CIS 643 Computer Security

Lab 4 Buffer Overflow Vulnerability Lab

**Instructor: Wenliang (Kevin) Du**

Contents

Initialization 2

Test malicious codes 2

Task 1: Exploiting the Vulnerability 2

Setup for task 2

Find return address 3

Create badfile 4

Buffer-overflow attack 5

Conclusion 5

Task 2: Protection in /bin/bash 5

Task 3: Address Randomization 6

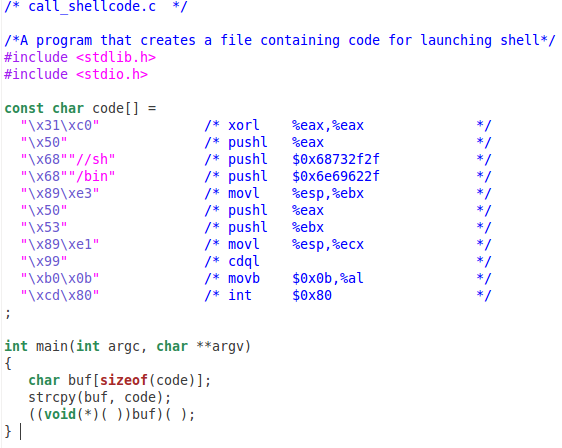
Task 4: Stack Guard 6

Task 5: Non-executable Stack 7

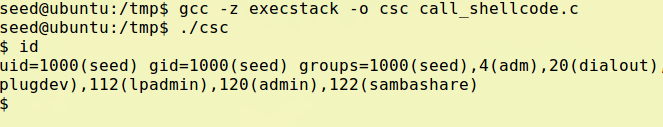
## Initialization

### Test malicious codes

The following codes are malicious codes used to attack the Set-UID program.



Since the Ubuntu that I used for lab has non-executable stack as default, so we need use “–z execstack” to set stack executable. Compile and test the code:



The results could tell that the codes work well.

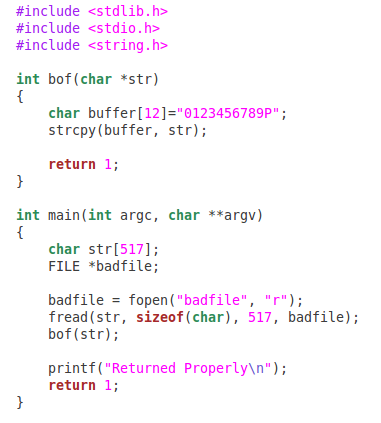
## Task 1: Exploiting the Vulnerability

### Setup for task

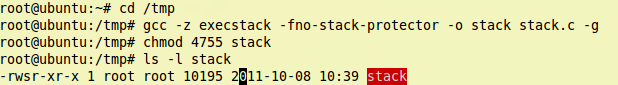
Turn off the memory randomization.

Macintosh HD:Users:Ider:Desktop:Screen Shot 2011-10-09 at 9.16.26 AM.png

The following program is a Set-UID program, which has buffer-overflow issue. To achieve goal of task 1, when we compile the codes, “-fno-stack-protector” will be included to turn off the stack protector, and “–z execstack” will be included to set stack executable.

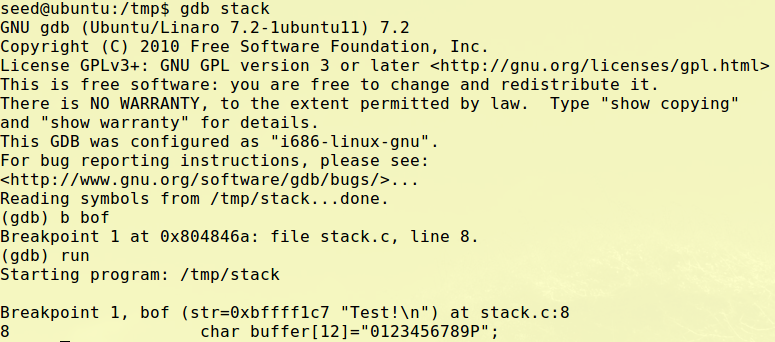


To make the lab easier, I include “-g” to further use “gdb” to figure out where is “return address” and what’s the address value should I put into it.

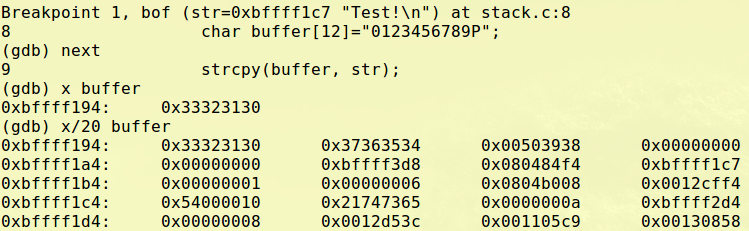


### Find return address

Now, login as normal user, and use gdb to debug the stack program. The “badfile” file only contains some text for testing. I also put initial values to buffer[] in bof function for locating address easily.

Set breakpoint at “bof(char \* str)” function, then run the program.

Run program to next lines, and then output the 32 addresses begin at buffer

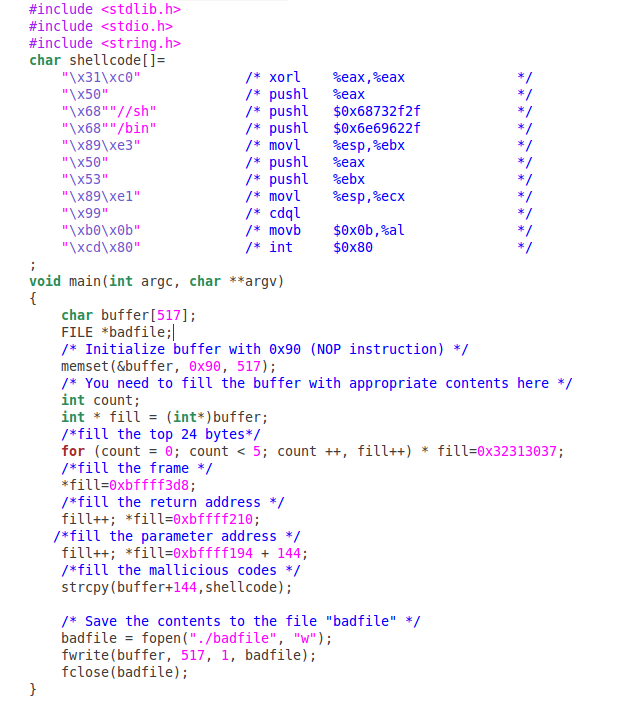


Match the output to he initial value of buffer, we could tell that fist three items in first line is the values of buffer. As the value of parameter “str” is 0xbffff1c7, so the last item in second line is where str located.

So, according to the structure of stack and memory frame, we could tell that third item in second line (whose value is 0x080484f4) is where “return address” located. And second item in second line (whose value is 0xbffff3d8) is where “frame pointer” point to.

### Create badfile

Since we got where is return address located, we could create badfile now:



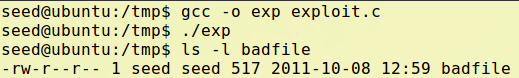
The most important value is the return address. I set to 0xbffff194 (the address where buffer begin in stack program) plus 144, and copy malicious codes at buffer+144.

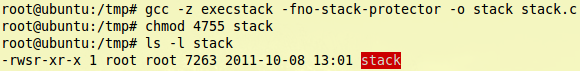
But we could set the return address value as any address that is bigger than 0xbffff1b4, and put the malicious codes begin at address which is not less than the return value. And the NOP will help to reach the malicious codes.

The values for buffer, frame pointer and parameter do not matter, but I just specify them with original value.

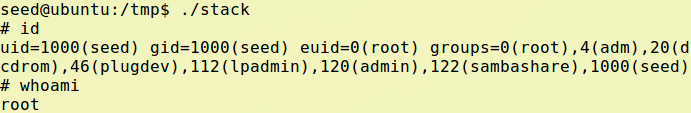
### Buffer-overflow attack

Compile the program and run it to generate the badfile:



Login as root, and compile the stack without debug tag

Now, let normal user run the Set-UID program to see if we run the shell of root



We could see that the effective id is 0, and we have got the permission of root.

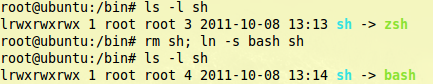
### Conclusion

In real life, we would not got chance to use gdb to find the address and attack the Set-UID program, but we still could guess an address because we have big range to set the value, as I mentioned before: the NOP will help to reach the malicious codes.

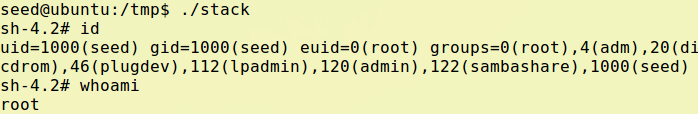
However, task 1 is just to exploite the vulnerability of program that has buffer overflow issues.

## Task 2: Protection in /bin/bash

We keep badfile, stack program unchanged. Login as root and change /bin/sh, let it links to bash shell



Now, run the stack again, to see if we could get into bash shell



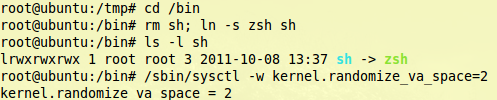
So, it is almost the same as before, the only different is the prompt for command, anyway, we have got root permission in bash shell.

The reason is that in Ubuntu 11.04, bash shell would not get rid of permission of effective user when running Set-UID program, so it is totally the same as zsh shell when we invoke the buffer-overflow attack.

But if we use the previous version of Ubuntu, we need to add “seteuid(0);” statement in badfile to manually set effective id as root, then get permission of root.

## Task 3: Address Randomization

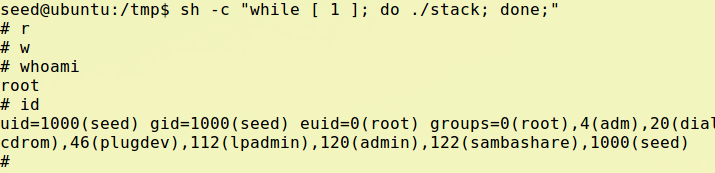
Now, link /bin/sh back to zsh shell, and turn on the memory randomization.



We still keep the badfile and stack program unchanged, but run the stack program in a shell script loop to get more chance to hit the malicious codes.

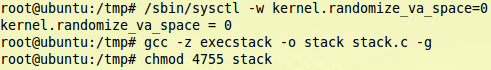
Macintosh HD:Users:Ider:Desktop:Screen Shot 2011-10-09 at 12.25.04 PM.png

In a long time, we would get nothing, but since the memory I assigned to the Virtual Machine that running Ubuntu is not very big. In a time, we still hit the malicious codes and get root permission:

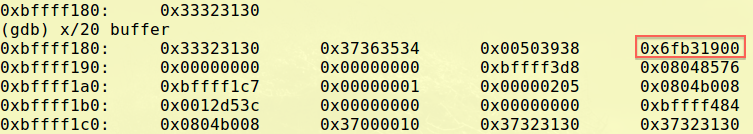


## Task 4: Stack Guard

Turn off the memory randomization, this time we compile the stack program with stack guard on, i.e. remove the “-fno-stack-protector” in gcc command

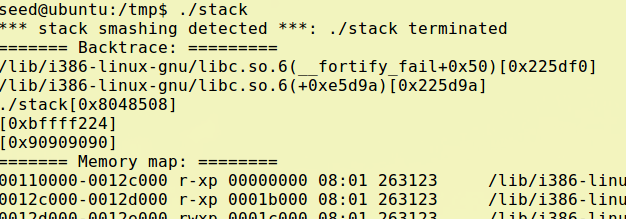


Use gdb to see the memory of stack program



We could see that there is one more item between buffer and return address. And if we debug the program again and again, we would find that the values in that address are different each time we run the program.

We still could find where is return address located, but we need do some modification in bad file to fill the new item. Since the value of item is random, I just assign value on that address arbitrarily.

Now run the stack program again

Instead of getting permission of root, we get “stack smashing detected” information.

And the new item we mentioned before is just the stack protect. When the program is running, it will be assigned a value, before stack shrink, the value will be checked to if it has been modified. If program find the value is different with before, it consider that program has buffer overflowed, so it reject to continue running the program.

## Task 5: Non-executable Stack

This time, we remove “-z execstack” in gcc command to make stack non-executable.

We don’t run the stack program, but run the test program, call\_shellcode.c, to see if we could run codes in stack.

Macintosh HD:Users:Ider:Desktop:Screen Shot 2011-10-09 at 12.51.28 PM.png

We had run this program with “-z execstack” on before, and I had got access to the shell.

But this time, we only “Segmentation fault”, which means program didn’t run appropriately, as the codes could not run at stack.

So, when the Set-UID program turn “non-executable stack” on, the normal user could not even get a chance to run the codes at stack. Hence, this would protect the Set-UID program from the malicious codes attack with buffer-overflow vulnerability.