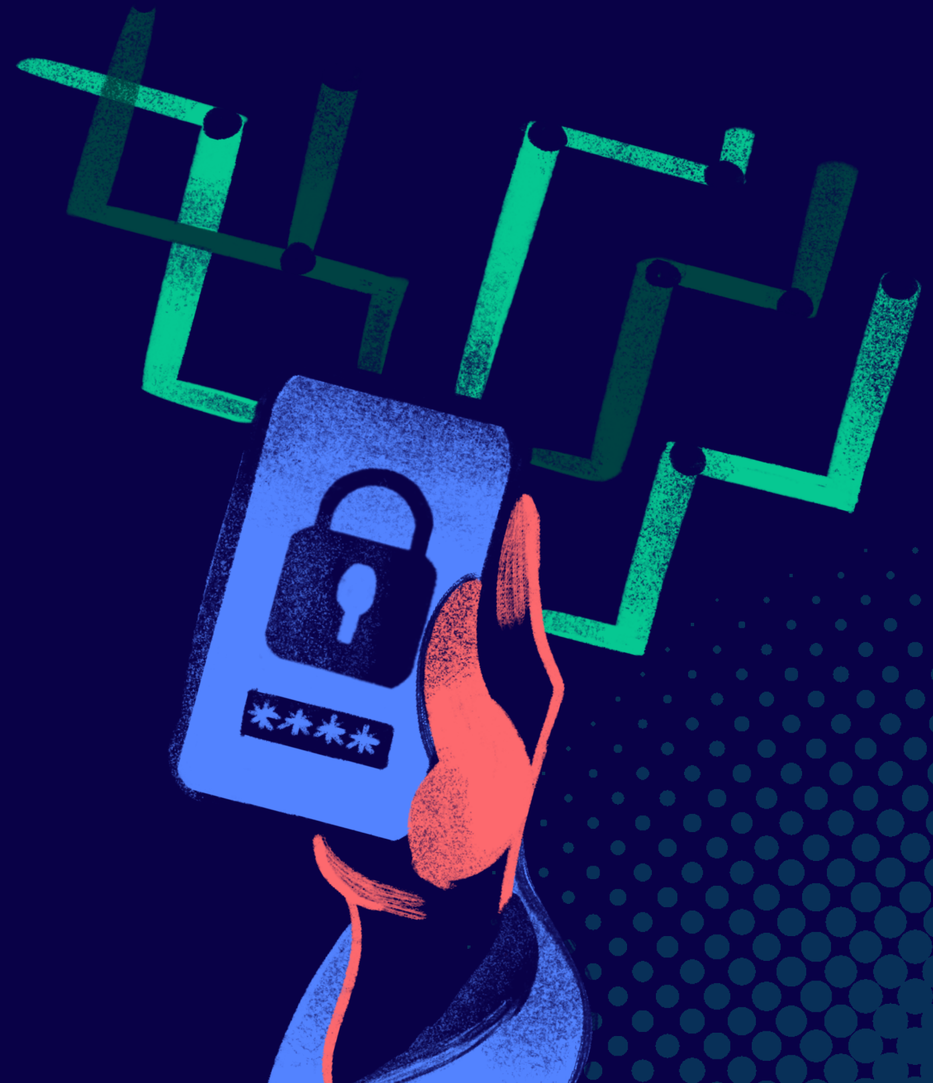
CYBER OFFENSE AND DEFENSE - A.Y. 2024-2025

WEB CACHE DECEPTION

Pierluigi Altimari - 257257
Emanuele Conforti - 252122
Francesca Filice - 252249
Jacopo Garofalo - 252093
Gianmarco La Marca - 252256

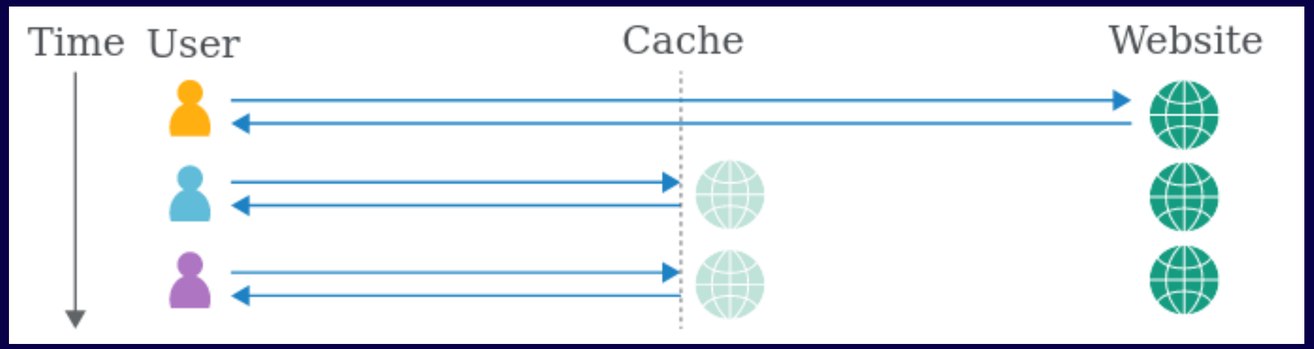


PRELIMINARIES (1)

What is a web cache?

A web cache is a system that sits **between** the origin server and the user.
When a client makes a request to a server, the cache server intercepts such a request and checks whether it contains the corresponding response:

- if yes (**cache hit**): the cache server serves the cached response to the client
- if no (**cache miss**): the cache server forwards the request to the origin server.



PRELIMINARIES (2)

CACHE RULES

Cache rules determine **what can be cached and for how long**.

They are set up to static resources, which:

- do not change frequently
- do not contain sensitive data

Dynamic content is **not** cached as it's more likely to contain sensitive information. *And that's what web cache deception exploits...*

WEB CACHE DECEPTION

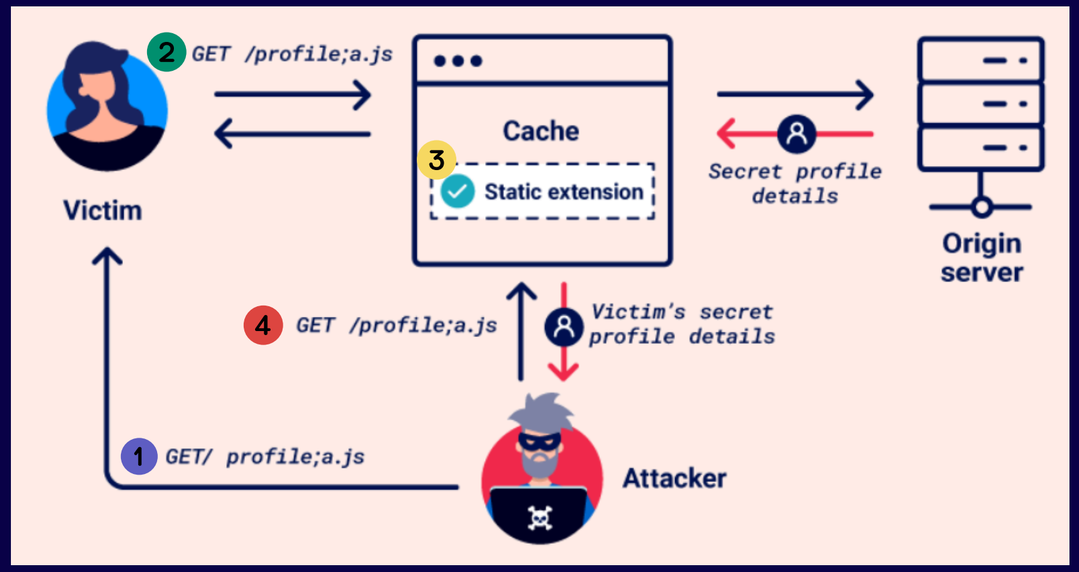
What is it?

An attack in which attacker **fools the cache server** to store a dynamic resource as if it was a static one.

How is it achieved?

Through **discrepancies** between how the cache server and the origin server handle requests.

WEB CACHE DECEPTION



Workflow

1. The attacker persuades the victim to visit a malicious URL.
2. The victim's browser is induced by the attacker to make an **ambiguous** request for sensitive content.
3. The cache **misinterprets** this as a request for a static resource and stores the response.
4. The attacker can then request the same URL to access the cached response, gaining unauthorized access to private information.

(WEB CACHE) DECEPTION VS POISONING

Web cache **deception**

Exploits cache rules to trick the cache into storing sensible data, which the attacker can later access

↓

✗ content manipulation

Web cache **poisoning**

Manipulates cache keys to inject malicious content into a web cache response

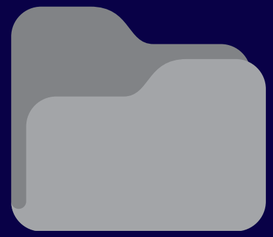
↓

✓ content manipulation

URL PATH-BASED CACHE RULES

Since web deception attacks exploit how cache rules are applied, it is important to understand how they work.
The **URL path-based rules** are among the most used ones. We can categorize them based on:

Static files extension rules



Static directory rules

File name rules

EXPLOITING STATIC EXTENSION CACHE RULES

PATH MAPPING

It is the process of **associating URL paths to resources on a server** (file, scripts, command executions)

The most common URL mapping styles are:

- **Traditional**, represents a direct path to a resource located on the file system.
  Example: http://example.com/path/in/filesystem/resource.html

- **RESTful**, doesn't match the physical file system structure; instead, they abstract file paths into logical parts of the API.
  Example: http://example.com/path/resource/param1/param2

MAPPING VULNERABILITIES

Discrepancies in how the cache and origin server map the URL path to resources can result in web cache deception vulnerabilities.
Consider the following example: **http://example.com/user/123/profile/wcd.css**

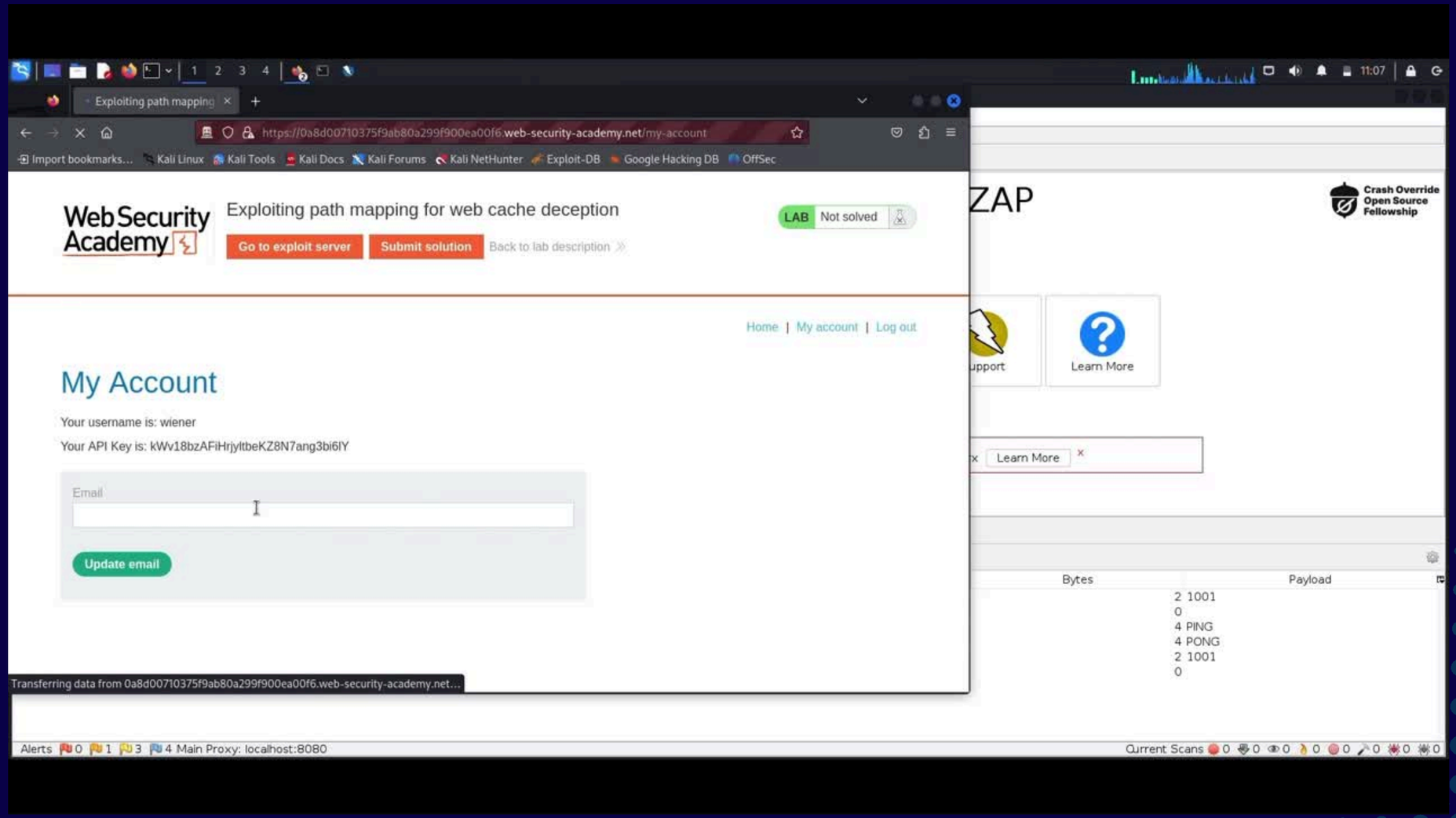
Origin Server

uses a **REST-style URL mapping** and interprets the URL as a **request for the /user/123/profile endpoint** and returns the profile information for user 123, **ignoring wcd.css as a non-significant parameter**

Cache Server

uses **Traditional URL mapping** and interprets the URL as a **request for a file named wcd.css** in the path user/123/profile. If the cache has a **cahe rule to store .css file**, it would cache the profile information as if it were a CSS file

LAB: EXPLOITING PATH MAPPING



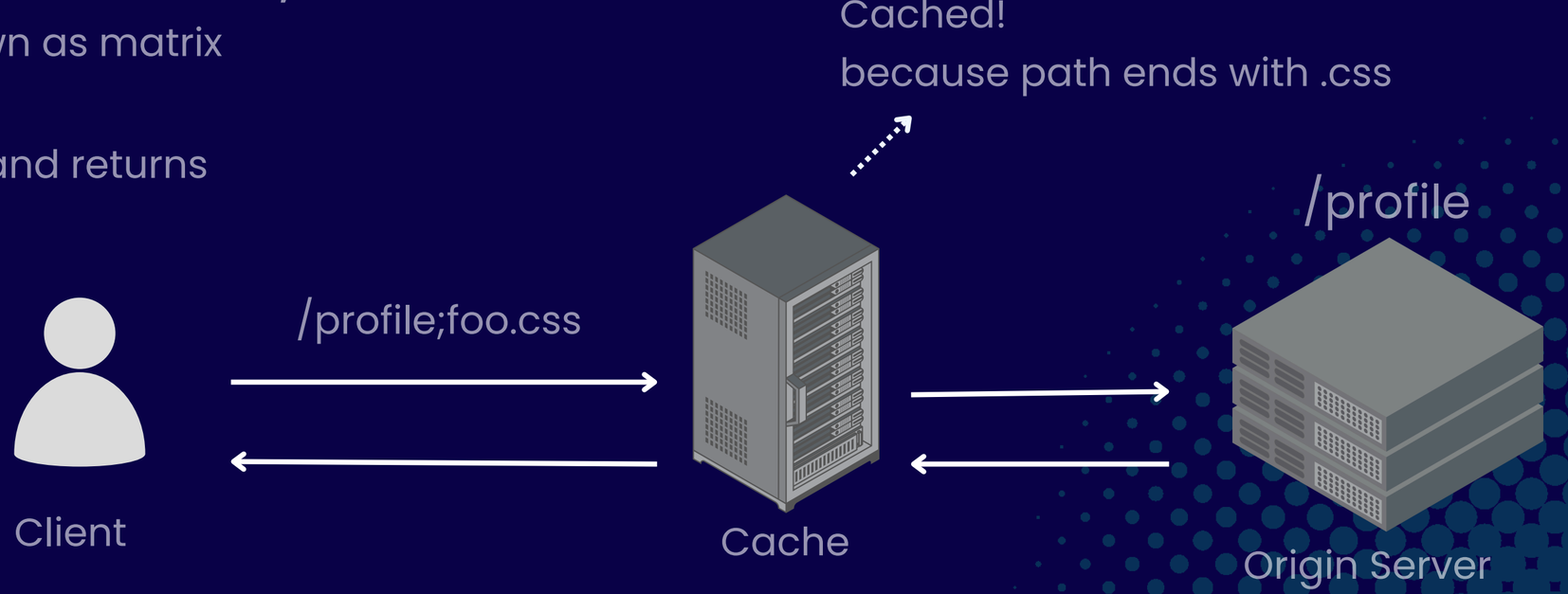
EXPLOITING DELIMITER DISCREPANCIES

How does it work

A delimiter is a special character that is included in the path viewed by the cache, but not by the origin server.

For example the Java Spring framework uses the **;** character to add parameters known as matrix variables.

It truncates the path after /profile and returns profile information



EXPLOITING DELIMITER DISCREPANCIES

```python
# this is the compromised url where we use the delimiter ";" to put wcd.js at the end of the path
# with this delimiter now;
#  the cache server sees: /my-account;wcd.js -> so it cached the response since it ends with .js
#  the origin server sees: /my-account -> so it serves the user's profile
def getCompromisedUrl():
    return f"{lab_url}/my-account;wcd.js"
```

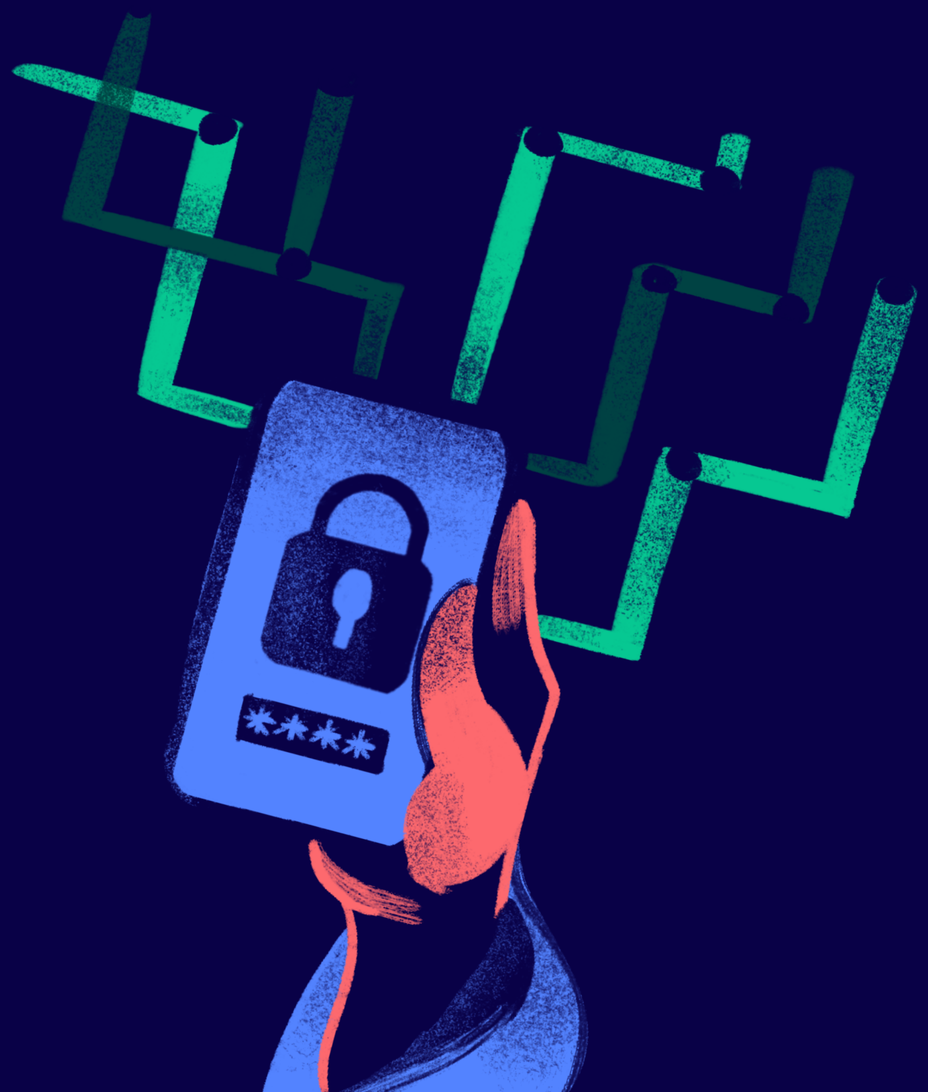
Script Available on github!

https://github.com/Absoluty02/Web-cache-deception

Workflow

1. find a valid delimiter used by the origin server
   a. If there is a character that after being added to the endpoint returns the exact same response then that character is a delimiter
2. confirm that the cache server does not use the delimiter
   a. otherwise we can try to **url encode** it be processed only by the origin server
3. determine if the cache has a rule that accept paths that end with a specific string

EXPLOITING STATIC DIRECTORY CACHE RULES



EXPLOITING STATIC DIRECTORY CACHE RULES

Static resources in specific directories:

- /static
- /resources
- /assets

Cache rules often target these directories

Normalization

Involves converting various representations of URL paths into a standardized format:

- resolving dot-segments
- decoding characters

Example: /abc/..%2fprofile is normalized into /profile

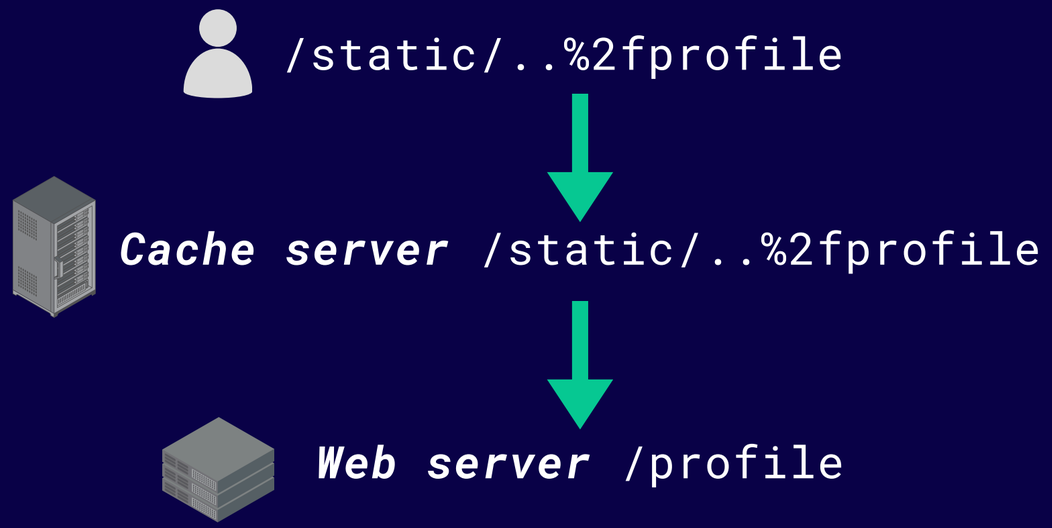
NORMALIZATION DISCREPANCIES

Discrepancies in how the cache and origin server normalize the URL can enable an attacker to construct a **path traversal payload**

We must be in one of the 2 following cases to create a normalization discrepancy:

- **Normalization by the origin server**: the web server normalizes the URL, while the cache server doesn't normalize it and has a cache rule for a specific folder
- **Normalization by the cache server**: the web server doesn't normalize the URL, but the cache server does normalize it and has a cache rule for /static folder

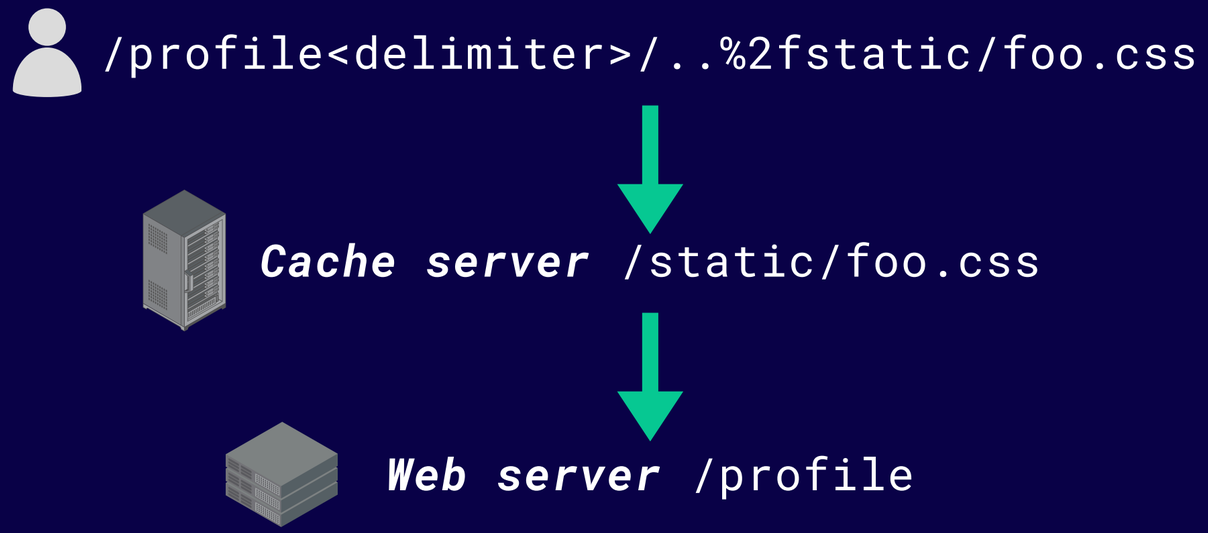
NORMALIZATION BY THE ORIGIN SERVER



- Consider the **path /static/..%2fprofile**
- The cache server doesn't normalize the URL and has a cache rule for /static folder. Hence, it caches /static/..%2fprofile
- The web server normalizes the URL, looking for /profile endpoint

/<static-directory-prefix>/..%2f<dynamic-path>

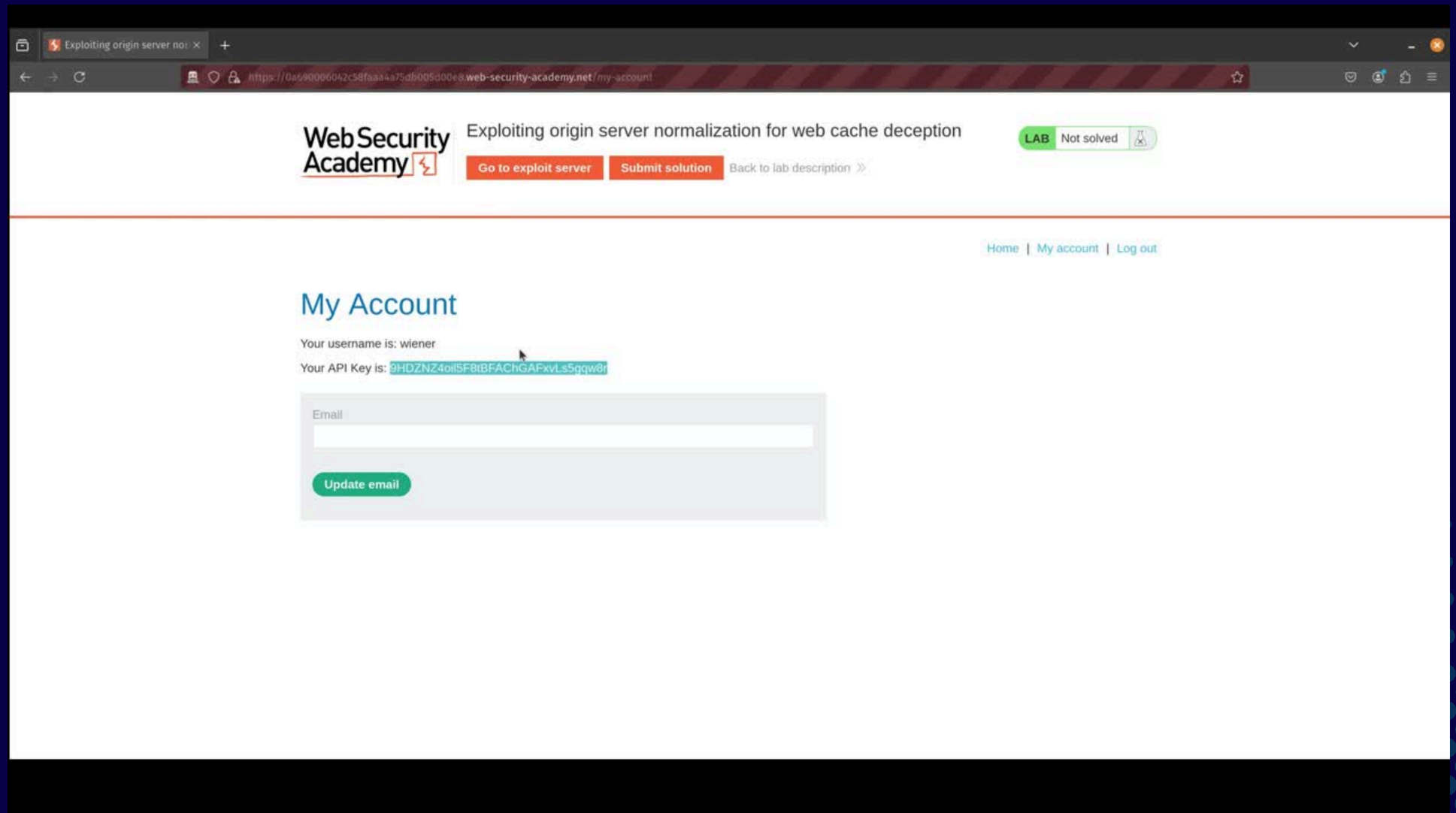
NORMALIZATION BY THE CACHE SERVER



- Consider the **path /profile<delimiter>/..%2fstatic/foo.css**
- The cache server does normalize the URL and has a cache rule for /static folder. Hence, it caches static/foo.css
- The web server doesn't normalize the URL, but the delimiter will truncate all the characters after it, then the server will directly look for the /profile endpoint

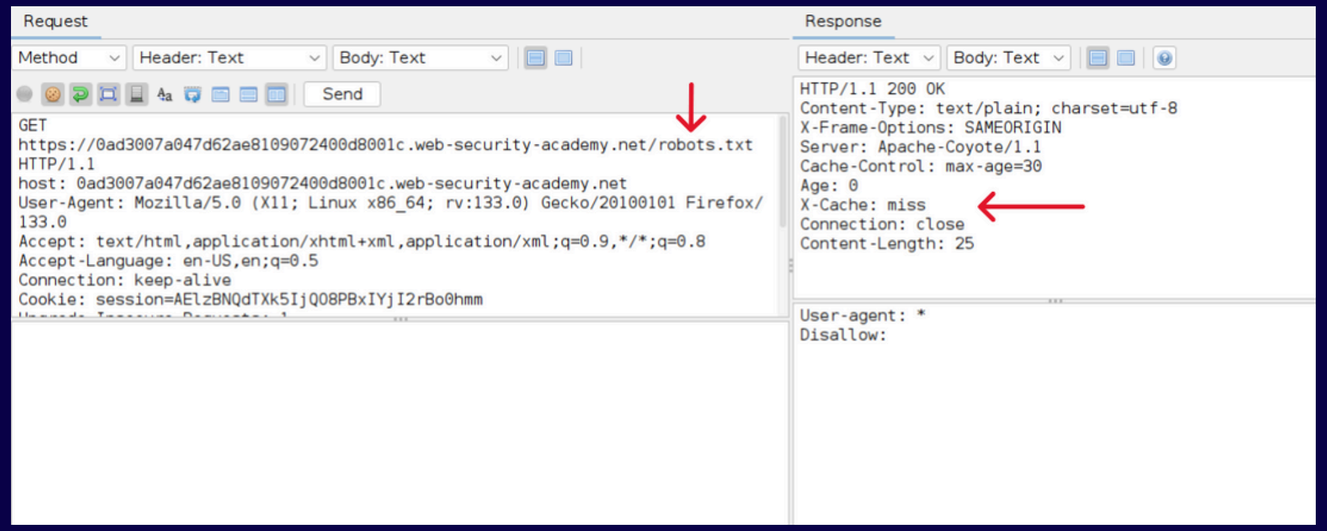
/<dynamic-path><delimiter>%2f%2e%2e%2f<static-directory-prefix>

LAB: EXPLOITING ORIGIN SERVER NORMALIZATION



EXPLOITING FILE NAME CACHE RULES

EXPLOITING FILE NAME CACHE RULES



There exists certain files that are very **common** in web servers, like:

- *robots.txt*
- *index.html*
- *favicon.ico*

They're often cached due to their infrequent changes. Cache rules target these files by matching the **exact** file name string

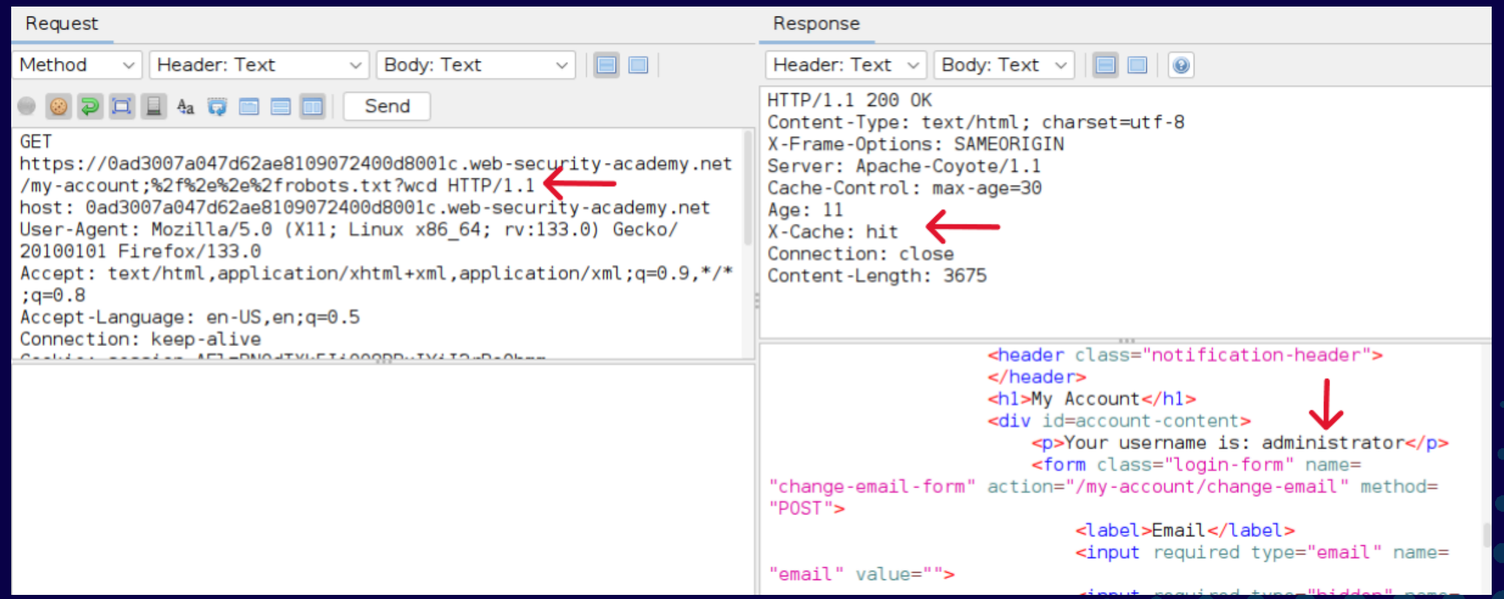
EXPLOITING FILE NAME CACHE RULES

We can use the same method used for exploiting static directory cache rules

Make a request with a path traversal sequence and an arbitrary directory before the file name. For example, */my-account%2f%2e%2e%2frobots.txt*

Response cached = path normalized
Response not cached = path interpreted as entered

Be careful! Response cached **only** if the request matches the exact file name



PREVENTION

Cache-Control headers for dynamic resources

Directives like *no-store* and *private* can save the day

Origin server behaviour = cache server behaviour

Verify that there aren't any discrepancies between how those two interpret URL paths

Use anti-web cache deception attacks tricks in your CDN

Set cache rules which verifies that the response *Content-Type* matches the request's URL file extension, like Cloudflare's Cache Deception Armor.

CDN properly configurated

Make sure that your caching rules don't override the Cache-Control header

THANK YOU FOR ATTENTION

See You Next →