Jeffry Jimenez
CSE 331.01
Professor Rahmati
Fall 2020

# Report

## Task1:

## h4ndY_d4ndY_sh311c0d3_55c521fe

Task1 only uses the libraries pwn and re. The flag is in the same directory as the vuln executable, however since I am not part of the required group, I cannot access it. To gain access to the flag, my script simply runs the vuln executable. This executable uses the dangerous function gets(), which allows me to send as input a string of any size. Using the sendline() function from pwntools, I sent shellcode  after the input string was requested. This gave me access to the groupid that the executable vuln had and with sendlineafter() I cat the flag. Lastly, I used re to parse the content of flag.txt.

## Task2:

## 3asY_P3a5yb197d4e2

The file vuln.c in task 2 uses a signal handler to print the flag anytime a SIGSEGV is activated. The file vuln.c also requires at least one cmd line argument, which it copies using the dangerous function strcpy() into a buffer of size 128. To get the flag my script simply runs the vuln executable with a cmd line argument of 200 A's which causes an overflow (SIGSEG), the script then proceeds to use re library and parse the output.

## Task3:

## n0w_w3r3_ChaNg1ng_r3tURn5a32b9368

Task3 only uses the libraries pwn and re. the script takes advantage of the fact that vuln.c uses gets() to fetch the user input. This allows me to overflow the buffer which gets() writes to, and by extension edit the memory around the buffer. The buffer size is 64 bytes. Knowing this by trial an error I just kept sending string larger than 64 bytes until I started to edit the return address of the vuln function (this took a string of size 76). The function flag() in vuln.c prints the flag into the console. Now all I had to do was to use the cmd < readelf -s vuln> or < objdump -D vuln> to find the address of the flag method (0x080485e6) and set it to the return of the vuln method. The script for task3 starts the vuln executable using the process() function from pwn, and then waits for input to be requested and sends a string of 76*A concatenated with the address of the flag() function in little endian (\xe6\x85\x04\x08) using the sendline() function in pwn. Finally using re library, the script parses the flag printed by the vuln executable.

## Task4:

## th4t_w4snt_t00_d1ff3r3nt_r1ghT?_d0b837aa

Task4 is very similar to task3, however there's one caveat: in this task we cant just jump directly to the flag() function because we will encounter a memory alignment issues. In order to fix this, we need to jump to a return instruction which according to an answer in https://stackoverflow.com/questions/18311242/retq-instruction-where-does-it-return, retq "moves the %rsp pointer to previous address on stack" (8 bytes) which is just what we need and therefore fixing our memory aligned issue. The first thing I did was to use the cmd < readelf -s vuln> or < objdump -D vuln> to find the address of the flag() function (0x00400767). The vuln.c file also uses the gets() which we can exploit, I found that it takes 72 'A' to start writing to the return address of the vuln() function. The script simply signs into the server, then proceeds to

change the working directory to the one where task 4 is. We then use the run function from pwn to pipe into the vuln executable a string of A*72 concatenated with the address of a return instruction, in our case I chose (0x0000000000400851) and the address of the flag() function (0x0000000000400767) . Lastly, using the recvall() from pwn we get the retuned values and use re to fetch the string inside the flag.

## Task5:

## sl1pp3ry_sh311c0d3_0fb0e7da

Task 5 is almost identical to task 1, however in this case the buffer can be located at an offset of anywhere from 0 to 256 bytes. To account for this, we can use the NOP 0x90 which has no effect. Sending 256 NOP 0x90 before the shellcode assures that no matter what the offset of the buffer is, it will eventually reach the shellcode. Just like task1, the flag is in a flag.txt in the same directory as the vuln executable. This executable uses the dangerous function gets(), which allows me to send as input any string of any size between 0-512. When the input string was requested, using the sendline() function from the pwn library, I sent a string of 256* '\x90' concatenated with a string of shellcode, which spawned a shell. This gave me access to the groupid that the executable vuln had and with sendlineafter() again I cat the flag, from the spawned shell. Lastly, I used re to parse the content of flag.txt.

## Task6:

## str1nG_CH3353_166b95b4

Task6 is very similar to task3. The script for task6 only uses the libraries pwn and re. the script takes advantage of the fact that vuln.c uses gets() to fetch the user input. This allows me to overflow the buffer which gets() writes to and by extension edit the memory around the buffer.

The buffer size is 176 bytes. Knowing this by trial and error I just kept sending string larger than 176 until I started to edit the address which the vuln function returns to (this took a string of size 188). The function flag() in vuln.c prints the flag into the console. Now all I had to do was to use the cmd < readelf -s vuln> or < objdump -D vuln> to find the address of the flag method (0x080485e6) and set it to the return of the vuln() method, what differentiates task6 with task3 is that the flag() function is task6 requires two parameter. According to this post https://security.stackexchange.com/questions/172053/how-to-pass-parameters-with-a-buffer-overflow , the parameter for the function we are returning to, should be located 4 bytes after EIP. The script for task6 starts the vuln executable using the process() function from pwn, and then waits for input to be requested and then, using the sendlineafter() function from pwn, the script sends a string of 188*A concatenated with the return address of the flag function in little endian (\xe6\x85\x04\x08) concatenate with 4*A and the two parameters for the flag() function with are, (\xEF\xBE\xAD\xDE) and (\x0D\xD0\xDE\xC0). Finally, the script receives the flag using recvall() form pwn, and then, using re library, the script parses the flag printed by vuln.

## Task7:

## cAnAr135_mU5t_b3_r4nd0m!_bf34cd22

Task7 is very much like task6 and task4 in a sense that it involve modifying the return address of the vuln() function in vuln.c. However, is this case there is a canary which alerts the program that I tried to edit smash the stack. The canary is located right after the end of the buffer which can contain 32 bytes. Because the canary is always the same as the content in canary.txt and vuln.c tells us that the length of the canary is KEY_LEN=4, we can try to brute force it. In the script the findcanary() function, implements a brute force approach to determine the canary.

For (i) starting at 1 in the length of the canary + 1 at each iteration, the function tries to find the correct character at the position (i-1) in the canary, if the right canary char in the range of (! to ~) => (33 to 126) in ASCII, is found at a position then the vuln executable returns a string that contains the word ‘ok’. If such string is returned, then the character is added to the global string canary. Once we’ve found all 4 chars of the global canary then we are ready to fetch the flag. all I had to do was to use the cmd < readelf -s vuln> or < objdump -D vuln> to find the address of the flag() method (\xed\x07\x00\x00) in little endian. We then start the vuln executable using the process() function from pwn, and wait for input to be requested and then, using the sendline() function from pwn we send a string of 32*A concatenated with the canary we found, 16*A (because the return address is not immediately after the canary, there is some trash in between, in this case the return address is 16 bytes after the canary) and the address of the flag function in little endian (\xe6\x85\x04\x08). Finally, the script receives the flag using recvall() form pwn, and then, using re library, the script parses the flag printed by vuln. Because the address of the flag method is position dependent the mentioned instructions run inside an infinite while loop that only stops once the until the flag is found.

## Task8:

## str1nG_CH3353_166b95b4

In task8 there only two important details to notice, one is that the flag is somewhere in the stack and two we can input whatever format string we want to the printf call in the printMessafe3() function. The task8 script uses the pwn and re libraries. Task8 is simple because I took a brute force approach. The powerhouse of the script is in the infinite while loop. Starting at i==1 every time an iteration happens we start a the vuln executable, and when input is prompted, we send the format string %{i}$s which prints the parameter at index (i) in the stack

as a string pointer.  If what we receive from the server does not contain the word pico (the flag) we increment (i) by one, close the process and loop again. If the flag is found we simply break out of the loop and use re to extract the message from the flag, lastly, we close that process.