CS-392

Tarusi Mittal
1901CS65.

Ques 1: Use PWD environment variable or getcwd() function to find out the current directory.

Ans:- We should use the getcwd approach instead of PWD because of the following reasons

1. The value of PWD does not get updated on the chdir command calls whereas the getcwd() gets updated and gives us accurate results

2. If our set-UID program which has privileged access runs the code then it can change the environment variables, and the chances of buffer-overflow attack are there.

eg

```
$ echo $PWD
/home/seed
$ PWD = xyz
$ echo $PWD
xyz
```

Hence not correct answer.

2. 'more' is a filter for paging through the texts one screenful at a time.
It is a very primitive version

When viewing the file using more;
~~when~~ we can however use the v command to view the file in an editor.

Now if the more command was run as root;
then the editor is also opened as root,
which gives us the unlimited access to all files on the system.
which can also make our code vulnerable and prone to attacks.

So, it is not ok or advised to do so.

3- No, the shell command echo word will not be executed. The vulnerability exsists only when the variable value starts with '() {'.

If not, then it is considered as a regular string and foo will be available as a variable in the child process.

If we want to see how this prcess will run

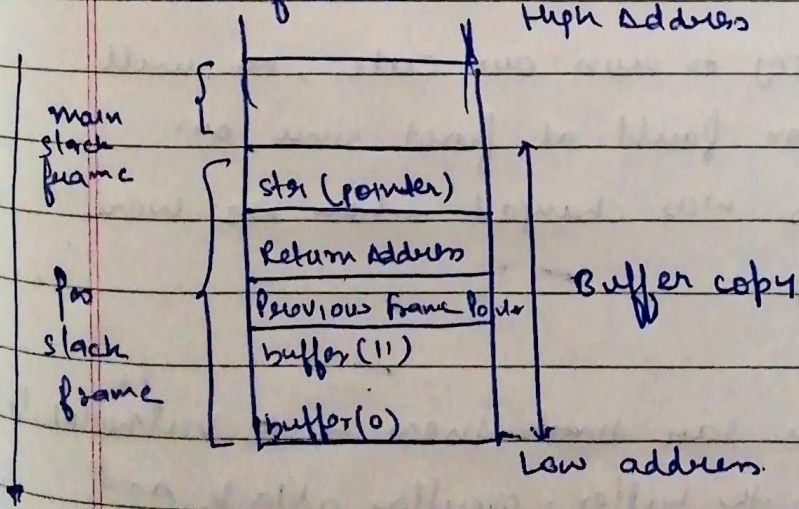
```
$ export foo = 'echo world ; () { echo hello ; }'
$ bash
$ foo
foo: command not found.
$ echo $foo
  echo world ; () { echo hello ; }
```

So, the shell command is not executed and only it was printed as part of echo $foo

4.

```
int f (char* str){
    char buffer[10];
    strcpy(buffer, str);
    return 1;
    }
int main(){
    char* str = "@@@@@@@@@@@@@@@@@@@@"
    f(str)
    return 1;
    }
```

→ Basically, when we run any program in our computer, a memory gets allocated to it for all the variables, The memory region is known as the stack frame



The function foo() in the diagram represent f. Now when we call our function it allocated the space in stack for our buffer array. And in our f() function the strcpy is copying our string into the str char array. However it will not stop copying the string unless it receives a particular character /0.

Now if we try to copy a string that is more than the length defined by our buffer array the strcpy will keep coping it in our previous frame pointer address and above. Hence arousing the condition of buffer - overflow.

If we will try to exec our code, we will get segmentation fault at first since our return address was changed when we were copying the string.

Therefore, we can say that there is a vulnerability in the code to the buffer - overflow attack and if we in case give the input our of char array in such a way that the return address is actually pointing to a malicious code, we can have the system do things of our interest

5. Given format-string vulnerability needs to be exploited but the stack cannot be changed to executable mode.

→ To find: which part of the stack needs to be modified and how to achieve that by exploiting a format-string vulnerability

→ We use the return-to-libc technique to return to the system() function, and use this to run the shell. This technique can be used to defeat the non-executable stack countermeasure.

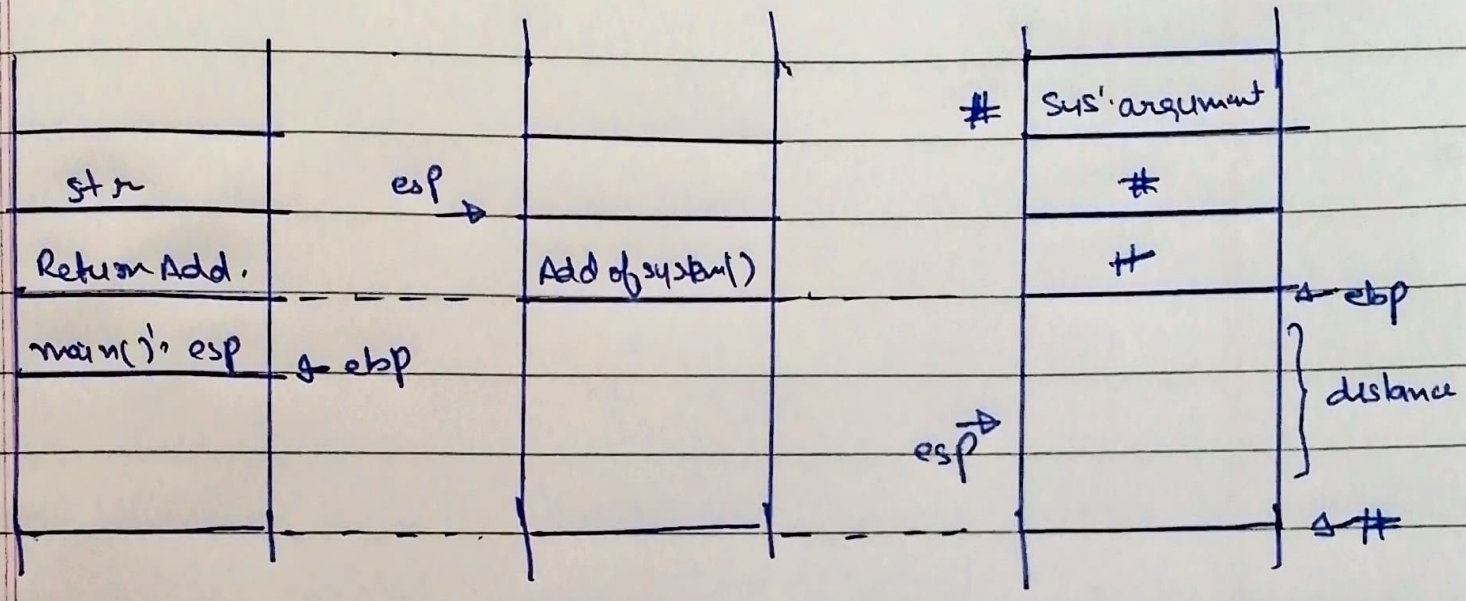
To use this method in format-string attack, we need to modify two places on the stack,

1) the return address field
2) the place where the system() function gets its argument.

Now, we know that the argument should be stored in a memory location four bytes above the return address.

Therefore to ~~attack~~ defeat - the non-executable stack countermeasure, we just need to use the format string attack to modify the

memory locations written earlier. We can acheive that with one - single format string.



- Inside the function
- Right after return from function. after running into function epilogue
- Inside system() after running the function's prologue