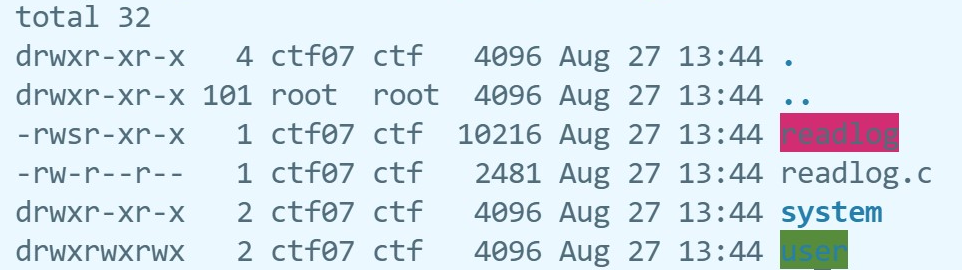
**Assignment 1 report**

**Question 1**

In this challenge, we first check what files and directories are there and their permissions (see below). there is a SUID binary readlog owned by ctf07, the system directory is readable and executable to everyone but it is only writable to ctf07, besides, the user directory readable, executable and writable to everyone. We find that only the ctf07 has permission on the target file system.log and it is encrypted.

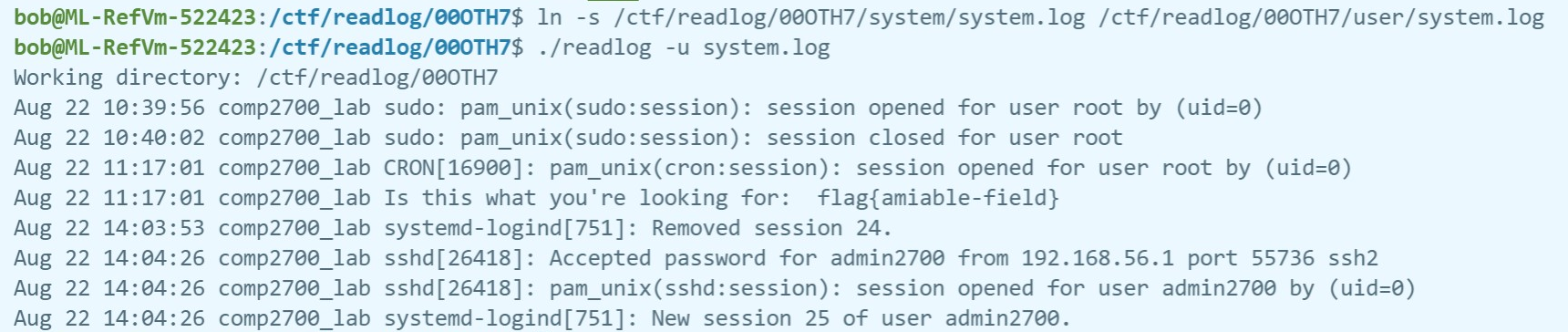


In the program, it requires us to type a file name and a specific mode. it first check whether the argv is satisfied the specific structure, and it will copy the user input to the filename buffer, which is performed correctly. Next, it will go to the userlog function or systemlog function according to the option the user given. The userlog function can open any files in the user directory without any authentication while systemlog function require a password that is hard to find to open files in system directory. I noticed that this program filters the non- alphanumeric characters by using the check function, so the Directory Traversal attack is impossible. We need to find out if there is a way you can make it read the files I want it to read.

* Vulnerability: All the users have read, write and execute permissions on the user directory, and the program reads file in this directory. So, we can create any files in this directory, which includes the file created by soft links. We cannot create the hard link because we do not have write permissions on system directory.

To test this hypothesis, I first creates some files on another directory, and using the soft link to create a symbolic-link files in the user directory, next using the “./readlog -u myfiles.txt” to read it. Its success means that we can use the soft link to read the files not in the user directory.

So, the attacking strategy is first create a symbolic link of the ./system/system.log in the ./user directory and name it system.log, next, run the binary using the “./readlog -u system.log” as argv, then this SUID program will read the system log (process shown below). The reasons why it works is that soft link contains a reference to another file in the form of an absolute path and that affects pathname resolution. So, when we use the soft link, the program will redirect to the system directory even we are in the user directory.



**Question 2: crypt\_flag**

For this challenge, there is an SUID binary crypt\_flag that requires a password to decrypt the flag.enc file and it will show the content of the encrypted file to the reader if the password is correct, otherwise it will print the Wrong password to the bash. There is another file decrypt.sh, which permission is all the users can read and execute it but only the owner “ctf01” can modify it. The last file is the flag.enc that contains the flag which is readable to all the user, but it is useless because it is encrypted. The permission detail is shown below in graph 2.1.



graph 2.1

In the main function, it first loads the environment variables MYKEY and MYIV to some specific values, and then it requires the user to type the password by using the standard input, and password will be stored in the char pass[32], it is impossible to implement the buffer overflow because the fgets only takes 31 characters from the user input. Besides, there's no way to do shell code injection because of the sanitise(pass) function. The password is then hashed and compared with the prestored hashed value, if the password is correct, it will run the shell script decrypt.sh which will use the correct environment variable to decrypt the password. Otherwise, it will run the Linux command to print the wrong password.

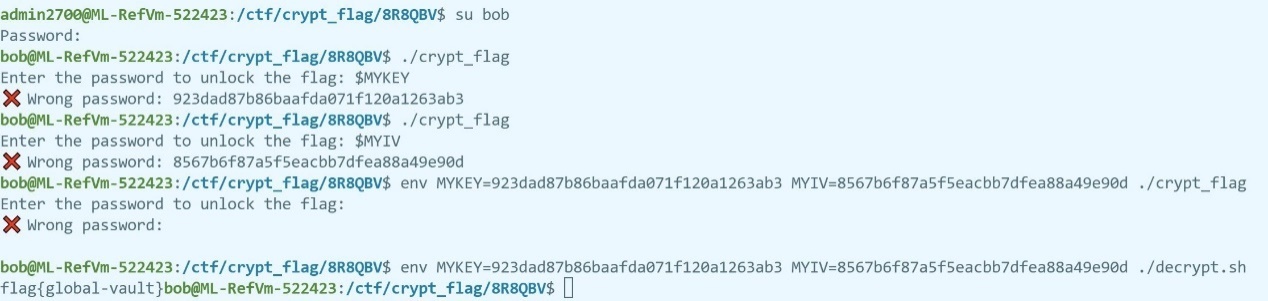
* Vulnerability: the following shell command can be exploited to output the environment variable message by typing the password as $x. (in line 79-80)

sprintf(cmd, "/bin/echo \"\u274c Wrong password: %s\"", pass);

system(cmd);

Let's test to see if we can exploit Vulnerability above. Checking the sanitise function, it seems that it only deletes the ';', '&', '|', '>', '<' to prevent the code injection but it does not prevent the ‘$’ character, this is a very serious vulnerability, especially for some programs using the environment variables like this program. In order to test this hypothesis, I enter the password like “$USER”. And I get the user name of current user which means this exploitation is applicable.

The attack strategy is first running the crypt\_flag and using the password as ‘$MYKEY’ to gain the key information from the output value then run it again but using different password “$MYIV” to gain the IV information. Lastly, running the decrypt.sh shell script with the $MYIV and $MYKEY values gained in the last step, which will return the flag (process shown below). The reason why this works is that we can access the value of an environment variable by prefixing it with a ‘$’ sign, and it allows the user to combine the user input with the command, on the other hand, the permission setting of the decrypt.sh is also an important reason since it allows anyone to execute it.



graph 2.2

**Question 3 num\_sort**

In this challenge, it only has a binary file num\_sort. Let look at the main function, this program implements some basic functions for the integer array like adding some elements, delete the array, sort the array and show the array. This specific integer array claims that it only can store 14 elements, and it also has an index pointer “max” which points to the last element’s index of the array. For the function of adding new elements to the array, its mechanism is first check whether the current number of elements in the array exceeds the size of buffer (14), if so, it will go back to the main function and print a buffer is full message, otherwise, it will allow the user recurrently adding new elements at most 7 times. In general, as long as the user passes the buffer size test, it can add at most 7 elements to the array. For the function of reset, it just only sets the index pointer “max” to 0. For the rest of the functions, there is nothing special. At the end of the main function, there is a test to check whether “a == GUARD”, a is initialized with 0 and GUARD is defined as 2984, if the test is passed, the flag will be displayed but obviously, this is impossible.

* Vulnerability: there is a potential buffer overflow in the following code. (line 84-85)

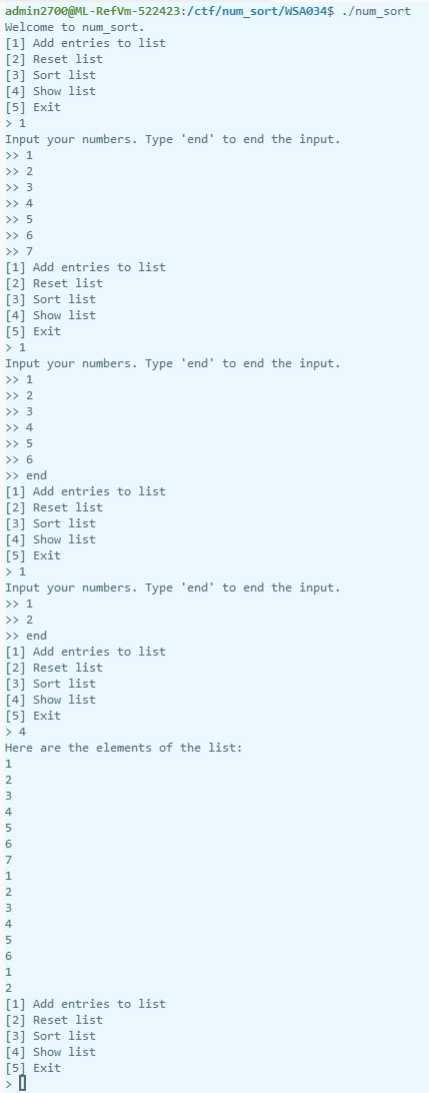
**int** a = 0;

**int** numbers[MAX\_ELEMENTS];

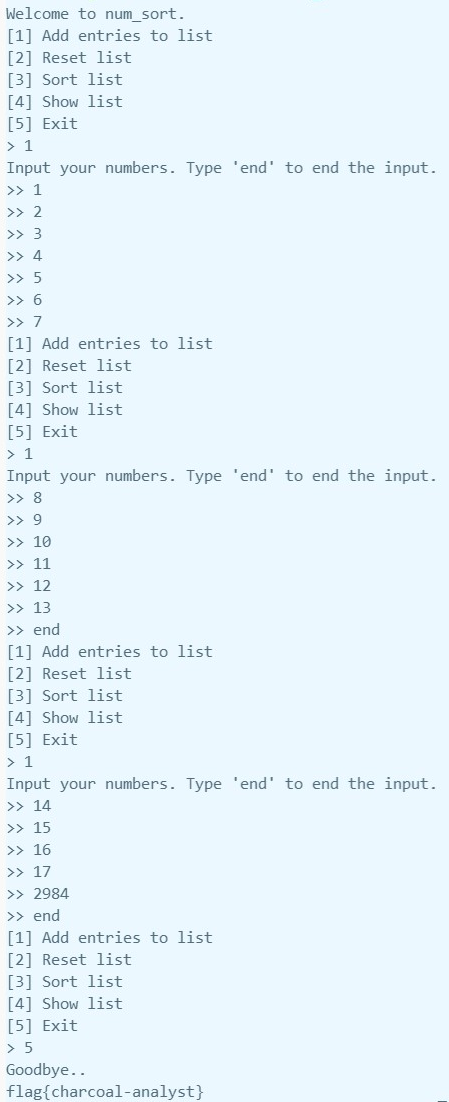
the variable “numbers” is allocated 14 bytes on the stack, but the user can potentially input 15 characters or more to the “numbers”. Assuming the stack layout for these variables respect the ordering given in the source code, it seems that an overflowed numbers buffer can potentially overwrite the value of a. so, if we can overflow the number buffer to write “a” to the GUARD value, we can pass the check at line 118 (f (a == GUARD)) and obtain the flag.

On the other hand, the mechanism of function of adding elements also has vulnerability. Although it has a protection of preventing user adding elements when the buffer is full, this protection check is not performed for each process of adding one element to the list. it only performs at the start of the function add\_entries. So, image that if we have an array that has 10 elements now, we can easily pass the buffer size test as its size is less than 14, and we can add at most 7 elements to the list without restriction, therefore, the size of the buffer will become at most 17, and the overflow occurs.

To test this hypothesis, I first reset the index pointer, and add 7 elements to the array, then in the next round, I add 6 elements to the array to control the buffer’s size less than 14, at last, I add 2 elements to the array. The process is shown below. Using the function of show array, we can see all the elements from numbers[0] to numbers[max], if the hypothesis is correct, it will show exactly 15 elements. And it did.



The attack strategy is first adding 7 elements to the array, and then adding another 6 elements, and in the final round we only need to add 2 elements, the second one should be 2984. We type “5” to quit and wait for testing. However, it still cannot pass the GUARD test, and I think this is the reason mentioned in the “Assignment 1 – Frequently Asked Questions”, which is there may be 'gaps' in the stack between two consecutively declared variables. Therefore, I test for some sizes and finally adjust the array’s size to 18. The steps are shown below in graph3.1. The reason why it works is because when we add the numbers to the array, and especially adding numbers outside the buffer will overwrite the buffer above (in this case a), so if we set the 18th value to 2984, we will overwrite the a and match the GUARD value.



graph 3.1

**Question 4 check\_passwd**

In this program, it first dynamically allocates a memory address for the msg pointer, and then it requires the user to type the password that will be checked whether it is the correct password “Q6VWQSXFZB”. If authentication is passed, it will display the content of the msg pointer. When running the program, it will display the flag memory address. So, the main idea of this exploitation will be using the buffer overflow to overwrite the msg pointer with the flag address, and we type the correct password, we can read the content of the flag address.

* Vulnerability: it allows the user to type 20-character input while it copies the content of the input to a 14-size of buffer. Potential buffer overflow. (line 37-38)

fgets(buf,20, stdin);

strcpy(pass, buf);

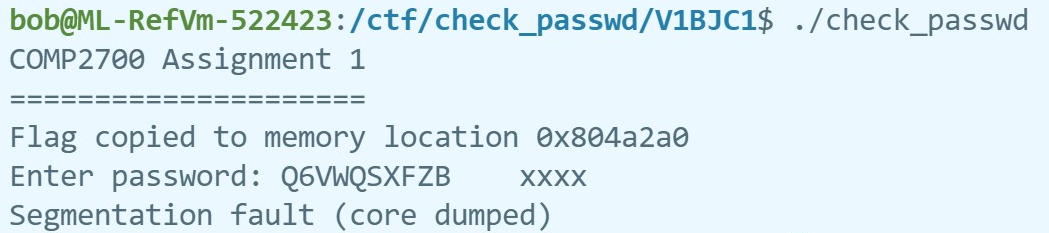
the buf is a 20-size buffer, and the pass is a 14-size buffer, so using the strpy, it can at most overwrite 6 characters. As the pass is the answer and the variables in the main function are declared in the following order:

**char**\* msg;

**char** answer[14];

And the user’s input is sanitized without any whitespace, and manually setting the index-13 to \0 that means we need to put the memory address after index-13. Therefore, if we using the “Q6VWQSXFZB xxxxxx” as the password, it will first cut the password according to the whitespace and then an overflowed answer buffer can potentially overwrite the value of msg with xxxxxx.

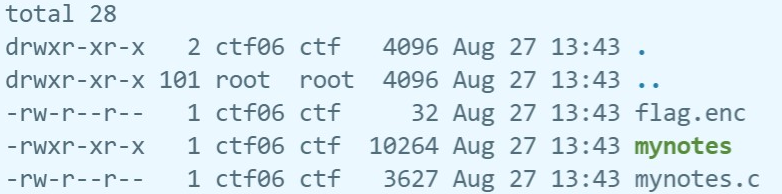
To prove this hypothesis, I run a simple test with user’s input as “Q6VWQSXFZB 123456”. If the hypothesis is correct, it will not return the "Sorry, try again\n" message. The test result (below) shows that hypothesis is correct.



The attack strategy is using the “echo -e "Q6VWQSXFZB \xa0\xa2\x04\x08" | ./check\_passwd” as the password, to first pass the password authentication and then overwrite the msg buffer with the flag’s memory address. Another thing should be noticed is that the input is a string, which means that if we typing the flag’s address directly, it will be converted to ASCII first, and the converted “new address” will be a different one. So, we have to first convert the address to some specific characters according to the ASCII table. And the strategy is using “echo -e” to print some unprintable characters and pipe the output to the program. The reason why it works is that this program performs a strcpy which source buffer’s size is 20 bytes while the destination buffer’s size is 14 bytes, so we have extra 6 bytes to overwrite the value of msg. and we change it to a different memory address which contains the flag value.

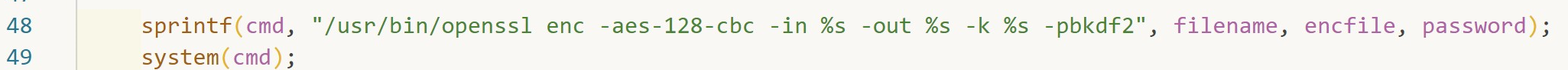
**Question 5**

When I check the permissions of this directory, I find that the mynotes binary is not SUID program, which means we do not have permission to create a new file in this directory during the running of the binary.



In the source code, this program using a user input while loop control flow to control the behaviors of the program, where user can switch notes, add notes, read notes, encrypt and decrypt notes. In the switch notes branch, I notice that the filename provided by user is sanitized to prevent code injection and all the buffer related operations are performed properly. In the add note branch, it first gets the note from the user, and using Linux command to write it to the filename, noticed that it does not sanitize the filename variables, instead it sanitizes the input variables, this is a potential vulnerability. After reading the source code of the encrypt note branch, I find that it using the shell commend to execute the encryption and the password provided by user is combined with it, if user does not provide any password, it will use the default password. For the decrypt branch, it first requires the user to type the name of the file to be decrypted, this is quite suspicious since it does not use the filename from the branch “switch note” and the input is not sanitized, next, we need to give a password to decrypt the file, and finally, using the shell command to decrypt the file. All the buffer’s usage is correct, so buffer overflow attack will not be available.

* Vulnerability: from the code analysis above, I notice that there is a potential shell commend injection attack in the function decrypt\_notes, encrypt\_notes.



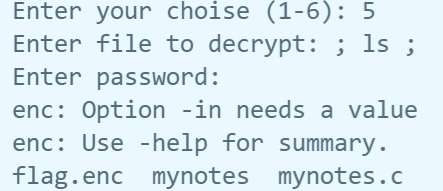
Encrypt\_notes



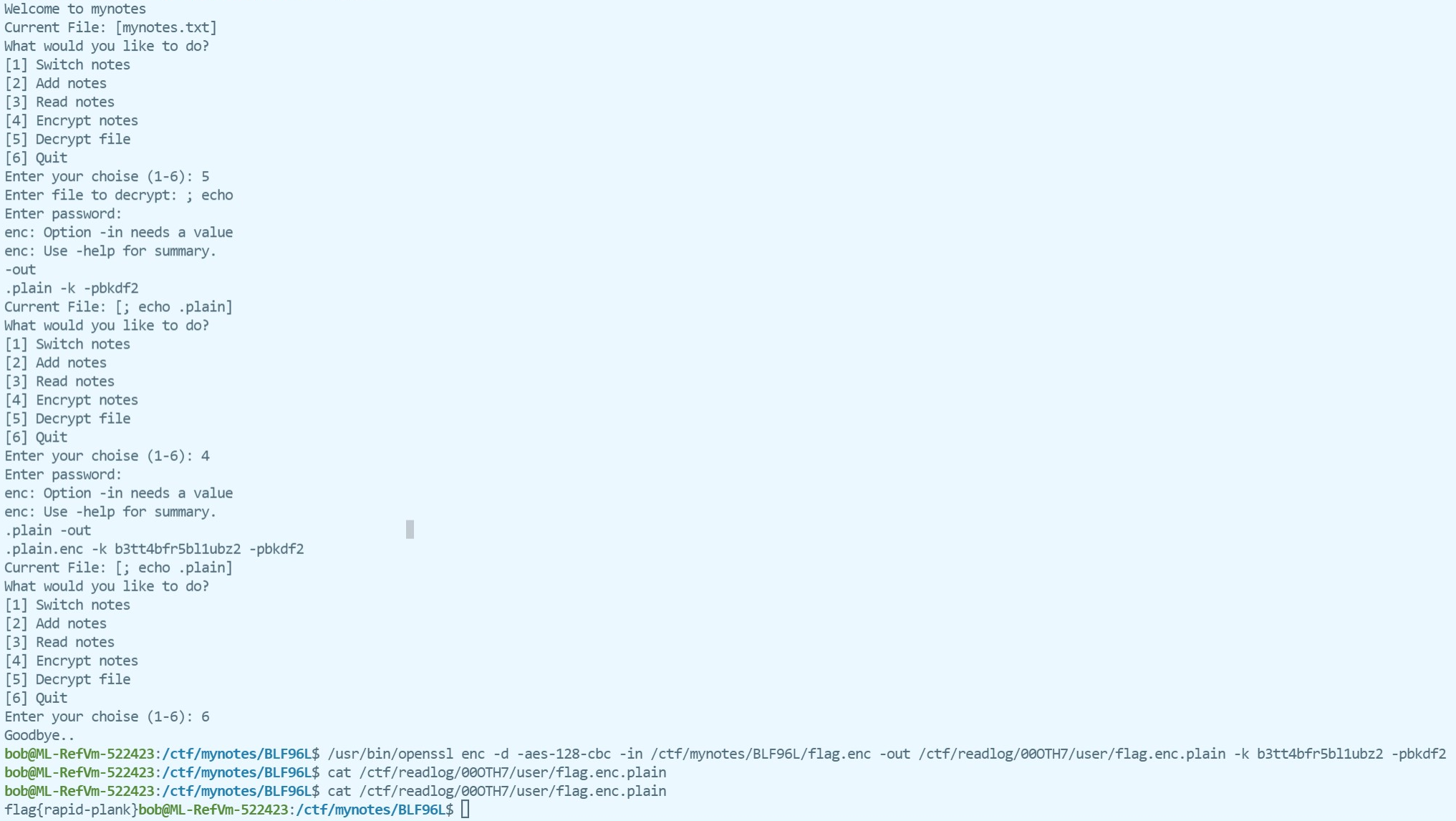
decrypt\_notes

In the decrypt\_note function, it first asks the user to type a filename to be decrypted (line 71-73), and overwrite the filename created in branch Switch notes with the new filename in the Decrypt file branch without any sanitization (line 79-80), which means this program has another way to switch to different filenames and it can also include some shell commands. In the encrypt\_note, it uses the filename in the shell command (line 48), so it can execute the injected shell command that we define it in the decrypt\_note. As a result, if we well defined the injected shell command, we can get the default password in encrypt\_note. Besides, another potential shell command injection attack is in the Add notes branch, but there is nothing to do with getting the default password.

To test the test the hypothesis, we can change the filename to “; ls ;” to check whether it will print the files and directories (test shown below). As we can see, it prints the files and directories, so this attacking technique is feasible.



The attack strategy is first change the file name to the shell command “; echo ” in the decrypt\_note function and then switch to the encrypt notes branch, using empty password to print the default password on to the screen. At last, we directly using the openssl command with another output file path to decrypt the flag.enc. This is because we do not have write permission on the current running directory and the mynotes binary is not a SUID program, so we have to put the decrypted plain files in a directory that we have sufficient permissions (process shown below). The reason why it works is that this program provides two ways of changing the filename, and the second one can contain the shell command injections. On the other hand, the program combines the shell command string with user input and does not check on it. And the “;” shell command allows to perform sequential execution, which means we can execute our own shell command in this program which leads to the leakage of the password.



**Question 6 even\_odd**

In this program, the user is required to play a game with the computer, we first choose the odd mode or the even mode, and in the next 10 games, computer will randomly select a number, user will need to type a number, the winner is decided by the odevity of the sum of the user’s number and computer’s number. In the main function, user first need to select the valid mode to jump out of the while loop, we noticed that in the gaming section, the random value for the computer is generated before the user type the number. If the user types an invalid number, it will not terminate the while loop but ask the user to type again. I observe that all the fgets functions are used properly, like

fgets(answer, BUFSIZE, stdin);

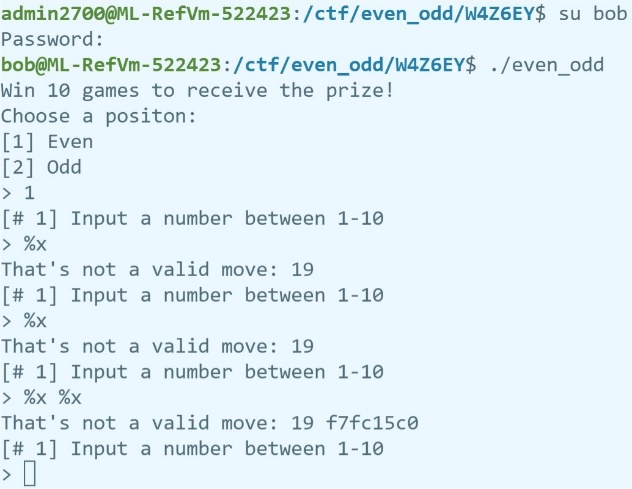
, the answer’s buffer size equals to the BUFSIZE, which means the buffer overflow technique will not be helpful here.

* Vulnerability: Format String Vulnerability using the format specifiers, for example in the following code. (line 88-94 in even\_odd.c)



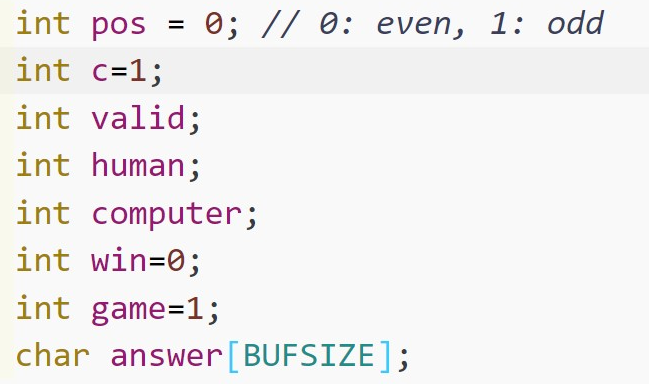
First the user takes the control of the answer buffer and the user can type any value not exceeding 25 characters. Then the value is transformed to integer and stored in human, the answer’s value is unchanged. After that, it uses the integer human value to test the feasibility. So, if we using the format specifier as input like “%x”, it will not pass the test in line 91 and run into the if-block (line92-94). In line 94, the printf(answer) is the source of the vulnerability, it prints the answer value rather than the human value. As a result, if we use the input as “%x”, the printf will show the content on the stack.

In order to test the hypothesis, we can just randomly select a mode, and type the user input as “%x”, and check whether it shows some hexadecimal value. The hypothesis is proved, details are shown below.

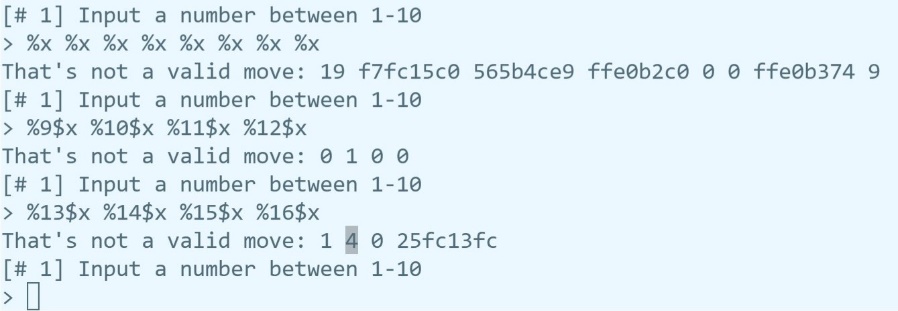


The attack strategy is divided into two steps. In the first step, the user needs to find out how many format specifiers should be used, and the second step is using the proper index format specifier to find the value of computer in the stack and using some math strategy to decide which value should the user to type according to the computer value.

For the first step, we can type “%x %x %x %x %x %x %x %x” to the first 8 stack value. Because of the BUFSIZE limit of the fgets we only can type 25 characters. So, for the follwoing buffers, we type “%9$x %10$x %11$x %12$x” and so on. We know that the integer value is between 1 and 10, and also the order of the variables in the source code is another strategy to find the computer value (graph 6.1), we know that computer is next to the win (win = 0). We can easily find that the 14th of format specifier is the human value. The find process shown in graph6.2.



graph 6.1



graph 6.2

In the second step, we just need to type the “%14$x” to gain the computer value, and using math strategy to determine which number should be used (the partial process is shown below). The reason why it works is that C compiler does not check that the number of format specifiers matches the number of arguments, so using the format specifiers we can gain all the information on the stack, especially the 14th index of the stack which stores the computer value.

