Jeffry Jimenez
CSE 331.01
Professor Rahmati
Fall 2020

Report

TASK1:

FLAG: picoCTF{no_clients_plz_ee2f24}

To get the flag in this task, a quick look of the source code did half the work. Pieces of the flag were scatter within the conditions of the "if statements", all I had to do was to look at which part of the substring the if statement was looking at and place that substring in the correct position of my Flag.

While implementing the above in python, I used 3 simple libraries to get the task done (requests, bs4, and re). I made I simple request with the requests library, and then parsed the html and extracted the <script> tags with bs4. I the used regular expressions to picked out the flag fragments and their indexing in the if statements. Finally, I used some simple logic to concatenate the flag fragments in the correct other using the indexing that belonged to each segment.

TASK2:

FLAG: picoCTF{not_this_again_39d025}

To get the flag in this task, a quick look of the source code did half the work. Pieces of the flag were scatter within the conditions of the "if statements". However, unlike task1 some of the conditions use the array _0x5a46 to hide some of the pieces of the flag. To find the correct flag all I needed to do was to look at the conditions of the if statements to find a substring of the flag, and the look at the indexes used for checkpass to find were in the string the substring goes in the overall flag.

Task 2 was almost identical to task 1, in this case I used 4 simple libraries to get the task done (requests, bs4, re, jsbeautifier). I made I simple request with the requests library, and then parsed the html and extracted the <script> tags with bs4. The task2 implementation differs from task1, because of obfuscation using regular expressions became a little to annoying, so to prettify the code I used the jsbeautifier library as a deobfuscator. After parsing the <scrip> tags with jsbeautifier, I then used regular expressions to pick out the flag fragments and their indexing from the if statements. Finally, I used some simple logic to concatenate the flag in the correct order using the indexing that belonged to each segment.

TASK3:

FLAG: picoCTF{p1c0_s3cr3t_ag3nt_7e9c671a}

To get the flag in this task I had to utilize Burp Suite to intercept the request. In the settings I used the regex, (^User-Agent.*$) to replace User-Agent with “User-Agent: picobrowser” and the response contained the flag.

In this implementation I utilized the libraries (requests, bs4, and re). I sent the request with the requests library which also allowed me to set my custom header which I set as "User-Agent": "picobrowser". I then parsed the response with bs4 and retrieved the flag with re using regular expressions.

TASK4:

FLAG: picoCTF{s0m3_SQL_93e76603}

To get the flag in this Task all I had to do was a simple SQL injection of the login form. For the username I entered (' or 1=1 --) which evaluates the username field to true and comments out the password section.

To create a script for task 4 I utilized the mechanize and re library. The code simply opens the login link and with mechanize I set the SQL injection (' or 1 = 1--) as a username and submit the form. I used re to use regular expressions to fetch out the flag from the response object returned by mechanized after I submit.

TASK5:

FLAG: picoCTF{m0R3_SQL_plz_015815e2}

In this Task I used SQL injection to the flag, unlike task 4 (' or 1=1 --) was being detected by the sever to stop me from being malicious. However, entering the username as admin and then commenting out everything else seem to work. This was my SQL injection (admin’--).

The code for task 5 is almost identical to task 4. I utilized the same libraries: mechanize and re. The code simply opens the login link and with mechanize I set the SQL injection ('admin-

-) as a username and submit the form. I used re to use regular expressions to fetch out the flag from the response object returned by mechanized after I submit.

TASK6:

FLAG: picoCTF{3v3n_m0r3_SQL_c2c37f5e}

To get the flag in this task I had to utilize Burp Suit. When I send a random password the POST request that Burp Suit intercepts has a debug option set to false (0) if I set it to true (1) when the password gets rejected, the response contain an encrypted version of the password I sent. abc -> nop which looks like a simple Caesar Cipher with shift 13. If I do a simple SQL injection and enter (' or 1=1 --) encrypted by the same Caesar Cipher (' be 1=1 --) as a password, it lets me bypass the login and returns the flag.

The code for task 6 is also like task 4 and 5. I utilized the same libraries: mechanize and re. The code simply opens the login link and with mechanize I set the SQL injection (' be 1=1 --) as a username and submits the form. I used re to use regular expressions to fetch out the flag from the response object returned by mechanized after I submit.

TASK7:

FLAG: picoCTF{th3_c0nsp1r4cy_l1v3s_a03e3590}

To get this flag there is a few simple things to do. The First one is to do a little SQL injection by signing in as an admin'--, this will make the password field be ignored. This method let me sign in as an admin, but the cookie returned by this method is set to false, and because of that you don't get the flag. To fix this all I had to do was change the admin cookie value to True and refresh the page and DONE!

Task 7 is a little more complicated than 5, 4, and 6, but its basically the same idea. In this task I used the libraries re and selenium. With selenium I open a firefox webdriver (note! you will need the geckodriver to open a firefox driver) and attempted to do a SQL injection. With selenium I simply search for the element with id 'email' set its value to (admin'--) and then click the submit button. After submitting the from I used selenium to delete the cookie named admin, replaced it with another cookie {name=admin, value=True} and refreshed the page. Lastly, I used selenium to find a tag <code> and parse it using re.

TASK8:

FLAG: picoCTF{jawt_was_just_what_you_thought_d571d8aa3163f61c144f6d505ef2919b}

In order to get the flag for this task, I signed in as John, and then took the cookie with the JWT key from the browser. I Then use John The Ripper with the file rockyou.txt as –wordlist

parameter, to crack the secret used in the JWT token, it returned ‘ilovepico’. After cracking the secret, I pasted the original JWT in the JWT debugger and replaced user John with admin and applied the cracked secret. Now all that was left to do was to replace the cookie in the browser with the modified JWT key.

The code in task 8 is also simple. I took the liberty or running John The Ripper <$ john --wordlist=/Users/jeffryjimenez/Desktop/rockyou.txt john.hash> and setting a constant ‘SECRET’ to the value returned by JTR ‘ilovepico’. In this task I used the libraries: re, selenium, and pyjwt. With selenium I opened a firefox webdriver to the desired link (you will need geckodriver to do this), I then looked for the element with id ‘name’, set its value to ‘John’ and used selenium to press enter. I then looked for the cookie named ‘jwt’. The reason why there is a while loop repeatedly trying to fetch the cookie is because sometimes selenium tries to get the cookie before it exits, the while loop just assures the cookie is fetched before proceeding to the next instruction. The script then decodes the jwt token that was inside the cookie, using the pyjwt library. I edit the user field in the token by replacing john with admin, and finally encode the token using pyjwt, with the new playload being {user: admin} and the secret ‘ilovepico’ which we got from JTR. We then use selenium to delete the cookie named ‘jwt’ from the browser, replaced it with another cookie also named ‘jwt’ but the new value is the jwt token we encoded, then refresh the page. Lastly, I used selenium to find a tag <textarea> and parse it using re.