Shellcode

bruce30262

Outline

- What is shellcode
- Review x86 linux system call
- How to write & test your own shellcode
- Basic shellcode tricks
- Advanced shellcode tricks

What is shellcode

- 長得像這樣
  - "\x31\xc0\x50\x68\x66\x68....."
- 指的是一連串可以被直接執行的 machine code
- 為什麼會被叫shellcode?
  - 因為我們常常藉由執行它來拿到shell

What is shellcode

- shellcode 的意義
  - 假設攻擊者可以執行 data 段的 code
  - 任意控制 data 的內容 → 任意執行想要的 code
  - 很多攻擊者常常會先嘗試用別種攻擊方式，呼叫 mprotect() 來更改程式的記憶體權限(將 x 權限打開)
  - 之後就可以將 shellcode 塞進記憶體，執行任意指令

Review x86 linux system call

- x86 linux system call table

- 總之要呼叫 system call，我們必須:
  - 在各個 register 裡面塞進對的值(參數，system call number...)
  - 執行 int 0x80 (x86)，發一個 interrupt 來呼叫 system call (x64 的話要用 syscall 指令)

sys_execve (syscall number : 0xb)

- int execve(const char *filename, char *const argv[], char *const envp[]);
  - filename (for ebx) : 欲執行的 binary 路徑
  - argv[] (for ecx) : 給程式的參數
  - envp[] (for edx) : 環境變數
- 目標 : 執行 execve(“/bin/sh”, NULL, NULL)
- 為了方便起見，這裡 argv[] 和 envp[] 我們直接清成 0 (NULL byte)

How to write your own shellcode

- machine code 是由 assembly 轉來的
- 因此要生 shellcode，我們必須要會一些 assembly programming

- 工具
  - nasm : assembler (生成 object file)
  - ld : GNU linker (生成 executable)
  - xxd : 方便觀察 shellcode

How to write your own shellcode

- git clone https://github.com/bruce30262/x86_shellcode_tutorial.git

- 進到 0_execve_binsh 資料夾
- 打開 shell.s，觀察一下裡面的 assembly

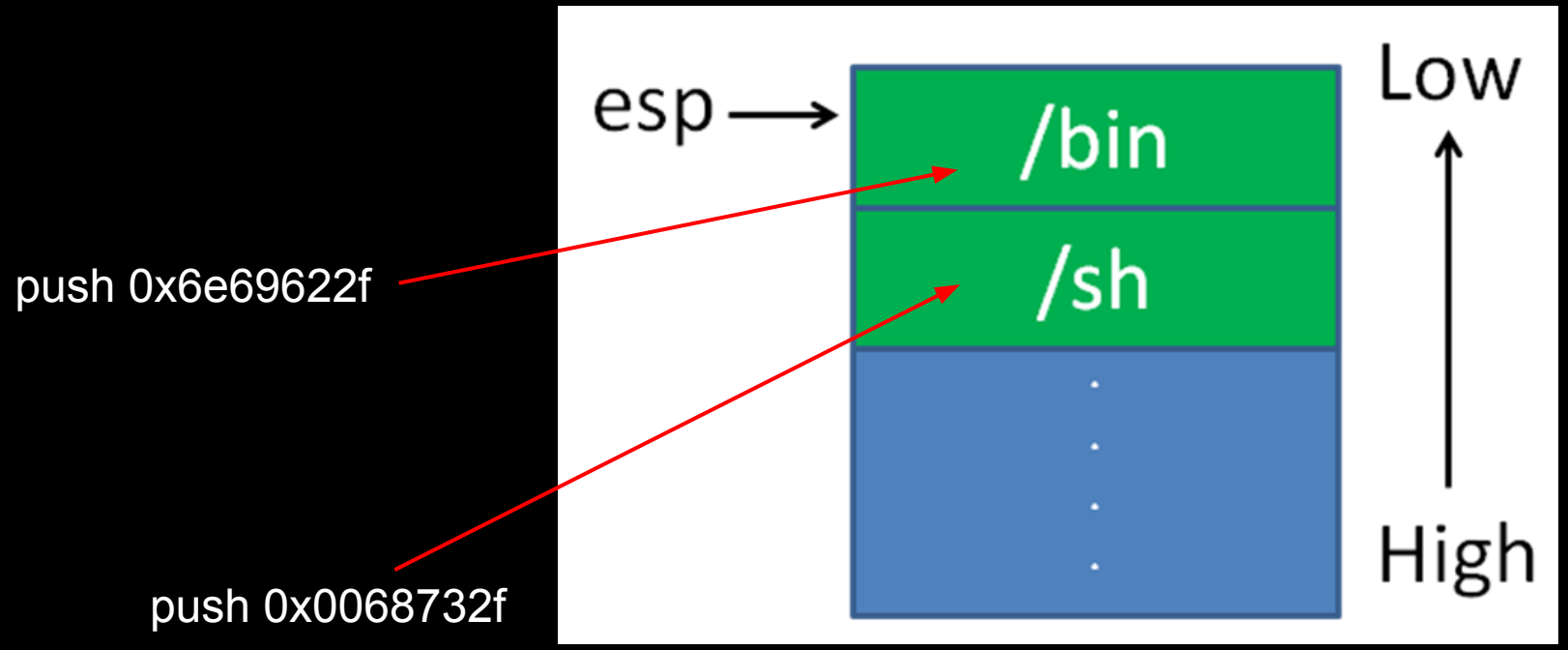
- 設定 eax, ecx 和 edx 的值那邊應該是沒有什麼問題
- ebx 的設定比較麻煩
  - 裡面必須放一個 pointer (char *)
  - 這個 pointer 必須指向字串 "/bin/sh"

How to write your own shellcode

- push 0x0068732f
  - 0x0068732f 可視為字串 "/sh" (null byte 結尾)
  - 同理 0x6e69622f 可視為字串 "/bin"
  - 此時，esp 將會指向字串 "/bin/sh"

- mov ebx, esp
  - 將 esp 存的記憶體位址 (指向 /bin/sh) 放入ebx裡面

How to write your own shellcode



How to test your own shellcode

- "make bin" 將 assembly 編成一個可執行檔
- "./shell.bin" 執行，成功的話將會拿到 shell

- 使用 gdb 來 debug
  - gdb ./shell.bin
  - disas _start，查詢所需要的記憶體位址
  - 下斷點 & debug

How to test your own shellcode

- Get the shellcode
  - "make shellcode"
  - char array 裡面存的那一連串 hex value 即為 machine code (shellcode)
- Test the machine code
  - 打開 shell.c, 將拿到的 machine code 塞入一個char array
  - "make cbin && ./shell.out"
  - 執行成功的話會拿到 shell

Basic shellcode tricks

- Null free
  - strcpy()，sprintf()...等函式遇到 null byte 的時候會停
  - 盡量避免含有 null byte 的 shellcode，順便減少長度
- 方法
  - mov eax, 0x0 → xor eax, eax
  - mov eax, 0xb → mov al, 0xb
  - shr eax,0x8 → set eax = 0x00xxxxxx

Basic shellcode tricks

- 進到 1_execve_binsh_nullfree 資料夾
- 打開 shell.s，觀察一下 assembly，看看跟前一個有什麼不同
- "make shellcode"，觀察一下產生出來的 shellcode

```
unsigned char code[] = {
  0x31, 0xc0, 0x50, 0x68, 0x2f, 0x2f, 0x73, 0x68, 0x68, 0x2f, 0x62, 0x69,
  0x6e, 0x89, 0xe3, 0xb0, 0x0b, 0x31, 0xc9, 0x31, 0xd2, 0xcd, 0x80
};
unsigned int code_len = 23;
```

Basic shellcode tricks

- 講下一個 trick 之前，先來講一下 open / read / write
- int open(const char *pathname, int flags)
  - *pathname (for ebx) : 檔案路徑
  - flags (for ecx) : 開啟模式 (ex. 唯讀)
- ssize_t read(int fd, void *buf, size_t count)
  - fd (for ebx) : file descriptor
  - *buf (for ecx) : buffer 位址
  - count (for edx) : 最多讀幾個byte
- ssize_t write(int fd, const void *buf, size_t count)
  - 參數意義基本上跟 read() 一樣

Basic shellcode tricks

- 先創一個文件
  - /home/shellcode/flag, 裡面塞一些內容
- 進入 2_open_read_write 資料夾
- 打開 orw.s, 觀察一下 assembly
- open(“/home/shellcode/flag”, O_RDONLY)
  - push 檔名到 stack, 之後 mov ebx, esp
  - ecx 直接清成 0 (O_RDONLY)

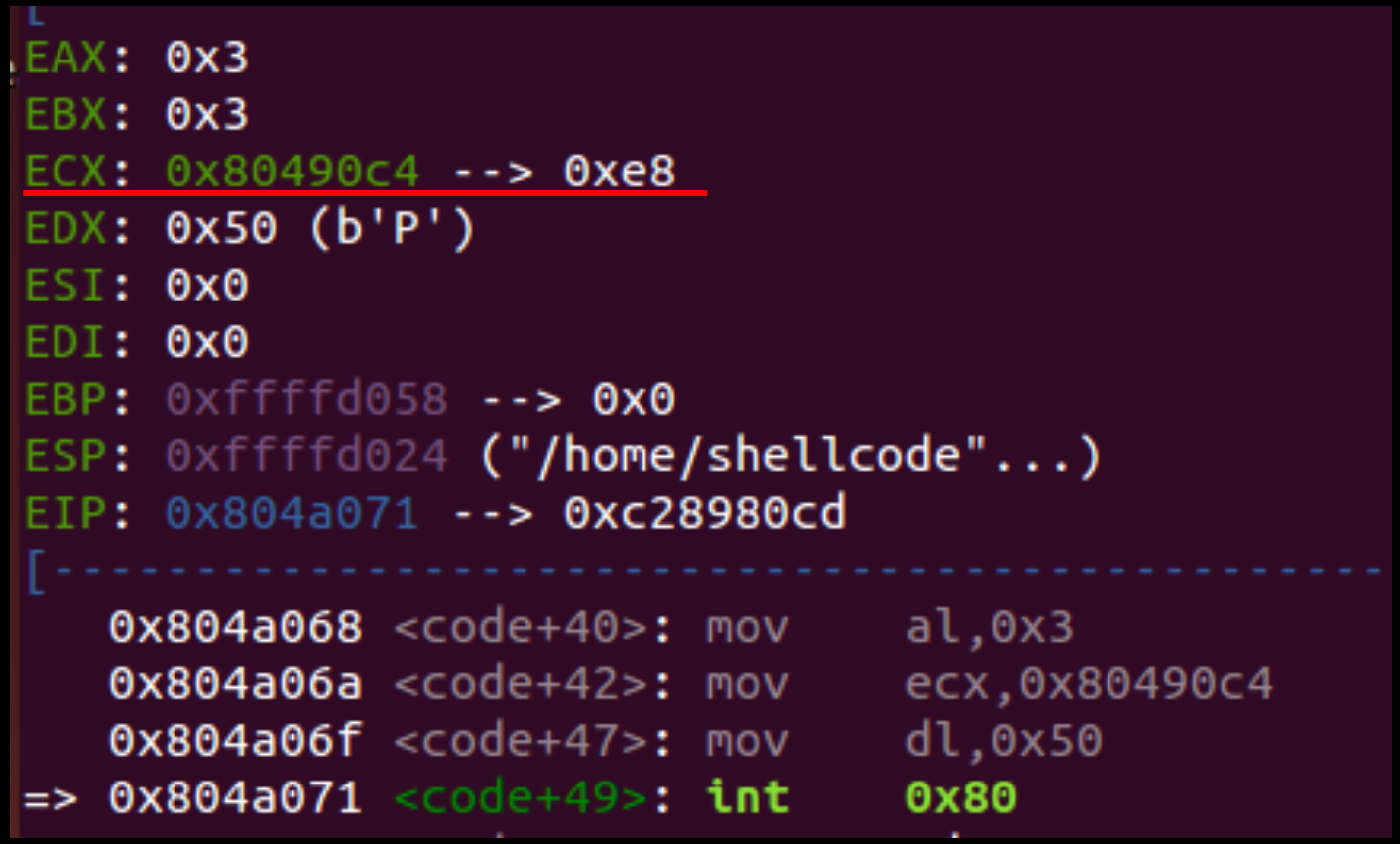
Basic shellcode tricks

- read(fd, mybuf, len)
  - open 完檔案之後，其回傳值即為 fd (放在 eax)
  - 先將 eax 的內容 move 到 ebx
  - 在.bss section 宣告一個 buffer，並 move 進 ecx
    - mybuf: resb 0x50 ＆ mov ecx, mybuf
  - 在 edx 塞入想要讀的長度
- write(fd, mybuf, len)
  - read的回傳值即為它讀到的長度
    - mov edx, eax 設定輸出長度
  - 要印到螢幕上，因此 fd 為 1 (STDOUT)
  - buffer 位址不變，因此 ecx 不需重新設定

Basic shellcode tricks

- 加個 exit(0) 避免程式輸出完檔案內容後直接 crash 掉
- make bin && ./orw.bin 查看執行結果
  - 執行成功的話會印出檔案內容
- make cbin && ./orw.out 查看 machine code 的執行結果
  - (理論上) 要出現 Segmentation fault
  - 使用 gdb debug 一下

Basic shellcode tricks



Basic shellcode tricks

- Position independent
  - 塞入 shellcode 時，我們通常無法確定 shellcode 所在的記憶體位址
  - 所有的 call, jmp, buffer 位址...均須要使用相對位置
- 進入 3_open_read_write_position_independent 資料夾
- 打開 orw.s 觀察一下 assembly
- read 的時候，先把 stack 拉長，再把資料讀到 stack 上
  - sub esp, 0x50 (此時 esp 即為 buffer 位址)
  - mov ecx, esp
- 測試一下是否能 work

Basic shellcode tricks

- NOP sled
  - shellcode 的位置常常抓不準
  - 前頭可以多加幾個 NOP 指令
  - 這樣子只要跳到其中一個 NOP，就可以順勢"滑"過去，最終執行到我們的 shellcode
- Jump relative
  - 有時候我們會無法控制整段 shellcode (被其他指令寫爛)
  - 可利用 jump relative 跳過這些爛掉的 shellcode
    - "\xeb\x08" → 往前跳 (skip) 8 個 byte

Advanced shellcode tricks

- alpha-numeric shellcode
  - alpha-numeric x86 op code
  - shellcode 中只包含 [0-9a-zA-Z]
  - 使用到的 opcode 均為 printable 字元
  - 使用 xor 來算出想要的 byte / word
- 進入 4_alphanumeric_shellcode 資料夾
  - alpha-numeric & 不包含 [BINSHbinsh] 的 shellcode
  - "make shellcode" 來查看 alpha-numeric shellcode
  - 觀察 alnum.c 的內容，並編譯執行看看 (make cbin)

Advanced shellcode tricks

- call - pop trick
- ~~抄台大 217 的~~
- 進入 5_call_pop_trick 資料夾
- 打開 shell.s，觀察 assembly
- 一開始先 jump 到 ed 那邊，再 call st
  - call st 的時候，會先把 return address push 進 stack 裡面，之後再 jump 進 st 裡面執行
  - return address = db ‘/bin/sh’ = pointer to "/bin/sh"
  - st 的第一行是 pop ebx，此時 ebx 的內容值就會是一個 pointer to "/bin/sh"

Advanced shellcode tricks

- pros & cons
  - 不需要對路徑做 double word 的轉換 (ex. "/bin" → 0x6e69622f), 比較好寫
  - 不過 shellcode 的長度也會因此增加
- 思考一下
  - null byte 為什麼要之後再加? 寫成 db ‘/bin/sh’, 0x0 會有什麼問題?
  - _start 那邊為什麼要先跳到 ed 再 call st ? 直接 call st 的話不行嗎?
  - 為什麼要將 section 設為 .data ? 設成 .text 的話又會發生什麼問題?

Reference

- Linux man page
- x86 linux system call table
- man nasm
- man ld
- reading from a file in assembly
- alpha-numeric x86 op code
- 如何撰寫 alpha-numeric shellcode (日文)
- call-pop trick

- http://shell-storm.org/shellcode/

Bonus

- nc 140.113.194.88 30001
- 原台大 CTF 課程題目
- 這題拿不到 shell (有設定過 sandbox)
- 請利用課程所學的知識，寫出一個 open-read-write 的 shellcode，讀取 /home/orw/flag 的檔案內容