CSE 545: Software Security Project Portfolio

Shreyas Chandrashekhar Kirtane
Arizona State University
ASU ID: 1225453736
skirtan1@asu.edu

***Abstract*—Software security is a fundamental field of computer and information systems. It is a critical pillar in today's digital landscape where the proliferation of interconnected systems and ever evolving threat landscape pose significant challenges to the integrity and confidentiality of data. This document is a portfolio report for the course project of Software Security undertaken in the semester of Spring 2024. It describes the candidate challenges from different modules covering techniques to identify, exploit and safeguard against various software vulnerabilities on pwn.college. This report describes techniques such as Stack overflow, Cross site request forgery and Cross site scripting attacks as used in these challenges. Furthermore, this report covers the solutions to these challenges and implementation of the exploits, comparison with related challenges and some ideas to prevent such vulnerabilities. Finally, lessons learned from these projects are described.**

***Index Terms*—stack, overflow, request, forgery, scripting**

I. Introduction

Maintaining the integrity and confidentiality of user data is very critical in today's age where all information of a person can be found online. The course Software Security, CSE 545 delves into some of the most common and critical software vulnerabilities. These vulnerabilities can be found in the programs for challenges on pwn.college. Pwn.college is a CTF style website designed to gain piratical experience learning about the vulnerabilities and how to exploit them. In CTF challenges the user is tasked with identifying vulnerabilities in a program or software system and getting access to a flag. The flag then can be submitted to the challenge website to get credit. In this report, Challenges 4.38 Little Dipper and 4.39 Big Dipper from module 4 (Hijacking Binary Power(Pwning)) and Challenges 5.07 Leak in my data, 5.13 Underground Data Escape and 5.14 Underground Data Escape II from module 5 (Wrecking the web world) will be covered. Most of the challenges in module 4 involve exploiting a program with SUID bit set. This allows the attacker to obtain a shell with super user privileges by exploiting the vulnerability in the program. The attacker can then access the flag file to obtain the flag. Challenges 4.38 and 4.39 involve exploiting a buffer overflow vulnerability in a tcp server program written in C. Module 5 involves exploiting various vulnerabilities found in systems involving HTTP protocol, HTML, JavaScript and server side code. Challenge 5.07 involves a sql injection vunlnerability and writing a brute force attack to guess the flag. In challenge 5.13 and 5.14, CSRF or cross site request forgery type vulnerabilities are exploited to obtain the contents of the flag file.

II. Description of Solution

A. Challenge 4.38: Little Dipper

The program for litter dipper starts a tcp server on a port supplied by through command line arguments. The program create a tcp socket with the supplied port and listens for connection on the socket. Whenever a client connects to the tcp server, the server forks a child process to handle the client connection. This is a common paradigm in network programming. The child process on the server redirects the standard input and output streams to the network socket. The child process also leaks the current stack pointer, which will be used later to build an exploit. The process then calls the function $manage\_tcp\_client$ to handle the communication with the client. The objective of the function is to read user input on a tcp connection into a buffer. The following figure is a snippet from the challenge program showing parts of the manage function with the chunk size.

Fig. 1: Code: manage_tcp_client

```
#define CHUNK_SIZE 65536
...
int manage_tcp_client()
{
  char buffer[CHUNK_SIZE];
  int ret = 0;

  printf("Ready to read!\n");
  fflush(stdout);

  mindex = 0;
  while ((ret = read(0, &buffer[mindex], 1)) >
       0) {
    mindex++;
#ifdef DEBUG
    fprintf(stderr, "mindex is %d\n", mindex);
#endif
    if (buffer[mindex - 1] == 0x0a) {
      buffer[mindex - 1] = '\0';
      break;
    }
  }
  ...
}
```

As seen in Fig. 1, function manage_tcp_client defines a char array called buffer of length 65536. Since its a char array, the buffer is of size 65536 bytes. The while loop in the code tries to read from the standard input one byte at a time into the buffer. Since the standard input is actually the network socket, the program is trying to read from the network socket into the buffer a byte at a time. It breaks out of the loop only when it encounters that the current byte is 0x0a. 0x0a is ASCII code for \n or the newline character. So the function won't stop reading into the buffer until it encounters the new line character. This is a clear case of a buffer overflow vulnerability. A malicious attacker can send a payload greater than 65536 bytes in size and overflow into the stack. This allows attacker to overwrite the contents of the stack. To exploit this vulnerability, buffer overflow is exploited to overwrite the stack. The stack contains the save rip and saved rbp addresses. RIP address is basically the return address or the address to which the function should return after it has executed. This allows the attacker to change the control flow of the program.

However, it isn't easy to move forward with only changing the control flow of the program, since there isn't any other function in the server code which can be used to get access to a shell. So, we use another method called shellcode injection. In this technique the attacker tries to insert bytecode for spawning a shell into the vulnerable program's memory. The attacker can then use other exploits to refer to this memory location and trick the program to run the bytecode for spawning a shell. There are various bytecode available on the internet which can be used for this exploit. Since, the attacker has access the the char array buffer,the bytecode can be injected into this buffer over a tcp connection. However, care is to be taken that the bytecode doesn't contain a 0x0a byte, which will terminate the loop and won't read from the connection further. After injecting the bytecode, the attacker can overwrite the stack by overflowing the buffer and modifying the saved rip address to point the the char array named buffer. Before we start construction the solution, some investigation is necessary to calculated the distance between the leaked rsp which we obtain from the server and the buffer. Also the distance between buffer and saved rip, and distance between saved rip and rsp will also be required. This can be easily accomplished using gdb. Gdb can be used to follow child process by doing *set follow-fork-mode child*.

To construct a solution, pwntools library is used. It is a utility library in python programming language with many useful functions to perform such exploits. The exploit program written in python first connects to a remote target using a tcp connection to the server port. Since the server leaks the rsp, the script listens for this information and calculates the address of buffer and saved rip. The script then creates a payload, which consists of no op sleds to fill the buffer and overflow it, bytecode for the shell, some padding and the address of the buffer. The payload is constructed such that when sent over the connection, it causes the buffer to overflow. It puts the bytecode with noop sled into the buffer and overwrites the saved rip with the address of the buffer. This results into program executing the bytecode from the buffer when it tries to return to the caller from manage_tcp_client.

Fig. 2: Code: payload construction

```
shellcode = b'\x48\xbb\x2f\x2f\x62\x69\x6e\
    x2f\x73\x68\x48\xc1\xeb\x08\x53\x48\x89\
    xe7\x48\x31\xd2\x52\x57\x48\x89\xe6\x48\
    x31\xc0\xb0\x3b\x0f\x05\x48\x31\xc0\xb0\
    x3c\x48\x31\xff\x0f\x05'

size_of_buffer = 65536
nop = b'\x90'

payload = nop * (size_of_buffer - len(
    shellcode)) + shellcode + b'a'*
    padding_size + p64(address_of_buffer)
```

Fig. 2 shows how the payload is constructed. The payload is filled with no op instructions. This instruction as the name suggest is a no operation instruction, it has no side effect. The instruction pointer then moves to the next instruction. This instruction can be used to fill up the buffer with a valid instruction. Next the shellcode is added. This shellcode executes */bin/sh*, spawning up a shell. Since the saved rip is not adjoining the buffer on the stack, some garbage bytes are added as padding. Next the address of the buffer, which was calculated using the leaked rsp is added as a byte string in little Indian format using p64 function from pwntools library. This completes the payload, which can be sent to the server over the tcp connection to get a shell with super user privileges.

B. Challenge 4.39: Big Dipper

The code for big dipper is very similar to that of challenge 4.38, little dipper. The function manage_tcp_client remains the same. The only thing that is changed is the size of the char array buffer. In Big Dipper, the buffer is only 16 bytes in size. In previous challenge, the buffer was used to store the bytecode for spawning the shell and was then referenced by overwriting the saved rip address by performing a buffer overflow. In this challenge, it is not possible to fit the bytecode for spawning shell into the buffer. To overcome this we can inject the buffer through an environment variable. Since we have control over the environment of the server, we can export the bytecode along with a huge no op sled in an environment variable. This works because child processes inherits the memory structure of the parent process. Fork works by creating another process pointing to the same memory pages of the parent process. Some of these pages would be modified depending on the program. However, it inherits the environment of the parent process. So the environment variable containing the shellcode is also available in the memory of the forked child process. Since, the process leaks the rsp when it gets forked, gdb can be used to calculate the distance between the environment variable containing the shellcode and the leaked rsp address. After this, a payload can be constructed, filling the buffer with garbage values and overwriting the stack where rip is

saved with the address of the environment variable. Similar to challenge 4.38, after the program returns from the function manage_tcp_client, the bytecode from the environment variable is executed.

Fig. 3: Code: Environment variables in memory

```
pwndbg> x/30s *((char **)environ)
...
0x7ffd3fdaaaa3: "HOSTNAME=practice~hijacking-
    binary-power~level-4-39-big-dipper"
...
0x7ffd3fdab193: "SHELLCODE=", '\220' <repeats
     190 times>...
0x7ffd3fdab25b: '\220' <repeats 200 times>...
...
0x7ffd3fdafeeb: '\220' <repeats 200 times>...
0x7ffd3fdaffb3: "
    \220\220\220\220\220\220\220\220\220H
    \273//bin/shH\301\353\bSH\211\347H1\322
    RWH\211\346H1\300\260;\017\005H1\300\260<
    H1\377\017\005"
```

The previous figure shows how to examine the memory of the server using gdb to find the addresses of the environment variables. According to the figure the SHELLCODE is located at 0x7ffd3fdab193, since the address of leaked rsp is known, It is possible to guess the location of SHELLCODE if the distance between leaked rsp and the SHELLCODE environment variable is known. This involves some trial and error since the distances can change during actual execution. To overcome this problem, a large no operation sled is added to SHELLCODE, so a number larger than the distance between SHELLCODE and leaked rsp can be used. This will land the saved instruction pointer on a no op operation, which will eventually lead to execution of the shellcode at the end.

C. Challenge 5.07: Leak in My Data

This challenge consists of a flask web server. The web server creates a users table and inserts an user with username flag and password as the secret flag value. Fig. 4 shows the code for the web server. The server only has a root path, which allows for 'GET' and 'POST' http methods. Is the user logged in or not is checked by checking if the session cookie is set. If the user is logged in, the server returns Hello username back. A post request can be made to the server with username and password as payload to log in. The server takes this username, password and runs a sql query on the database to verify if the credentials are valid. However, the server doesn't sanitize the user input data. This leaves an opportunity to insert malicious data in the payload to manipulate the sql query. This is known as sql injection.

Using this knowledge, it is possible to get logged in. However, this does not get the attacker the flag. Looking at the code, it is not possible to get the flag directly. To get the flag, a special sql query can be constructed which checks if the password contains a character at a position. It is possible to run this query for all positions of the password string and using all possible printable characters. If it matches the character at the position, the request logs the user in and returns Hello, username.

Fig. 4: Code: 5.07 web server code

```
def level7():
   # create user table and insert flag user,
       password into it

   if request.method == "POST":
      # get username and password

      user = db.execute(f'SELECT rowid, *
          FROM users WHERE username = "{
          username}" AND password = "{
          password}"').fetchone()
      assert user, "Invalid `username` or `
          password`"

      session["user"] = int(user["rowid"])
      return redirect(request.path)

   if session.get("user"):
      user_id = int(session.get("user", -1))
      user = db.execute("SELECT * FROM users
          WHERE rowid = ?", (user_id,)).
          fetchone()
      if user:
         username = user["username"]
         return f"Hello, {username}!\n"
   return form(["username", "password"])
```

The following figure shows how this query is constructed. The username field is substituted with malicious input. The value is set such that the expected query is executed. The query compares the sub string of password of length 1 starting at position $i+1$ with letter j. If the letter j matches with the $i+1^{th}$ letter of password, the web server logs in and returns 'Hello, flag'.

Fig. 5: SQL injection with username field

```
#expected query:
# SELECT rowid, * FROM users WHERE username =
     "flag" AND substr(password, i+1, 1)=j

username = f'flag" AND substr(password, {i
    +1}, 1) =\'{j}\' --'
```

Using this query, we can write a python to script to perform a brute force attack and build the value of the flag by incrementally checking each position for all possible characters.

D. Challenge 5.13: Underground Data Escape

In this challenge, the webserver consists of multiple paths for different purposes. It consists of a '/login' path which allows the user to login. After logging in with the correct credentials, the webserver set a session cookie, which is used to authenticate further requests. A request to the '/info' path gives back information about the username given the user id. If the user is logged in however, the info path also gives back the password of the user along with the username. The '/visit'

path starts up a local browser session and logs in using the credentials of the flag user. It then visits the url supplied by the user. The url should have the same hostname as the hostname of this webserver i.e 'challenge.localhost' otherwise it returns 'Invalid url' back. Then there is '/echo' path. A request on this path converts the echo argument to html and sends it back. The server sets SESSION_COOKIE_HTTPONLY option to False in the application config. This option allows browser JavaScript access to session cookies. In this challenge, it is possible to perform a XSS or cross site scripting attack by inject a script as the echo parameter. The JavaScript which the attacker can send through the echo parameter can access the document cookies and send it as a payload to a different server hosted by the attacker. After obtaining the user session cookie, the attacker can authenticate further requests to the web server. The attacker can then call the /info path with the cookie to get the flag. The echo request can be made by sending it through visit path. The visit path logs into the web service and hence gets the session cookie for subsequent requests. However, the payload should be URL encoded so that it is parsed correctly by the web server. Since the JavaScript sent as the echo parameter can also contain URLs, these URLs should be double encoded, so that it unwinds correctly after reaching the /echo path.

Fig. 6: Code: web server code

```
def level13():
   app.config['SESSION_COOKIE_HTTPONLY'] =
       False
   if request.path == "/info":
      # retrieve session id and request user
          id

      info = [request_user["username"]]
      if request_user_id == session_user_id:
         info.append(request_user["password"
             ])
      return " ".join(info) + "\n"

   if request.path == "/visit":
      ...
      assert url_arg_parsed.hostname ==
          challenge_host, f"Invalid `url`,
          hostname should be `{challenge_host
          }`"
      ....
      with run_browser() as browser:
         ...
         do login
         ...
         WebDriverWait(browser, 10).until(EC.
             staleness_of(submit_field))
         browser.get(url)
         time.sleep(1)

      return "Visited\n"

   if request.path == "/echo":
      echo = request.args.get("echo")
      assert echo, "Missing `echo` argument"
      return html(echo)

   return "Not Found\n", 404
```

E. Challenge 5.14: Underground Data Escape II

The code for the web server for this challenge is the same as the code for 5.13 with the change that now, the SESSION_COOKIE_HTTPONLY option is set to be True. Due to this it is no longer possible to access the session cookies by injecting a script through /echo. To overcome this, it is possible to run another server with the same host name as the challenge server but a different port. Since urllib.urlparse returns the same hostname for both 'http://challenge.localhost/' and 'http://challenge.localhost:8080/', it is possible to make visit go to the url on a differnt server. Since the browser automatically puts the session cookie on this requests, the attacker can just log the session cookie and use it to authenticate malicious request. After obtaining the cookie, a simple request with the session cookie to the web service on the /info path will reveal the flag to the attacker. This exploit also works on challenge 5.13.

III. Results

The script for solving challenge 4.38 and 4.39 was written in python using pwntools library. The script connects to the remote target over a tcp connection. The script listens for the marker 'Current stack pointer' and receives the leaked rsp. The script then calculates the buffer address, the saved rip address using the distances which were obtained by inspecting the execution of the program using gdb. The script then constructs the payload and sends it over the tcp connection. The buffer overflow causes the saved rip to be overwritten with the address of the buffer. This opens up a shell with the standard input and output redirected over the tcp connection. I was able to obtain the flag using this shell.

The script for challenge 4.38 can be modified to be used in challenge 4.39. The buffer however this time is not big enough to hold the bytecode for the shell. So I added a big no op sled to the shellcode and exported it to a an environment variable. The location of the environment variable can be found using gdb and the distamce between the leaked rsp and environment variable was calculated. Since it is not a exact difference, I added a number larger than the distance so the program execution lands on the no op sled. This way I was able to obtain a shell and the flag.

For challenge 5.07, I wrote a script which brute forced over the possible positions and possible characters of the flag using the solution described earlier. Using the script, I was able to obtain the flag.

For challenge 5.13 underground escape, I wrote a python web server which prints the contents of the requests it receives. I also wrote some JavaScript code which was encoded in the echo parameter passed through the /visit path. The JavaScript used the cookie from the document as payload and made a post requests to the python web server. The server the logged the payload. This way I obtained the session cookie. After this I used this cookie to make a request to /info path and obtained the flag.

For challenge 5.14 underground escape, I wrote a python web server which printed the headers of the requests it received and hosted it using the same name 'challenge.localhost' but on a different port. After this I just made a request to /visit path with the URL to my web server. The server logged the request and it's header. This way I obtained the cookie and was able to obtain the flag by sending the cookie along with the request to /info path.

IV. My Contributions

Since this was a solo project, all of the contributions were individual efforts by me. I prepared all the scripts and found all the vulnerabilities. In doing so, I took help of course slides and class lectures. The following list describes all of the tasks which I undertook to complete this project:

1) Learn about buffer overflow and stack overflow vulnerabilities to modify the saved rip.
2) Identify the vulnerabilities in challenges 4.38/4.39.
3) Explore the code using gdb and calculate the approximate distance between the leaked rsp and location of the buffer.
4) Learn how to use pwntools library to prepare shell injection exploits.
5) Prepare the shellcode for the injection attack.
6) Prepare a python script which performs the exploits by preparing the payload and communicating with the target.
7) Export the shellcode to environment variable and find it's location in the memory of the process.
8) Modify script used in 4.38 for 4.39
9) For 5.07 identify the vulnerability.
10) Device a way to obtain the flag.
11) Prepare a script which automates the brute force attack and returns the flag.
12) Identify the vulnerabilities in 5.13 and 5.14.
13) Write python code to host attacker's server.
14) Prepare a script which leaks the document cookie to the attacker's web server

V. Lessons Learned

Working on the challenges has made me aware of the subtle vulnerabilities which can be found in the code developers write. It is essential to prevent such attacks as these can compromise user data, there by reducing the trust on computer systems among people. Being aware of such vulnerabilities has made me conscious of the mistakes that can be made during development of software systems. I also learned how to exploit these vulnerabilities to get access to privileged data. Working through the challenges, I have honed my terminal skills and am well versed with bash syntax now. I have learned various tools like GDB and have gotten aware of features such as follow-fork-mode. I have improved my understanding of assembly code and how programs are structured in memory. Working through challenges 4.38 and 4.39, I learned about techniques like shellcode injection, injecting code through environment variables. I learned how to use requests library in python to make web requests.

The challenges from module 5 improved my understanding of web protocols. I learned how to use curl to make web requests from terminal. I learned how to write web servers in using Flask in python. I learned how to write basic JavaScript to make web requests. I learned how cookies are used to identify users, and how some options to safeguard cookies work. I learned about cookie hijacking to make malicious requests. I learned how to inject JavaScript code with these requests using course slides and various tutorials online [2]. I learned about SQL and SQL injection techniques.

I also learned about various prevention techniques which can be used to prevent such attacks. Studying these vulnerabilities, I also learned about ASLR [3] which is used to prevent attackers from guessing the memory location of data in a program. Although ASLR can be broken through some clever exploits [4] , it still remains a useful deterrent from preventing simple attacks I also learned about how stack guards and stack canaries [5] [6] can be used to prevent against stack overflow attacks. Both the challenges from module 4 and module 5 taught me the importance of sanitising user input. All of these attacks arise from trusting user input blindly. It is necessary to use html safe functions while handling user input from the web. Finally, I have also improved my researching skills using these challenges by looking through documentation and searching for various techniques to get through the challenges.

References

[1] https://pwn.college/
[2] J. Kallin, I. Valbuena, https://excess-xss.com/#xss-attacks
[3] P. Team, "PaX Address Space Layout Randomization (ASLR)," Documentation for the PaX Project, 2003. Available at https://pax.grsecurity.net/docs/aslr.txt.
[4]] T. Durden, "Bypassing PaX ASLR Protection," Phrack, vol. 59, no. 9, p. 9, 2002.
[5] https://ctf101.org/binary-exploitation/stack-canaries/
[6] https://www.sans.org/blog/stack-canaries-gingerly-sidestepping-the-cage/