

Spring 2025

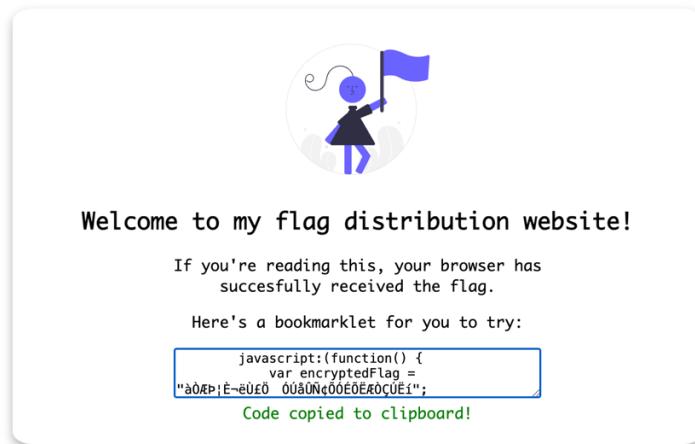
Cyber Security Fundamentals (INFT-3508 - 20332)

Aysel Panahova

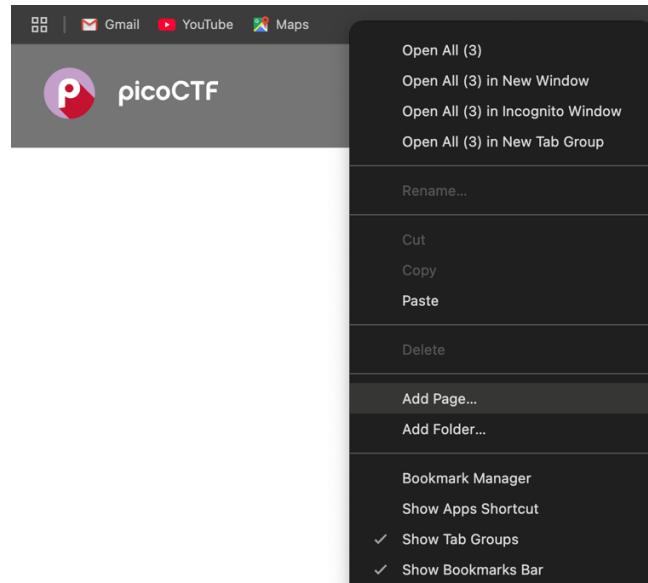
Homework 2

1. Bookmarklet

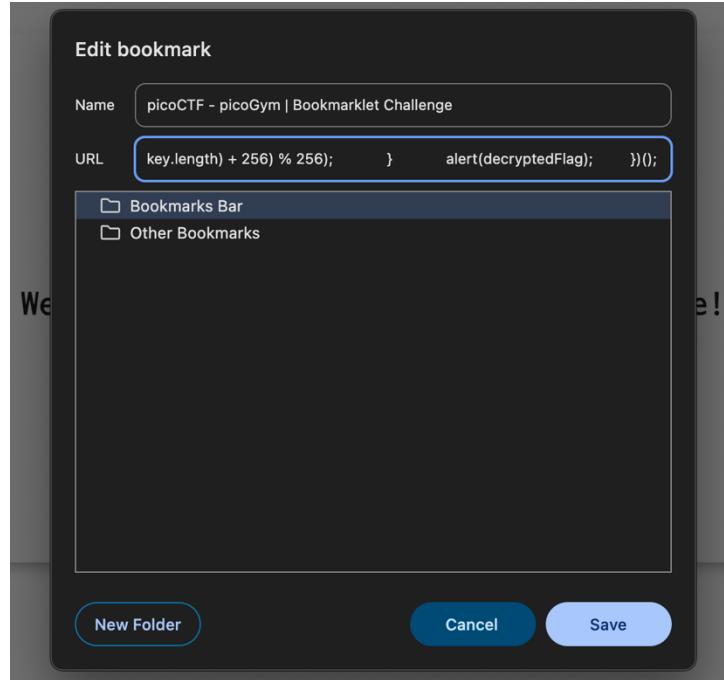
In this task, we need to click “Launch instance” to see the instructions. After clicking we can see the javascript code. Here, we need to copy the code first.



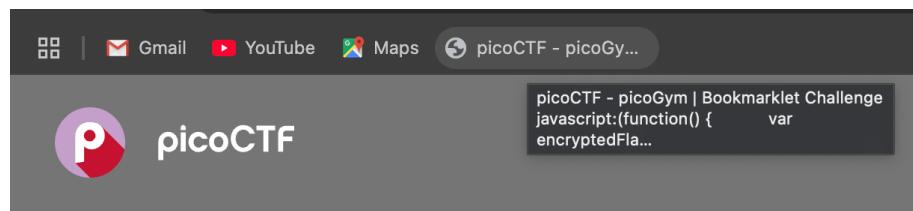
Then, we go to the “Bookmarklet” part of our page and add a page.



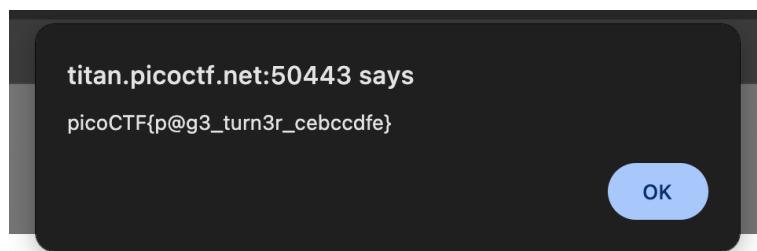
After this step, we paste the link we copied to the “URL” section.



It will be visible like this in the “Bookmarklet” section.

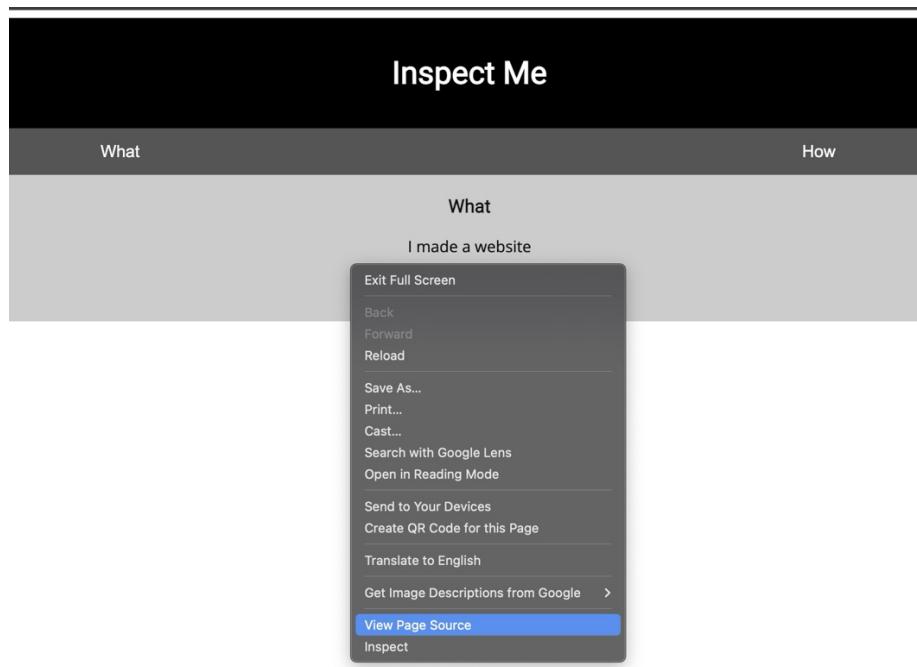


When we click on this it will display the flag itself.



1. Insp3ct0r

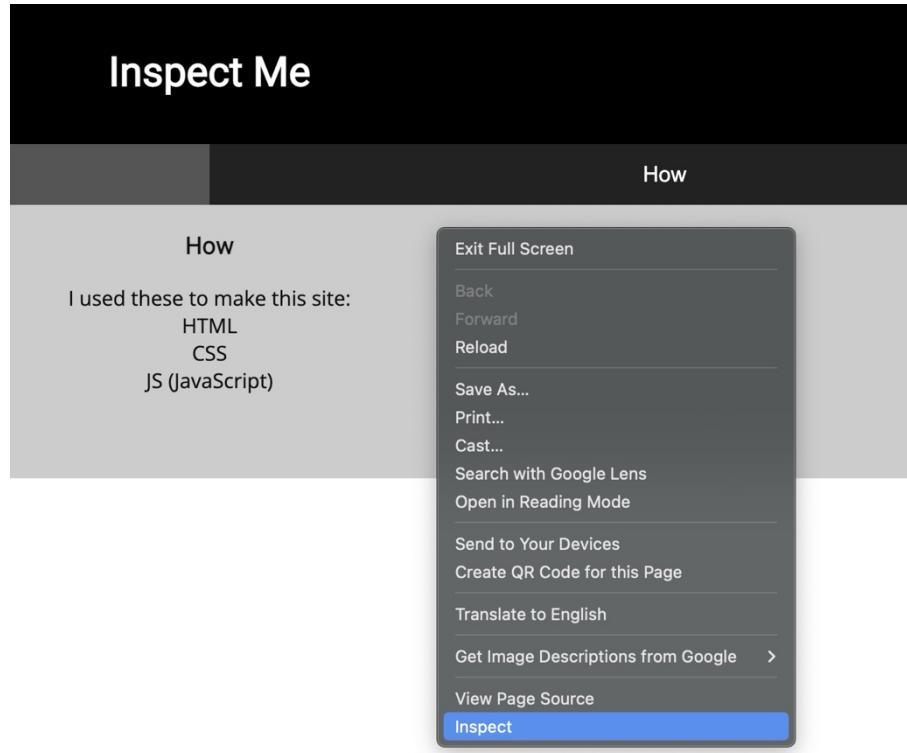
The first step is to click on the link provided in the task description. Once we click the link, a webpage will open displaying the text “Inspect Me”. It has two parts – What and How. First, we right-click anywhere on the “What” page, and from the context menu that appears, select “View Page Source.” This option allows us to view the HTML code behind the webpage, which helps in analyzing or understanding how the content is structured and possibly where hidden information is located.



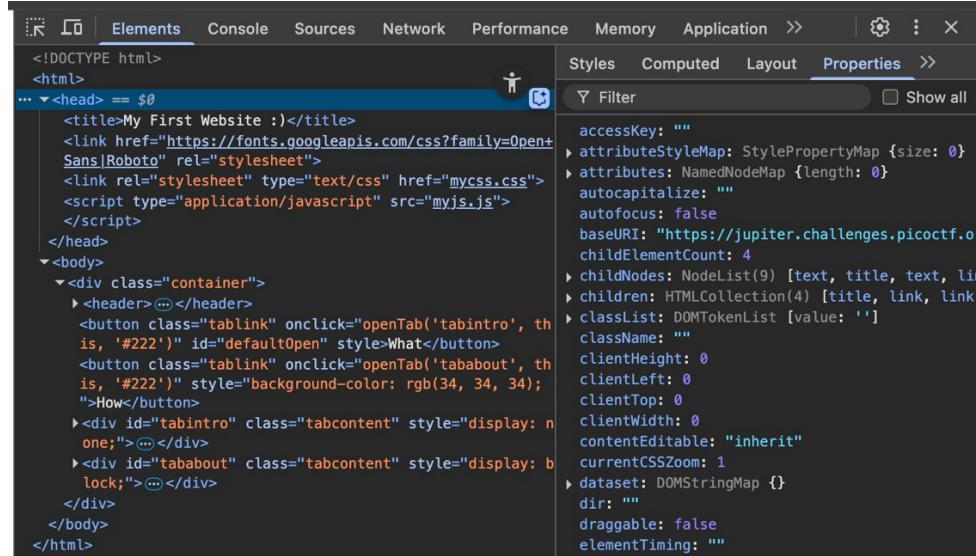
Here we will see the first part of the flag (written in green color).

```
15      <button class="tablink" onclick="openTab('tabintro', this, '#222')" id="defaultOpen">What</button>
16      <button class="tablink" onclick="openTab('tababout', this, '#222')>How</button>
17
18      <div id="tabintro" class="tabcontent">
19          <h3>What</h3>
20          <p>I made a website</p>
21      </div>
22
23      <div id="tababout" class="tabcontent">
24          <h3>How</h3>
25          <p>I used these to make this site: <br/>
26              HTML <br/>
27              CSS <br/>
28              JS (JavaScript)
29          </p>
30          <!-- Html is neat. Anyways have 1/3 of the flag: picoCTF{tru3_d3 -->
31          </div>
32
33      </div>
34
35  </body>
36
37  </html>
```

To find the next parts of the flags we need to inspect the other part of the page which is “How”.



After clicking the “Inspect”, we will be directed to a page that looks like this:



We need to examine the <head> section of the HTML code.

Why? Because the <head> section contains important resources such as CSS and JavaScript files. These external files can include hidden parts of the flag, which are essential for solving this challenge.

In this case, we notice a reference to a CSS file name “mycss.css” in the <head section>. To view the content of this file, we can manually open it by pasting its full path in the browser’s address bar. To do that, take the original URL of the challenge and append “mycss.css” to the end of it. The final URL should look like this:

<https://jupiter.challenges.picoctf.org/problem/44924/mycss.css>

When pressing enter to load the file and examine its contents, we will see the second part of the flag (2/3) at the end of the code.

```
#tabintro { background-color: #ccc; }
#tababout { background-color: #ccc; }

/* You need CSS to make pretty pages. Here's part 2/3 of the flag: t3ct1ve_0r_ju5t */
```

Now only the third part of the flag remains to be found. Since we have already found the second part of the flag in the CSS file, it is logical to check the JavaScript file, next to find the last part. To proceed, we follow the same steps we used for the CSS file. The full URL should look like this: <https://jupiter.challenges.picoctf.org/problem/44924/myjs.js>

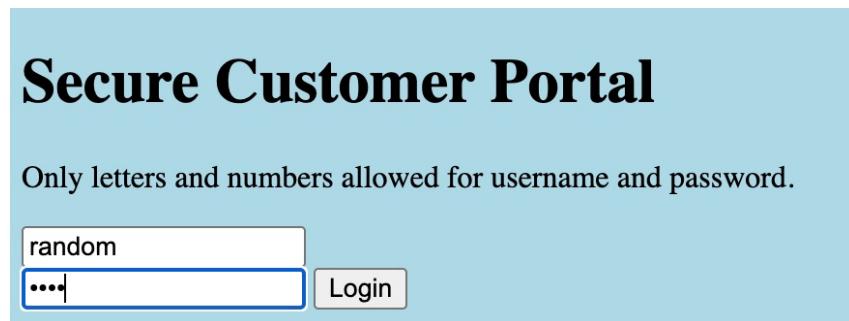
```
document.getElementById(tabName).style.display = "block";
if(elmnt.style != null) {
    elmnt.style.backgroundColor = color;
}
}

window.onload = function() {
    openTab('tabintro', this, '#222');
}

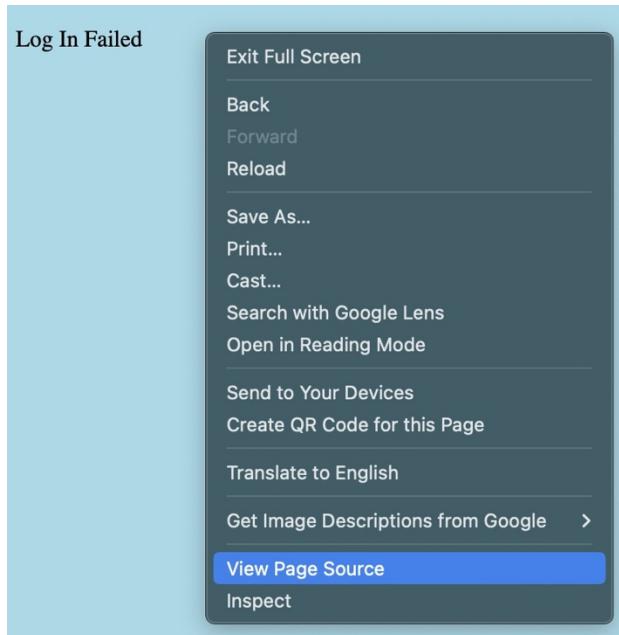
/* Javascript sure is neat. Anyways part 3/3 of the flag: _lucky?f10be399 */
```

2. Local Authority

The first step of the assignment is to click on the “Launch Instance” button. Once it launches, we will see a brief problem statement which will be a link to a login page of a website that we need to access. On this age, we are asked to enter a username and password, but these credentials are not provided in the instructions.



To explore further, we can try entering a random username and password, which will result in an error message such as “Login failed.” However, this does not mean we have reached the end. In fact, it is a hint that we should investigate how the website is functioning behind the scenes. To do this, right-click anywhere on the page and select “View Page Source.” This action allows us to see the HTML code that was used to create the web page.



While examining the source code, we notice a line like this: `<script src="secure.js"></script>`.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <link rel="stylesheet" href="style.css">
    <title>Login Page</title>
  </head>
  <body>
    <script src="secure.js"></script>
```

This line tells the browser to load an external JavaScript file called “secure.js.” The `src="secure.js"` means that the web page is importing a script from that file to perform a specific logic, which can possibly include the login information.

We can click on the link “secure.js” to view its contents. Inside this JavaScript file, we find the actual username and password in the code. The Username is “admin”, and the Password is “strongPassword098765.”

```
function checkPassword(username, password)
{
  if( username === 'admin' && password === 'strongPassword098765' )
  {
    return true;
  }
  else
  {
    return false;
  }
}
```

After successfully logging in with these values, the system will display the flag.

3. Logon

The first step of the task is to click on the “Launch Instance” button again. Once the instance loads, a brief problem statement appears, which includes a link to a webpage. This page turns out to be a login page. Since we do not know the username or password, we just enter something random. Even if we log in without entering any username and password at all, it still lets us log in.

A screenshot of a web-based login interface titled "Factory Login". At the top right are "Home" and "Sign Out" buttons. The main area contains two input fields: one for "Username" containing "test" and another for "Password" containing "....". Below the inputs is a large green "Sign In" button. The entire form is set against a light gray background.

© PicoCTF 2019

After this step, the website displays messages like: “Success: You logged in! Not sure you'll be able to see the flag though.” And “No flag for you.” These messages indicate that while the login was accepted, the actual goal, which is finding the flag is not straightforward. It hints that the flag may be hidden somewhere within the inner parts of the page. I noticed four different links in the source code, so I explored each of them, hoping to find some useful information. However, none of them helped, and I could not find any mention of the flag or anything that looked relatable. Since this method did not lead to a result, the next step was to use Developer Tools.

Developer Tools, often called DevTools, provide a much deeper look into how a webpage functions which means they let us view JavaScript files, examine data stored in cookies or local storage, and even what scripts are running in the background. They are especially useful for examining dynamic content that does not appear in the static source code.

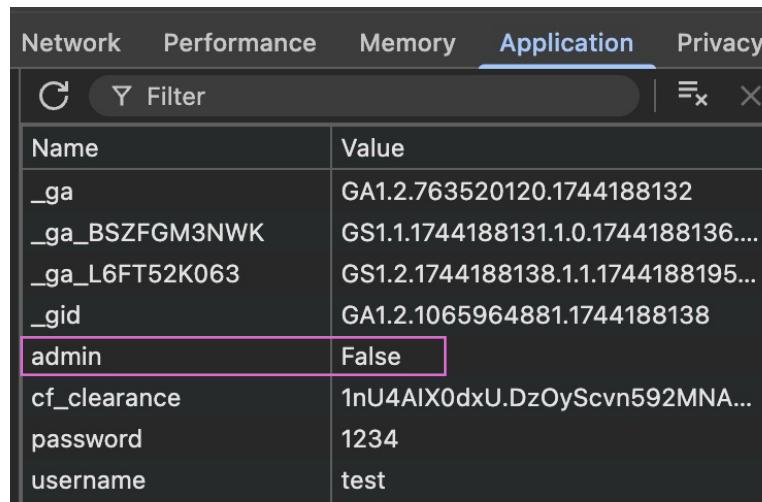
The screenshot shows a browser window with the title "Factory Login". Below the title, a green bar displays the message "Success: You logged in! Not sure you'll be able to see the flag though." The main content area contains the text "No flag for you". At the bottom left, it says "© PicoCTF 2019". To the right of the browser window is the Chrome context menu, which is open near the top-right corner. The menu includes options like "New Tab", "New Window", "New Incognito Window", "Passwords and Autofill", "History", "Downloads", "Bookmarks and Lists", "Tab Groups", "Extensions", "Delete Browsing Data...", "Zoom" (set to 100%), "Print...", "Search with Google Lens", "Translate...", "Find and Edit", "Cast, Save, and Share", "More Tools", "Help", and "Settings". The "More Tools" option is highlighted with a pink rectangle. Below the context menu, a separate developer tools menu is shown, also with "Developer Tools" highlighted by a pink rectangle.

To access Developer Tools, we can click the three vertical dots in the top-right corner of the browser window, near the address bar. From there, we go to “More Tools” and then choose “Developer Tools.”. Alternatively, there are shortcuts like F12 or Ctrl+Shift+I on Windows, or Cmd+Option+I on Mac.

The screenshot shows the "Application" tab of the Chrome DevTools interface. On the left, a sidebar lists various storage types: Manifest, Service workers, Storage, Local storage, Session storage, Extension storage, IndexedDB, Cookies, Shared storage, Cache storage, and Storage buckets. Under "Cookies", a section for the URL "https://jupiter.challenges.picoctf.org" is expanded, showing entries for "_ga", "_ga_BSZFGM3NWK", "_ga_L6FT52K063", "_gid", "admin", "cf_clearance", "password", and "username". The main panel displays a table of cookies with columns for Name, Value, Do..., Path, Expi..., Size, Http..., Sec..., Sam..., Part..., Cro..., and Prio... . A filter bar at the top of the table allows filtering by name. A message at the bottom of the table reads "No cookie selected Select a cookie to preview its value".

In Development Tools, we will see Main Navigation (top tabs), which are Elements, Console, Sources, Network, and Application. We need to choose the Application tab because it lets us view cookies, local storage, and session storage which often store login tokens. It is basically useful for seeing what data the site stores in our browser and debugging app behavior.

Additionally, there is a sidebar (Application Sections). Here we need to go to the “Cookies” section which includes the website <http://jupiter.challenges.picoctf.org:15796> because it shows all cookies saved by that website. Because cookies store login info, user sessions, and additional tokens, by clicking on it, we can see what this site remembered about us and find issues with login or sessions.



A screenshot of the Chrome DevTools Application tab. The tab bar at the top has tabs for Network, Performance, Memory, Application (which is underlined in blue), and Privacy. Below the tabs is a toolbar with a refresh icon, a filter input field, and other icons. The main area is a table titled 'Cookies'. The table has two columns: 'Name' and 'Value'. There are nine rows in the table. The 'admin' row is highlighted with a pink border. The data is as follows:

Name	Value
_ga	GA1.2.763520120.1744188132
_ga_BSZFGM3NWK	GS1.1.1744188131.1.0.1744188136....
_ga_L6FT52K063	GS1.2.1744188138.1.1.1744188195...
_gid	GA1.2.1065964881.1744188138
admin	False
cf_clearance	1nU4AIX0dxU.DzOyScvn592MNA...
password	1234
username	test

Here we can see the username, password, and other details. The site has saved my login credentials with the username set to test and password as 1234. Moreover, there is another cookie named as admin, which is set to “False.” This indicates that I am recognized as a regular user, not someone with administrative access (not authorized). Because of this, certain content like a hidden flag is not being displayed on the page.

Network	Performance	Memory	Application	Privacy
Name	Value			
_ga	GA1.2.763520120.1744188132			
_ga_BSZFGM3NWK	GS1.1.1744188131.1.0.1744188136...			
_ga_L6FT52K063	GS1.2.1744188138.1.1.1744188195...			
_gid	GA1.2.1065964881.1744188138			
admin	True			
cf_clearance	1nU4AIx0dxU.DzOyScvn592MNA...			
password	1234			
username	test			

Now that we understand the reason, we can take a simple step to bypass this restriction. By manually changing the value of the admin cookie from “False” to “True”, we can make the website believe that I am an admin user. Once this change is made, all we need to do is refresh the page. After refreshing, the page reloads with admin-level access, and the hidden flag becomes visible.

4. Mod 26

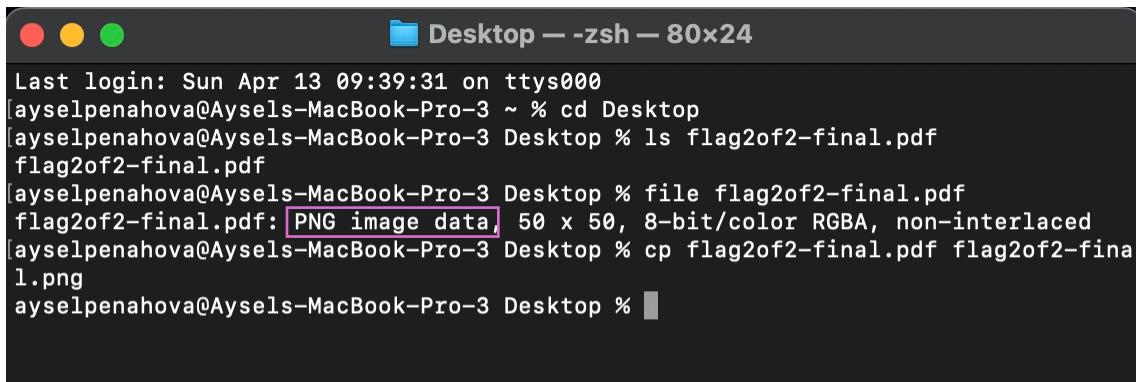
Cryptography is the process of transforming information to make it unreadable to unauthorized people. ROT13 is a simple encryption method where each alphabet letter is shifted 13 places forward. For example, the letter “A” becomes “N”, and “B” becomes “O”. If we apply this method to the

“cvpbPGS{arkg_gvzr_V’yy_gel_2_ebhaqf_bs_ebg13_Ncualgvd}”, each letter will be replaced by the letter 13 places forward. The decoded text will be
 “picoCTF{next_time_I'll_try_2_rounds_of_rot13_Aphnytiq}”

5. Secret of the Polyglot

In this task, we are asked to extract all the information from a given file. After downloading the file mentioned in the instructions, we received a PDF named “flag2of2-final.pdf”. When I opened it, I saw a piece of text that looked like the last part of a flag, which was “**1n_pn9_&_pdf_53b741d6}**”. However, the first part of the flag was missing.

There was a hint which suggested opening the file in different ways, therefore I decided to use the terminal. First, I used the “cd Desktop” command to navigate to the Desktop, where the file was saved. After this command, I used “ls flag2of2-final.pdf” to confirm the file was there. Then, we need to inspect the actual file. Therefore, I used the “file flag2of2-final.pdf” command.



```
Last login: Sun Apr 13 09:39:31 on ttys000
[ayselpenahova@Aysels-MacBook-Pro-3 ~ % cd Desktop
[ayselpenahova@Aysels-MacBook-Pro-3 Desktop % ls flag2of2-final.pdf
flag2of2-final.pdf
[ayselpenahova@Aysels-MacBook-Pro-3 Desktop % file flag2of2-final.pdf
flag2of2-final.pdf: [PNG image data, 50 x 50, 8-bit/color RGBA, non-interlaced
[ayselpenahova@Aysels-MacBook-Pro-3 Desktop % cp flag2of2-final.pdf flag2of2-final
1.png
ayselpenahova@Aysels-MacBook-Pro-3 Desktop %
```

Although the file had a “.pdf” extension, the “file” command revealed that it was actually a PNG image, not a real PDF. To reveal its contents more clearly, I used the “cp flag2of2-final.pdf flag2of2-final.png”. This command makes an exact duplicate of flag2of2-final.pdf but names it “flag2of2-final.png”. If you are using macOS, you can also use the “sips -s format png flag2of2-final.pdf --out flag2of2-final.png” command. This command converts the PDF into a PNG image unlike cp, which copies without conversion.



Now we can see the first part of the flag: “**picoCTF{f1u3n7-**“.

6. Verify

```
apanahova16022-picoctf@webshell:~$ ssh -p 56575 ctf-player@rhea.picoctf.net
ctf-player@rhea.picoctf.net's password:
Welcome to Ubuntu 20.04.3 LTS (GNU/Linux 6.8.0-1021-aws x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.
```

In this task, we start by connecting to a remote server using SSH. The command “ssh -p 52706 ctf-player@rhea.picoctf.net” tells our system to securely log into the server at rhea.picoctf.net using port 565775 and the username ctf_player. After entering the password, we are successfully logged into the server. When we type the password, no characters appear on the screen. It is hidden because of the security.

```
ctf-player@pico-chall$ ls
checksum.txt  decrypt.sh  files
```

Once inside the server, we need to check the contents of the current directory with the “ls” command. It shows three items. A file named “checksum.txt”, a script named “decrypt.sh”, and a folder named “files”.

```
ctf-player@pico-chall$ cat checksum.txt
fba9f49bf22aa7188a155768ab0dfdc1f9b86c47976cd0f7c9003af2e20598f7
```

Next, we open “checksum.txt” because it is indicated in the instructions using “cat checksum.txt”. The cat command stands for “concatenate” and is often used to display the contents of the file directly in the terminal. When running this, the terminal prints out a long string of characters. That string is a SHA-256 checksum. SHA-256 is a type of mathematical

formula that takes some input like the contents of a file and produces a string of characters that looks very random. In the case of SHA-256, it always gives a 64-character log string, no matter how big or small the file is. This 64-character result is called a checksum, which acts like a unique ID for the contents of that file. Every file has a unique SHA-256 checksum. If we find a file that exact checksum, we have found the correct one.

```
ctf-player@pico-chall$ sha256sum files/* | grep fba9f49bf22aa7188a155768ab  
0dfdc1f9b86c47976cd0f7c9003af2e20598f7  
fba9f49bf22aa7188a155768ab0dfdc1f9b86c47976cd0f7c9003af2e20598f7  files/87  
590c24
```

To search for the file that matches this checksum, we run a command that calculates the SHA-256 checksums for all the files inside the “files” folder – “`sha256sum files/*`”. “`files/*`” means applying the command to all the files inside the “files” directory. However, this command will give us a full list. We do not need to scroll through hundreds of lines looking for a match manually. Therefore we add “`grep`” and copy and paste the checksum (the one from `checksum.txt`) right after it. This is done by using the `|` symbol, which connects the two commands. The full version of the command looks like “`sha256sum files/* | grep fba9f49bf22aa7188a155768ab0dfdc1f9b86c47976cd0f7c9003af2e20598f7`”

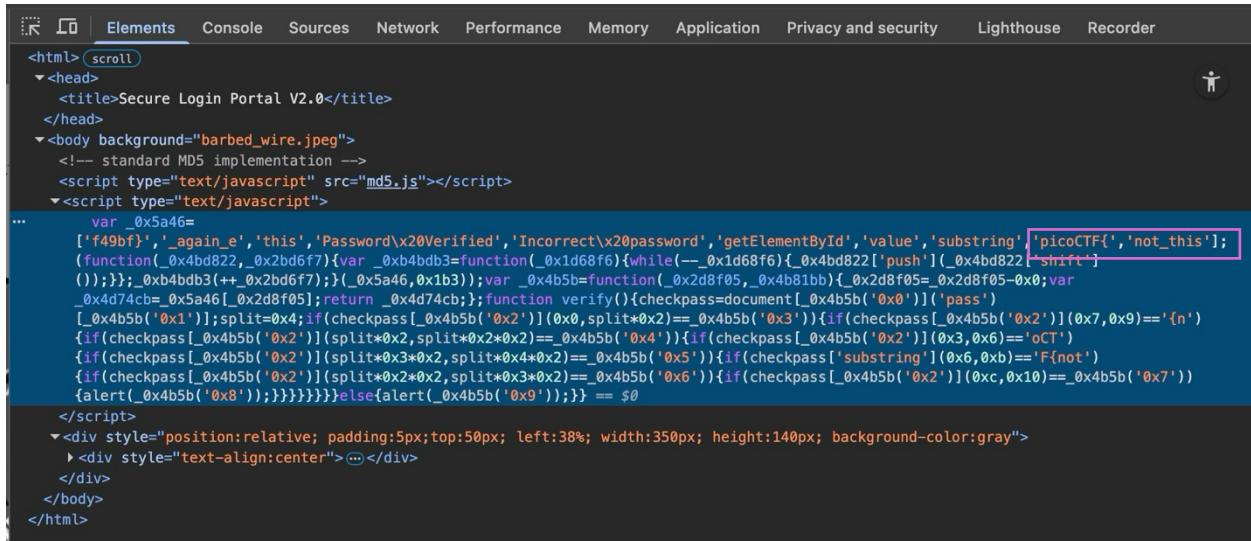
One of the files inside the files folder, named “`87590c24`”, has the exact same checksum. That means this is the file that we needed.

```
ctf-player@pico-chall$ ./decrypt.sh files/87590c24  
picoCTF{trust_but_verify_87590c24}
```

Finally, we use a script named `decrypt.sh` and pass it the path to the matching file with the command “`./decrypt.sh files/87590c24`”. This command processes the file and outputs a flag in the picoCTF format.

7. Client-side AGAIN

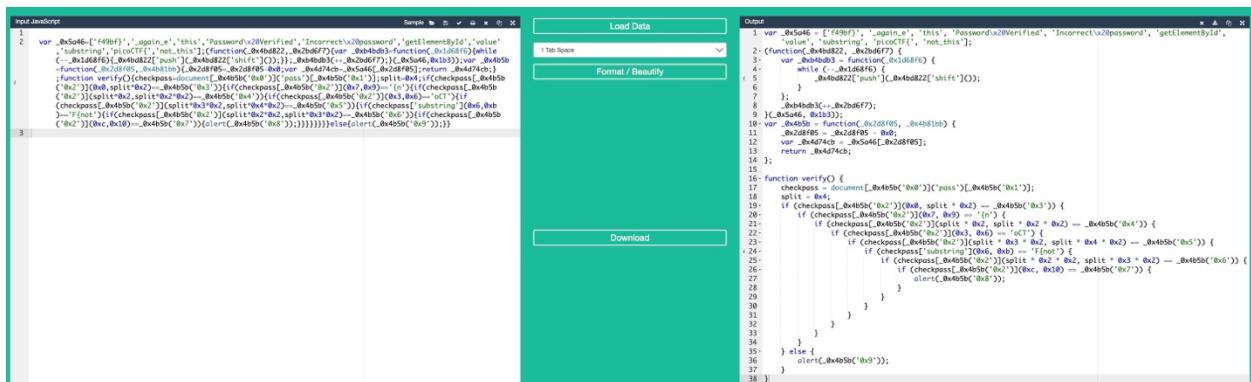
In this task, after clicking the provided link, a webpage opens. The first step that should do is to inspect the website to see if there are any hidden details or messages that might help us find the flag. To do this, we right-click anywhere on the page and select “Inspect” from the menu. This opens the browser’s developer tools.



The screenshot shows the browser's developer tools with the "Elements" tab selected. The page source code is displayed, containing a large block of obfuscated JavaScript. A specific line of code is highlighted in blue, which contains the string "picoCTF{'not_this'". This indicates that the code is likely part of a challenge related to the picoCTF competition.

```
<html> scroll
<head>
  <title>Secure Login Portal V2.0</title>
</head>
<body background="barbed_wire.jpeg">
  <!-- standard MD5 implementation -->
  <script type="text/javascript" src="md5.js"></script>
<<script type="text/javascript">
  var _0x5a46=
    ['f49bf','_again_e','this','Password\x20Verified','Incorrect\x20password','getElementById','value','substring'];
    picoCTF{'not_this'};
  (function(_0x4bd822,_0xbdf6f){var _0xb4bdb3=function(_0x1d68f6){while(--_0x1d68f6){_0x4bd822['push'](_0x4bd822['shift'])}}};_0xb4bdb3(++_0xbdf6f)};{_0x5a46,_0xb1b3});var _0x4b5b=function(_0x2d8f05,_0x4b81bb){_0x2d8f05=_0x2d8f05-0x0;var _0x4d74cb=_0x5a46[_0x2d8f05];return _0x4d74cb};function verify(){checkpass=document[_0x4b5b('0x0')](pass')
  [_0x4d74cb[_0x1']);split=_0x4;if(checkpass[_0x4b5b('0x2')](0x0,split*0x2)==_0x4b5b('0x3')){if(checkpass[_0x4b5b('0x2')](0x7,0x9)=='n')
  {if(checkpass[_0x4b5b('0x2')](split*0x2,split*0x2*0x2)==_0x4b5b('0x4')){if(checkpass[_0x4b5b('0x2')](0x3,0x6)=='oCT')
  {if(checkpass[_0x4b5b('0x2')](split*0x3*0x2,split*0x4*0x2)==_0x4b5b('0x5')){if(checkpass['substring'](0x6,0xb)=='F(not)')
  {if(checkpass[_0x4b5b('0x2')](split*0x2*0x2,split*0x3*0x2)==_0x4b5b('0x6')){if(checkpass[_0x4b5b('0x2')](0xc,0x10)==_0x4b5b('0x7'))
  {alert(_0x4b5b('0x8'))};}}}}}}}}else{alert(_0x4b5b('0x9'))};}} == $0
</script>
<div style="position:relative; padding:5px; top:50px; left:38%; width:350px; height:140px; background-color:gray">
  <div style="text-align:center">...</div>
</div>
</body>
</html>
```

Inside the Inspect, we look through the head section of the page and notice a very long and complex line of JavaScript code. Even though it looks confusing at first, we can see the term “picoCTF” inside it, which tells us this code is probably important and might contain the flag.



The screenshot shows the browser's developer tools with the "Elements" tab selected. The page source code is displayed, but it is heavily compressed into a single long line. To the left, the "Input JavaScript" panel shows the original compressed code, and to the right, the "Output" panel shows the code after being beautified by an online tool. The beautified code is much more readable, showing the structure of the functions and their logic.

```
Input JavaScript
1 var _0x5a46=['f49bf','_again_e','this','Password\x20Verified','Incorrect\x20password','getElementById','value','substring'];
2   picoCTF{'not_this'};
3   (function(_0x4bd822,_0xbdf6f){var _0xb4bdb3=function(_0x1d68f6){while(--_0x1d68f6){_0x4bd822['push'](_0x4bd822['shift'])}}};_0xb4bdb3(++_0xbdf6f)};{_0x5a46,_0xb1b3});var _0x4b5b=function(_0x2d8f05,_0x4b81bb){_0x2d8f05=_0x2d8f05-0x0;var _0x4d74cb=_0x5a46[_0x2d8f05];return _0x4d74cb};function verify(){checkpass=document[_0x4b5b('0x0')](pass')
4   [_0x4d74cb[_0x1']);split=_0x4;if(checkpass[_0x4b5b('0x2')](0x0,split*0x2)==_0x4b5b('0x3')){if(checkpass[_0x4b5b('0x2')](0x7,0x9)=='n')
5   {if(checkpass[_0x4b5b('0x2')](split*0x2,split*0x2*0x2)==_0x4b5b('0x4')){if(checkpass[_0x4b5b('0x2')](0x3,0x6)=='oCT')
6   {if(checkpass[_0x4b5b('0x2')](split*0x3*0x2,split*0x4*0x2)==_0x4b5b('0x5')){if(checkpass['substring'](0x6,0xb)=='F(not)')
7   {if(checkpass[_0x4b5b('0x2')](split*0x2*0x2,split*0x3*0x2)==_0x4b5b('0x6')){if(checkpass[_0x4b5b('0x2')](0xc,0x10)==_0x4b5b('0x7'))
8   {alert(_0x4b5b('0x8'))};}}}}}}}}else{alert(_0x4b5b('0x9'))};}} == $0
9   </script>
10  <div style="position:relative; padding:5px; top:50px; left:38%; width:350px; height:140px; background-color:gray">
11    <div style="text-align:center">...</div>
12  </div>
13  </body>
14 </html>

Load Data
Format / Beautify
Download
Output
```

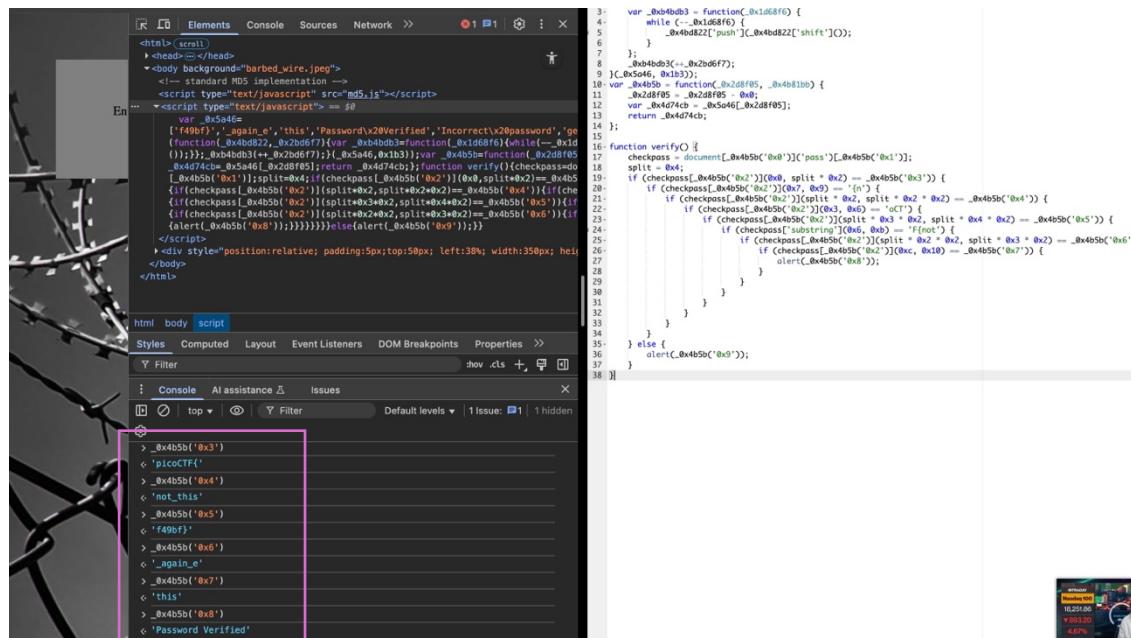
Since the code is compressed into one long unreadable line, we can make it more readable using an online “beautifie” tool. I used “<https://jsonformatter.org/jsbeautifier>”. After pasting the code there and “beautifying” it, the structure became much clearer.

The screenshot shows a browser developer tools console with the following content:

```
function verifyO {
    var checkpass = document[_0x4b5b('0x0')].value;
    split = _0x4b5b('0x3');
    if (checkpass[_0x4b5b('0x0')](_0x4b5b('0x0')) == _0x4b5b('0x1')) {
        if (checkpass[_0x4b5b('0x0')](_0x4b5b('0x2'))(_0x4b5b('0x0')) == _0x4b5b('0x2')) {
            if (checkpass[_0x4b5b('0x0')](_0x4b5b('0x3'))(_0x4b5b('0x0')) == _0x4b5b('0x3')) {
                if (checkpass[_0x4b5b('0x0')](_0x4b5b('0x4'))(_0x4b5b('0x0')) == _0x4b5b('0x4')) {
                    if (checkpass[_0x4b5b('0x0')](_0x4b5b('0x5'))(_0x4b5b('0x0')) == _0x4b5b('0x5')) {
                        if (checkpass[_0x4b5b('0x0')](_0x4b5b('0x6'))(_0x4b5b('0x0')) == _0x4b5b('0x6')) {
                            if (checkpass[_0x4b5b('0x0')](_0x4b5b('0x7'))(_0x4b5b('0x0')) == _0x4b5b('0x7')) {
                                if (checkpass[_0x4b5b('0x0')](_0x4b5b('0x8'))(_0x4b5b('0x0')) == _0x4b5b('0x8')) {
                                    alert(_0x4b5b('0x9'));
                                }
                            }
                        }
                    }
                }
            }
        }
    }
} else {
    alert(_0x4b5b('0x9'));
}
```

While exploring the formatted code, I found a “function verify() {}” part of the code, which includes several checkpass function calls. Each of these seems to refer to something like “_0x4b5b(‘0x3’), “_0x4b5b(‘0x4’), “_0x4b5b(‘0x5’), and so on. These calls all use the same base function (_0x4b5b), but with different values inside the parentheses. Therefore, each of these might represent a different part of the flag.

I went back to the website and opened the Inspect tool again. This time in the Console tab at the bottom, I tried running “`_0x4b5b('0x3')`”, which is the first value used in the `verify()` function. It returned the beginning of the flag –“`picoCTF{`“, confirming that this function helps decode the flag.



The screenshot shows a browser developer tools interface with the "Elements" tab selected. The page source code is visible, showing a script that performs MD5 hashing on user input and compares it against a stored password hash. The exploit code injects a function to intercept the password hash and alert it to the user.

```
4: var _0x4b5b = Function(_0x1d68f0) {
5:   while (..._0x4bd82["push"](_0x4bd82["shift"]));
6: }
7: _0x4bd82[..._0xb2d6f7];
8: }(_0x4bd82, _0x1d68f0);
9: _0x4bd82["push"](_0x2d8f95, _0x4b81b0);
10: var _0x2d8f95 = _0x2d8f95 - _0x8;
11: var _0x4d74cb = _0x5a46[_0x2d8f03];
12: return _0x4d74cb;
13;
14;
15: function verify() {
16:   checkpass=document[_0x4b5b('0x0')]("pass")[_0x4b5b('0x1')];
17:   split = _0x4b5b('0x1');
18:   if (checkpass[_0x4b5b('0x2')](0x0, split + _0x2) == _0x4b5b('0x3')) {
19:     if (checkpass[_0x4b5b('0x2')](0x1, split + _0x2) == _0x4b5b('0x4')) {
20:       if (checkpass[_0x4b5b('0x2')](0x2, split + _0x2) == _0x4b5b('0x5')) {
21:         if (checkpass[_0x4b5b('0x2')](0x3, split + _0x2) == _0x4b5b('0x6')) {
22:           if (checkpass[_0x4b5b('0x2')](0x4, split + _0x2) == _0x4b5b('0x7')) {
23:             if (checkpass[_0x4b5b('0x2')](0x5, split + _0x2, split + _0x4 + _0x2) == _0x4b5b('0x8')) {
24:               if (checkpass[_0x4b5b('0x2')](0x6, split + _0x2, split + _0x4 + _0x2) == _0x4b5b('0x9')) {
25:                 if (checkpass[_0x4b5b('0x2')](0x7, split + _0x2, split + _0x4 + _0x2) == _0x4b5b('0xa')) {
26:                   if (checkpass[_0x4b5b('0x2')](0x8, split + _0x2, split + _0x4 + _0x2) == _0x4b5b('0xb')) {
27:                     if (checkpass[_0x4b5b('0x2')](0x9, split + _0x2, split + _0x4 + _0x2) == _0x4b5b('0xc')) {
28:                       if (checkpass[_0x4b5b('0x2')](0x10, split + _0x2, split + _0x4 + _0x2) == _0x4b5b('0xd')) {
29:                         if (checkpass[_0x4b5b('0x2')](0x11, split + _0x2, split + _0x4 + _0x2) == _0x4b5b('0xe')) {
30:                           if (checkpass[_0x4b5b('0x2')](0x12, split + _0x2, split + _0x4 + _0x2) == _0x4b5b('0xf')) {
31:                             if (checkpass[_0x4b5b('0x2')](0x13, split + _0x2, split + _0x4 + _0x2) == _0x4b5b('0x10')) {
32:                               if (checkpass[_0x4b5b('0x2')](0x14, split + _0x2, split + _0x4 + _0x2) == _0x4b5b('0x11')) {
33:                                 if (checkpass[_0x4b5b('0x2')](0x15, split + _0x2, split + _0x4 + _0x2) == _0x4b5b('0x12')) {
34:                                   if (checkpass[_0x4b5b('0x2')](0x16, split + _0x2, split + _0x4 + _0x2) == _0x4b5b('0x13')) {
35:                                     if (checkpass[_0x4b5b('0x2')](0x17, split + _0x2, split + _0x4 + _0x2) == _0x4b5b('0x14')) {
36:                                       if (checkpass[_0x4b5b('0x2')](0x18, split + _0x2, split + _0x4 + _0x2) == _0x4b5b('0x15')) {
37:                                         alert(_0x4b5b('0x9'));
38:                                       }
39:                                     }
40:                                   }
41:                                 }
42:                               }
43:                             }
44:                           }
45:                         }
46:                       }
47:                     }
48:                   }
49:                 }
50:               }
51:             }
52:           }
53:         }
54:       }
55:     }
56:   }
57: }
58: } else {
59:   alert(_0x4b5b('0x9'));
60: }
```

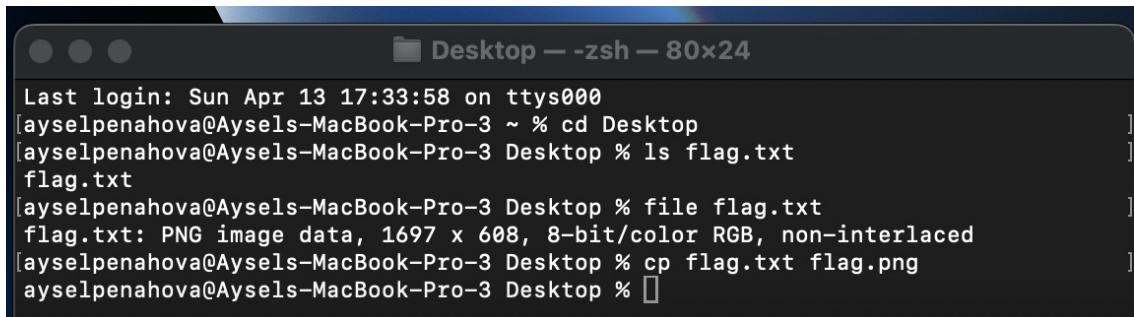
After that, I continued running the next values in order, like “`_0x4b5b('0x4')`”, “`_0x4b5b('0x5')`”, and so on, up to “`_0x4b5b('0x8')`”. Each one revealed the next piece of the flag. I did not run “`_0x4b5b('0x9')`” because that part appears in an “else” block in the code.

By collecting all the valid pieces from the main checkpass calls, I was able to logically decode the full flag: **picoCTF{not_this_again_ef49bf}**

8. Extensions

This task is similar to the “Secret of the Polyglot” challenge, where the key idea is to recognize that a file may not actually be what it appears based on its name or extension.

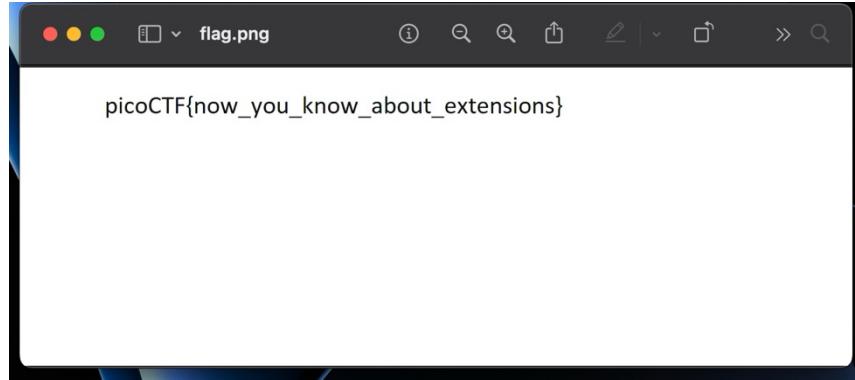
After downloading the file provided in the instructions, we see that it is named as “flag.txt”. At first sight, it seems to be a txt file. However, based on the challenge description, we already suspect that it is not a real txt file, but actually, some other file type that has been renamed as a .txt extension.

A screenshot of a terminal window titled "Desktop — zsh — 80x24". The terminal shows the following session:

```
Last login: Sun Apr 13 17:33:58 on ttys000
[ayselpenahova@Aysels-MacBook-Pro-3 ~ % cd Desktop
[ayselpenahova@Aysels-MacBook-Pro-3 Desktop % ls flag.txt
flag.txt
[ayselpenahova@Aysels-MacBook-Pro-3 Desktop % file flag.txt
flag.txt: PNG image data, 1697 x 608, 8-bit/color RGB, non-interlaced
[ayselpenahova@Aysels-MacBook-Pro-3 Desktop % cp flag.txt flag.png
ayselpenahova@Aysels-MacBook-Pro-3 Desktop % ]
```

The terminal window has a dark background with light-colored text. The title bar is "Desktop — zsh — 80x24". The command history shows navigating to the Desktop folder, listing "flag.txt", running the "file" command which correctly identifies it as a PNG image, and then copying it to "flag.png". The prompt ends with a closing bracket and a cursor.

To figure out what kind of file it really is, I used a few commands in my terminal. First, I navigated to the folder where I saved the file (Desktop) by using the “cd” command. Then i ran “ls flag.txt” to check whether the file is there or not. Next, to inspect the actual type of the file, I used the command “file flag.txt”. Even though the file was named “flag.txt”, the output showed that it was actually a PNG image.



Since the file is really an image, we need to give it the corrected extension so that it can be opened properly. To do that, I made a copy of the file and renamed it with a .png extension using the “cp flag.txt flag.png”.

Now, when we open “flag.png”, we will be able to see the hidden flag.

9. So Meta

In this task, we need to search for the image’s metadata to find the flag. Therefore I open the terminal and access the file using the “cd” command. Then, “ls” to check whether the file is there or not.

Now, here comes the important part, which is checking the metadata. Metadata is hidden information stored inside a file. For images, it can include information like creation date, software, some notes, etc. To look at the metadata, we need to use the tool called “exiftool”. This command helps to scan the file and print out all the metadata that is inside the image. When I first tried to use the “exiftool” command in my terminal, I got a message that this command is not found. To fix this, I used “Homebrew”, which is a package manager for macOS. With the help of this package I installed “exiftool”.

```
Desktop — -zsh — 108x45
Last login: Sun Apr 13 19:18:02 on ttys000
ayselpenahova@Aysels-MacBook-Pro-3 ~ % cd Desktop
ayselpenahova@Aysels-MacBook-Pro-3 Desktop % ls pico_img.png
pico_img.png
ayselpenahova@Aysels-MacBook-Pro-3 Desktop % exiftool pico_img.png
ExifTool Version Number      : 13.25
File Name                   : pico_img.png
Directory                   : .
File Size                   : 109 kB
File Modification Date/Time : 2025:04:13 19:11:16+04:00
File Access Date/Time       : 2025:04:13 19:12:14+04:00
File Inode Change Date/Time: 2025:04:13 19:12:14+04:00
File Permissions            : -rw-r--r--
File Type                   : PNG
File Type Extension         : png
MIME Type                   : image/png
Image Width                 : 600
Image Height                : 600
Bit Depth                   : 8
Color Type                  : RGB
Compression                 : Deflate/Inflate
Filter                      : Adaptive
Interlace                    : Noninterlaced
Software                     : Adobe ImageReady
XMP Toolkit                 : Adobe XMP Core 5.3-c011 66.145661, 2012/02/06-14:56:27
Creator Tool                 : Adobe Photoshop CS6 (Windows)
Instance ID                  : xmp.iid:A5566E73B2B811E8BC7F9A4303DF1F9B
Document ID                  : xmp.did:A5566E74B2B811E8BC7F9A4303DF1F9B
Derived From Instance ID     : xmp.iid:A5566E71B2B811E8BC7F9A4303DF1F9B
Derived From Document ID    : xmp.did:A5566E72B2B811E8BC7F9A4303DF1F9B
Warning                      : [minor] Text/EXIF chunk(s) found after PNG IDAT (may be ignored by some readers)
Artist                       : picoCTF{s0_m3ta_fec06741}
Image Size                   : 600x600
Megapixels                   : 0.360
ayselpenahova@Aysels-MacBook-Pro-3 Desktop %
```

As we can see from the screenshot, there is a field called “Artist”, where the flag is hidden.