert metacharacters of any type
- Result
  - The site is vulnerable to XML injection
  - The structure of the XML format has been discovered.

Possible Attacks using XML injection

- XSS Cross Site Scripting
- External Entity
- Tag Injection

Use CDATA for XSS

- Suppose we have a node containing some text that will be displayed back to the user

```
<html>
$HTMLCode
</html>
```

- Then an attacker can provide the following input

```
$HTMLCode = <![CDATA[<]]>script<![CDATA[>]]>alert('xss')
            <![CDATA[<]]>/script<![CDATA[>]]>
```

- And we obtain the following node

```
<html>
<![CDATA[<]]>script<![CDATA[>]]>alert('xss')
<![CDATA[<]]>/script<![CDATA[>]]>
</html>
```

Use CDATA for XSS (Cont.)

- **Durring the process, CDATA delimiters are eliminated, so the following HTML code is generated**

  <script>alert('XSS')</script>

External Entity

- **The set of valid entities can be extended by defining new entities.**
  - If the definition of an entity is a URI, the entity is called an external entity.
  - External entities force the XML parser to access the resource specified by the URI (Unless configured to do otherwise).
- **Such an application is exposed to XML eXternal Entity (XXE) attacks.**
  - For performing a denial of service of the local system
  - gain unauthorized access to files on the local machine
  - scan remote machines
  - perform denial of service of remote systems.

Test for XXE vulnerability

```
<?xml version="1.0" encoding="ISO−8859−1"?>
 <!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///dev/random" >]>
  <foo>&xxe;</foo>
```

- **This test could crash the web server (on a UNIX system),**
  - if the XML parser attempts to substitute the entity with the contents of the `/dev/random` file

Other XXE tests

- **Access the content of `/etc/passwd` file**

Tag Injection

- The tester has gained information about the XML structure
- It is possible to inject data and tags
- Example: priviledge escalation attack in the previous example
- Suppose we have the following inputs

```
Username: tony
Password: Un6R34kb!e
E-mail: s4tan@hell.com</mail><userid>0</userid><mail>s4tan@hell.com
```

Tag Injection (Cont.)

- The database becomes

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
    <user>
            <username>gandalf</username>
            <password>!c3</password>
            <userid>0</userid>
            <mail>gandalf@middleearth.com</mail>
     </user>
    <user>
                 <username>tony</username>
                 <password>Un6R34kb!e</password>
                 <userid>500</userid>
                 <mail>s4tan@hell.com</mail>
                 <userid>0</userid>
                 <mail>s4tan@hell.com</mail>
        </user>
</users>
```

Tag Injection (Cont.)

- Result
  - User Tony gets the userid 0 (super-user)
- Problem
  - Userid tag appears twice for Tony
  - If XML documents is associated with a shema or a DTD, it will be rejected
  - UserID tag has cardinality 1.
- Comment out the superfluous userid

```
Username: tony
Password: Un6R34kb!e</password><!--
E-mail: --><userid>0</userid><mail>s4tan@hell.com
```

Tag Injection (Cont.)

- The final XML is

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
  <user>
    <username>gandalf</username>
    <password>!c3</password>
    <userid>0</userid>
    <mail>gandalf@middleearth.com</mail>
  </user>
 <user>
    <username>tony</username>
    <password>Un6R34kb!e</password><!--</password>
    <userid>500</userid>
    <mail>--><userid>0</userid><mail>s4tan@hell.com</mail>
  </user>
</users>
```

Conclusion

- **Shell Injection**
  - Attacker inherits the priviledges of the user running the web server
  - Solutions: Filter/Sanitize input + reduce the priviledges to the minimum
- **XML Injection**
  - Attacker can force the server to load entities from outside
  - He can change the content of an XML database, and gain illegal priviledges in the application.
  - Solution: Filter/Sanitize input, allow no metacharcters in your normal inputs, or escape them.

References

- **OWASP Top 10 - 2007**
  `http://www.owasp.org/index.php/Top_10_2007`
- **A Guide for Building Secure Web Applications and Web Services**
  `http://www.lulu.com/content/1401012`
- OWASP Testing for XML Injection
  `http://www.owasp.org/index.php/Testing_for_XML_Injection_%28OWASP-DV-008%29`
- Wikipedia.org `Code injection`.