# Crypto Failures [THM]

First exploit the encryption scheme in the simplest possible way, then find the encryption key.

## Reconnaissance and Enumeration [Phase 1]

### Nmap Scanning

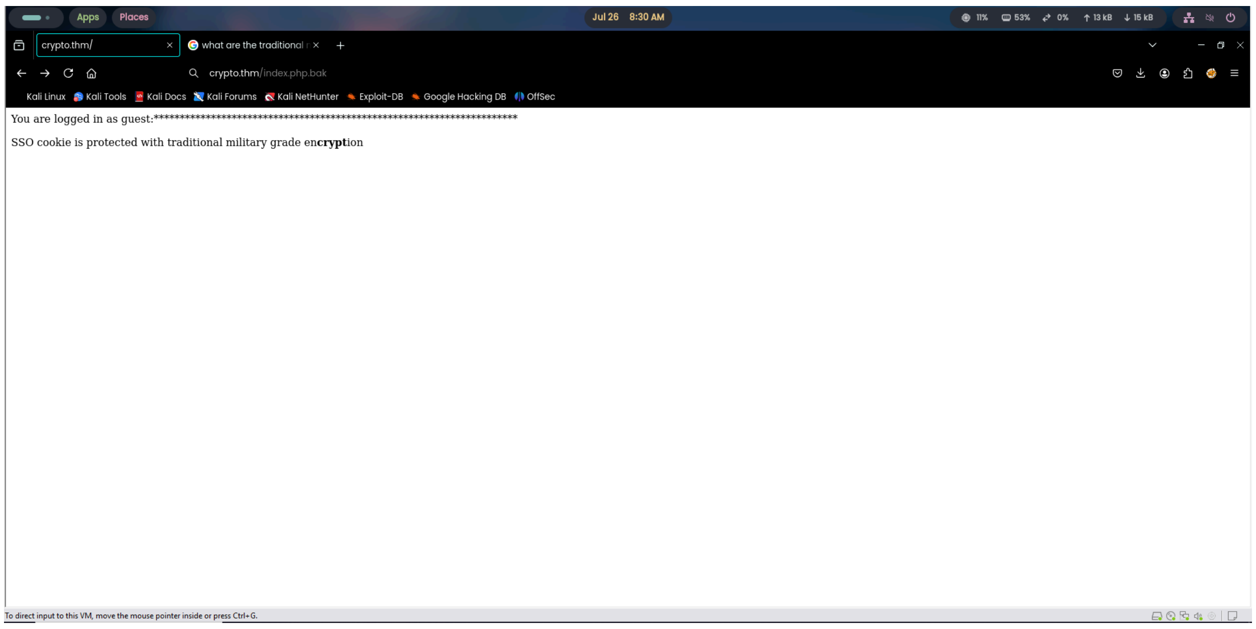
**command** ⇒ nmap -sC -sV -T5 -A -O crypto.thm

```
PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 8.9p1 Ubuntu 3ubuntu0.7 (Ubuntu Linux; protocol 2
| ssh-hostkey:
|   256 57:2c:43:78:0c:d3:13:5b:8d:83:df:63:cf:53:61:91 (ECDSA)
|_  256 45:e1:3c:eb:a6:2d:d7:c6:bb:43:24:7e:02:e9:11:39 (ED25519)
80/tcp open  http    Apache httpd 2.4.59 ((Debian))
|_http-title: Did not follow redirect to /
|_http-server-header: Apache/2.4.59 (Debian)
Aggressive OS guesses: Linux 3.1 (95%), Linux 3.2 (95%), AXIS 210A or 211 Netw
No exact OS matches for host (test conditions non-ideal).
Network Distance: 2 hops
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

TRACEROUTE (using port 993/tcp)
HOP RTT       ADDRESS
1   212.81 ms 10.8.0.1
2   219.85 ms crypto.thm (10.10.158.48)
```

Port 80 and 22 are open basic ports as web port and the ssh port. Let's explore the web app running on the port 80.

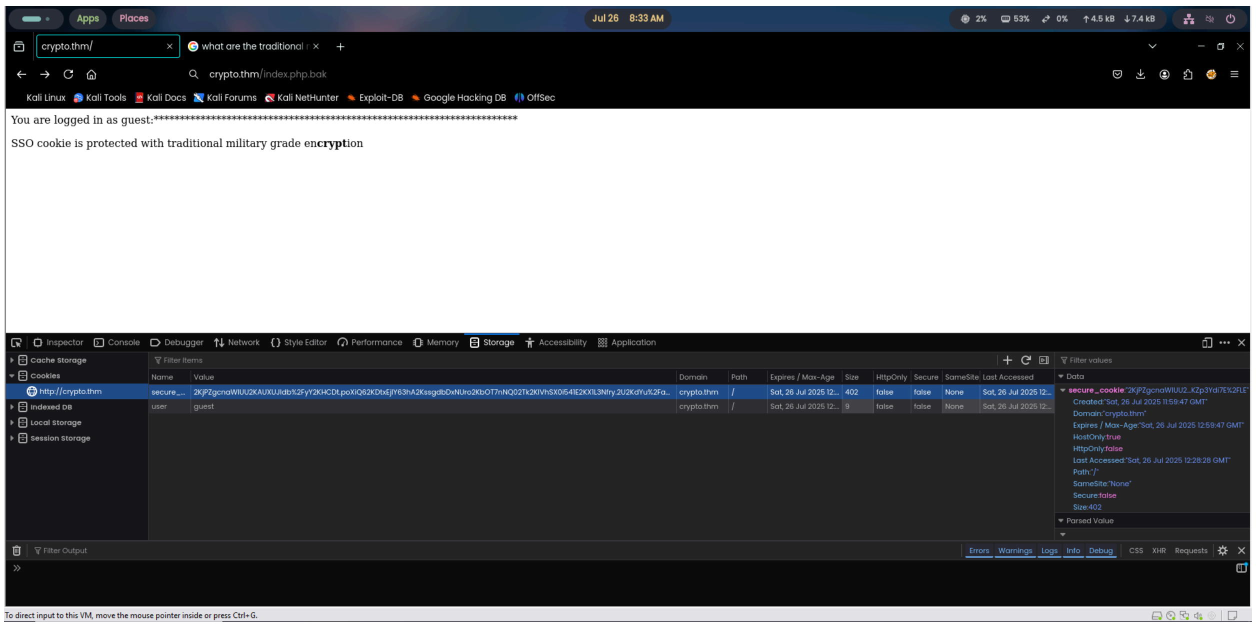
### Visiting the web application

http://crypto.thm



```
You are logged in as guest:*******************************************************************************
SSO cookie is protected with traditional military grade encryption
```

Here we can see that the we are logged in as guest in this and also it says that the sso cookie is protected with traditional military grade encryption. If we are logged in as user we should have a session cookie that we can grab. Let’s grab the session cookie from the dev tool.



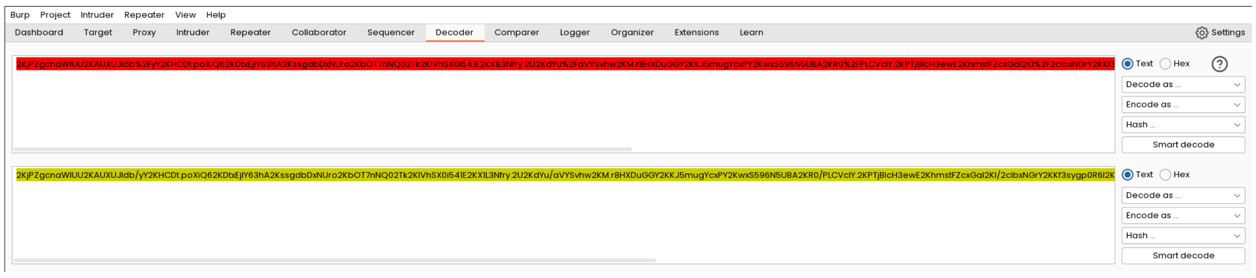
```
You are logged in as guest:*******************************************************************************
SSO cookie is protected with traditional military grade encryption
```

| Name | Value | Domain | Path | Expires / Max-Age | Size | HttpOnly | Secure | SameSite | Last Accessed |
|---|---|---|---|---|---|---|---|---|---|
| secure_... | 2KjPZgcnaWlUU2KAUXUJldb%2FyY2KHCDt.poXiQ62KDtxEjlY63hA2KssgdbDxNUro2KbOT7nNQ02Tk2KIVhSX0i541E2KX1L3Nfry.2U2KdYu%2Fa... | crypto.thm | / | Sat, 26 Jul 2025 12:... | 402 | false | false | None | Sat, 26 Jul 2025 12:... |
| user | guest | crypto.thm | / | Sat, 26 Jul 2025 12:... | 9 | false | false | None | Sat, 26 Jul 2025 12:... |

```
secure_cookie:"2KjPZgcnaWlUU2...KZp3Ydi7E%2FLE"
  Created:"Sat, 26 Jul 2025 11:59:47 GMT"
  Domain:"crypto.thm"
  Expires / Max-Age:"Sat, 26 Jul 2025 12:59:47 GMT"
  HostOnly:true
  HttpOnly:false
  Last Accessed:"Sat, 26 Jul 2025 12:28:28 GMT"
  Path:"/"
  SameSite:"None"
  Secure:false
  Size:402
```

Ok the cookie has url encoding as it has %2f thingy in the cookie lets url decode this and save it to a txt file in our local system.

## Url Decoding the cookie

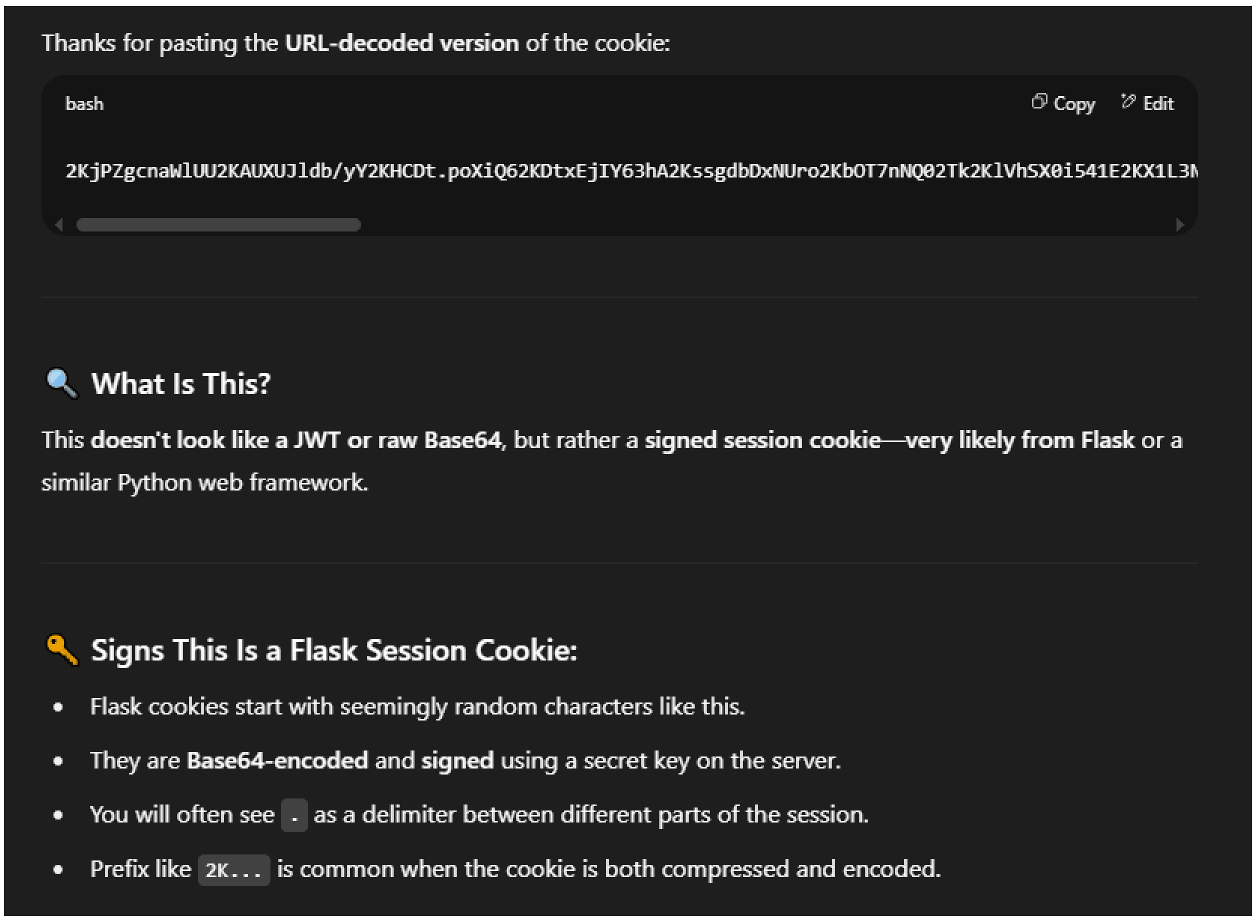


```
2KjPZgcnaWlUU2KAUXUJldb%2FyY2KHCDt.poXiQ62KDtxEjlY63hA2KssgdbDxNUro2KbOT7nNQ02Tk2KIVhSX0i541E2KX1L3Nfry.2U2KdYu%2FaVYSvhw2KM.r8HXDuGGY2KKJ5mugYcxPY2KwxS596N5UBA2KR0%2FPLCVclY.2KPTjBlcH3ewE2KhmstFZcxGal2KI%2F2clbxNGrY2KKf
```

```
2KjPZgcnaWlUU2KAUXUJldb/yY2KHCDt.poXiQ62KDtxEjlY63hA2KssgdbDxNUro2KbOT7nNQ02Tk2KIVhSX0i541E2KX1L3Nfry.2U2KdYu/aVYSvhw2KM.r8HXDuGGY2KKJ5mugYcxPY2KwxS596N5UBA2KR0/PLCVclY.2KPTjBlcH3ewE2KhmstFZcxGal2KI/2clbxNGrY2KKf3sygp0R6I2K
```

We used the burp suite's decoder tab to decode the cookie from url to plain text. Now let’s find out what type of token it is looking at it it does not resemble the

JWT so lets do a quick gpt prompting to find what type of toking it is.



Thanks for pasting the **URL-decoded version** of the cookie:

```bash
2KjPZgcnaWlUU2KAUXUJ1db/yY2KHCDt.poXiQ62KDtxEjIY63hA2KssgdbDxNUro2KbOT7nNQ02Tk2KlVhSX0i541E2KX1L3N
```

### 🔍 What Is This?

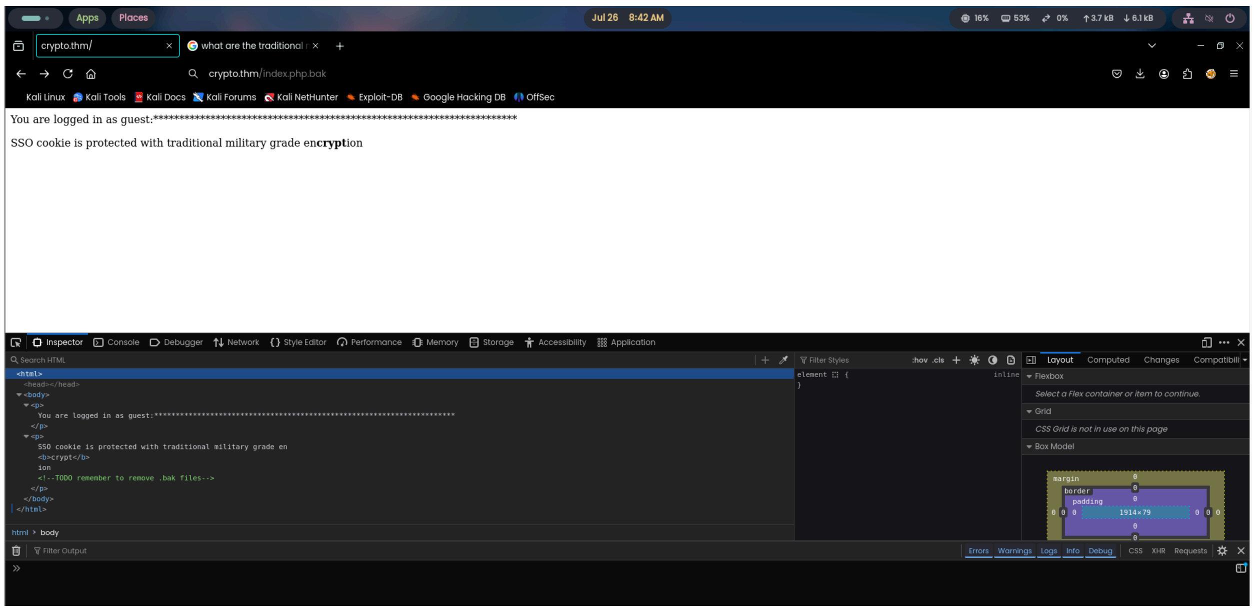
This **doesn't look like a JWT or raw Base64**, but rather a **signed session cookie—very likely from Flask** or a similar Python web framework.

### 🔑 Signs This Is a Flask Session Cookie:

- Flask cookies start with seemingly random characters like this.
- They are **Base64-encoded** and **signed** using a secret key on the server.
- You will often see `.` as a delimiter between different parts of the session.
- Prefix like `2K...` is common when the cookie is both compressed and encoded.

Ok so the token is from the flask session and as per chatgpt these flask keys are signed with HMAC-SHA1 or SHA256 to prevent tampering also we saw on the page that it is encrypted by a military grade encryption keeping this aside for a moments lets inspect the web page if there is something that can help us

## Inspecting the web page



crypto.thm/index.php.bak

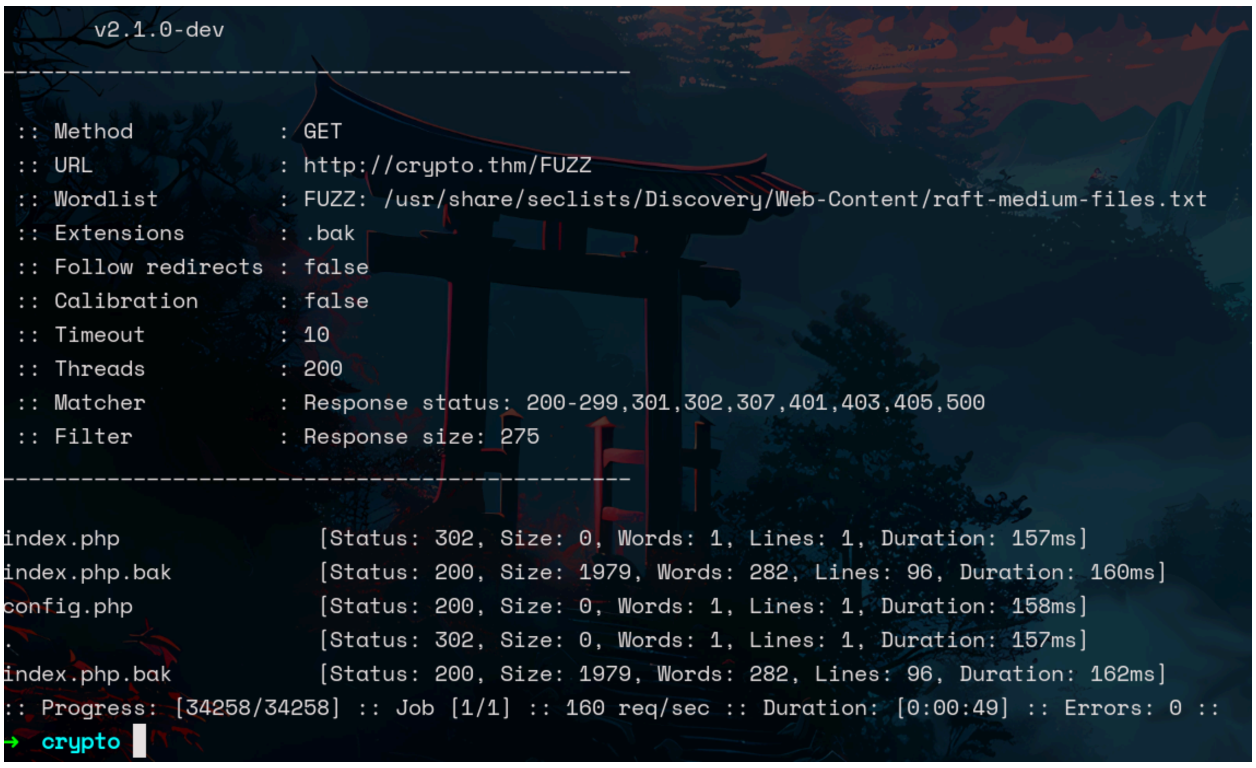
You are logged in as guest:**********************************************************************

SSO cookie is protected with traditional military grade en**crypt**ion

```
<html>
  <head></head>
  <body>
    <p>
      You are logged in as guest:**********************************************************************
    </p>
    <p>
      SSO cookie is protected with traditional military grade en
      <b>crypt</b>
      ion
      <!--TODO remember to remove .bak files-->
    </p>
  </body>
</html>
```

Ok we got a huge clue to look for the .bak (backup files). We can use ffuf to find the files that have the extension .bak

**command** ⇒ ffuf -w /usr/share/seclists/Discovery/Web-Content/raft-medium-files.txt -u http://crypto.thm/FUZZ -e .bak -t 100 -fs 275



```
       v2.1.0-dev
________________________________________________

 :: Method           : GET
 :: URL              : http://crypto.thm/FUZZ
 :: Wordlist         : FUZZ: /usr/share/seclists/Discovery/Web-Content/raft-medium-files.txt
 :: Extensions       : .bak
 :: Follow redirects : false
 :: Calibration      : false
 :: Timeout          : 10
 :: Threads          : 200
 :: Matcher          : Response status: 200-299,301,302,307,401,403,405,500
 :: Filter           : Response size: 275
________________________________________________

index.php               [Status: 302, Size: 0, Words: 1, Lines: 1, Duration: 157ms]
index.php.bak           [Status: 200, Size: 1979, Words: 282, Lines: 96, Duration: 160ms]
config.php              [Status: 200, Size: 0, Words: 1, Lines: 1, Duration: 158ms]
.                       [Status: 302, Size: 0, Words: 1, Lines: 1, Duration: 157ms]
index.php.bak           [Status: 200, Size: 1979, Words: 282, Lines: 96, Duration: 162ms]
:: Progress: [34258/34258] :: Job [1/1] :: 160 req/sec :: Duration: [0:00:49] :: Errors: 0 ::
➜  crypto
```

Here we got a few hits and one of them is the now let’s take a look at the backup file that we got the index.php.bak

## index.php.bak

**command** ⇒ cat index.php.bak

```
<?php
include('config.php');

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

    $secure_cookie_string = $user.":".$_SERVER['HTTP_USER_AGENT'].":".$ENC_S

    $secure_cookie = make_secure_cookie($secure_cookie_string,$SALT);

    setcookie("secure_cookie",$secure_cookie,time()+3600,'/','',false);
    setcookie("user","$user",time()+3600,'/','',false);
}

function cryptstring($what,$SALT){

return crypt($what,$SALT);

}
```

```
function make_secure_cookie($text,$SALT) {

$secure_cookie='';

foreach ( str_split($text,8) as $el ) {
    $secure_cookie .= cryptstring($el,$SALT);
}

return($secure_cookie);
}


function generatesalt($n) {
$randomString='';
$characters = '0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQ
for ($i = 0; $i < $n; $i++) {
    $index = rand(0, strlen($characters) - 1);
    $randomString .= $characters[$index];
}
return $randomString;
}



function verify_cookie($ENC_SECRET_KEY){


    $crypted_cookie=$_COOKIE['secure_cookie'];
    $user=$_COOKIE['user'];
    $string=$user.":".$_SERVER['HTTP_USER_AGENT'].":".$ENC_SECRET_KEY;

    $salt=substr($_COOKIE['secure_cookie'],0,2);

    if(make_secure_cookie($string,$salt)===$crypted_cookie) {
        return true;
    } else {
        return false;
    }
}


if ( isset($_COOKIE['secure_cookie']) && isset($_COOKIE['user']))  {

    $user=$_COOKIE['user'];

    if (verify_cookie($ENC_SECRET_KEY)) {
```

```
    if ($user === "admin") {

        echo 'congrats: ******flag here******. Now I want the key.';

            } else {

        $length=strlen($_SERVER['HTTP_USER_AGENT']);
        print "<p>You are logged in as " . $user . ":" . str_repeat("*", $length) . "\n";
        print "<p>SSO cookie is protected with traditional military grade en<b>cryp
    }

} else {

    print "<p>You are not logged in\n";


}

}
  else {

    generate_cookie('guest',$ENC_SECRET_KEY);

    header('Location: /');


}
?>
```

The PHP application generates a session cookie using the crypt() function with a 2-character salt, which invokes the legacy DES-based encryption. While historically considered military-grade, DES is now deprecated due to security weaknesses. The cookie is formed by applying DES to chunks of the string username:User-Agent:ENC_SECRET_KEY, making it possible to tamper or forge if the secret key or hash format is guessed.

## How the Protected Session Key was Formed

### Step 1)

```
username : user_agent : ENC_SECRET_KEY
```

```
function generate_cookie($user,$ENC_SECRET_KEY) {
    $SALT=generatesalt(2);
    $secure_cookie_string = $user.":".$_SERVER['HTTP_USER_AGENT'].":".$ENC_S

    $secure_cookie = make_secure_cookie($secure_cookie_string,$SALT);

    setcookie("secure_cookie",$secure_cookie,time()+3600,'/','',false);
    setcookie("user","$user",time()+3600,'/','',false);
}
```

The php code first creates a string with all three of the above the username , user-agent and the secret key.

### Step 2)

```
function make_secure_cookie($text,$SALT) {

$secure_cookie='';

foreach ( str_split($text,8) as $el ) {
    $secure_cookie .= cryptstring($el,$SALT);
}

return($secure_cookie);
}
```

Once the string is created with all three it is then split into 8 character chunks and applies the crypt() to the 2 character salt on each chunk. Now since the 2 character salt is used in php it defaults to the des (56-bit encryption).

### Step 3)

All the des hashed chunks are concatenated together and we get the protected session cookie for the guest or the admin

### Bonus

```
function generatesalt($n) {
$randomString='';
$characters = '0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQ
for ($i = 0; $i < $n; $i++) {
    $index = rand(0, strlen($characters) - 1);
    $randomString .= $characters[$index];
}
return $randomString;
}
```

The salt that the generatesalt() function is creating is used in hashing the first two character of the chunks that were created.

# Exploitation [Phase 2]

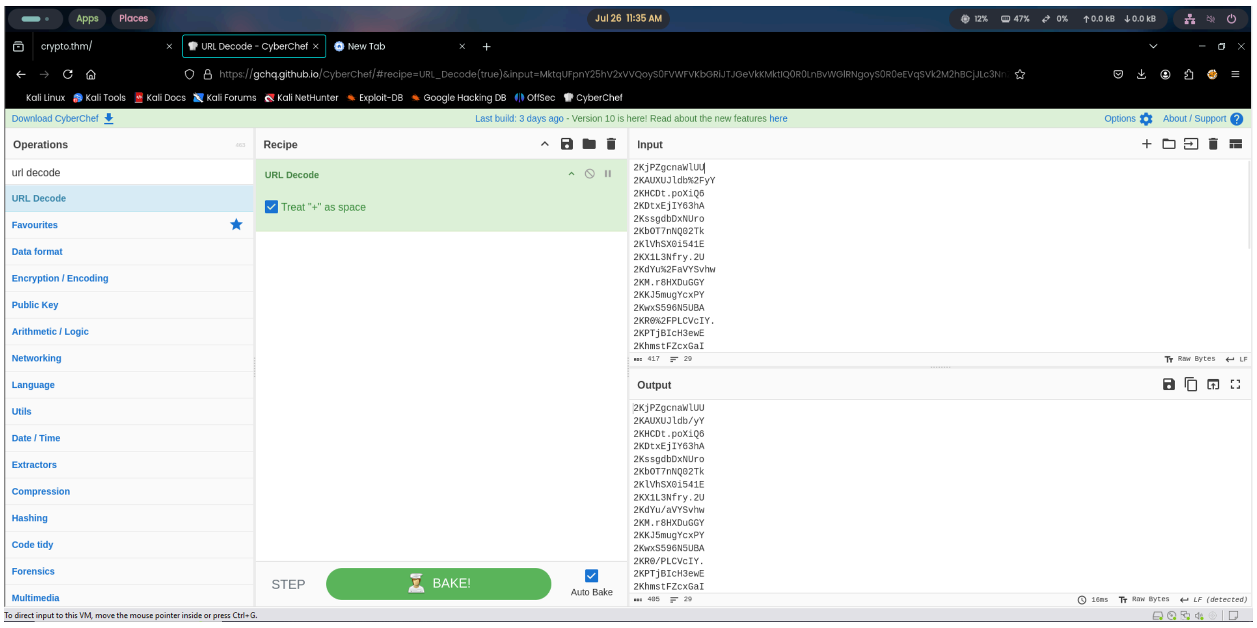
## How does crypt() works in php

```
<?php

$x = "admin:Mo";      //First 8 character of the step 1 ( 1st chunk )
$y = "2K";            // Found from the start of every chunk in guest cookie
$z = crypt($x, $y);   // using the crypt() with both the param the chunk and the sa
echo $z;              // print out the hashed chunk

?>
```

So here is the basic script to get the admin access which i will explain later but how does crypt() work is it first takes the string hashes it with the salt provided and then concats itself with the hashed string like this “salt + hashed_string” .

```
2KRKYFc/NU2xc      // output
2K                 // salt used to hash the string
RKYFc/NU2x         // string that is hashed
```

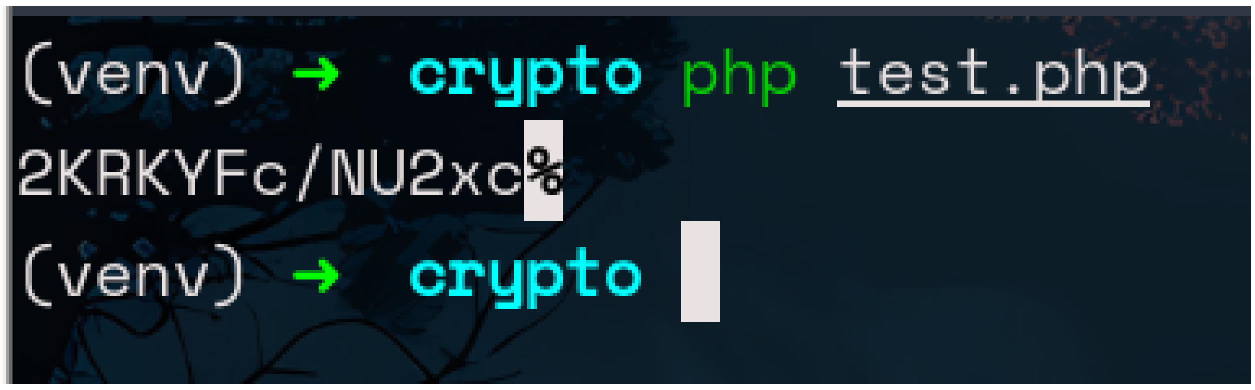
Now that you understood how does the crypt() work in php and how are we creating the protected hash of the cookie. Now lets understand the script that we wrote. in that script we are assigning two variables that hold the 1st chunk of 8 characters and the salt that we got from the guests cookie then we use crypt to create a hash of this and save it variable z and print it out using echo here is how it looks



Recipe: URL Decode — Treat "+" as space ✓

Input:
```
2KjPZgcnaWlUU
2KAUXUJldb%2FyY
2KHCDt.poXiQ6
2KDtxEjIY63hA
2KssgdbDxNUro
2KbOT7nNQ02Tk
2KlVhSX0i541E
2KX1L3Nfry.2U
2KdYu%2FaVYSvhw
2KM.r8HXDuGGY
2KKJ5mugYcxPY
2KwxS596N5UBA
2KR0%2FPLCVcIY.
2KPTjBIcH3ewE
2KhmstFZcxGaI
```

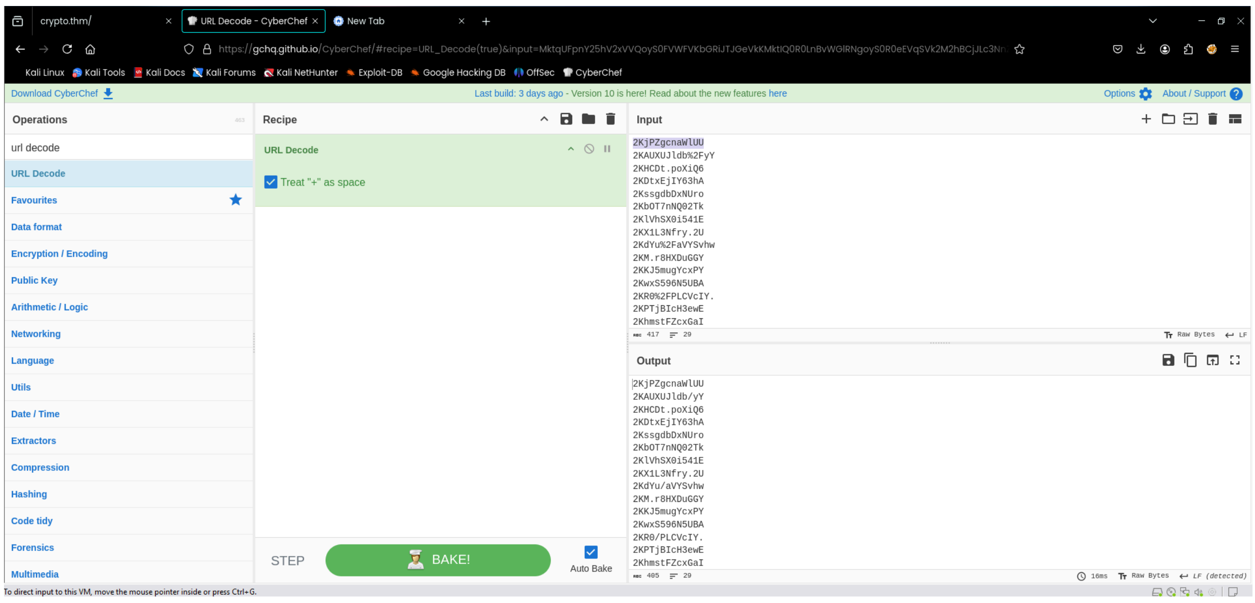
Output:
```
2KjPZgcnaWlUU
2KAUXUJldb/yY
2KHCDt.poXiQ6
2KDtxEjIY63hA
2KssgdbDxNUro
2KbOT7nNQ02Tk
2KlVhSX0i541E
2KX1L3Nfry.2U
2KdYu/aVYSvhw
2KM.r8HXDuGGY
2KKJ5mugYcxPY
2KwxS596N5UBA
2KR0/PLCVcIY.
2KPTjBIcH3ewE
2KhmstFZcxGaI
```



```
(venv) ➜  crypto php test.php
2KRKYFc/NU2xc%
(venv) ➜  crypto
```

Now we can just change the first chunk of the cookie and we should get the admin access as secure was false when i saw it in dev tools let’s change the 1st chunk and get the flag.

## Web Flag

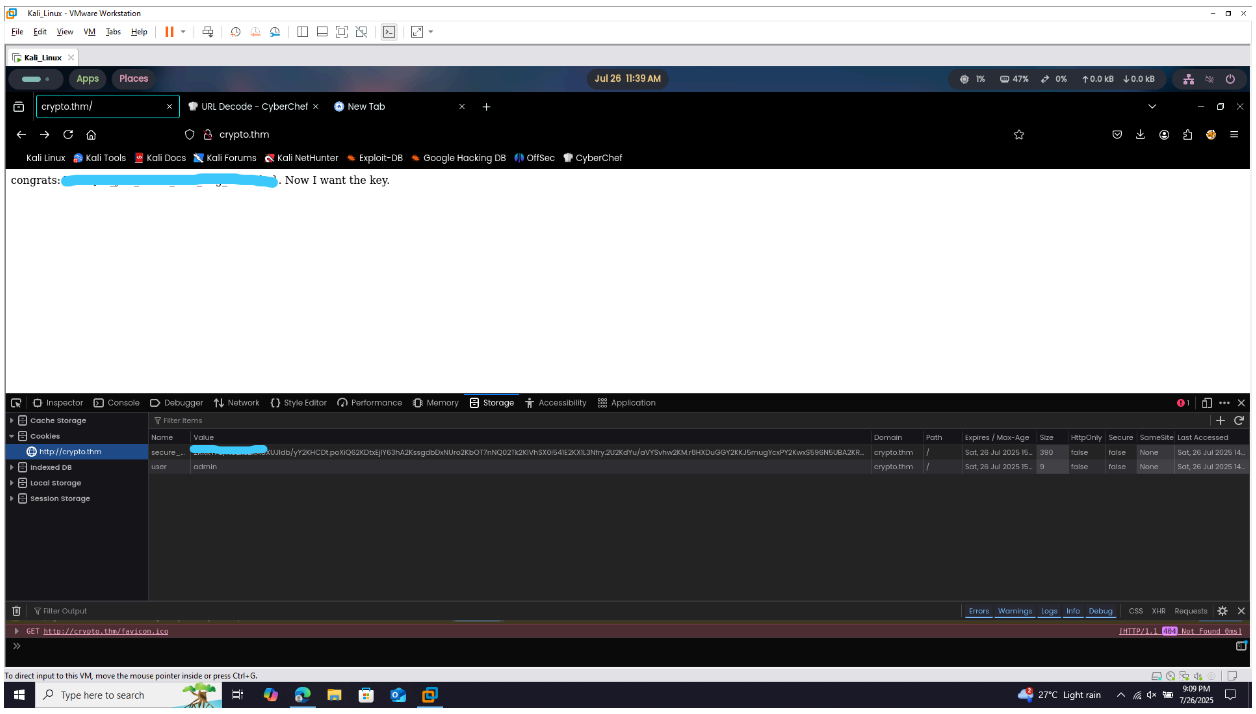


```
2KjPZgcnaWlUU
2KAUXUJldb%2FyY
2KHCDt.poXiQ6
2KDtxEjIY63hA
2KssgdbDxNUro
2KbOT7nNQ02Tk
2KlVhSX0i541E
2KX1L3Nfry.2U
2KdYu%2FaVYSvhw
2KM.r8HXDuGGY
2KKJ5mugYcxPY
2KwxS596N5UBA
2KR0%2FPLCVcIY.
2KPTjBIcH3ewE
2KhmstFZcxGaI
```

```
2KjPZgcnaWlUU
2KAUXUJldb/yY
2KHCDt.poXiQ6
2KDtxEjIY63hA
2KssgdbDxNUro
2KbOT7nNQ02Tk
2KlVhSX0i541E
2KX1L3Nfry.2U
2KdYu/aVYSvhw
2KM.r8HXDuGGY
2KKJ5mugYcxPY
2KwxS596N5UBA
2KR0/PLCVcIY.
2KPTjBIcH3ewE
2KhmstFZcxGaI
```

```
2KRKYFc/NU2xc
```

We need to replace the highlighted part with the hash we just created in the console and we will get the admin access.


congrats: [redacted]. Now I want the key.

| Name | Value | Domain | Path | Expires / Max-Age | Size | HttpOnly | Secure | SameSite | Last Accessed |
|---|---|---|---|---|---|---|---|---|---|
| secure_... | [redacted]XUJldb/yY2KHCDt.poXiQ62KDtxEjlY63hA2KssgdbDxNUro2KbOT7nNQ02Tk2KlVhSX0i541E2KXlL3Nfry.2U2KdYu/aVYSvhw2KM.r8HXDuGGY2KKJ5mugYcxPY2KwxS596N5UBA2KR... | crypto.thm | / | Sat, 26 Jul 2025 15... | 390 | false | false | None | Sat, 26 Jul 2025 14... |
| user | admin | crypto.thm | / | Sat, 26 Jul 2025 15... | 9 | false | false | None | Sat, 26 Jul 2025 14... |

```
GET http://crypto.thm/favicon.ico    [HTTP/1.1 404 Not Found 0ms]
```

And Boom we got our first flag the web flag lets submit it and continue on finding the encryption key flag.

## Padding method

```
guest::1
23456789

guest:AA
AAAAA:12
3456789

and so on...
```

This is called the padding method at least i call it the padding method. Basically after hours of staring at the screen and wasted enumeration i realised that we can manipulate the user-agent section of the website to reveal the encryption key letter by letter or character by character so here is how it work.

As you can see above that is how our cookie looks like but the user-agent section is empty and that is for a reason

```
userame : user-agent : encryption_key
```

This is how our payload should be but what if we keep the user-agent empty or we send a request where our user-agent is null

```
username :: encryption_key
```

Now that we have removed the user-agent lets divide it into 8 byte chunks like the php code we found. For understanding lets keep the encryption key as 123456789

```
guest::1    // chunk 1
23456789    // chunk 2
```

if you see carefully we already know what 'guest::' is so we just brute force character by character and crypt it character by character and then compare it to the chunk that we got from the server

```
// Server Side

guest::hidden_server_encrpytion_key // request sent to the server with empty us


guest::h   // chunk 1
idden_se   // chunk 2
rver_enc   // chunk 3
rpytion_   // chunk 4
key        // chunk 5

$salt = "6u"

crypt($chunk1, $salt)

guest::h

6uO9TwqCmDBj. // for example this is the chunk generated by the server
```

As we can see this is basically how the protected cookie is generated but that is the concern we were able to manipulate the server into making this chunk which is a huge benefit for us now lets see how on the client side.

```
// Client Side

guest::+<Loop_added_to_run_chr_by_chr>  //this is how we will loop till we find tl
guest::0   // failed match 6uTLlFZ.N/JBY != 6uO9TwqCmDBj.
guest::1   // failed match 6uGnelrUCFZZg != 6uO9TwqCmDBj.
guest::h   // success match 6uO9TwqCmDBj. == 6uO9TwqCmDBj.

Match Found → h
```

Look what happened by manipulating the server side we were able to find the first character of our encryption key. what happened above is we started a loop and after every loop we user crypt to hash it with the salt we get from the cookie the first two character and then try to match it with the chunk we found on the server if the client chunk and the server chunk are a match we get the character by doing this over a loop via a script we can get the entire encryption key but now u may ask how do we get the next bits and pieces of the encryption key. Lets see that below

.

```
guest:AA
AAAAA:12
3456789

guest:AA
AAAA:123
456789

guest:AA
AAA:1234
56789

guest:AA
AA:12345
6789

guest:AA
A:123456
789

guest:AA
:1234567
89

guest:A:
12345678
9

guest::123456789
```

Now that A's you see are the padding we are sending the 7A's to the server via user-agent to now we know the first letter of the encryption key is 1 here and we also know the amount of A's we entered we just don't know the end character so once again we brute force it

```
AAAAA:character_found + <char_loop_like_the_first_one>

AAAAA:10
AAAAA:11
```

```
AAAAA:12
AAAAA:13

and so on...
```

Loop → hash → compare

if match print letter

else move to the next letter and repeat

And remember as you progress the now of A’s will decrease and there will be a time when it all starts again and it goes on till the end of the encryption key.

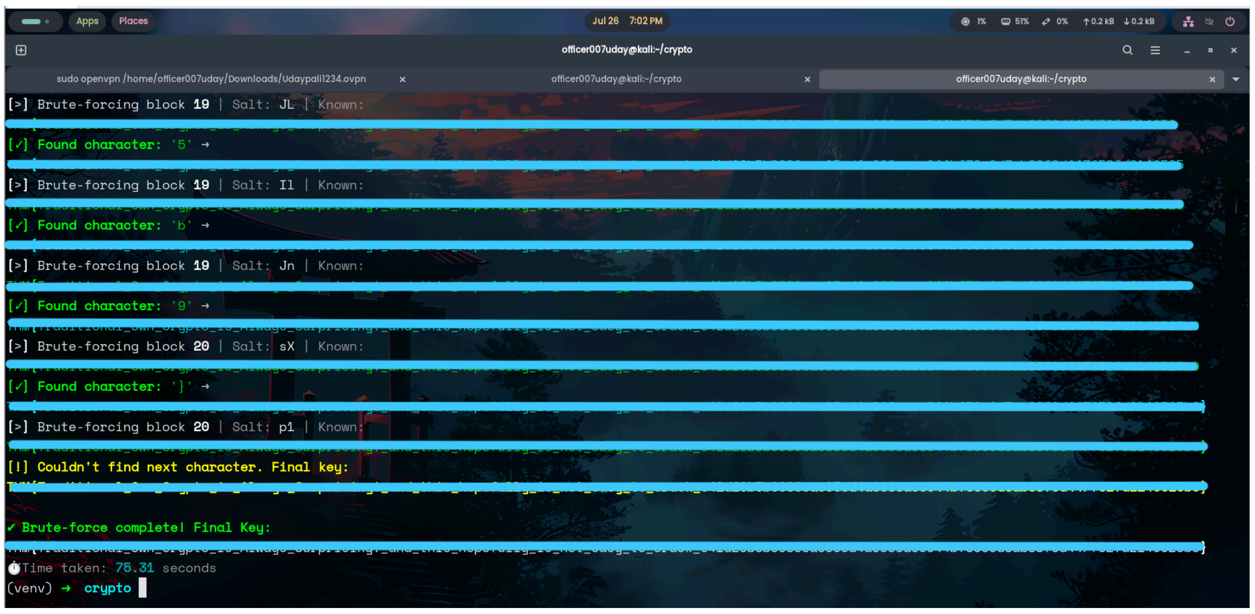
## Encryption_key_Finder.py

```
import socket
import re
import urllib.parse
import crypt
from math import floor
from time import time, sleep
from concurrent.futures import ThreadPoolExecutor, as_completed
from rich.console import Console

# ========== CONFIG ==========
HOST = "crypto.thm"
PORT = 80
USERNAME = "guest:"
SEPARATOR = ":"
CHARSET = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQR
TIMEOUT = 5
THREADS = 20
# ============================

console = Console()


def send_request(user_agent: str) → str | None:
    """Send HTTP request and return decoded secure_cookie."""
    req = (
        f"GET / HTTP/1.1\r\n"
        f"Host: {HOST}\r\n"
        f"User-Agent: {user_agent}\r\n"
        f"Accept: */*\r\n"
        f"Connection: close\r\n\r\n"
    )
    try:
        with socket.create_connection((HOST, PORT), timeout=TIMEOUT) as sock:
            sock.sendall(req.encode())
            response = b""
            while True:
                chunk = sock.recv(1024)
                if not chunk:
                    break
                response += chunk
    except Exception:
        return None

    match = re.search(rb"Set-Cookie: secure_cookie=([^;]+)", response)
    if match:
        cookie = match.group(1).decode()
        return urllib.parse.unquote(cookie)
    return None


def split_blocks(cookie: str) → list[str]:
    return [cookie[i:i + 13] for i in range(0, len(cookie), 13)]


def try_char(candidate_char, prefix, salt, target_block):
    candidate = prefix + candidate_char
    block = candidate[-8:]
    hashed = crypt.crypt(block, salt)
    if hashed == target_block:
        return candidate_char
    return None


def brute_force_key():
    known_key = ""
    position = 0
    start_time = time()

    console.print("[bold cyan]🔓 Starting secure_cookie brute-force...[/]\n")

    while True:
        prefix_len = len(USERNAME + SEPARATOR + known_key)
        pad_len = ((7 - prefix_len) % 8 + 8) % 8
        user_agent = "A" * pad_len
        prefix = USERNAME + user_agent + SEPARATOR + known_key

        cookie = send_request(user_agent)
        if not cookie:
            console.print("[bold red][!] Failed to retrieve cookie. Exiting.[/]")
            break

        blocks = split_blocks(cookie)
        block_idx = floor(len(prefix) / 8)

        if block_idx >= len(blocks):
            console.print(f"\n[bold yellow][✓] Key might be complete: [white]{known_
            break

        target_block = blocks[block_idx]
        salt = target_block[:2]

        console.print(
            f"[white][>] Brute-forcing block {block_idx}[/] "
            f"[dim]| Salt:[/] {salt} [dim]| Known:[/] [green]{known_key}[/]"
        )

        found_char = None
        with ThreadPoolExecutor(max_workers=THREADS) as executor:
            futures = {
                executor.submit(try_char, ch, prefix, salt, target_block): ch
                for ch in CHARSET
            }

            for future in as_completed(futures):
                result = future.result()
                if result:
                    found_char = result
                    break

        if found_char:
            known_key += found_char
            console.print(f"[bold green][✓] Found character:[/] '{found_char}' → [cya
        else:
            console.print(f"[bold yellow][!] Couldn't find next character. Final key: {kn
            break

        position += 1
        sleep(0.05)  # just enough to avoid socket issues on some servers

    console.print(
        f"\n[bold green]✔ Brute-force complete! Final Key:[/] [white]{known_key}[/]
        f"[dim]⏱ Time taken: {time() - start_time:.2f} seconds[/]"
    )


if __name__ == "__main__":
    try:
        brute_force_key()
    except KeyboardInterrupt:
        console.print("\n[bold red][!] Interrupted by user. Exiting...[/]")
```

Here is my python code that automates the stuff i told above and gives your that key lets see it in practice and find the key to end this challenge.

## Encryption Key

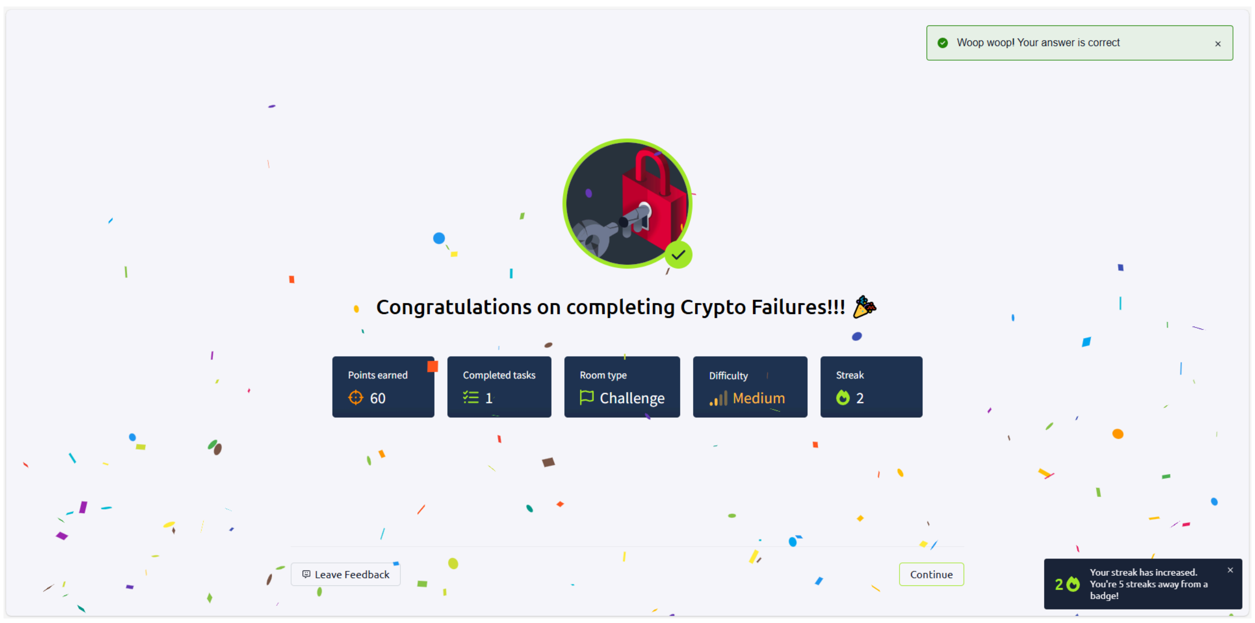
**command** ⇒ python3 Encryption_key_Finder.py



```
[>] Brute-forcing block 19 | Salt: JL | Known:
[redacted]
[✓] Found character: '5' →
[redacted]
[>] Brute-forcing block 19 | Salt: Il | Known:
[redacted]
[✓] Found character: 'b' →
[redacted]
[>] Brute-forcing block 19 | Salt: Jn | Known:
[redacted]
[✓] Found character: '9' →
[redacted]
[>] Brute-forcing block 20 | Salt: sX | Known:
[redacted]
[✓] Found character: '}' →
[redacted]
[>] Brute-forcing block 20 | Salt: p1 | Known:
[redacted]
[!] Couldn't find next character. Final key:
[redacted]

✓ Brute-force complete! Final Key:
[redacted]
⏱Time taken: 75.31 seconds
(venv) → crypto
```

And there we go we got the final flag let's submit it and end this challenge

# TryHackMe



Woop woop! Your answer is correct

Congratulations on completing Crypto Failures!!! 🎉

Points earned 60
Completed tasks 1
Room type Challenge
Difficulty Medium
Streak 2

Leave Feedback

Continue

Your streak has increased. You're 5 streaks away from a badge!

**Note**: This challenge took me way longer than it should have sure it made me very confused for hours i look for something to find the encryption key but at last i got the answer.