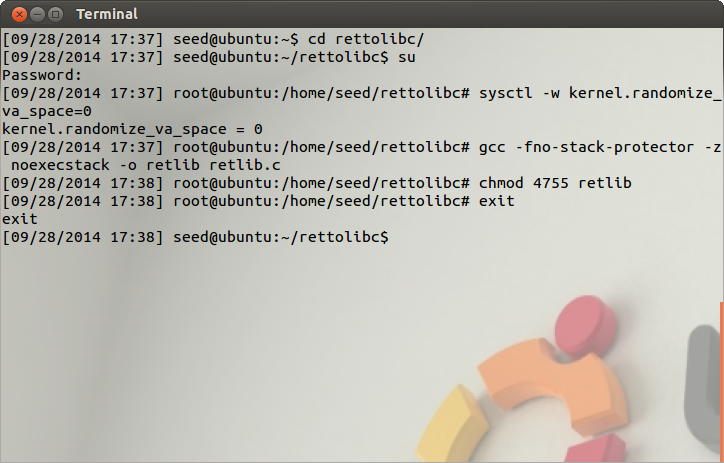
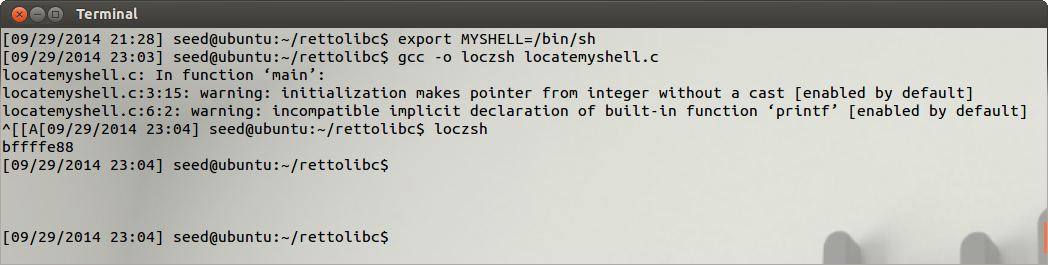
**Return-to-libc Attack Lab Report**

**Task 1: **

Set-UID program

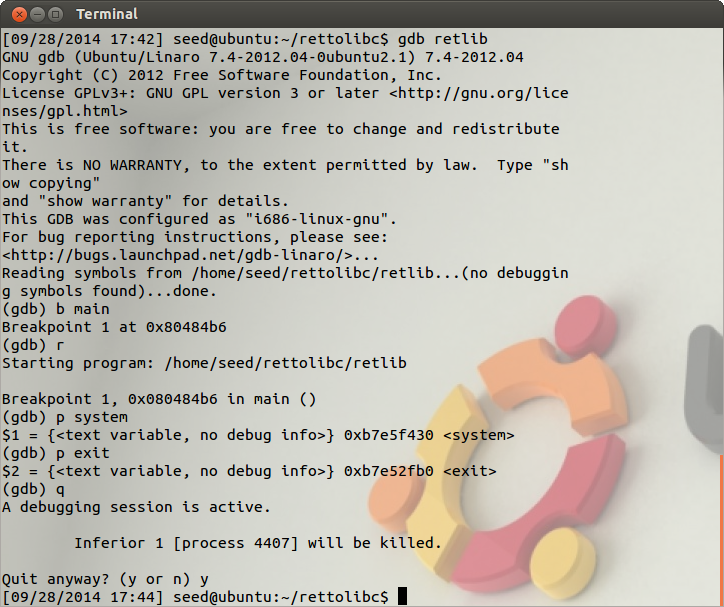
ASLR disabled

Figure 1.1



Address of /bin/sh

Figure 1.2

****

NON Set-UID retlib files run using gdb

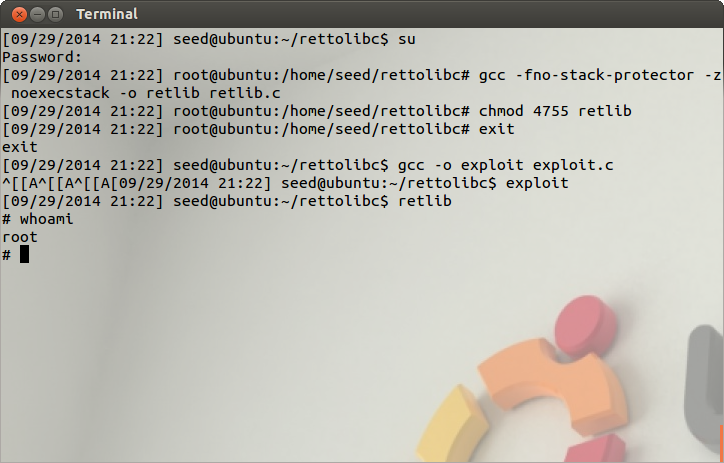
Address of exit function on the stack

Address of system function on the stack

Figure 1.3

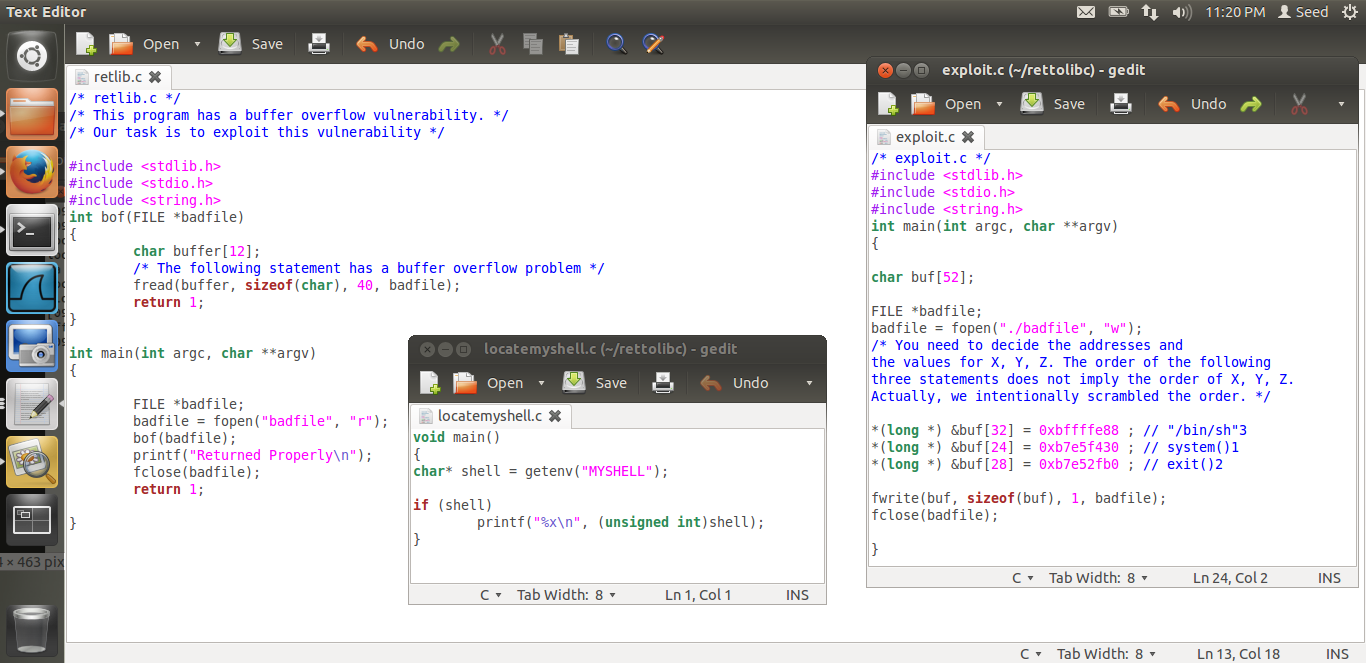
****

Figure 1.4

****

Root access granted, attack successful

Figure 1.5

****

Addresses input to buf[X], buf[Y], buf[Z]

Figure 1.6

**Observations and Explanations:**

1. We turn off Address Space Randomization to be able to perform the attack,

***$ su root***

***Password: (enter root password)***

***#sysctl -w kernel.randomize\_va\_space=0***

We also compile the retlib program which contains the vulnerability, as a root Set-UID program in Figure 1.1

1. We add /bin/sh to the environment variables and find the address of this on the stack using the loczsh program that displays the address of the environment variable.

Address of **/bin/sh** is found to be, **0xbffffe88**

1. We compile retlib as a non-root Set-UID program and use gdb to be able to get the address of system() and exit() from the stack. (Figure 1.4)

We find that they are located at,

**system() 0xb7e5f430**

**exit() 0xb7e52fb0**

1. We then locate the address of the buffer and the address of the ebp to be able to determine where to overflow the address of **system(), exit()** and **/bin/sh.**

We run gdb with retlib as non-Set-UID program and get the values as follows,

**buffer** **0xbffff300**

**ebp 0xbffff318**

with these values we calculate the size of the buffer till the previous frame pointer is reached to be as 24. We want to load the address of system to the address where previous frame pointer is and so we add systems address on the stack to buffer[24].

Above this address at 4 bytes we want to add the address of exit() so that when the call to system is made with /bin/sh, and fails, the process should leave neatly without leaving any traces of an attack.

So at buffer[28] we add the address of exit() on the stack.

Next we know that system needs an argument, and this argument in our case needs to be **/bin/sh** for the attack to gain root access.

So at buffer[32] we add the address to the stack.(Figure 1.6)

So the buffer is loaded with the following addresses,

**\*(long \*) &buf[32] = 0xbffffe88 ; // "/bin/sh"3**

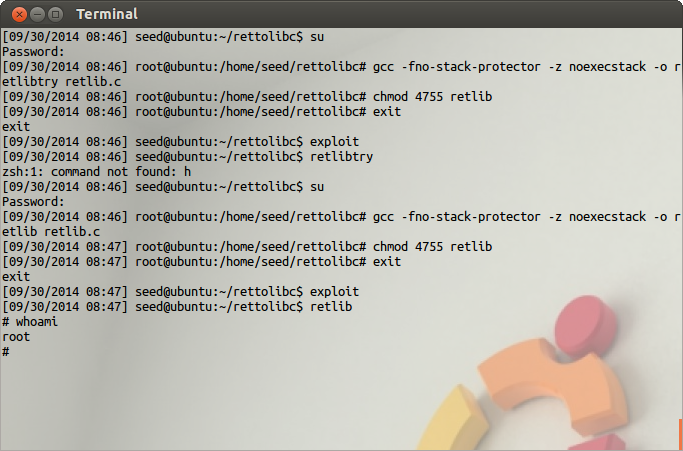
**\*(long \*) &buf[24] = 0xb7e5f430 ; // system()1**

**\*(long \*) &buf[28] = 0xb7e52fb0 ; // exit()2**

1. We run exploit where we write the value to the stack using the buffer overflow vulnerability.

We then execute retlib and are able to get the root privileges, meaning that the return-to-libc attack is successful.

1. A key observation that was made was that when the file name for retlib was changed from root Set-UID to non-root Set-UID, the attack failed and command not found instraction was noticed, the reason for this is that the file name is part of the stack, it is a part of the data stored on the stack. If the length of the file changes the values of where the system address and /bin/sh address should be stored needs to be changed accordingly.

****

Root access obtained with same length filename.

On changing the size, the address on the stack will need to be re-calculated

1. The stack looks like this when the attack is going to be successful,

Address of the previous stack frame pointer

buffer[11]  
…….  
buffer[0]

Address of exit()

Address of system()

**0xbffffe88**

Address of /bin/sh

Stack grows in this direction

**0xbffff31C**

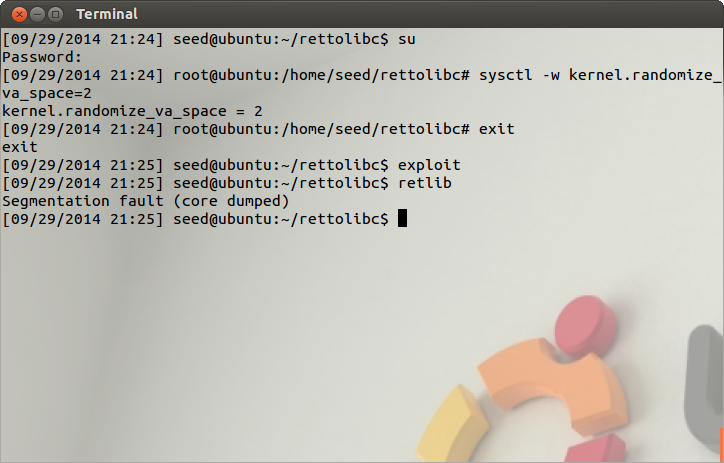
**0xbffff318**

**0xbffff300**

Address grows in this direction

* This figure shows the contents of stack after buffer overflow has been done. First 20 bytes are actual memory allocated to stack, of which 12 bytes belong to the buffer.
* Next 4 bytes would be the pointer to the previous stack frame. So this contains the memory address of the stack frame of the main() function. So first 24 bytes of the buffer array are empty.
* The next 4 bytes are supposed to be the return address of bof() function, and this is where we attack the program, we overflow this region with the address of the libc function system()
* After completion of system call, execution will return to the address specified in this 4 bytes block. So address of the exit function is fed at this address, buffer[28]
* During the execution of system() function, program looks for available arguments to the system function call, So we put /bin/sh address so that we can get root access.

**Task 2:**

****

ASLR enabled, to reduce probability of right combination of addresses for buffer

Figure 2.1

**Observations and Explanations:**

1. We enable address space randomization to make check if attack still succeeds with ASLR enabled.

***$ su root***

***Password: (enter root password)***

***#sysctl -w kernel.randomize\_va\_space=2***

1. We notice that segmentation fault occurs. If address space randomization is enabled.
2. Observe that a “return-to-libc” attack needs to know the virtual addresses of the libc functions to be written into a function pointer or return address. If the base address of the memory segment containing libc is randomized, then the success rate of such an attack significantly decreases.
3. We can try to run a while loop and exit retlib attack repeatedly to try and get root access but that is only because the randomizations in a 32-bit address space is limited, if it were to be executed in a 64-bit environment then the probability ofgetting root access would decrease to a minimum level.

**Task 3:**

****Figure 3.1

Stack smashing detected as ‘canary’ is corrupted

**Observations and Explanations:**

1. We now compile the code with noexec stack flag enabled.

***$ su root***

***Password***

***# gcc -z noexecstack -o retlib retlib.c***

***# chmod 4755 retlib***

***# exit***

1. Since the “canary” is checked before the ret instruction is executed, your exploit will fail if you overwrite the canary (which in most cases you have to do in order to overwrite the return address on the stack). Since ROP and Return to Lib c also overwrite the return address, both methods will not work. Figure 3.1
2. The stack looks like this with the stack guard turned on in general.

**CANARY(StackGuard)**

Arguments/Parameters

Stack grows in this direction

**0xbffff31C**

**0xbffff318**

**0xbffff300**

Address grows in this direction

Return Address

Previous FP

buffer[11]  
…….  
buffer[0]

1. This is what the stack looks like to in case of StackGuard being enabled for our return-to-libc attack,

Address of previous frame pointer

buffer[11]  
…….  
buffer[0]

**CANARY(StackGuard)**

**0xbffffe88**

Address of exit

Address of system

Address of /bin/sh

Stack grows in this direction

**0xbffff31C**

**0xbffff318**

**0xbffff300**

Address grows in this direction