Security Engineering
Coursework Submission

Instructions:

This is a template file for providing explanations for your solutions of the coursework challenges.

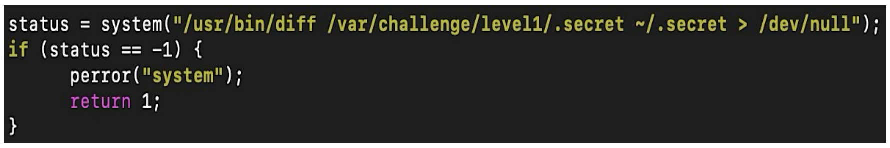
1. First, please provide your details in the fields above (name, email, ID, VM username).
2. Second, for each solved challenge, explain the identified vulnerability as well as your exploitation method. For more information about what to include in your explanation, please refer to the example template given below. Please provide your answers for each challenge on the corresponding page, starting with Challenge 1 on page 2.
3. Finally, after typing your answers, please save this file as PDF (Explanations.pdf) and include it in the submitted archive along with your exploit files (as explained in the Coursework Guide).

Example template for your answer

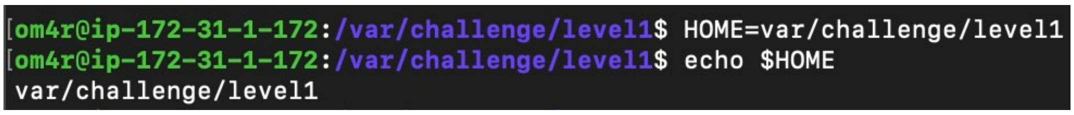
Challenge 1

1. Explain the Vulnerability

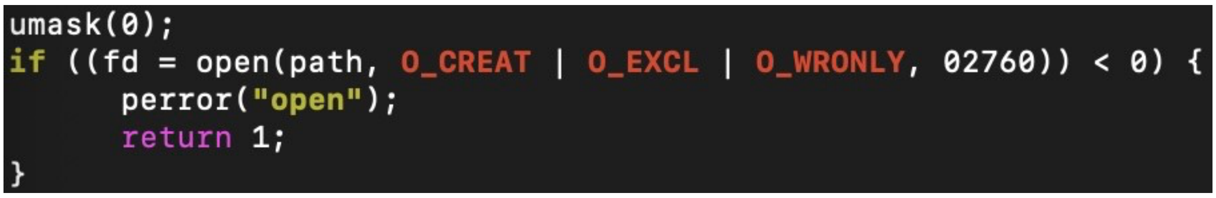
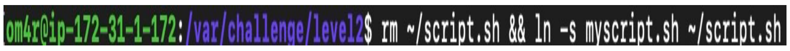
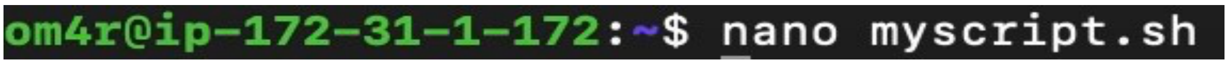
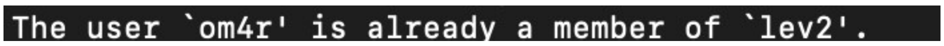
I exploited a vulnerability in the program by using an environmental attack. The program's source code relies on the HOME environment variable to find the .secret file in the user's home directory. Since users can modify the home directory, the program can be misled into thinking it's in the user's home directory when it's actually elsewhere.



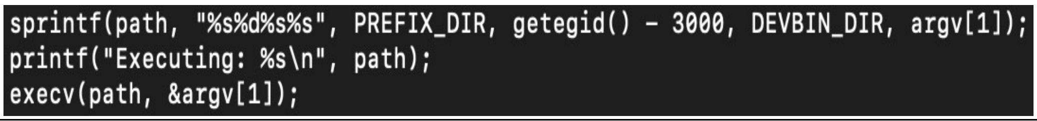
```
status = system("/usr/bin/diff /var/challenge/level1/.secret ~/.secret > /dev/null");
if (status == -1) {
      perror("system");
      return 1;
}
```

2. Explain Your Exploit	For this challenge, I exploited the program's reliance on the HOME environment variable to bypass its security measures. I modified the HOME variable to point to /var/challenge/level1, effectively redirecting the program to check the .secret file in the same directory instead of comparing files across different directories. This manipulation caused the program to compare the .secret file in /var/challenge/level1 with itself, bypassing the intended password verification process. By exploiting the program's dependency on HOME for determining file locations, I tricked it into performing an unnecessary self-check, allowing me to bypass the verification entirely and proceed without the need for a valid password.  `om4r@ip-172-31-1-172:/var/challenge/level1$ HOME=var/challenge/level1` `om4r@ip-172-31-1-172:/var/challenge/level1$ echo $HOME` `var/challenge/level1`

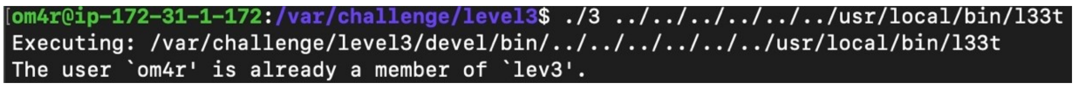
Challenge 2

1. Explain the Vulnerability	In Challenge 2, the program has two security flaws a Symbolic Link Attack and a TOCTOU vulnerability. The program's source code creates and writes to a file named script.sh without checking if it's a regular file or a symbolic link. This allows an attacker to replace script.sh with a symbolic link to a file of their choice. The program also uses umask(0), which creates files with highly permissive access rights. This means script.sh can be easily manipulated. Since the program doesn't restrict access to the file, an attacker can replace or modify it before execution, increasing the risk of a security breach. figure 1.  `umask(0);` `if ((fd = open(path, O_CREAT | O_EXCL | O_WRONLY, 02760)) < 0) {` `perror("open");` `return 1;` `}`
2. Explain Your Exploit	For this challenge, I exploited a TOCTOU (Time of Check to Time of Use) vulnerability caused by the program's five-second delay before executing the target file. During this delay, I had a window to replace script.sh with a symbolic link pointing to my malicious script. To execute this, I created a shell script named level2.sh that contained the l33t command to escalate privileges. In another terminal, I prepared the commands rm ~/script.sh && ln -s level2.sh ~/script.sh. When I ran the vulnerable program (2.c), the five-second delay allowed me to quickly execute the rm and ln -s commands, replacing script.sh with my malicious level2.sh. As a result, the program executed my script instead of the original, successfully elevating my privileges to Level 2. This exploit worked perfectly because the program failed to ensure the integrity of the file during the delay period.  `om4r@ip-172-31-1-172:/var/challenge/level2$ rm ~/script.sh && ln -s myscript.sh ~/script.sh`  `om4r@ip-172-31-1-172:~$ nano myscript.sh`  ``The user `om4r' is already a member of `lev2'.``

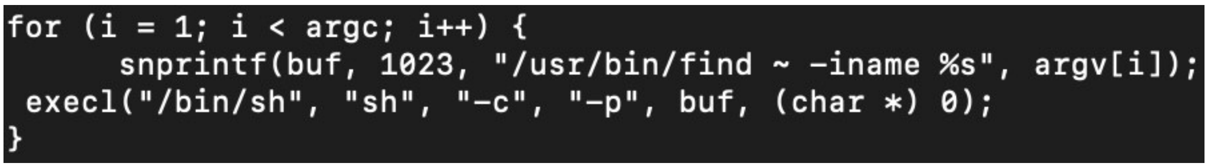
Challenge 3

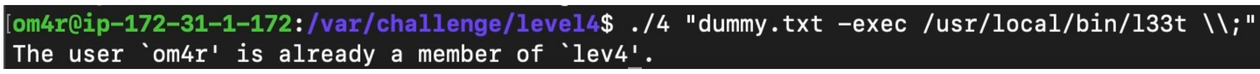
1. Explain the Vulnerability	In Challenge 3, I identified a vulnerability caused by improper sanitization of the path variable. I analyzed the code and noticed that the sprintf function combined user input from argv[1] with predefined directory paths such as PREFIX_DIR and DEVBIN_DIR without adequate validation. Although the program checked for specific characters, it failed to account for relative path sequences like ../, which allowed me to escape the intended directory structure. By crafting a malicious input for argv[1], I was able to traverse directories and access files outside the expected paths. This issue occurred because the program did not strictly validate user input when constructing critical directory paths, leaving it vulnerable to directory traversal attacks during the execv call. 

```
sprintf(path, "%s%d%s%s", PREFIX_DIR, getegid() - 3000, DEVBIN_DIR, argv[1]);
printf("Executing: %s\n", path);
execv(path, &argv[1]);
```

2. Explain Your Exploit	For this challenge, I exploited a path traversal vulnerability by taking advantage of the program's inadequate handling of directory paths. I crafted my input as ../../../../../../usr/local/bin/l33t, which allowed me to traverse up the directory structure repeatedly until I reached the root directory. From there, I directed the path to /usr/local/bin/l33t. This bypassed the intended restrictions that limited execution to specific directories like PREFIX_DIR and DEVBIN_DIR. When the program processed my input and executed the command, it ended up running the l33t binary from the unintended location. This worked because the program didn't sanitize or validate the input path properly, enabling me to trick it into executing something outside its restricted scope.  `om4r@ip-172-31-1-172:/var/challenge/level3$ ./3 ../../../../../../usr/local/bin/l33t` `Executing: /var/challenge/level3/devel/bin/../../../../../../usr/local/bin/l33t` ``The user `om4r' is already a member of `lev3'.``

Challenge 4

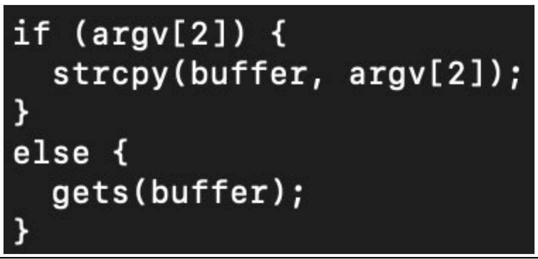
1. Explain the Vulnerability	In Challenge 4, I identified a vulnerability in how the program constructs a command string buffer (buf) using user input from argv[i]. After analyzing the code, I noticed that the program executed this string using execl with sh -c, interpreting it as a shell command. While the program attempted to block certain characters in the input, I found that this protection was not sufficient. Since the entire buf string was executed within a shell environment, I was able to craft input in argv[i] to inject unintended commands. By exploiting this, I influenced the execution flow and successfully demonstrated a command injection attack.  `for (i = 1; i < argc; i++) {`
	`      snprintf(buf, 1023, "/usr/bin/find ~ -iname %s", argv[i]);`
	` execl("/bin/sh", "sh", "-c", "-p", buf, (char *) 0);`
	`}`

2. Explain Your Exploit	In Challenge 4, I exploited a command injection vulnerability in the program. The program constructed a shell command using snprintf(buf, 1023, "/usr/bin/find ~ iname %s", argv[i]); and executed it with execl("/bin/sh", "sh", "-c", "-p", buf, (char *) 0);. I analyzed the code and noticed that it directly inserted user input from argv[i] into the command without proper validation. To exploit this, I provided input like ./4 "dummy.txt -exec /usr/local/bin/l33t \;", crafting a payload where dummy.txt was a placeholder and -exec /usr/local/bin/l33t \; appended a command to execute the l33t program. The command executed regardless of whether dummy.txt existed, as the exec flag forced execution. By leveraging this lack of input sanitization, I was able to inject arbitrary commands and exploit the vulnerability successfully.  `[om4r@ip-172-31-1-172:/var/challenge/level4$ ./4 "dummy.txt -exec /usr/local/bin/l33t \\;"` `The user `om4r' is already a member of `lev4'.`

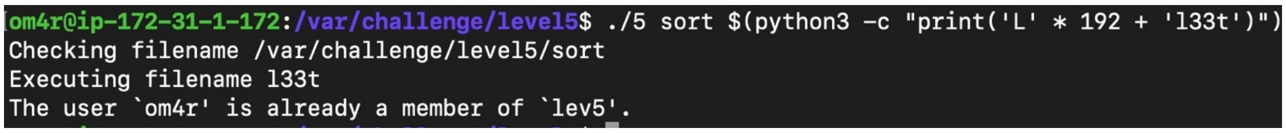
Challenge 5

1. Explain the Vulnerability

In Challenge 5, I identified a vulnerability buffer overflow caused by the unsafe use of the gets function, which lacked bounds checking. The program used a buffer with a fixed size of 192 bytes, but gets allowed me to input data exceeding this size. By crafting input that overflowed the buffer, I was able to overwrite adjacent memory locations, including the return address on the stack. This gave me control over the program’s execution flow, allowing me to inject and execute malicious code or cause the program to crash. Using this approach, I successfully demonstrated a buffer overflow exploit.

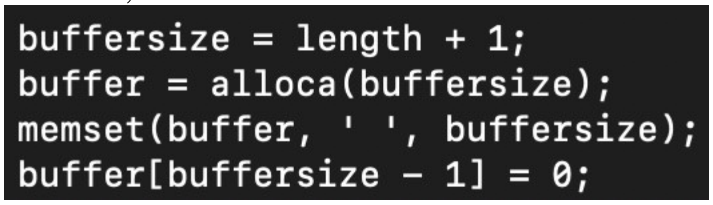


```
if (argv[2]) {
  strcpy(buffer, argv[2]);
}
else {
  gets(buffer);
}
```

2. Explain Your Exploit	For Challenge 5, I exploited a buffer overflow vulnerability caused by the program's use of the gets(buffer) function, which reads user input without any bounds checking. This meant I could input more data than the allocated buffer size of 192 bytes, allowing me to overwrite adjacent memory, including the filename variable. To exploit this, I crafted input consisting of 192 A characters, followed by the path to the l33t binary. This overflowed the buffer and replaced the value of filename with my desired path. As a result, when the program attempted to execute the original file, it instead executed the l33t binary. The lack of input size restrictions and proper validation of critical variables made this exploit possible. This challenge showed me just how dangerous unbounded functions like gets can be when proper safeguards aren't in place. 

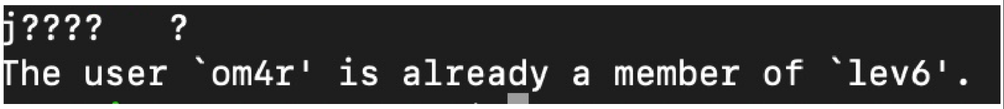
```
om4r@ip-172-31-1-172:/var/challenge/level5$ ./5 sort $(python3 -c "print('L' * 192 + 'l33t')")
Checking filename /var/challenge/level5/sort
Executing filename l33t
The user `om4r' is already a member of `lev5'.
```

Challenge 6

1. Explain the Vulnerability	In challenge 6, the code has memory allocation and bounds checking vulnerabilities which can lead to buffer overflow. The use of alloca allocates a stack-based buffer with a user-provided length. An attacker can provide a large length, potentially causing a stack overflow if it exceeds available space. Additionally, adding 1 to buffersize can cause an integer overflow, leading to an unexpectedly small allocation. This miscalculation may allow an out-of-bounds write when accessing buffer[index]. While index > length is checked, it doesn't fully protect against all possible out-ofbounds writes, especially when considering integer overflow bypasses. These vulnerabilities expose the program to stack overflow and memory corruption risks. The number 65535 is often linked to buffer overflows because it's the biggest unsigned 16-bit integer. This number is super important in computing since many programs and systems use 16-bit integers to handle buffer sizes or data limits. If a size goes over 65535, it can cause an integer overflow, 

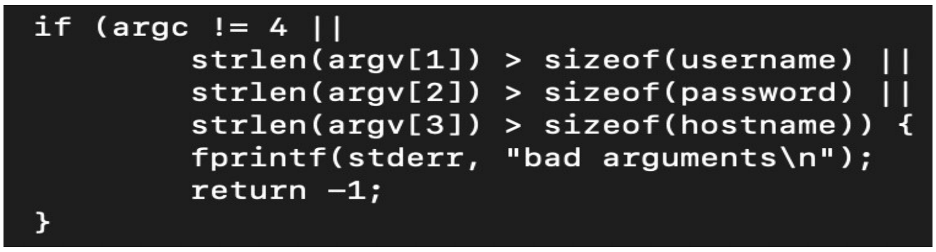
```
buffersize = length + 1;
buffer = alloca(buffersize);
memset(buffer, ' ', buffersize);
buffer[buffersize - 1] = 0;
```

2. Explain Your Exploit	In Challenge 6, I utilized gdb to analyze and exploit vulnerabilities in the program. I started by passing the values A0, A1, A2, and A3 using the starti command and set breakpoints at key locations. By inspecting the main frame, I identified the saved return address and used the layout asm function to trace the execution flow to the final printf() statement in the main function. This process allowed me to examine the stack and confirm that my input AAAA was represented as 0x41414141. From this, I determined that the offset between the buffer and the return address was 28 bytes. To exploit the vulnerabilities, I leveraged a stack overflow and an integer overflow in the program's memory allocation and boundschecking mechanisms. I utilized a shellcode provided by Mohamed Abouhashem via an email to execute the l33t command and this will be used in all of my other exploits by storing it in an environment variable to bypass input size restrictions. Shellcode is used to exploit vulnerabilities, bypass normal program flow, and achieve tasks like privilege escalation or delivering malicious payloads.The memory address for the shellcode was calculated using the formula which again was provided by Mohamed addr = 0xC0000000 - strlen(prog_path) - strlen(shellcode), which resolved to \xB0 28, \xFF 29, \xFF 30, \xBF 31. This address was passed to the program in little-endian format along with the calculated offset of 28 bytes. By passing a large size, such as 65535, as an argument, I triggered excessive allocation using alloca, causing a stack overflow. Additionally, I exploited an integer overflow that occurred when adding one to the buffer size, resulting in a smaller-than-expected allocation. This allowed me to perform out-of-bounds writes, bypassing the bounds check. Through precise positioning of the

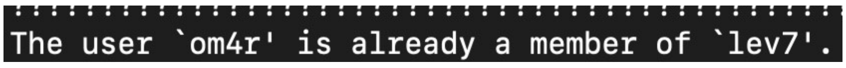
	shellcode and careful manipulation of arguments, I successfully overwrote the return address. This enabled my payload to execute, granting access to the next level. All the exploits I ran from now till level 10 were created in my home directory and I used execve to target the /var/challenge/level/etc. 

```
j????   ?
The user `om4r' is already a member of `lev6'.
```

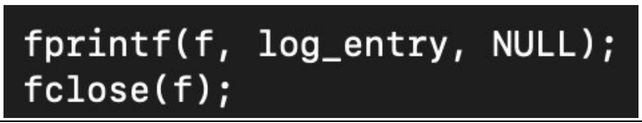
Challenge 7

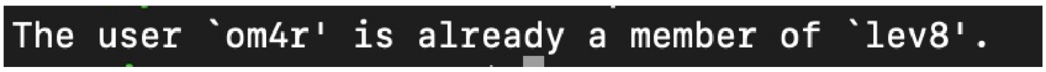
1. Explain the Vulnerability	In Challenge 7, I discovered a stack-based buffer overflow vulnerability caused by improper handling of null-terminated strings and unsafe string operations. The program uses strcpy to copy the user inputs argv[1], argv[2], and argv[3] into the username, password, and hostname buffers. Although the program limits the length of each input to 64 characters using strlen(argv[i]) > sizeof(buffer), it fails to account for the null terminator when concatenating these inputs into the result buffer with strcat. I found that if an input is exactly 64 characters long, the null terminator does not fit into the buffer and is overwritten by the next input. When strcat is called, it cannot distinguish the boundaries between the buffers and continues reading from memory until encountering a null byte. This allowed me to write
	data far beyond the allocated 256 bytes for the result buffer, causing a stack-based buffer overflow. 

```
if (argc != 4 ||
        strlen(argv[1]) > sizeof(username) ||
        strlen(argv[2]) > sizeof(password) ||
        strlen(argv[3]) > sizeof(hostname)) {
        fprintf(stderr, "bad arguments\n");
        return -1;
}
```

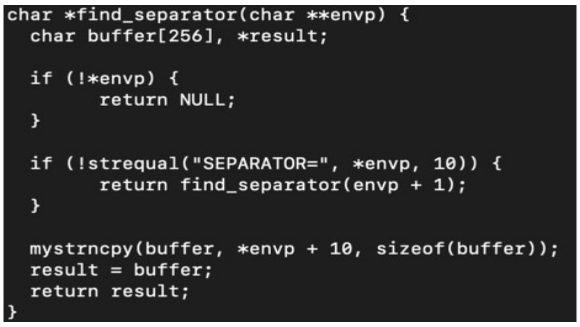
2. Explain Your Exploit	In Challenge 7, I used gdb to craft and execute my exploit, following a similar approach to Challenge 6. I began by sending a payload and setting breakpoints at critical locations to analyze the program's behavior. Using the main frame, I located the EIP and then traced the execution flow to the final printf() statement in the main function using the layout asm command. This allowed me to view my payload in memory as 0x41414141. Once I had this information, I crafted a payload that included a NOP sled, the shellcode provided by Mohamed Abouhashem to execute the l33t command, and an overwritten return address to redirect execution to the shellcode. Using memory inspection with the x/350x $esp command, I identified the exact return address to overwrite, which was \x40\xff\xff\xbf and I target the same memory 3 times so I can land my shellcode in the right location. After fine-tuning the payload for alignment, I set the return address to point to the NOP sled, ensuring reliable execution of the shellcode. The most teadius process was finding the correct location that would execute my shellcode but the guidelines in the cw helped a lot.  `The user `om4r' is already a member of `lev7'.`

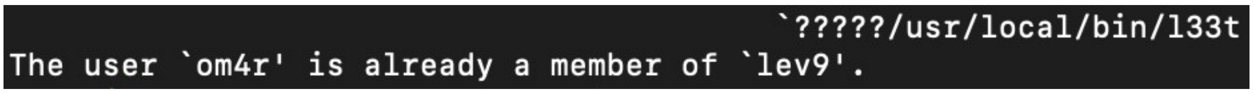
Challenge 8

1. Explain the Vulnerability	In Challenge 8, I identified a format string vulnerability in the sudoexec function's logging functionality. The program directly passed log_entry as the format string to fprintf and snprintf without validating or sanitizing it. Since log_entry included user-controlled input, such as the command, I was able to insert malicious format specifiers like %x and %n to exploit the vulnerability. This allowed me to read arbitrary memory locations and write controlled values to specific addresses, giving me a path to manipulate the program's execution.
	
	Using the guidelines provided in CW Challenge 8, I analyzed the binary with objdump to locate the Global Offset Table entries. I identified the GOT entry for fopen at 0x08040a10, as this function was called soon after the vulnerable fprintf statement. To exploit this, I crafted a payload containing format specifiers that overwrote the GOT entry for fopen with the address of my shellcode which I found by crafting a c code shellcoe_loader that first loads the shellcode in the envi and then another c code shellcode_finder that shows the address of my shellcode which was \xe9\xfc\xff\xbf and I will be using this mostly. This redirected the execution flow to my shellcode when the program attempted to call fopen.
	
	
	`fprintf(f, log_entry, NULL);`
	`fclose(f);`

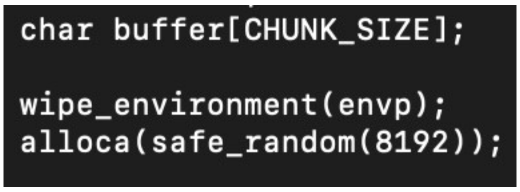
2. Explain Your Exploit	In Level 8, I exploited a format string vulnerability in the program's use of fprintf, which processed user-controlled input as the format string. Following the guidelines provided in Challenge 8, I began by analyzing the binary with objdump (objdump -R /var/challenge/level8/8) to locate the Global Offset Table entry for fopen, which I found at 0x08040a10. This address became my primary target for the exploit. I was able to find this out though the guidance of the cw guideline level 8. I crafted a payload designed to overwrite the GOT entry for fopen with the address of my shellcode, stored in an environment variable. To achieve this, I used four pointers pointing to successive bytes of the fopen GOT entry: \x10\xa0\x04\x08, \x11\xa0\x04\x08, \x12\xa0\x04\x08, and \x13\xa0\x04\x08. I determined these addresses corresponded to the 68th, 69th, 70th, and 71st positions on the stack. Using the %n and %hhn format specifiers, I was able to write specific values to these addresses one byte at a time. Futher I had to use Padding of "AA" so the exploit can fit In the correct address with it it will not be at the correct loction. To calculate the values, I accounted for alignment and the stack setup, which included 24 bytes printed before the format string was processed. For the least significant byte, I padded the output using %166x and wrote the value to the address at the 71st position using %71$hhn. For the second byte, I used %64x for padding and wrote the value to the address at the 70th position with %70$hhn. For the third byte, I added padding with %254x and wrote the value to the address at the 69th position using %69$hhn. Finally, for the most significant byte, I padded the output with %10x and wrote the value to the address at the 68th position using %68$n. This calculated approach ensured precise writes to each byte of the target address. After getting everything ready I use the the c code shellcode_loader and then run the exploit 8 for it to work. As using my normal shellcode was not working I kept on getting permission errors.  `The user `om4r' is already a member of `lev8'.`

Challenge 9

1. Explain the Vulnerability	In Level 9, the vunlnerability of the program is buffer overflow which I discovered by finding a flaw in the find_separator function, which returns a pointer to a local stack-allocated buffer. The buffer became invalid once the function returned, therefore any subsequent use resulted in crashes and unpredictable behavior. After reviewing the source code, I discovered that the application utilized the mystrncpy method to transfer the SEPARATOR environment variable into the buffer without sufficient bounds checking. When the SEPARATOR surpassed 256 bytes, it overwrote nearby memory, including the stack's return address. 
	```
	char *find_separator(char **envp) {
	  char buffer[256], *result;

	  if (!*envp) {
	        return NULL;
	  }

	  if (!strequal("SEPARATOR=", *envp, 10)) {
	        return find_separator(envp + 1);
	  }

	  mystrncpy(buffer, *envp + 10, sizeof(buffer));
	  result = buffer;
	  return result;
	}
	```

2. Explain Your Exploit	I exploited the vulnerability in the find_separator function of Level 9, where a pointer to a local stack-allocated variable is returned, leading to undefined behavior. First, I created a LEET environment variable containing a large NOP sled followed by shellcode to execute /usr/local/bin/l33t. Then, I crafted a SEPARATOR environment variable filled with repeated memory addresses \xe9\xfc\xff\xbf which is a memory address of my shellcode that point to the desired location in memory. To ensure proper alignment, I added a filler string of 384 A characters as an argument. When the program executed and processed the environment variables, it dereferenced the invalid pointer from find_separator, which I had set to point to my crafted memory. This redirected the program's execution to the shellcode in LEET, giving me control and successfully triggering the exploit. 
	```
	                                          `?????/usr/local/bin/l33t
	The user `om4r' is already a member of `lev9'.
	```

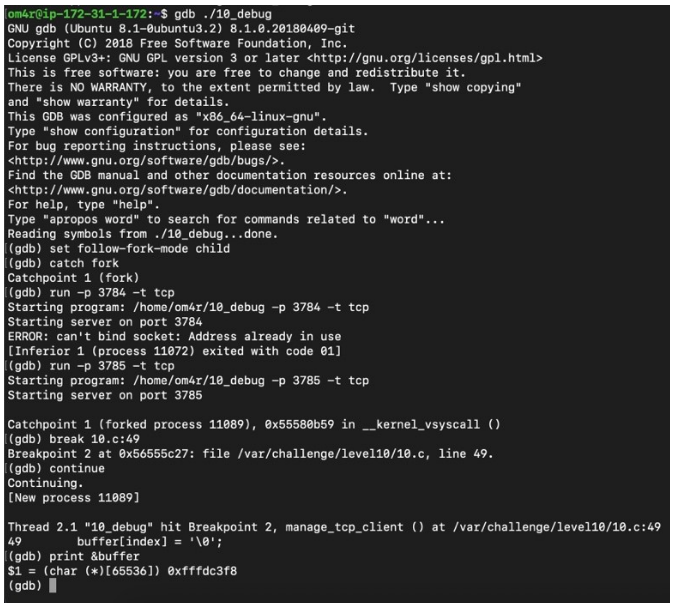
Challenge 10

1. Explain the Vulnerability	In level 10 the vulnerability is buffer overflow this can be found in the code lies in the unbounded use of alloca(safe_random(8192)), which allocates a random-sized buffer on the stack without any safeguards, combined with the allocation of large fixed-size stack buffers like buffer[CHUNK_SIZE] where CHUNK_SIZE is 65536 bytes. This combination significantly increases the risk of stack exhaustion, as alloca does not enforce stack size limits, allowing excessive memory usage that can corrupt the stack and lead to crashes or undefined behavior. While this is not a classic buffer overflow, the impact is similarly severe, as it involves unbounded memory usage on the stack. This vulnerability is particularly dangerous in scenarios involving high workloads or recursive operations, where repeated allocations can quickly overwhelm system resources, making it a critical flaw in the program's memory management.  `char buffer[CHUNK_SIZE];` `wipe_environment(envp);` `alloca(safe_random(8192));`

2. Explain Your Exploit

I discovered two ways to exploit this program. The first method was unexpected and based on luck. While crafting C code to inspect the memory stack, I opened and retained ownership of port 2222. At some point, another student in the coursework sent their payload through the same port. Since I was the owner of port 2222, the program prioritized my session, and I was credited with progressing to the next level instead of them. For my own exploit, I followed the guidelines provided by Mohamed. Using gdb, I set follow-fork-mode child to track the program's execution in the child process. This allowed me to identify the top address of the stack which was 0xfffdc3f8, enabling me to craft a precise payload to exploit the program effectively.

Shown below:



```
om4r@ip-172-31-1-172:~$ gdb ./10_debug
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./10_debug...done.
(gdb) set follow-fork-mode child
(gdb) catch fork
Catchpoint 1 (fork)
(gdb) run -p 3784 -t tcp
Starting program: /home/om4r/10_debug -p 3784 -t tcp
Starting server on port 3784
ERROR: can't bind socket: Address already in use
[Inferior 1 (process 11072) exited with code 01]
(gdb) run -p 3785 -t tcp
Starting program: /home/om4r/10_debug -p 3785 -t tcp
Starting server on port 3785

Catchpoint 1 (forked process 11089), 0x55580b59 in __kernel_vsyscall ()
(gdb) break 10.c:49
Breakpoint 2 at 0x56555c27: file /var/challenge/level10/10.c, line 49.
(gdb) continue
Continuing.
[New process 11089]

Thread 2.1 "10_debug" hit Breakpoint 2, manage_tcp_client () at /var/challenge/level10/10.c:49
49          buffer[index] = '\0';
(gdb) print &buffer
$1 = (char (*)[65536]) 0xfffdc3f8
(gdb) 
```

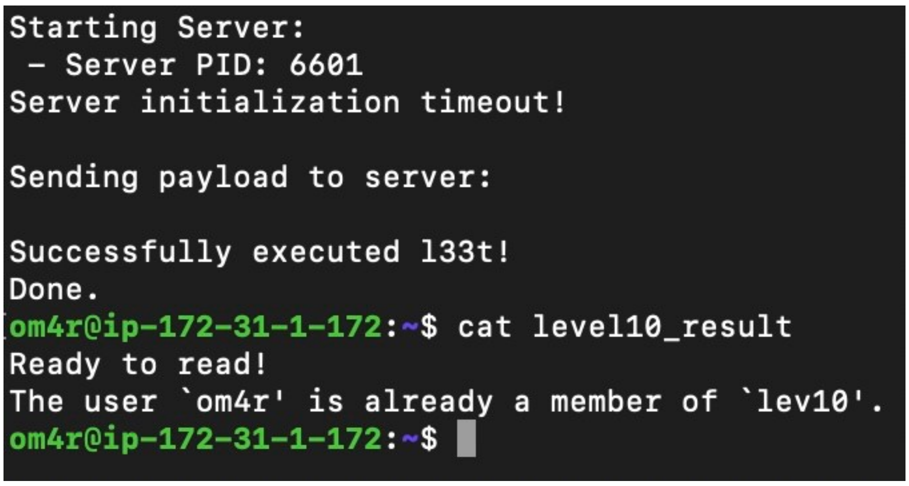
After working with gdb, I wrote a Python script to exploit level10's buffer overflow vulnerability. The Python scripts start a server that runs on port 3784, which is unique to me. I started by creating an output file named level10_result to hold temporary data during the process, which can be used to store the exploit results. I then developed a function, run_exploit, to manage the full exploitation sequence because it was too time-consuming to restart the application manually each time, so I decided to write a script that could do it manually.

First, I started the server using the command /var/challenge/level10/10 -p 3784, and I watched its initialization by lool nd running, I ran
in the level10_result file. Once the server was
a payload, which I then delivered to the serv
server response showed success, such as the w

"lev10" in the output, I terminated the loop. If
I

created logic to destroy the server process and restart the operation, allowing the script to attempt exploitation indefinitely until it succeeded or I manually ended it.

In level10.py, I wrote a program that generated the exploit's payload. I constructed shellcode to run /usr/local/bin/l33t and saved it in the variable payload_shellcode. I then created a contrived return address, \xe9\xfc\xff\xbf, that directed execution to the shellcode. To ensure dependability, I put a NOP sled (\x90) before the shellcode, establishing a buffer to absorb minor alignment issues. Finally, I repeated the return address several times and inserted a line break to complete the payload. I merged all of these components into payload_vector and printed it so that exploit10.py could use it during the exploitation process.


```
Starting Server:
 - Server PID: 6601
Server initialization timeout!

Sending payload to server:

Successfully executed l33t!
Done.
om4r@ip-172-31-1-172:~$ cat level10_result
Ready to read!
The user `om4r' is already a member of `lev10'.
om4r@ip-172-31-1-172:~$ 
```