Modern Web Exploitation Via priorities , desyncs and confusions in parameters

BY mohammed sultan

What is prioritizing inputs in server side ?

Server-side input prioritization refers to the order in which a backend application **evaluates, selects, or overrides** conflicting values coming from multiple input vectors representing the *same logical parameter*.

In other words:

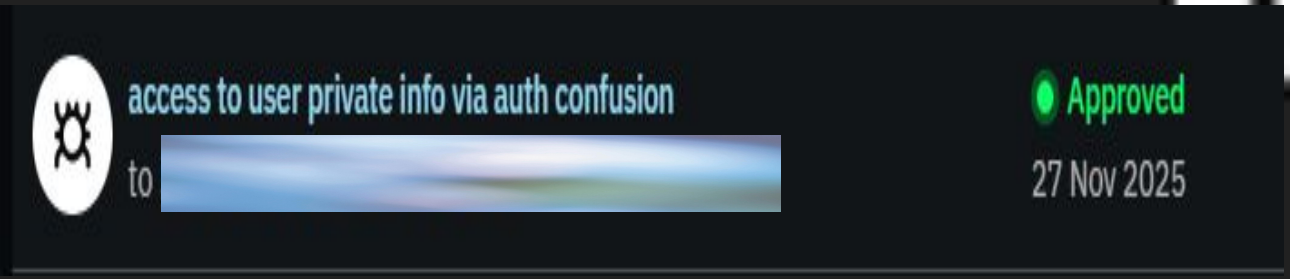
> **When the same parameters intent arrives from different sources — GET, POST, cookies, headers, JSON body, URL path, or even internal defaults — the server must choose which value "wins."**
> **That decision is the server's input priority.**

Where does it happen ?

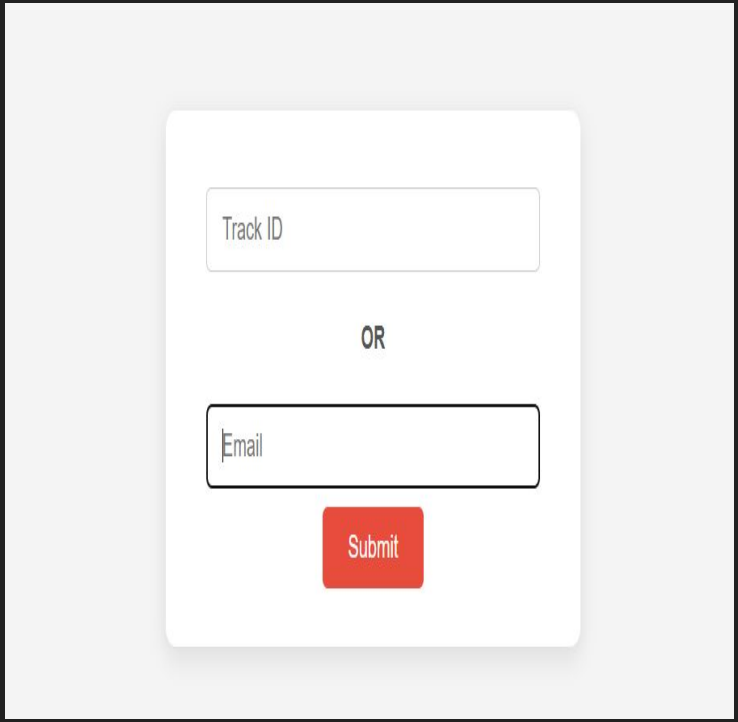
Weak binding and priority-overwriting issues typically occur in backend functions where multiple parameters or data sources influence the same value, but the server never clearly defines which one should take priority. For example, an API might accept both a **URL parameter** and a **JSON-body field** for the same setting, and because the backend doesn’t consistently compare or validate them, an attacker can override one with the other. Similarly, in functions where user-controlled fields are merged with server-side defaults, the application may “trust” whichever value is processed last, allowing the attacker to manipulate logic that developers assumed was protected. This mismatch between intended and actual parameter priority is what creates the vulnerability.

REAL WORLD WEBSITE FINDINGS :) – > best to understand

REAL



WHAT HAPPENED ?

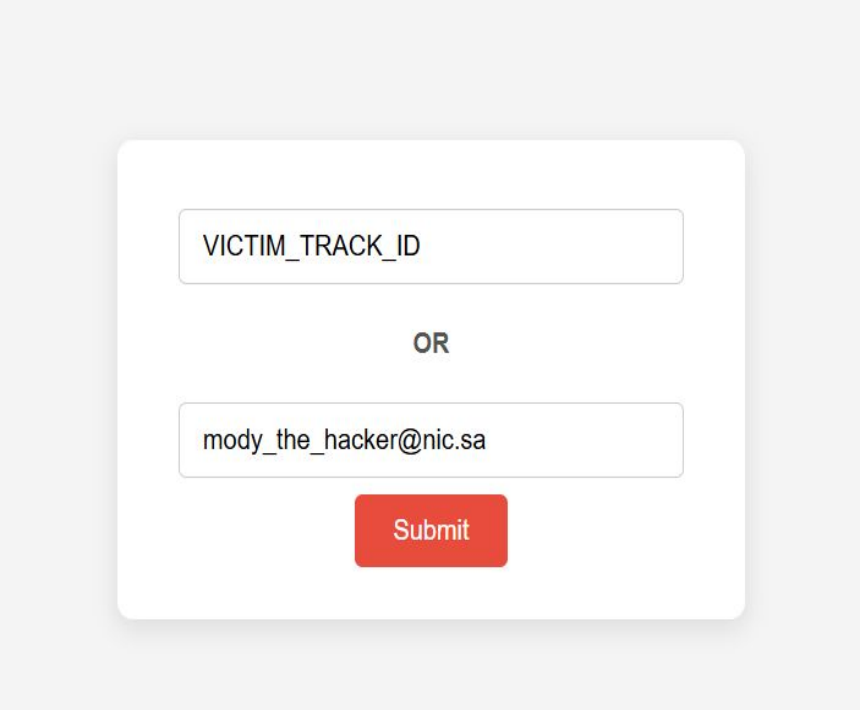


Request

Pretty Raw Hex

```
1 GET /XX/XX/XX?email=&tracking=123&otp=111111 HTTP/1.1
2 Host: XXXXXXXXXXXXXXXXXXXX.XXXXXXXXXXX.XXXXXXXXXXX
3 Sec-Ch-Ua-Platform: "Windows"
4 Accept-Language: en-US,en;q=0.9
5 Accept: application/json, text/plain, */*
6 Sec-Ch-Ua: "Not_A Brand";v="99", "Chromium";v="142"
```

What if we put an email and tracking id ?



What happened in the backend ?

The backend has a critical flaw in how it handles tracking IDs. When a user submits a tracking ID along with their email, the server simply does

```
receive( tracking_id , email )
```

```
and sendOTP( email ) -> improper binding + priority overwrite
```

without checking whether the email actually belongs to the tracking ID owner. When the OTP is entered, the backend validates it with

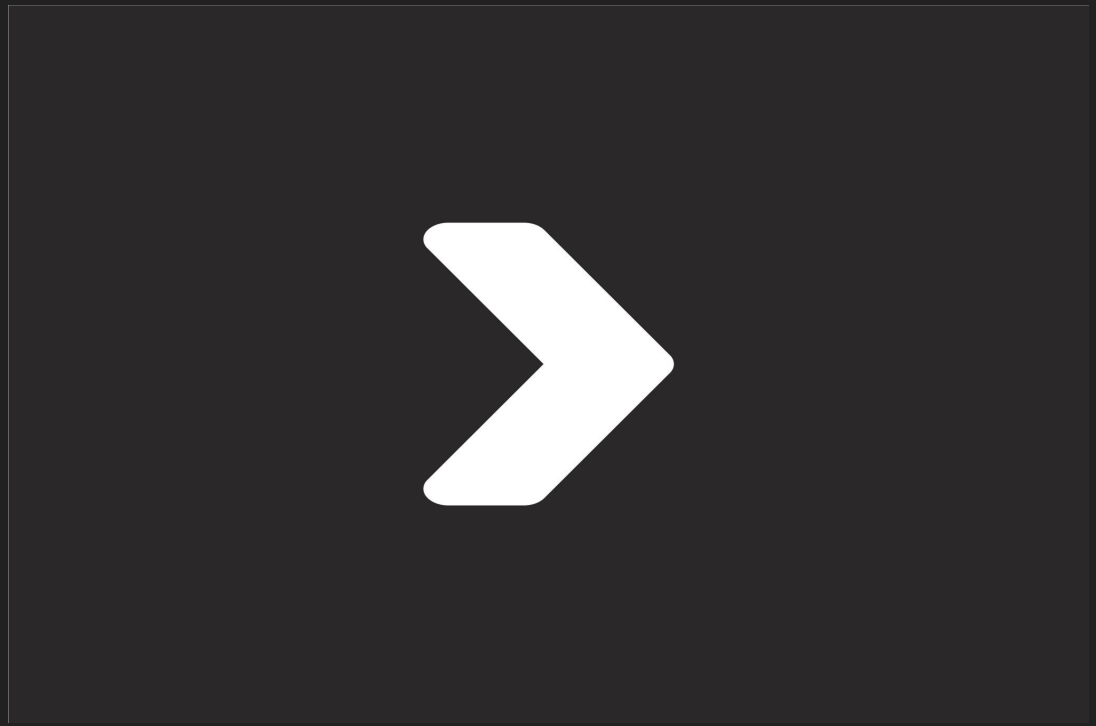
```
if OTP_is_valid( email ) : authorized = true,
```

again ignoring the connection between the email and the tracking ID. Finally, if the OTP is valid, the server returns the tracking information using

```
if authorized : return database.getTracking( tracking_id ).
```

This means that anyone with a valid OTP can access **any tracking ID**, creating a broken access control or IDOR vulnerability.

Advanced exploitation of priority overwriting via known parameter injection

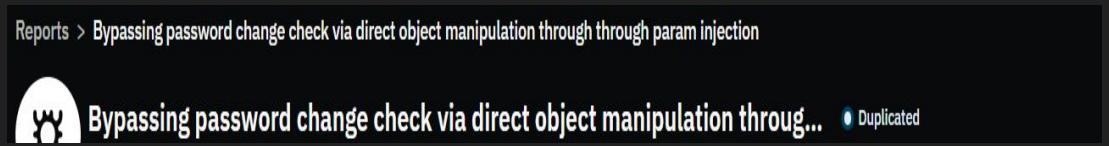


What is known parameter injection ?

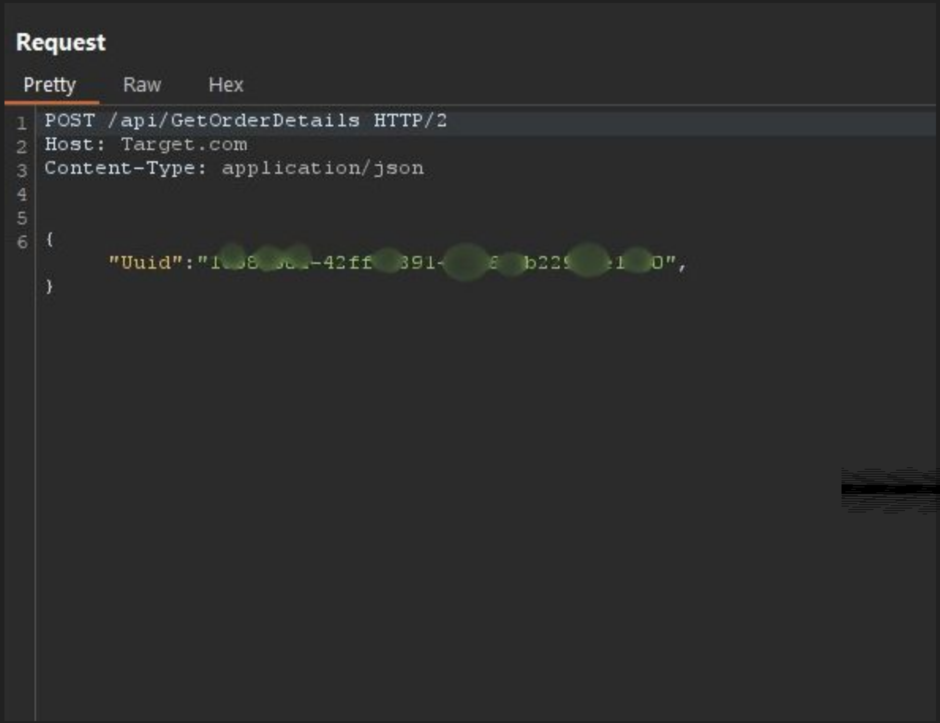
1 - lets say a server have the parameter (Password = “your_password”) when you sign in and account_name as parameter name of your account name

2 - but when updating your account name for example we can inject the parameter Password and see if it updates something

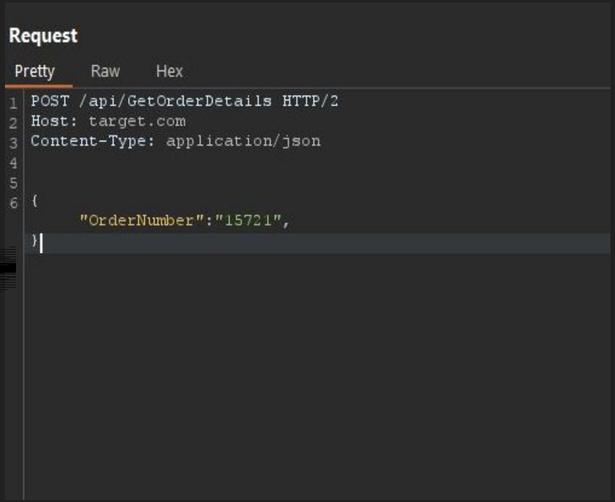
3 - and this technique is known as DOM – > DIRECT OBJECT MANIPULATION



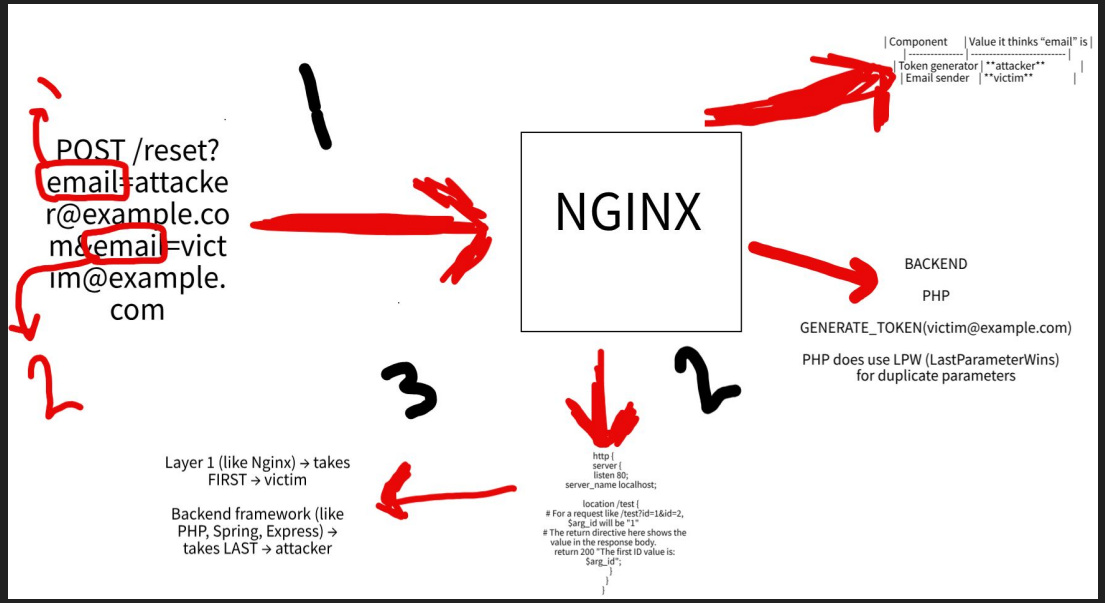
Causing priority bugs with Parameter injection ! ?



In the server response there is a parameter that says : ORDER NUMBER



Exploiting parameter pollution to exploit server side input Desynchronization (SSID)in modern frameworks



DEEP DIVE AND CODE ANALYSIS IN APACHE ?!

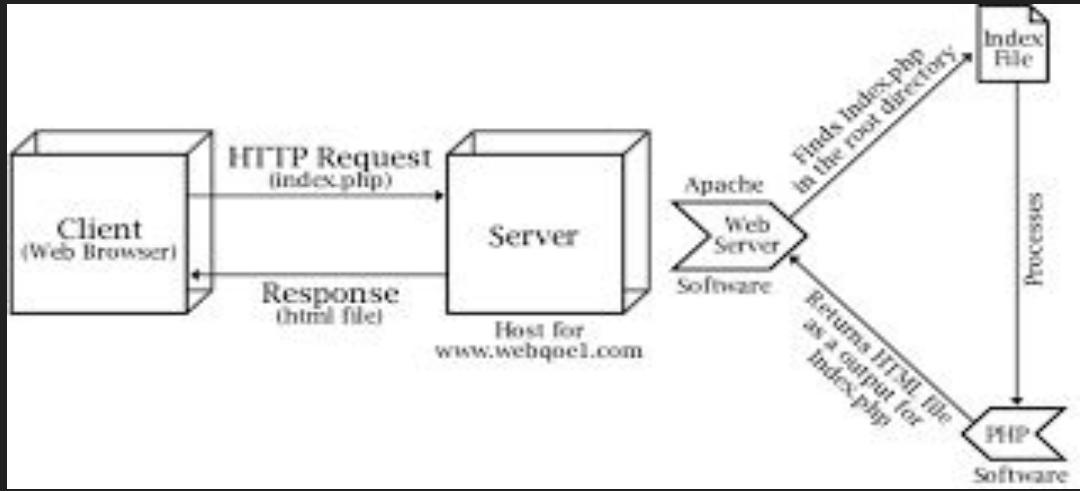
Apache is a web server that handles web requests.

You type a URL in your browser → Apache receives the request.

Apache decides which file or resource to serve.

Example: `http://example.com/index.html` → Apache serves `/var/www/html/index.html`.

Apache = the “manager” of all web requests.



How is apache built ?

Apache is older than most modern web stacks — it started in **1995**.
Its architecture is based on **modules** that plug into a central core.

Think of it like this:

`[Core Apache] ← loads dozens of modules → [mod_rewrite, mod_proxy, mod_auth, mod_php, etc.]`

Apache itself does very little.
Instead, modules do almost everything:

- URL parsing
- Authentication
- Proxying
- Rewriting
- Handling PHP
- Logging
- Access control

Apache's key idea:

"Every feature should be a module."

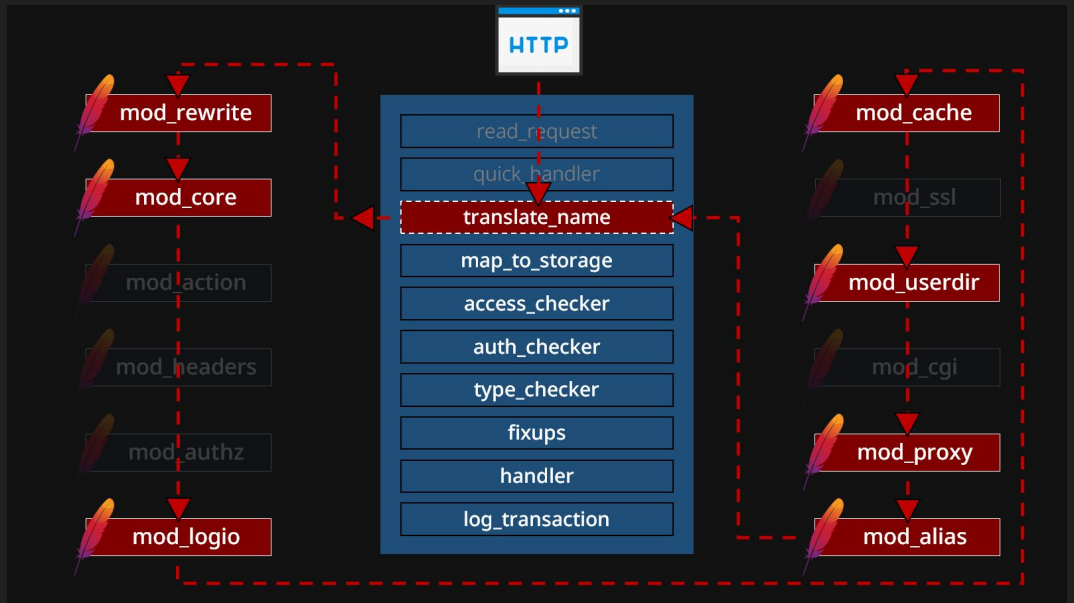
Good idea for flexibility.
BAD idea for security.

The entire Httpd service relies o
hundreds of small modules workin
together to handle a client's HTT
request. **Among the 136 modules listed by the official documentation, about half are either enabled by default or frequently used by websites**!

Apache is basically a **giant request pipeline**.

Why is this build interesting ?

What's even more surprising is that these modules also maintain a colossal `request_rec` structure while processing client HTTP requests. This structureincludes all the elements involved in handling HTTP, with its detailed definition available in include/httpd.h. All modules depend on this massive structure for synchronization, communication, and data exchange. As an HTTP request passes through several phases, modules act like players in a game of catch, passing the structure from one to another. Each module even has the ability to modify any value in this structure according to its own preferences!




```
typedef struct request_rec {
    char *method;      // "GET", "POST"
    char *uri;         // "/index.php"
    char *filename;    // Filesystem path
or URL
    char *args;        // Query string
    char *path_info;   // Extra path

    int status;

    struct request_rec *prev;
    struct request_rec *next;

    struct conn_rec *connection;
    struct server_rec *server;

    apr_table_t *headers_in;
    apr_table_t *headers_out;
} request_rec;
```

Confusing the code to bypass ACL ?

ACL in Apache:
An **Access Control List** that controls who can access a website or directory. It uses rules (like IP, user, or group) to **allow or deny** access.

Super short:
“ACL = rules that allow/deny access in Apache (based on IP, user, etc.).”

```
<Directory "/var/www/html/admin">
    Require ip 127.0.0.1
</Directory>
```

Exploiting modern setups !

🐘 What Is PHP-FPM?

PHP-FPM = *PHP FastCGI Process Manager*

It is a separate program (not part of Apache) whose job is:

- Run PHP code
- Manage PHP worker processes
- Handle concurrency
- Keep PHP running efficiently

Apache does **not** execute PHP directly.
Apache sends `.php` files to PHP-FPM, PHP-FPM executes them, and returns the output.

Think of PHP-FPM like this:

Browser → Apache → PHP-FPM → Apache → Browser
Browser requests /index.php.

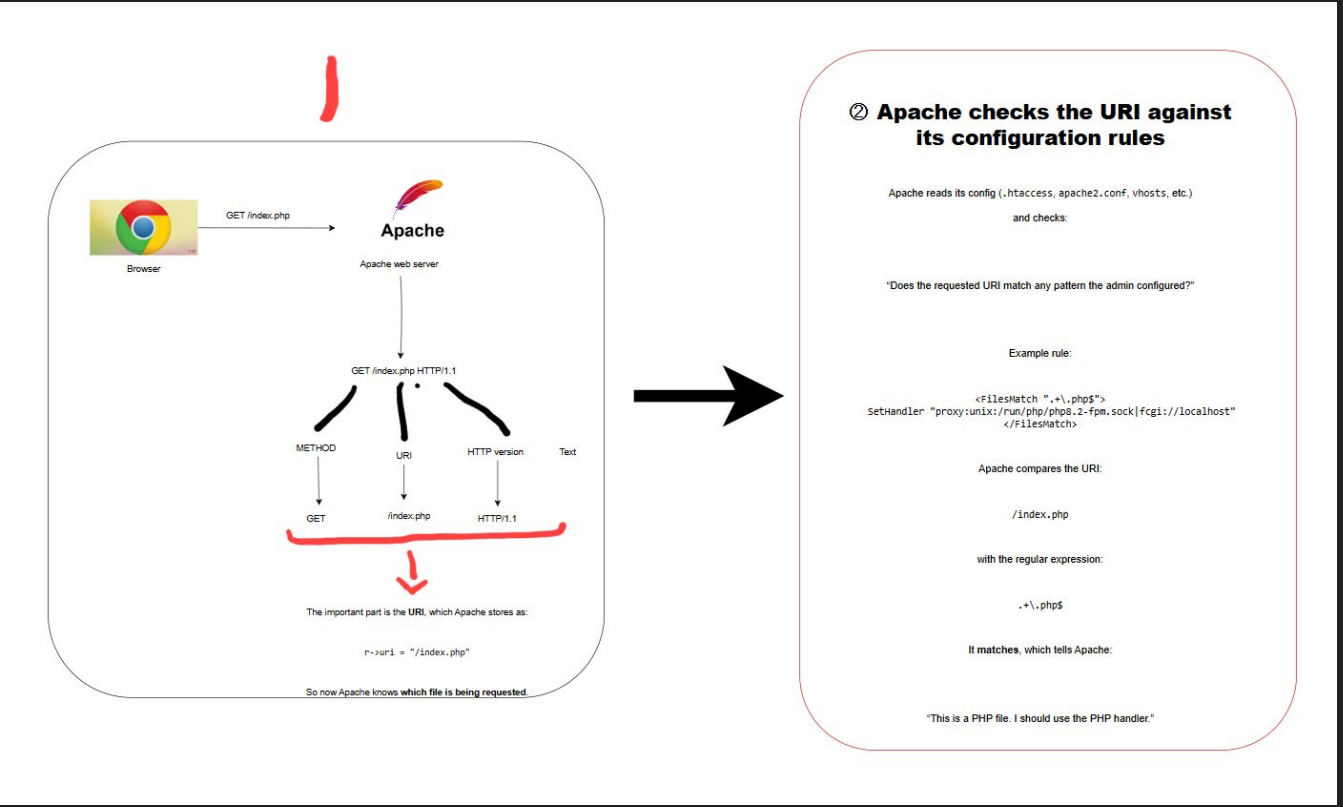
Apache sees .php → uses handler → forwards to PHP-FPM.

PHP-FPM executes PHP code → returns HTML.

Apache sends HTML back to browser.

How do they talk ?

Apache cannot execute PHP directly, so it uses the module **mod_proxy_fcgi** to forward `.php` requests to PHP-FPM. This module acts as a bridge, implementing the FastCGI protocol to send requests to PHP-FPM and return the responses back to Apache. In configuration, a typical setup uses `<FilesMatch ".+\.php$">` with `SetHandler "proxy:unix:/run/php/php8.2-fpm.sock|fcgi://localhost"`, which matches all PHP files and tells Apache to use the handler that communicates with PHP-FPM via a Unix socket.



Where is the error ?

Most Apache modules — including:

- Authentication (`mod_authz_core`)
- Access control (`<Files>`, `<FilesMatch>`)
- Static file serving
- Logging
- Authorization

all assume:

`r->filename` = **a real filesystem path on disk**

Example:

`/var/www/html/admin.php`

These modules use `r->filename` to decide:

- Is this file protected?
- Is the user allowed?
- Should access be denied?

So for them, `r->filename` **MUST be accurate**, or decisions will be wrong.

AUTH MODULE sees → "admin.php?23"
PHP-FPM sees → "/var/www/html/admin.php"

🌐 2. Proxy modules (mod_proxy + mod_proxy_fcgi)

These modules think:

> "`r->filename` is NOT a file — it's a URL describing the backend target."

Example of what THEY put into `r->filename`:

```
proxy:fcgi://localhost/var/www
/html/admin.php?abc
```

This is **not** a real file path — it's a URL used internally to tell PHP-FPM what to run.

EXAMPLE TO GET THE POINT EASILY

Your request:
`/admin.php?test`
What Apache AUTH module expects:
`r->filename = /var/www/html/admin.php`
This would match:
`<Files "admin.php">`
→ auth triggered (good)

What actually happens:
Because you add a tricky ? in the filename (URL encoded),
Apache thinks:
`r->filename = admin.php?test`
Auth module compares:
`admin.php?test ≠ admin.php`
→ auth **NOT** triggered (bad)

Then mod_proxy rewrites it AGAIN to:
`r->filename =`
`proxy:fcgi://localhost/var/www/html/admin.php?test`
This is a **URL**, not a file path.
Auth modules do NOT understand this format at all — but they already ran and it's too late.

Then PHP-FPM normalizes it BACK to the real path:
`/var/www/html/admin.php`

Final exploit diagram !