Basic Concept

Reverse Engineering

- 透過逆向工程分析程式來找出程式的漏洞或是修改程式
- static analysis program without running
  - objdump
  - strings
- dynamic analysis program with running
  - ltrace, strace
  - gdb

Exploitation (Pwn)

- 利用漏洞來達成攻擊者目的
- 一般來說主要目的在於取得程式控制權！
  - 本地提權 ==> 手機越獄
  - Remote Code Execution ==> 取得 remote shell

Basic Tool

GDB

run

- run - 執行程式
- si - 下一行指令，跟進 function call
- ni - 下一行指令，不跟進 function call
- c[ontinue] - 繼續執行
- record - 開始記錄，有記錄時可以反相執行
  - rsi - 上一行指令，同 si
  - rni - 上一行指令，同 ni
  - rc - 反相繼續執行，同 continue
- 直接按 enter 會執行上一個輸入的命令

break points

- b[reak] - 可以直接對目前位置下 breakpoint
- b[reak] *<address> - 在某個位置下 breakpoint
- del[ete] <break point id> - 刪除第幾個 breakpoint
- info b[reakpoint] - 查看所有 breakpoints

catch [syscal|signal...]

- 可以偵測 syscal, signal... 等建立 breakpoint，執行 syscal, signal... 後停下來

show info

- info registers - 查看目前暫存器狀態
- Info proc map - 查看目前的 memory map
- backtrace - 顯示上層的 Stack Frame 資訊

x/x

- x/k[b|h|w|g]x <address> - 查看某個位置的內容
  - /k 後可以放入要印出幾個單位出來，e.g. x/40gx
  - b/h/w/g，分別是取 1, 2, 4, 8 bytes 為 1 單位
  - u/d/x/s/i，分別是 unsigned int/dec/hex/string/instruction - 數值顯示方式

set

- set <reg>=<value> - 把某個 register 改成數值
- set *<address>=<value> - 把某個地址填入數值 (4 byte)
- * 可換成 {char} {short} {long} 來改變填入的長度
  - e.g. set {long}0x400000=1 - 把 0x400000 填入 0x00000001

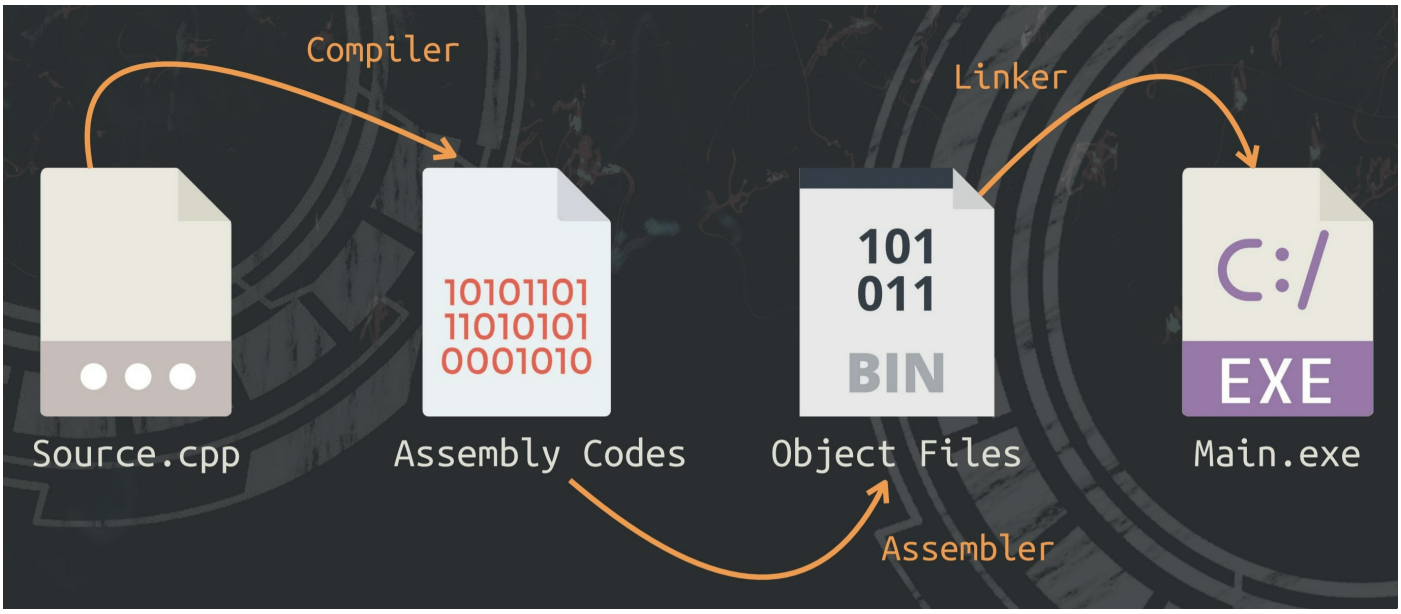
attach

- 可以配合 ncat 來 debug exploit
  - $ ncat -vc <elfpath> -kl 127.0.0.1 <port>
- 需要 root 權限

Pwntools - 這裡沒有筆記！

Basic about Code

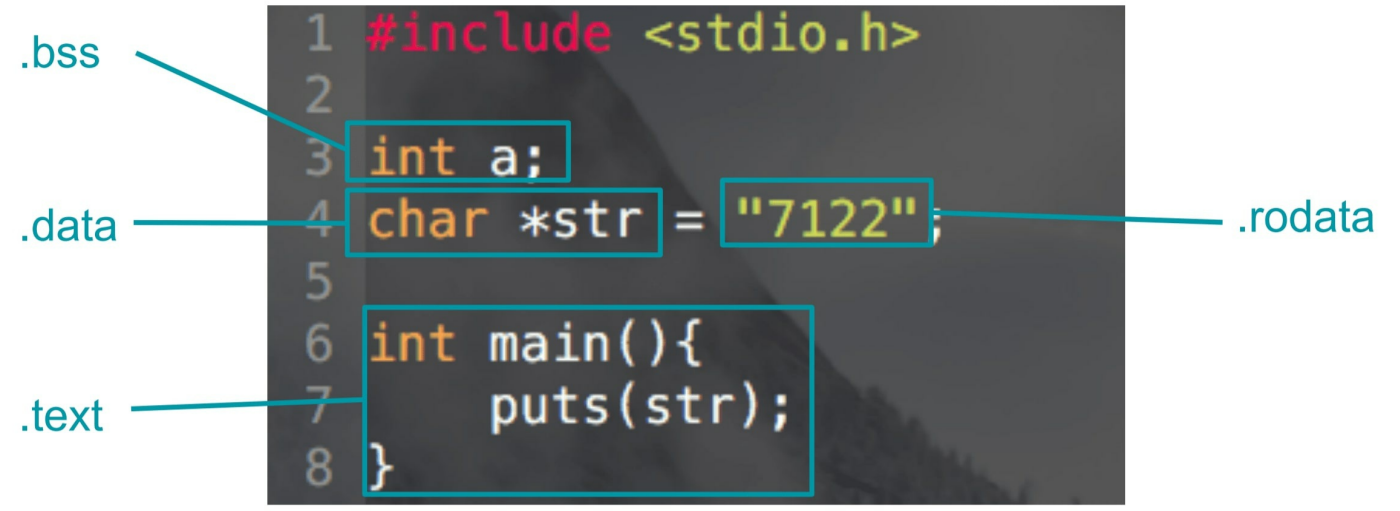
General Compiler



Section, Segment

程式碼會分成 text, data, bss 等 section ，並不會將 code 跟 data 混在一起

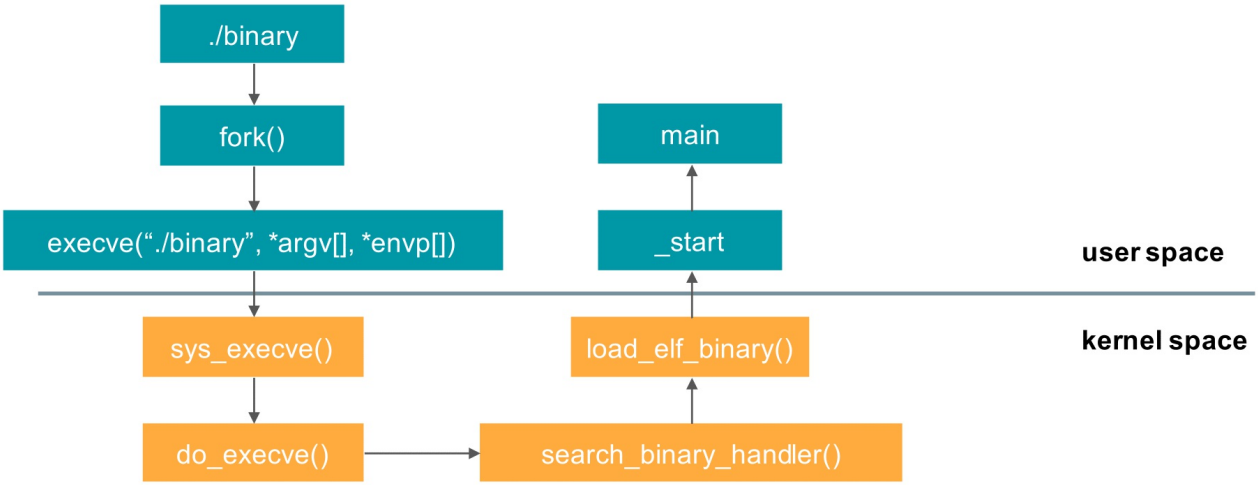
- .text
  - 存放 code 的 section
- .data
  - 存放有初始值的全域變數
- .bss
  - 存放沒有初始值的全域變數
- .rodata
  - 存放唯讀資料的 section



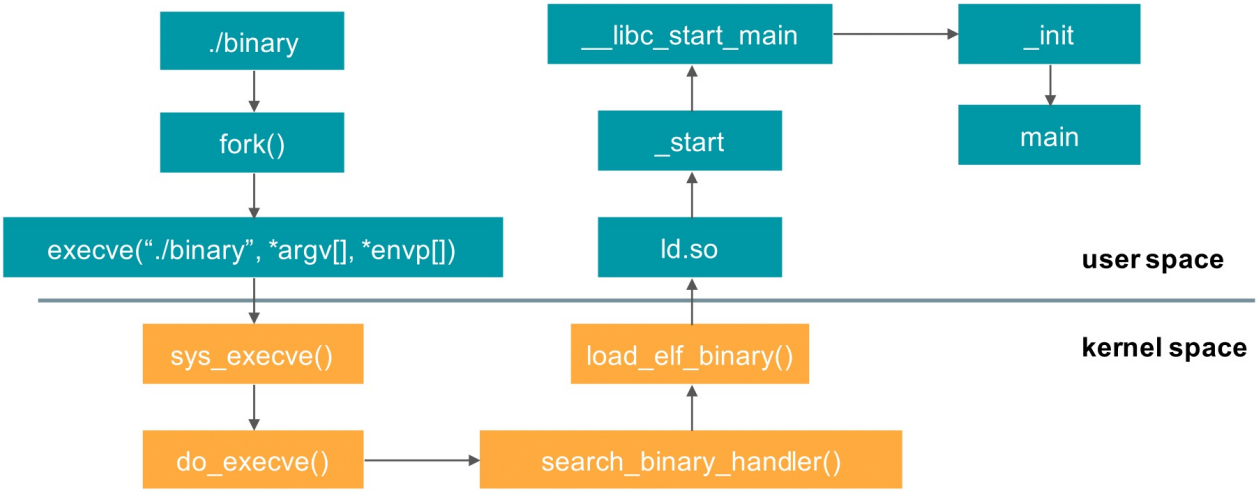
而 segment 是在程式執行時期才有的概念，基本上會根據讀寫執行權限及特性區分，常用到的可分為 data, code, stack, heap

Execution Flow

Execution Flow (Static Linking)



Execution Flow (Dynamic Linking)



Assembly

Syntax

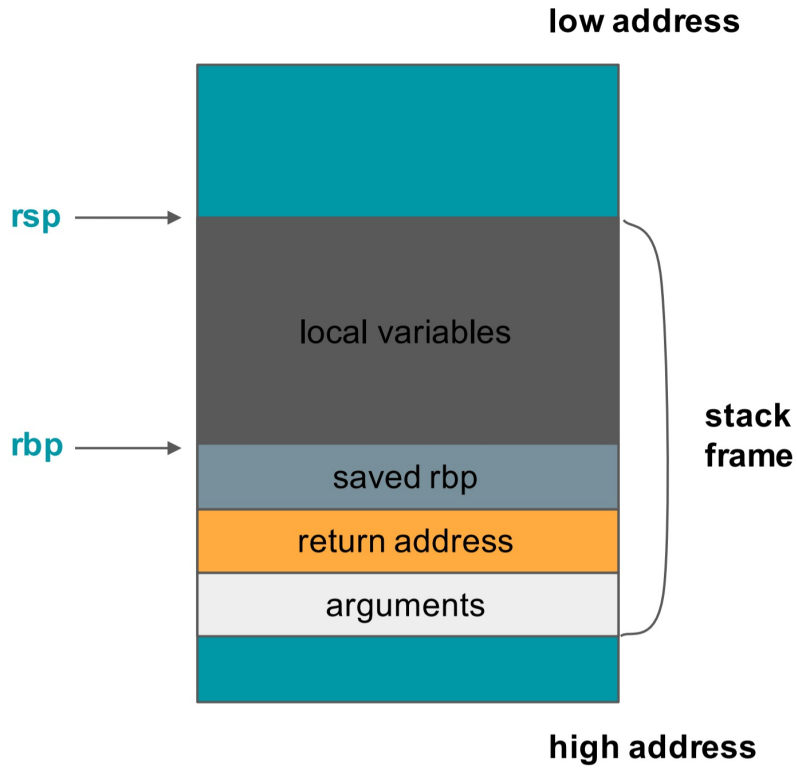
- AT&T (很醜，不要用)
- Intel (比較好看)
  - 用 objdump 時可以下 -M intel 來輸出此 Syntax

Registers

- x86 只有 32 bits，所以 R 開頭的只出現在 x64 架構上

64 bit	RAX	RBX	RCX	RDX	RSI	RDI	r8	r9	r10	r11	r12
32 bit	EAX	EBX	ECX	EDX	ESI	EDI	r8d	r9d	r10d		
16 bit	AX	BX	CX	DX	SI	DI	r8w	r9w	r10w		
08 bit							r8b	r9b	r10b		

- Stack Pointer Register
  - RSP (x64) / ESP(x86) - 指向 stack 頂端
- Base Pointer Register
  - RBP (x64) / EBP(x86) - 指向 stack 底端
- Program Counter Register
  - RIP (x64) / EIP(x86) - 指向目前前執行位置
- SP 到 function 參數範圍稱為該 function 的 Stack Frame



Calling Convention (function call)

function 參數傳遞

- x64 是先用 register 傳遞，register 用完則 push stack
  - register 順序為 rdi, rsi, rdx, rcx, r8, r9
- x86 則是直接 push stack 傳遞

Function prologue

- compiler 在 function 開頭加的指令
- 主要在保存 rbp 和分配區域變數所需空間。

```
push rbp
mov rbp, rsp
sub rsp, 0x50
```

Function epilogue

- compiler 在 function 結尾加的指令
- 主要在利用保存的 rbp 恢復 call function 前的 stack 狀態。

```
leave
ret
```

System call

- x64 使用 syscall，x86 則是使用 int 0x80 來呼叫 system call
- 而在呼叫之前會先在特定的 registers 設置好需要的參數，詳細如下
  - x64: http://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/
  - x86: https://syscalls.kernelgrok.com/

Basic instructions

lea v.s. mov

- mov reg/mem, imm/reg/mem
  - mov rax, [rsp+8] => rax = 0xdeadbeef，將值d rax
- lea reg/mem, imm/reg/mem
  - lea rax, [rsp+8] => rax = 0x7fffffffe4c8，將位置給 rax

add/sub/or/xor/and

- add/sub/or/xor/and reg, imm/reg

push/pop

- push/pop reg/mem
  - push rax == sub rsp, 8; mov [rsp], rax;
  - pop rdi = mov rdi, [rsp]; add rsp, 8;

jmp/call/ret/leave

- jmp 跳到程式某處
  - jmp A = mov rip, A
- call 儲存本將執行的下一行指令，再跳到程式某處
  - call A = push next_rip; jmp A
- ret 反回儲存位置
  - ret = pop rip
- leave 還原上一個 stack frame
  - leave = mov rsp, rbp; pop rbp

nop (no operation)

- 不做任何事，常常拿來 patch
  - 例如程式呼叫了我們不想要的某個函式 A，就可以把 call A 改成 nop
- 一個 byte，opcode = 0x90

Shellcode

- 顧名思義，攻擊者主要注入程式碼後的目的為拿到 shell ，故稱 shellcode
- 由一系列的 machine code 組成，最後目的可做任何攻擊者想做的事
- 用 pwntools 產生 shellcode
  - asm() - assembly
  - disasm() - disassembly

Lazy binding

- Dynamic linking 的程式在執行過程中，有些 library 的函式可能到結束都不會執行到，所以 ELF 採取 Lazy binding 的機制，在第一次 call library 函式時，才會去尋找函式真正的位置進行 binding。
- library 的位置再載入後才決定，因此無法在 compile 後，就知道 library 中的 function 在哪，該跳去哪。
- GOT 為一個函式指標陣列，儲存其他 library 中，function 的位置，但因 lazy binding 的機制，並不會一開始就把正確的位置填上，而是填 plt 位置的 code
- 當第一次執行到 library 的 function 時，會跳到 plt 去，plt 會去呼叫 _dl_fixup()，才會真正去尋找 function，最後再把 GOT 中的位置填上真正 function 的位置，這樣之後再 call 到這個 function 就有

offset 直接跳到 function 裡。

保護機制

ASLR (Address Space Layout Randomization)

- 看目標主機是否開啟，和程式無關
- 記憶體位置隨機變化
- 每次執行程式時，stack 、heap、library 位置都不一樣
- 查看是否有開啟 ASLR
  - cat /proc/sys/kernel/randomize_va_space
  - 0/1/2 - disable/only stack/for all

RELRO (RELocation Read Only)

- No RELRO - link map 和 GOT 都可寫
- Partial RELRO - link map 不可寫，GOT 可寫
- Full RELRO - link map 和 GOT 都不可寫
  - 會在 load time 時將全部 function resolve 完畢
  - No lazy binding

Canary

- 在 rbp 之前塞一個 random 值，在 ret 之前檢查那個 random 值有沒有被改變，有的話代表有 overflow，就讓程式 abort
- 主要防止 buffer overflow

NX (No eXecute)

- 又稱 DEP (Data Execution Prevention)
- 可寫的不可執行，可執行的不可寫

PIE (Position Independent Executable)

- 一般預設沒開啟的情況下程式的 data 段及 code 段會位置是固定的
- 但開啟之後 data 及 code 也會跟著 ASLR

Some tips

LD_PRELOAD

當本機的 libc 跟題目給的 libc 不一樣的時候，要用題目的 libc 跑程式，可以這樣下命令：`LD_PRELOAD=./libc.so.6 ./elf`

alarm -> isnan

很多題目會有 alarm，動態逆向的時候很煩，可以直接用文字編輯器打開 binary，把所有的 alarm 改成 isnan！