

Re:0 從零開始的逆向工程

2021/10/02

Presented by LJP

whoami

- LJP / LJP-TW
- 台科大 → 交大碩班
- CTF 戰隊 10sec 隊員



大綱

- 工具安裝
- 逆向工程簡介
- 組合語言
- 分析方法
- ELF 逆向工程
- 逆向工程技巧

工具安裝

工具安裝

- 在這一切開始之前
- 先讓我們準備一下工具吧
- 建議準備虛擬機，並記得將虛擬機斷網
- 畢竟研究病毒還是要關在實驗室裏面做，病毒跑出來會很慘 QQ

工具安裝

- 虛擬機準備這邊請自行研究
 - Windows (無限制版本, 建議 win7 以上)
 - Linux (無限制發行版, 推薦 Ubuntu / Kali)
- 本篇簡報範例沒有病毒, 可以直接運行在本機環境
- 但外面撿來的樣本, 還是請關在虛擬機運行

工具安裝

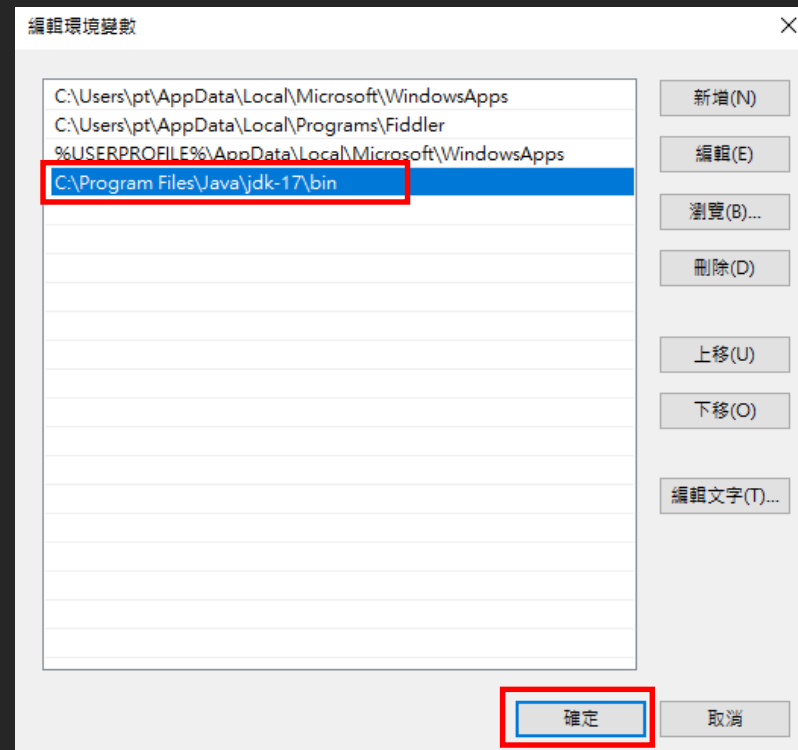
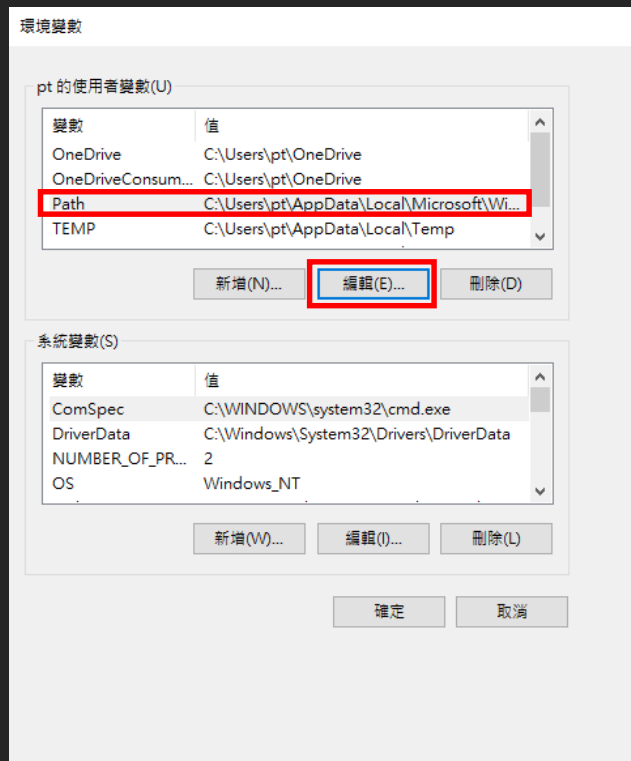
- 可安裝在本機上的
 - Ghidra
 - PEBear
- Windows 虛擬機
 - x64dbg
- Linux 虛擬機
 - gdb

JDK

- 安裝 Ghidra 前要先安裝 JDK
- jdk-17_windows-x64_bin.exe
- 一直按下一步就對了
- 設置環境變數



JDK



Ghidra

- ghidra_10.0.3_PUBLIC_20210908.zip
- 解壓縮
- JDK 環境變數有設定好, 就能成功運行 ghidraRun.bat

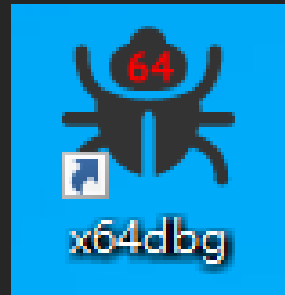
名稱	修改日期	類型	大小
docs	2020/12/29 下午 05:22	檔案資料夾	
Extensions	2020/12/29 下午 05:22	檔案資料夾	
Ghidra	2020/12/29 下午 05:22	檔案資料夾	
GPL	2020/12/29 下午 05:22	檔案資料夾	
licenses	2020/12/29 下午 05:22	檔案資料夾	
server	2020/12/29 下午 05:22	檔案資料夾	
support	2020/12/29 下午 05:22	檔案資料夾	
ghidraRun	2020/12/29 下午 05:22	檔案	1 KB
ghidraRun.bat	2020/12/29 下午 05:22	Windows 批次檔案	1 KB
LICENSE	2020/12/29 下午 05:22	檔案	12 KB

PE-Bear

- PE-bear_0.5.4_x64_win_vs17.zip
- 解壓縮
- 執行 PE-bear.exe 即可

x64dbg

- snapshot_2021-07-01_23-17
- 解壓縮
- 執行 release/x96dbg.exe
- 一開始初始化選項 全部選確定
- 桌面就會出現 x32dbg 和 x64dbg 的捷徑



gdb

- 開啟終端機執行以下指令
- `sudo apt-get install gdb`
- 建議額外安裝 `gdb-gef` 套件
- 參考 <https://github.com/hugsy/gef> 文中敘述的安裝方式

```
# or manually
$ wget -O ~/.gdbinit-gef.py -q http://gef.blah.cat/py
$ echo source ~/.gdbinit-gef.py >> ~/.gdbinit
```

gdb

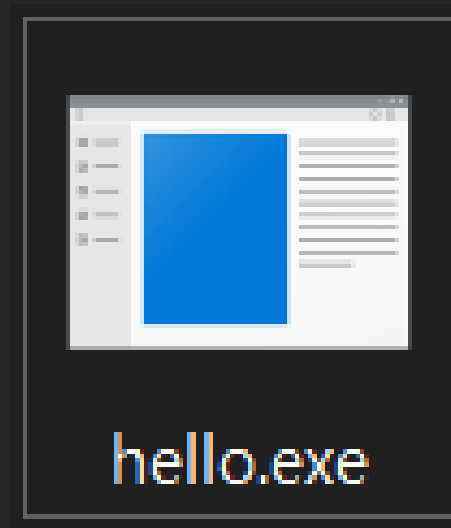
- gef config 可以參考以下
- <https://gist.github.com/LJP-TW/2edf8b66b61e91a232f76acc487bbd10>
- 請先註解掉 glibc heap 設定的部分
- 主要保留以下的部分

```
1  source ~/.gdbinit-gef.py
2  # source ~/peda/peda.py
3
4  set print asm-demangle on
5  set auto-load safe-path /
6  # set debug-file-directory ~/gdb-debug-symbol
7
8  ####
9  # gef setting
10
11  gef config dereference.max_recursion 2
12  gef config context.layout "regs code args source memory stack trace"
13  gef config context.nb_lines_backtrace 3
14
15  gef config context.redirect /dev/pts/4
```

逆向工程簡介

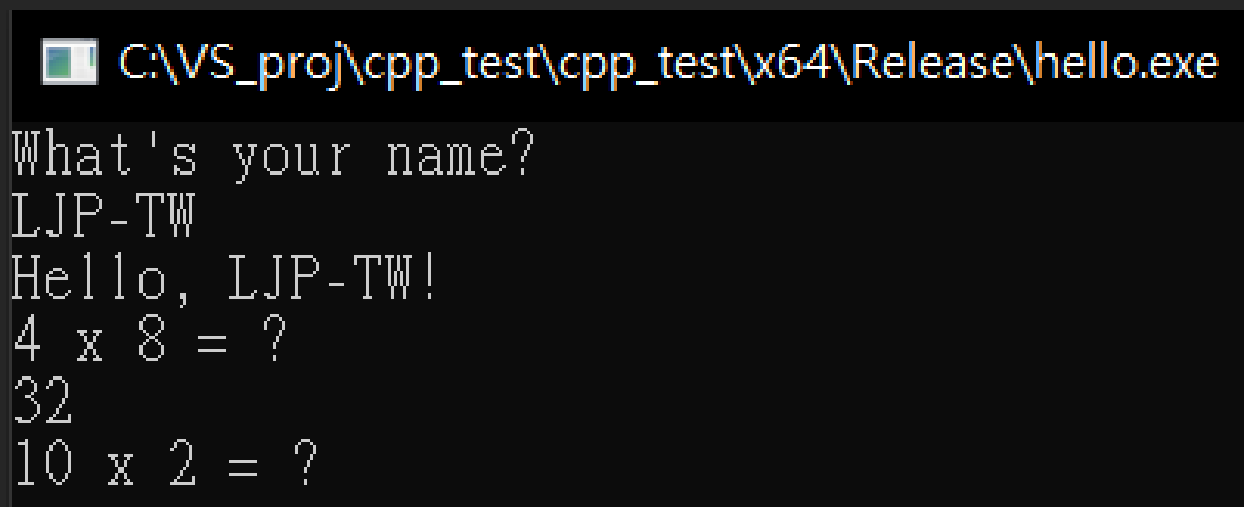
逆向工程簡介

- 逆向工程 Reverse Engineering
- 如果今天你想知道一個程式在做什麼，要怎麼做？



逆向工程簡介

- 直接執行看看?



```
C:\VS_proj\cpp_test\cpp_test\x64\Release\hello.exe
What's your name?
LJP-TW
Hello, LJP-TW!
4 x 8 = ?
32
10 x 2 = ?
```

逆向工程簡介

- 如果你剛好有原始碼，那就看 code 就好

```
int main(void)
{
    char name[32] = { 0 };
    int a, b, ans;

    srand(time(NULL));

    printf("What's your name?\n");

    scanf("%32s", name);

    printf("Hello, %s!\n", name);
}
```

```
for (int i = 0; i < 5; ++i) {
    a = (rand() % 10) + 1;
    b = (rand() % 10) + 1;

    printf("%d x %d = ?\n", a, b);

    scanf("%d", &ans);

    if (ans != a * b) {
        printf("Wrong!\n");
        evilcatboy();
    }
}

return 0;
}
```

逆向工程簡介

- 那如果沒有 source code 呢?

逆向工程簡介

- 當你想…
 - 破解程式
 - 修改程式
 - 分析惡意程式
 - 挖掘漏洞
- 卻又沒有原始碼
- 你就需要逆向工程！

想知道程式到底有沒有在
偷挖礦



逆向工程簡介

- 沒有原始碼，要怎麼知道程式在幹嘛？
- 先轉換一下問題
- 程式怎麼產生的？
- 程式怎麼跑起來的？

程式產生過程

- Q: 程式怎麼產生的?
- Visual Studio 按一下 F5, 程式碼就變程式了(ノ_>、)
- 喂, 我是問更詳細的過程

程式產生過程

- 首先程式碼經過編譯器，經過解析後，產生出組合語言
 - 關於編譯器是怎麼解析的，我們以後會專門做一期視頻給大家講解

```
int main(void)
{
    // blablabla...
}
```

原始程式碼



編譯器
Compiler



```
$LL4@main:
; Line 25
    call    QWORD PTR __imp_rand
    mov     edi, eax
    mov     eax, 1717986919
    imul    edi
    sar     edx, 2
    mov     ecx, edx
    shr     ecx, 31
    add     edx, ecx
    lea     ecx, DWORD PTR [rdx+rdx*4]
    add     ecx, ecx
    sub     edi, ecx
    inc     edi
```

組合語言

程式產生過程

- 組合語言再通過組譯器，將組合語言組譯成 object code

```
$LL4@main:  
; Line 25  
    call    QWORD PTR __imp_rand  
    mov     edi, eax  
    mov     eax, 1717986919  
    imul    edi  
    sar     edx, 2  
    mov     ecx, edx  
    shr     ecx, 31  
    add     edx, ecx  
    lea     ecx, DWORD PTR [rdx+rdx*4]  
    add     ecx, ecx  
    sub     edi, ecx  
    inc     edi
```

組合語言



組譯器
Assembler



object code

程式產生過程

- Object code 再通過連結器，最終連結成執行檔



程式產生過程

- 統整一下

```
int main(void)
{
    // blablabla...
}
```

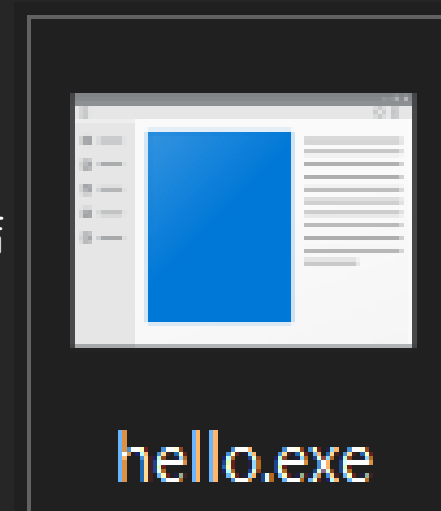
編譯

```
$LL4@main:
; Line 25
    call    QWORD PTR __imp_rand
    mov     edi, eax
    mov     eax, 1717986919
    imul    edi
    sar     edx, 2
    mov     ecx, edx
    shr     ecx, 31
    add     edx, ecx
    lea     ecx, DWORD PTR [rdx+rdx*4]
    add     ecx, ecx
    sub     edi, ecx
    inc     edi
```

組譯



連結



原始程式碼

組合語言

object code

執行檔

程式產生過程

- 整個過程，程式碼越來越不適合人類閱讀
- 從人類的語言一路慢慢變成一堆 0 跟 1

```
int main(void)
{
    // blablabla...
}
```

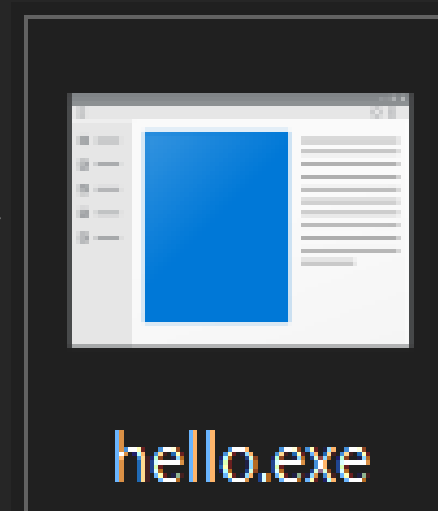
編譯

```
$LL4@main:
; Line 25
    call    QWORD PTR __imp_rand
    mov     edi, eax
    mov     eax, 1717986919
    imul    edi
    sar     edx, 2
    mov     ecx, edx
    shr     ecx, 31
    add     edx, ecx
    lea     ecx, DWORD PTR [rdx+rdx*4]
    add     ecx, ecx
    sub     edi, ecx
    inc     edi
```

組譯



連結



原始程式碼

組合語言

object code

執行檔

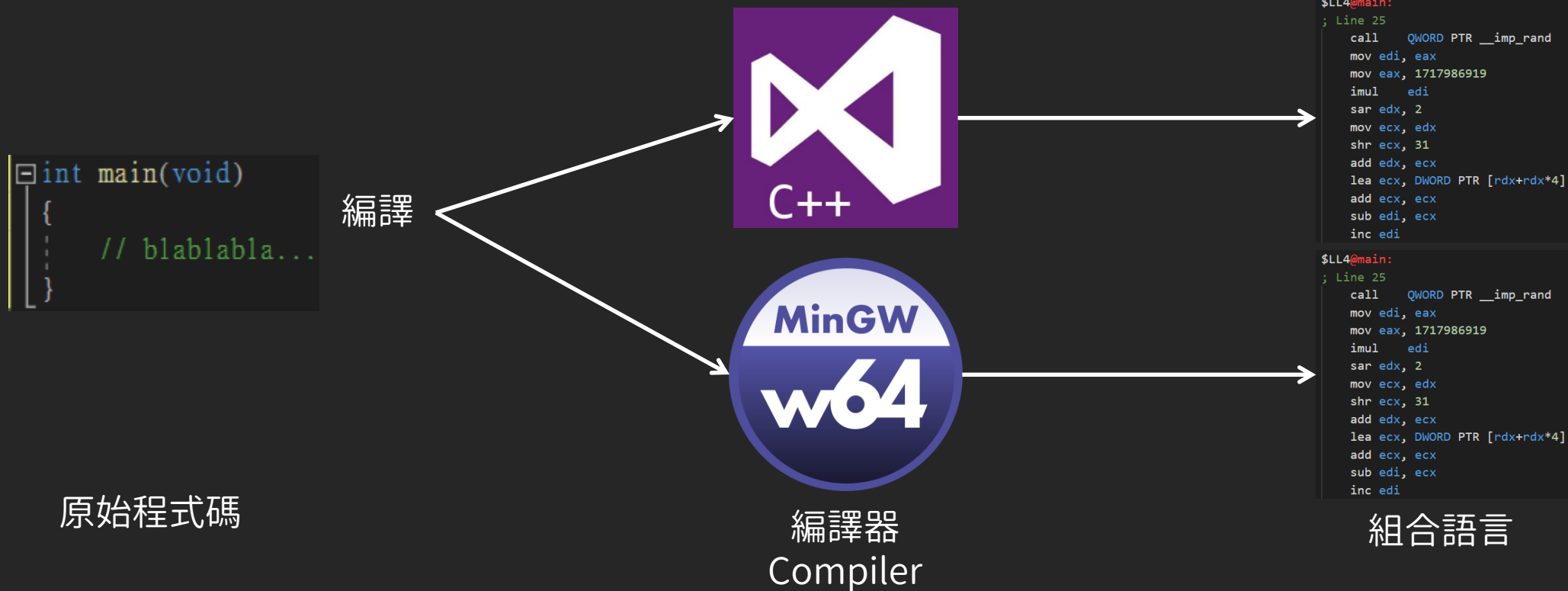
編譯/反編譯

- 來談一下編譯與反編譯



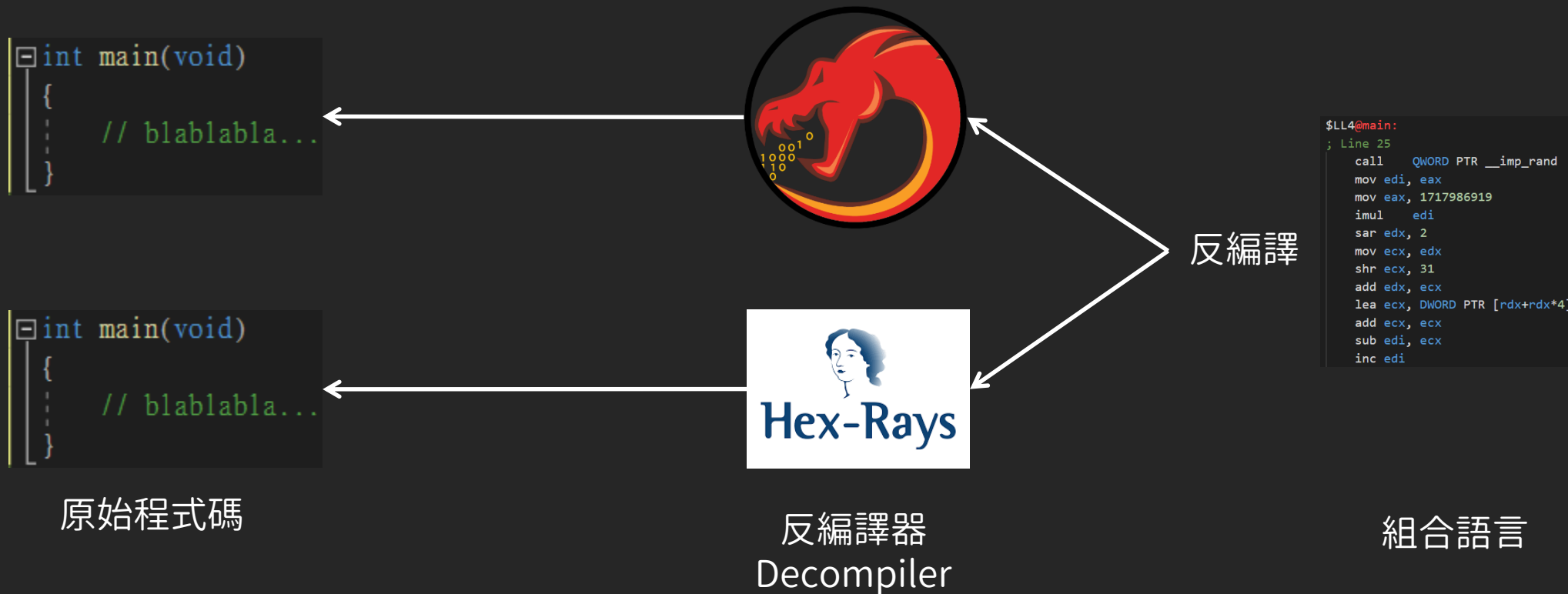
編譯/反編譯

- 程式碼編譯成組合語言，每種編譯器都不盡相同
- 同一句 C 可以用多種組合語言表達



編譯/反編譯

- 因此反編譯沒有那麼簡單
- 雖然程式邏輯一樣，但程式碼長相始終無法與原始程式碼一樣



組譯/反組譯

- 再來談一下組譯與反組譯

```
int main(void)
{
    // blablabla...
}
```

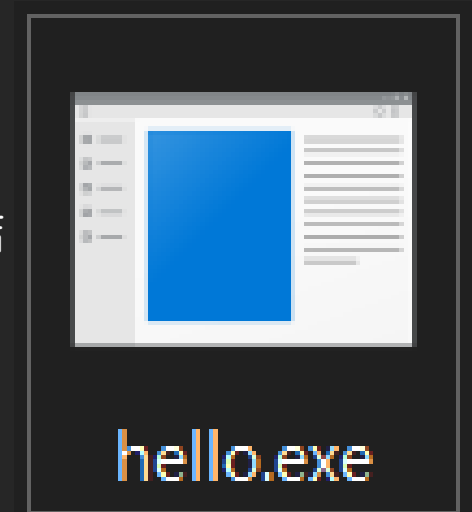
編譯

```
$LL4@main:
; Line 25
    call    QWORD PTR __imp_rand
    mov     edi, eax
    mov     eax, 1717986919
    imul    edi
    sar     edx, 2
    mov     ecx, edx
    shr     ecx, 31
    add     edx, ecx
    lea     ecx, DWORD PTR [rdx+rdx*4]
    add     ecx, ecx
    sub     edi, ecx
    inc     edi
```

組譯



連結



原始程式碼

組合語言

object code

執行檔

組譯/反組譯

- 每一句組合語言都只和一組機械碼互相對應
- 比如說 `mov rax, rbx` 翻成機械碼就是 `0x48 0x89 0xd8`
- `0x48 0x89 0xd8` 翻成組合語言也只有 `mov rax, rbx` 這種可能
- 所以反組譯器相對來說比較好做

組譯/反組譯

- 每一句組合語言都只和一組機械碼互相對應

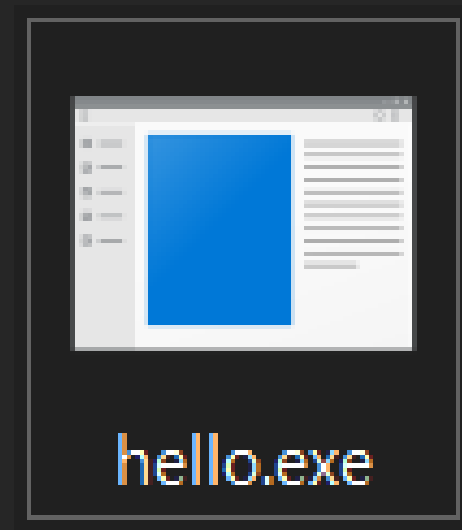
```
$LL4@main:  
; Line 25  
call    QWORD PTR __imp_rand  
mov edi, eax  
mov eax, 1717986919  
imul    edi  
sar edx, 2  
mov ecx, edx  
shr ecx, 31  
add edx, ecx  
lea ecx, DWORD PTR [rdx+rdx*4]  
add ecx, ecx  
sub edi, ecx  
inc edi
```

組合語言



反組譯器
Disassembler

反組譯



hello.exe

執行檔

程式產生過程

- 溫馨提醒：不同語言的機制可能不太一樣
 - C / C++
 - C#
 - Java
 - Python
 - ...
- 本篇簡報只針對 C / C++

程式產生過程

- 知道了程式的產生過程後，你多少應該能知道逆向工程的原理
- 通過反組譯，將人類看不懂的 0101 變成看得懂的組合語言，就能知道程式在執行什麼
- 有些工具提供反編譯功能，能更有效的看懂程式在幹嘛
- ~~當然，如果有 source code 最好~~

程式運作過程

- Q: 程式怎麼跑起來的?
- 對著 exe 點兩下他就跑起來了 (ノ_>、)
- 喂，我是問更詳細的過程

程式運作過程

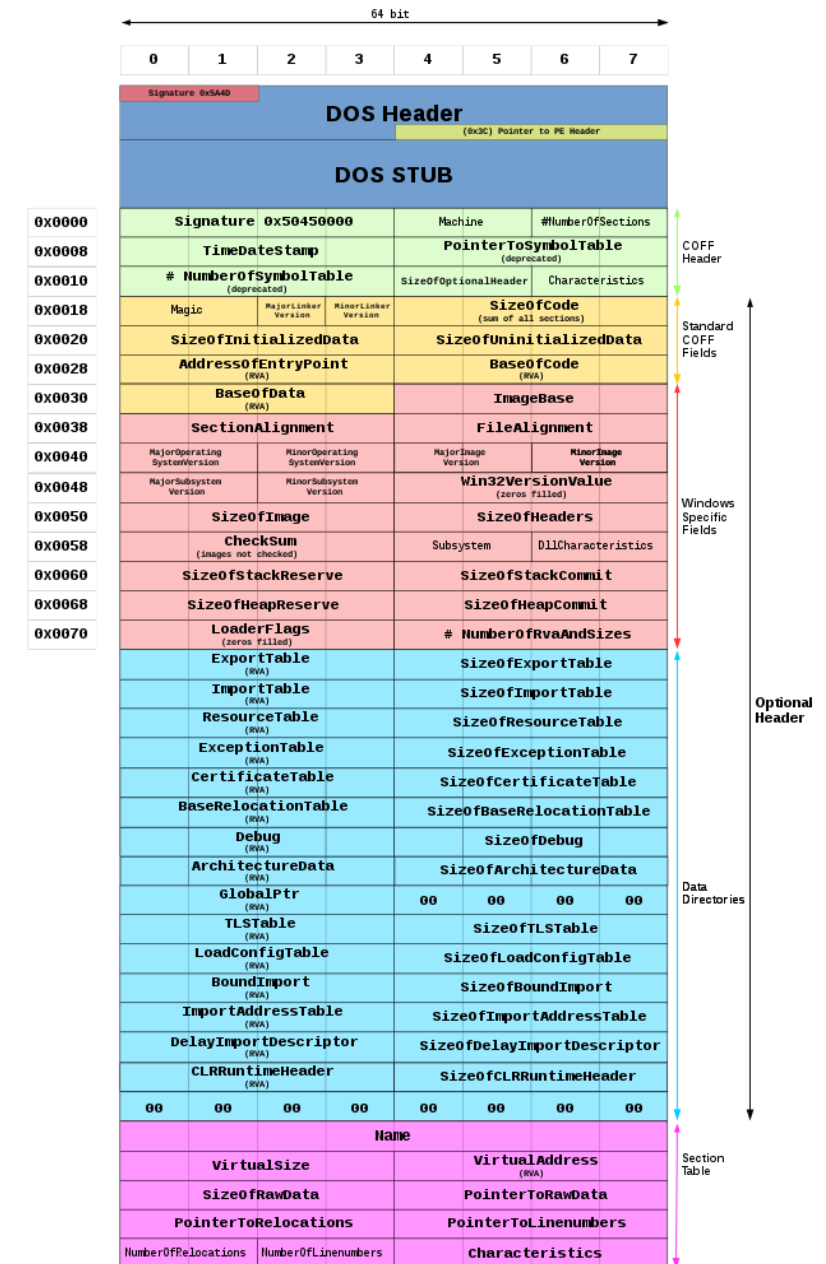
- 你點兩下程式，告訴作業系統 (OS) 你想執行他
- 所以問題更具體地問是，OS 怎麼將程式跑起來的？
- OS 會從程式檔案的頭部讀取資訊
- 這些資訊包含怎麼把它放到記憶體裡、程式進入點在哪...

程式運作過程

- 不同 OS 是如何載入程式的方式大同小異
- 頭部結構
 - Windows: PE (Portable Executable) Header
 - Linux: ELF (Executable and Linkable Format)
- 載入後，就從程式進入點開始執行
- 先帶大家看看熟悉的 exe 內部結構

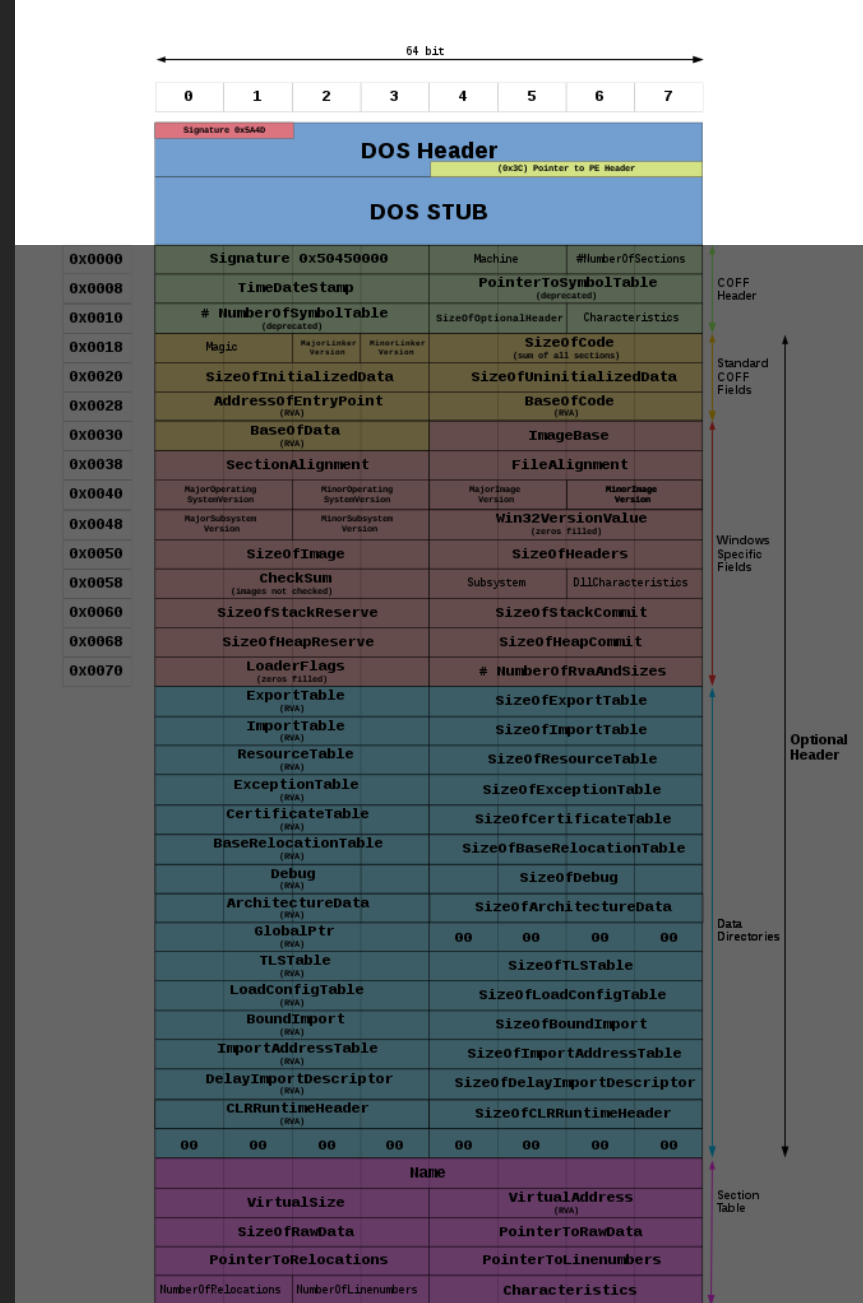
PE format

- 這就是你熟悉的 exe 的結構
- 是不是看了就頭痛
- 讓我們一點一點地拆開來說



PE format

- 首先看最上面的 DOS Header
- 用 PE-Bear 來展示一下



PE format

PE-bear v0.5.4 [C:/Users/pt/Downloads/hello.exe]

File Settings View Compare Info

hello.exe

- DOS Header
- DOS stub
- NT Headers
 - Signature
 - File Header
 - Optional Header
- Section Headers
 - .text
 - EP = 980
 - .rdata
 - .data
 - .pdata
 - .rsrc
 - .reloc

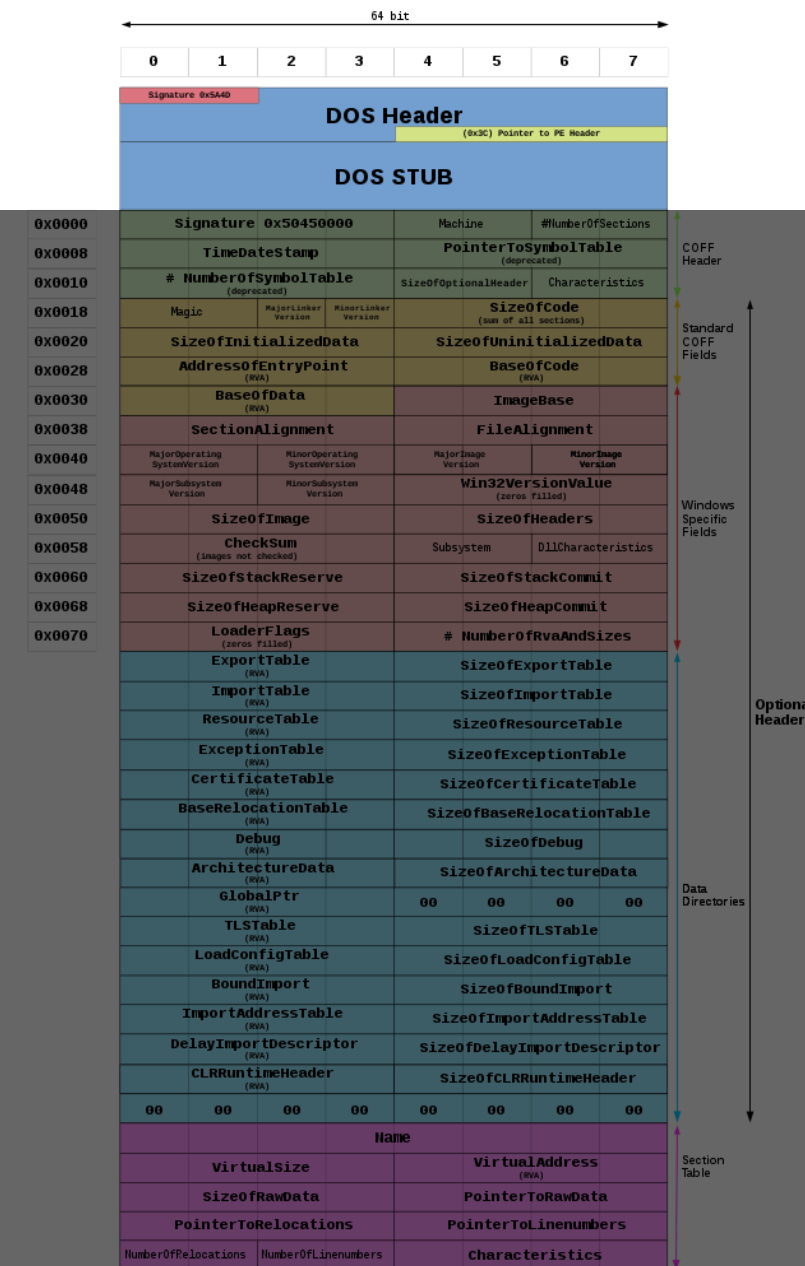
Disasm General DOS Hdr Rich Hdr File Hdr Optional Hdr Section Hdrs

Offset	Name	Value
0	Magic number	5A4D
2	Bytes on last page of file	90
4	Pages in file	3
6	Relocations	0
8	Size of header in paragraphs	4
A	Minimum extra paragraphs needed	0
C	Maximum extra paragraphs needed	FFFF
E	Initial (relative) SS value	0
10	Initial SP value	B8
12	Checksum	0
14	Initial IP value	0
16	Initial (relative) CS value	0
18	File address of relocation table	40
1A	Overlay number	0
1C	Reserved words[4]	0, 0, 0, 0
24	OEM identifier (for OEM information)	0
26	OEM information; OEM identifier specific	0
28	Reserved words[10]	0, 0, 0, 0, 0, 0, 0, 0, 0, 0
3C	File address of new exe header	100

Check for updates

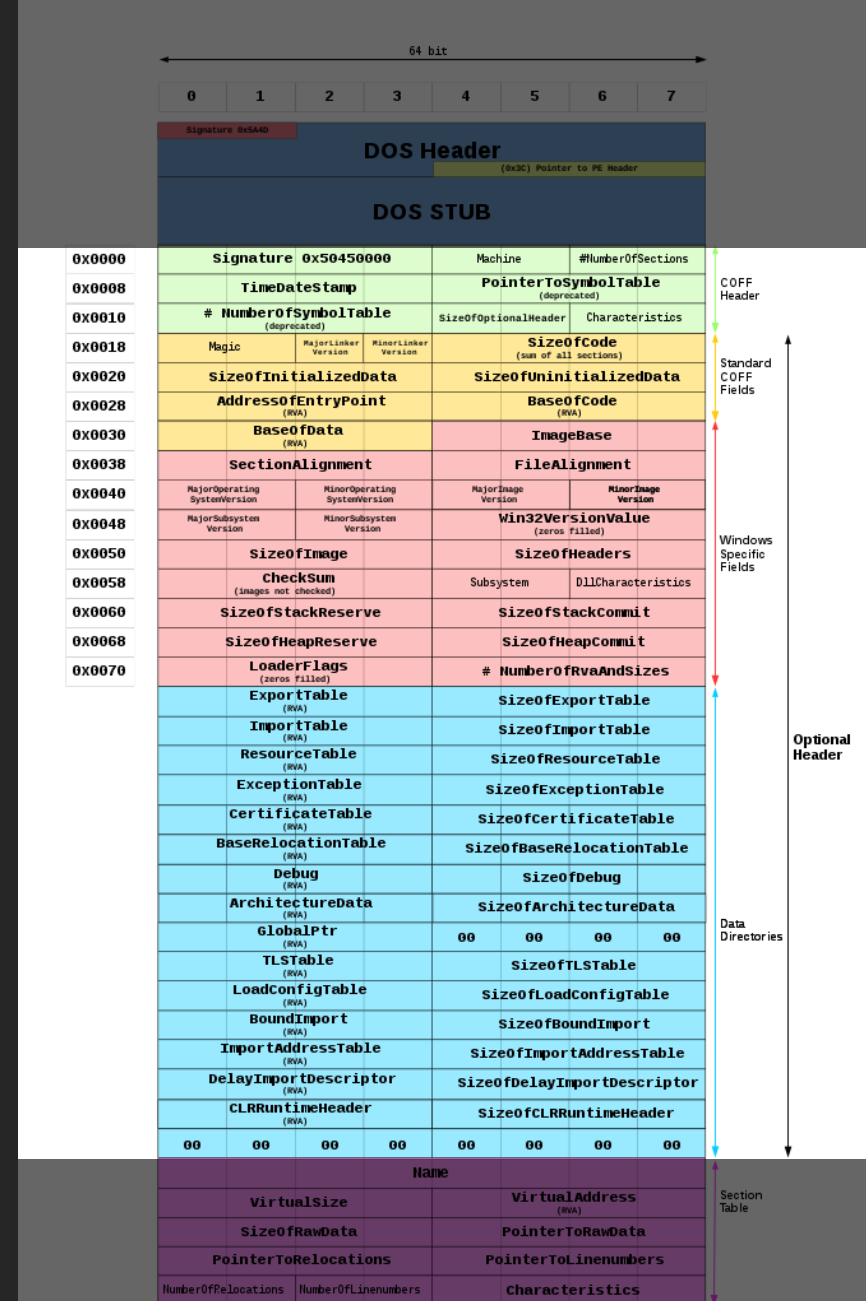
DOS Hdr 以 “MZ” 作為開頭

紀錄 NT Hdr 偏移量(offset)多少



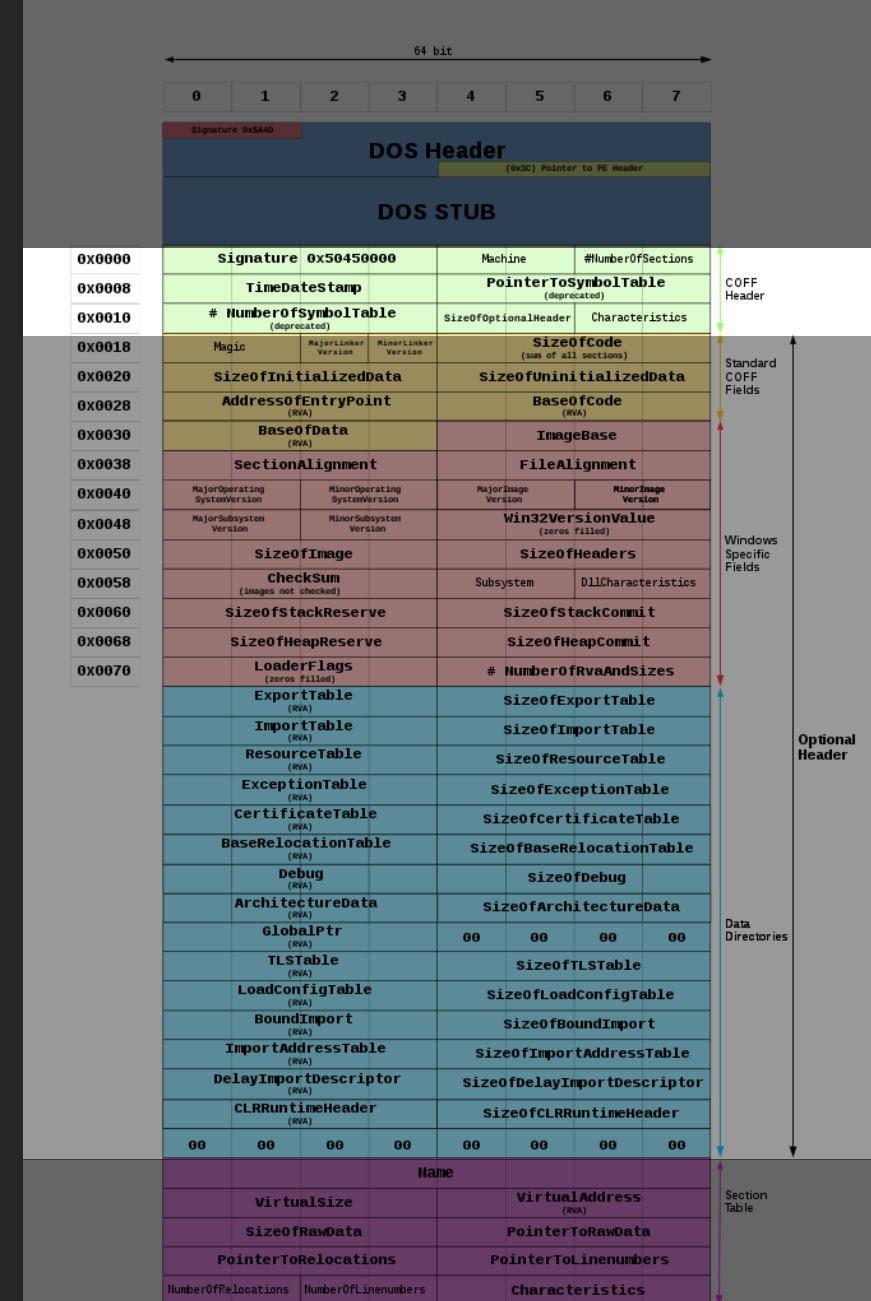
PE format

- 再來是 NT Hdr (或稱 PE Hdr)
- 其包含了



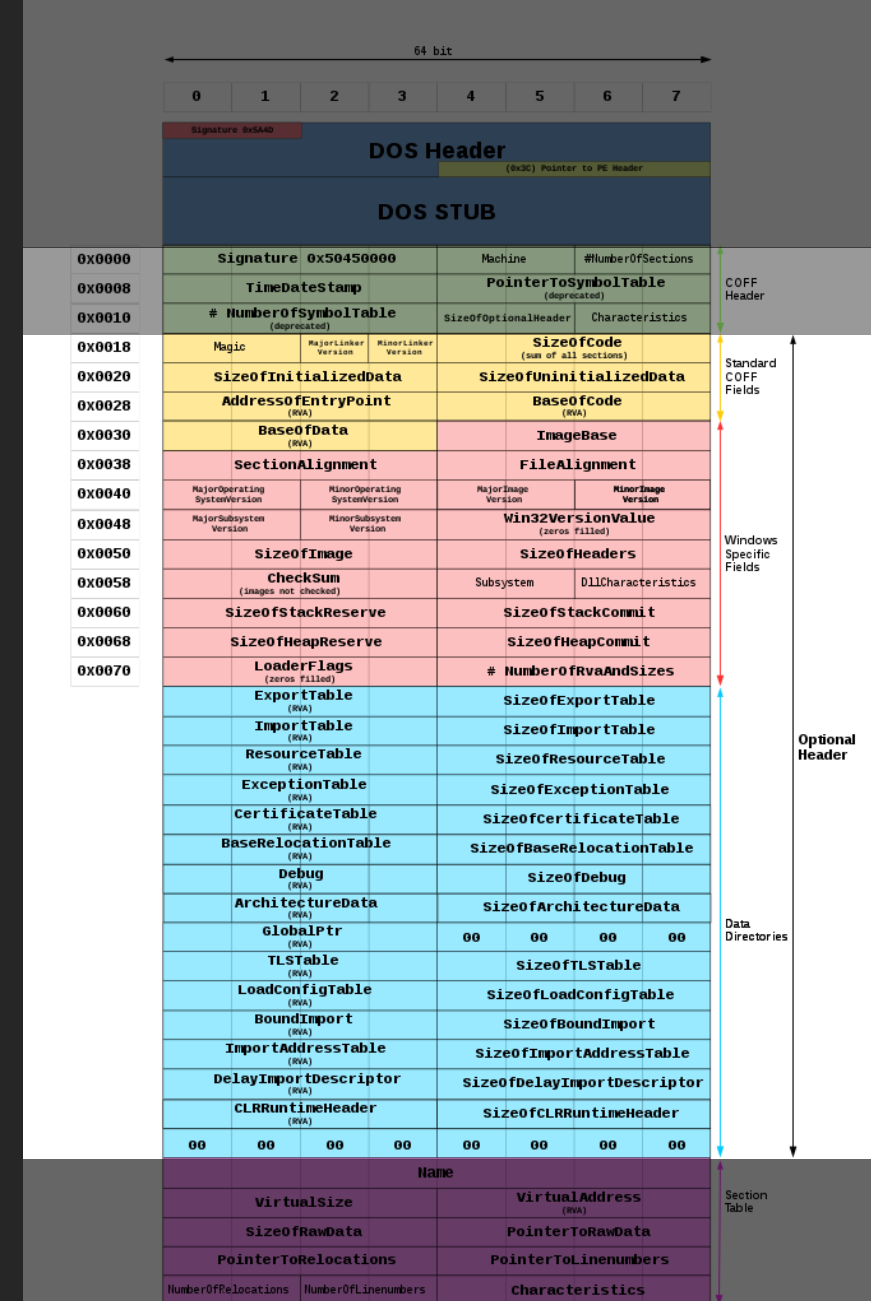
PE format

- 再來是 NT Hdr (或稱 PE Hdr)
- 其包含了
 - COFF Hdr (或稱 File Hdr)



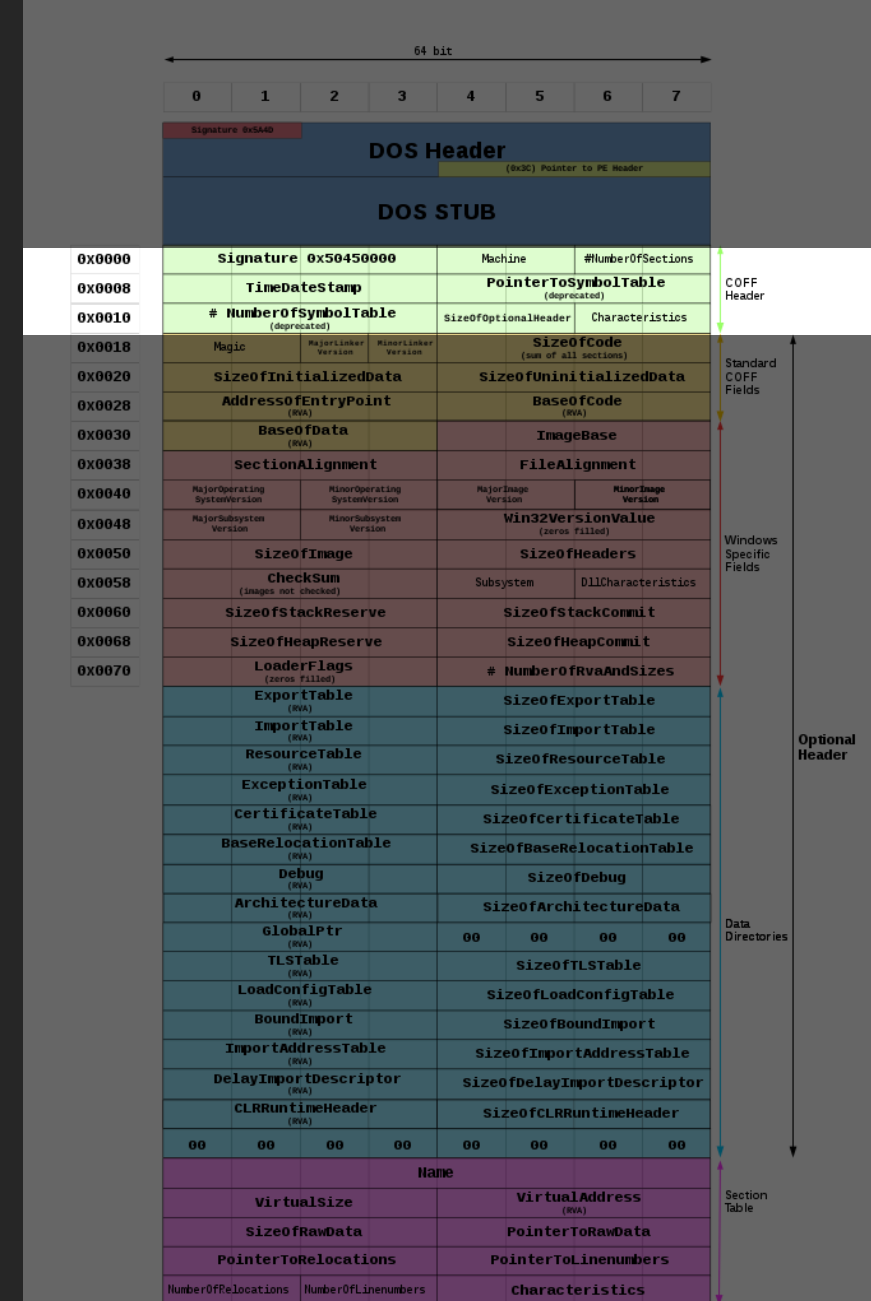
PE format

- 再來是 NT Hdr (或稱 PE Hdr)
- 其包含了
 - COFF Hdr (或稱 File Hdr)
 - Optional Hdr



PE format

- COFF Hdr (或稱 File Hdr)
- 用 PE-Bear 來展示一下



PE format

PE-bear v0.5.4 [C:/Users/pt/Downloads/hello.exe]

File Settings View Compare Info

hello.exe

- DOS Header
- DOS stub
- NT Headers
 - Signature
 - File Header
 - Optional Header
 - Section Headers
- Sections
 - .text (EP = 980)
 - .rdata
 - .data
 - .pdata
 - .rsrc
 - .reloc

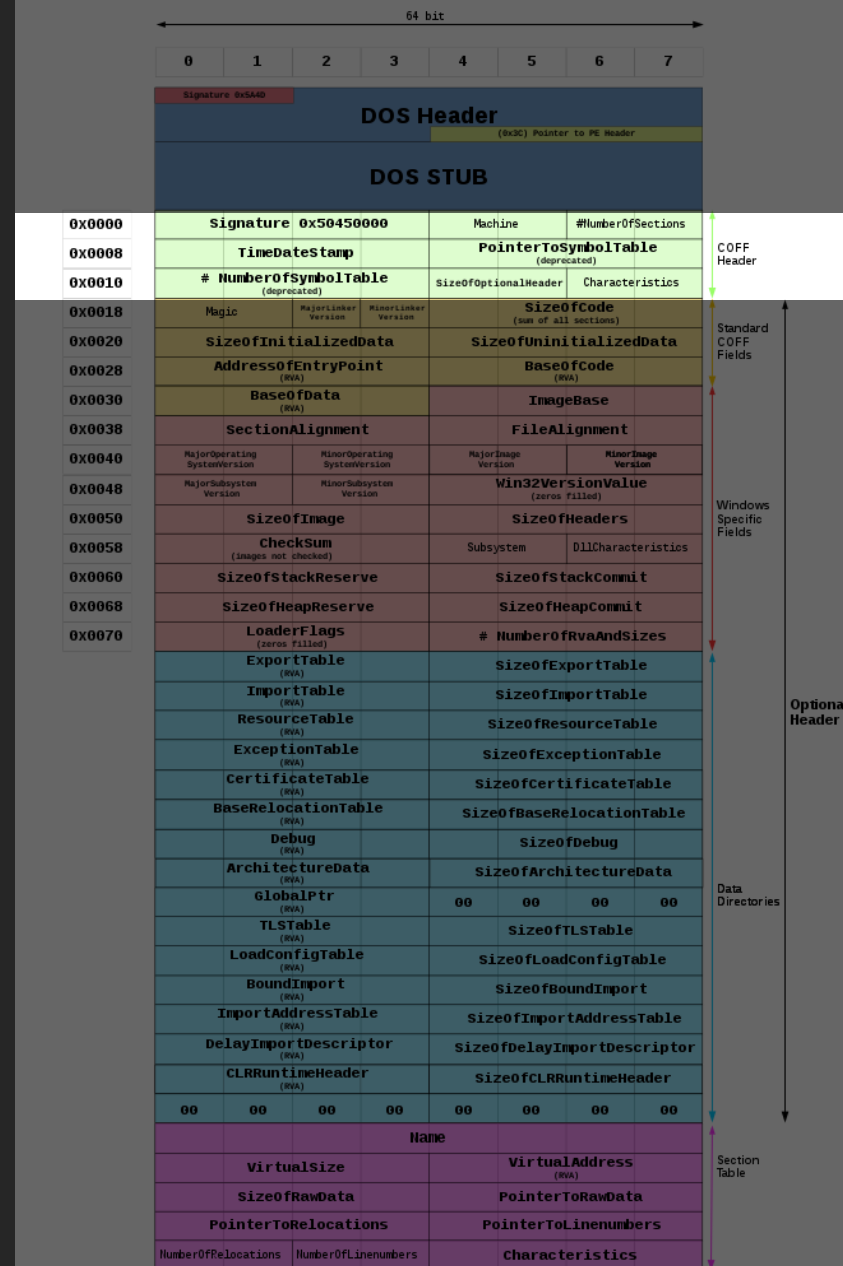
PE Hdr 以 "PE" 開頭

紀錄有幾個 sections

紀錄 Optional Hdr 多大

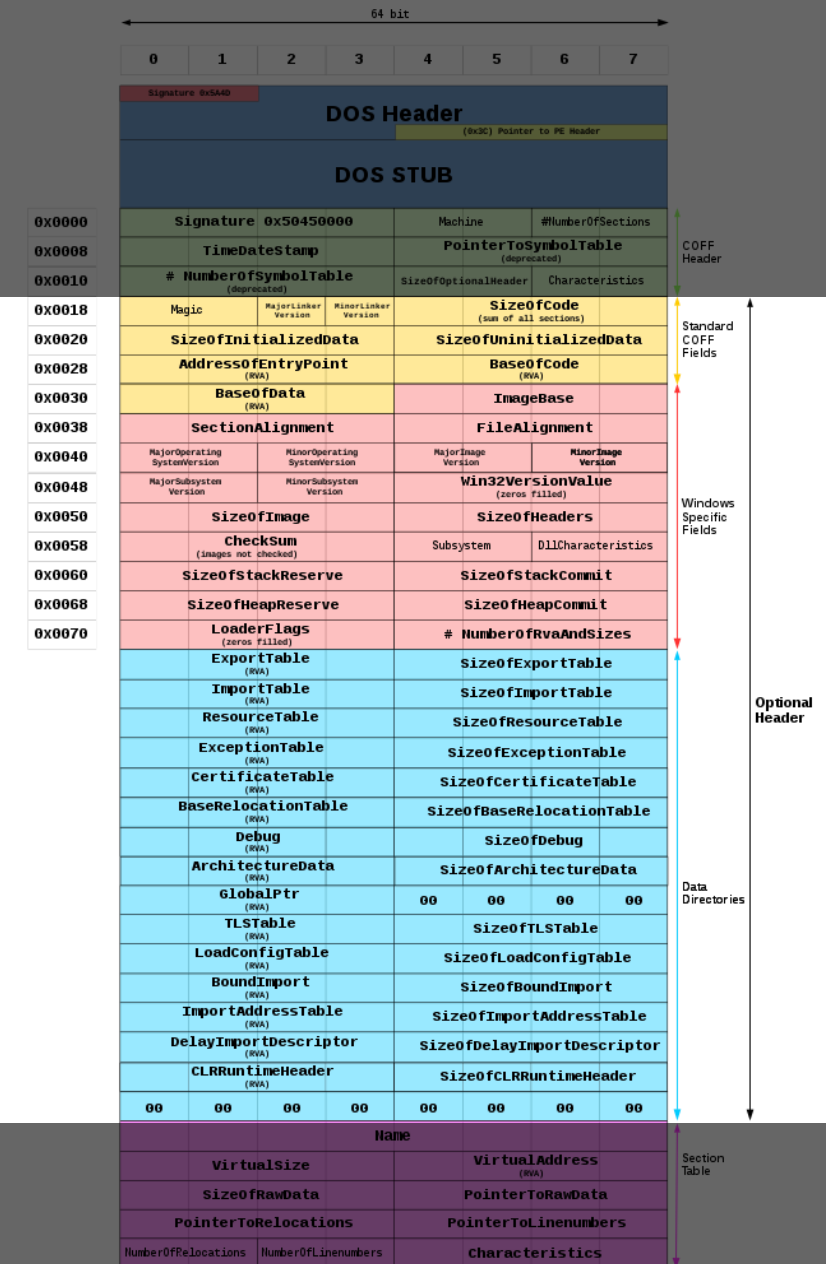
紀錄有哪些特殊設定

Offset	Name	Value	Meaning
104	Machine	8664	AMD64 (K8)
106	Sections Count	6	6
108	Time Date Stamp	6142bfd9	星期四, 16.09.2021 03:54:01 UTC
10C	Ptr to Symbol Table	0	0
110	Num. of Symbols	0	0
114	Size of OptionalHeader	f0	240
116	Characteristics	22	File is executable (i.e. no unresolved external r... App can handle >2gb addresses



PE format

- Optional Hdr
- 用 PE-Bear 來展示一下



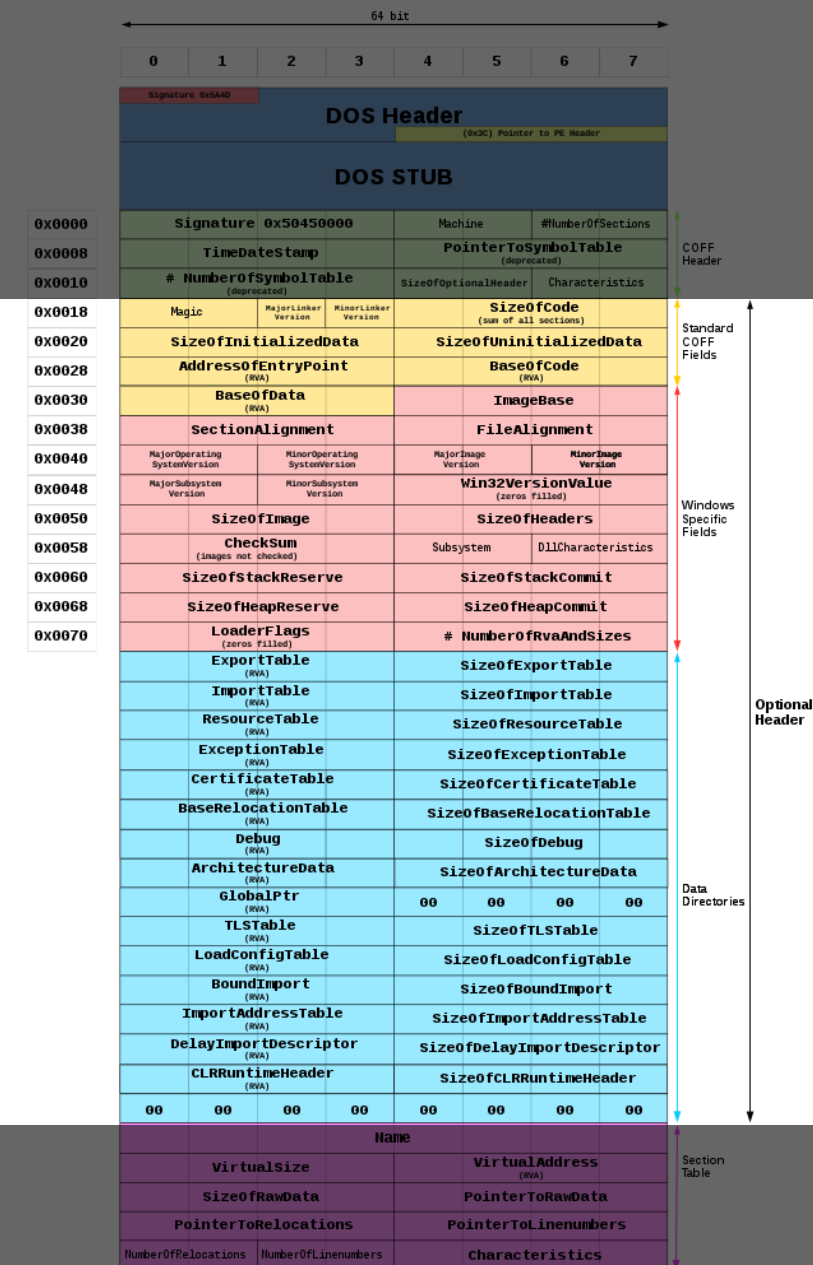
PE format

Disasm	General	DOS Hdr	Rich Hdr	File Hdr	Optional Hdr	Section Hdrs
Offset	Name	Value	Value			
118	Magic	20B	NT64			
11A	Linker Ver. (Major)	E				
11B	Linker Ver. (Minor)	1D				
11C	Size of Code	1200				
120	Size of Initialized Data	2000				
124	Size of Uninitialized Data	0				
128	Entry Point	1580				
12C	Base of Code	1000				
130	Image Base	14000000				
138	Section Alignment	1000				
13C	File Alignment	200				
140	OS Ver. (Major)	6	Windows Vista / Server 2008			
142	OS Ver. (Minor)	0				
144	Image Ver. (Major)	0				
146	Image Ver. (Minor)	0				
148	Subsystem Ver. (Major)	6				
14A	Subsystem Ver. Minor)	0				
14C	Win32 Version Value	0				
150	Size of Image	9000				
154	Size of Headers	400				
158	Checksum	0				
15C	Subsystem	3	Windows console			
15E	DLL Characteristics	8160				
		40	DLL can move			
		100	Image is NX compatible			
		8000	TerminalServer aware			

程式進入點 RVA

程式基址

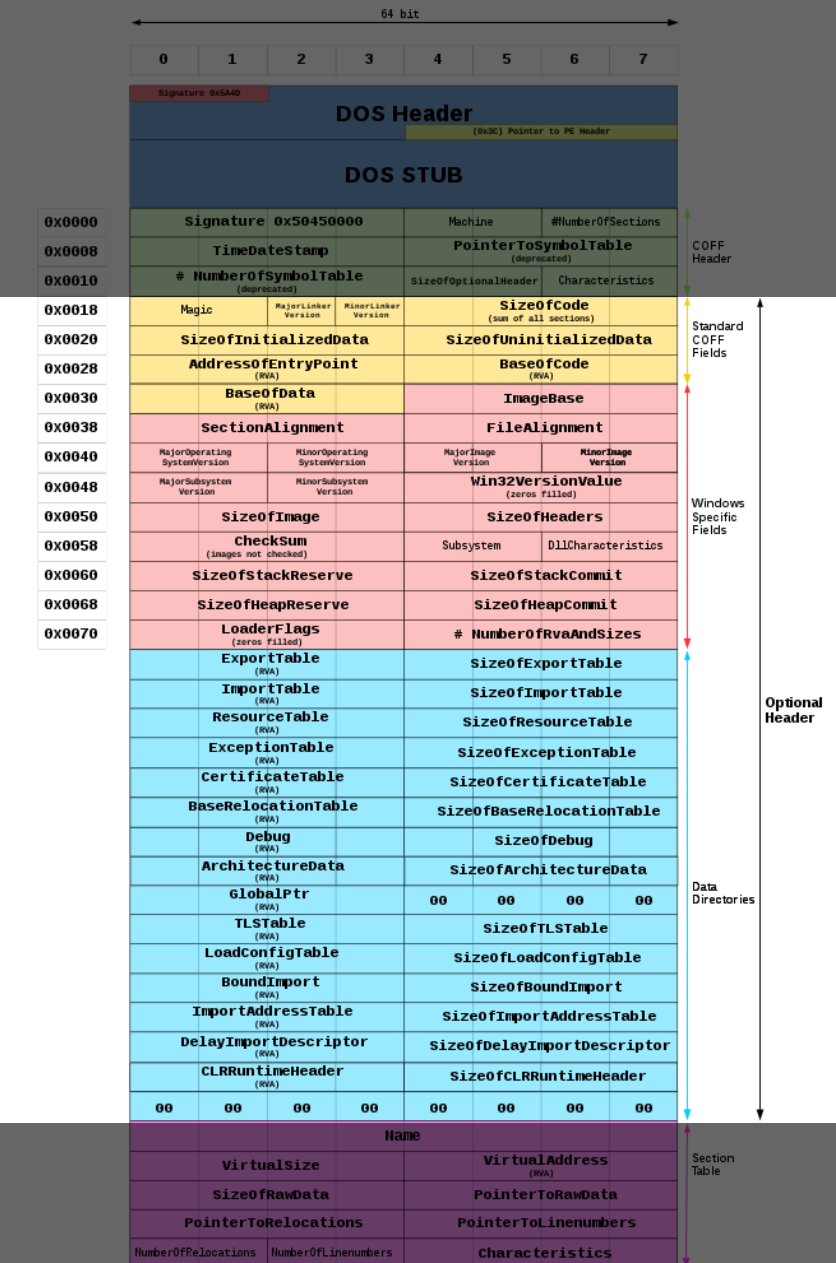
紀錄有哪些特殊設定



PE format

160	Size of Stack Reserve	100000
168	Size of Stack Commit	1000
170	Size of Heap Reserve	100000
178	Size of Heap Commit	1000
180	Loader Flags	0
184	Number of RVAs and Sizes	10
Data Directory		
	Address	Size
188	Export Directory	0
190	Import Directory	396C
198	Resource Directory	7000
1A0	Exception Directory	6000
1A8	Security Directory	0
1B0	Base Relocation Table	8000
1B8	Debug Directory	3310
1C0	Architecture Specific Data	0
1C8	RVA of GlobalPtr	0
1D0	TLS Directory	0
1D8	Load Configuration Directory	3380
1E0	Bound Import Directory in headers	0
1E8	Import Address Table	3000
1F0	Delay Load Import Descriptors	0
1F8	.NET header	0

各種 Directory 位址及大小



PE format

- 解釋一下 RVA (Relative Virtual Address)
- 首先先直接展示一個程式在跑的時候，記憶體位址的樣子

PE format

位址	大小	資訊	內容	類型	保護	初始保護
0000002DAA5A4000	0000000000005C000	Reserved (0000002DA		PRV		-RW--
0000002DAA600000	000000000000FC000	Reserved		PRV		-RW--
0000002DAA6FC000	0000000000004000			PRV	-RW-G	-RW--
000001FAD43A0000	00000000000010000			MAP	-RW--	-RW--
000001FAD43B0000	0000000000001000	\Device\HarddiskVo1		MAP	-R---	-R---
000001FAD43C0000	0000000000001D000			MAP	-R---	-R---
000001FAD43E0000	0000000000004000			MAP	-R---	-R---
000001FAD43F0000	0000000000001000			MAP	-R---	-R---
000001FAD4400000	0000000000002000			PRV	-RW--	-RW--
000001FAD4410000	000000000000C9000	\Device\HarddiskVo1		MAP	-R---	-R---
000001FAD44E0000	0000000000001000			MAP	-R---	-R---
000001FAD44F0000	00000000000010000			PRV	-RW--	-RW--
000001FAD4500000	000000000000F0000	Reserved (000001FAD		PRV		-RW--
000001FAD45F0000	0000000000001000			MAP	-R---	-R---
000001FAD4600000	0000000000001000			MAP	-R---	-R---
00007FF4F0140000	0000000000005000			MAP	-R---	-R---
00007FF4F0145000	000000000000FB000	Reserved (00007FF4F		MAP		-R---
00007FF4F0240000	0000000100020000	Reserved		PRV		-RW--
00007FF5F0260000	0000000002000000	Reserved		PRV		-RW--
00007FF5F2260000	0000000000001000			PRV	-RW--	-RW--
00007FF5F2270000	0000000000001000			MAP	-R---	-R---
00007FF5F2280000	00000000000033000			MAP	-R---	-R---
00007FF676150000	0000000000001000	hello.exe		IMG	-R---	ERWC-
00007FF676151000	0000000000002000	".text"	可執行代碼	IMG	ER---	ERWC-
00007FF676153000	0000000000002000	".rdata"	唯讀的已初始化的資料	IMG	-R---	ERWC-
00007FF676155000	0000000000001000	".data"	已初始化的資料	IMG	-RW--	ERWC-
00007FF676156000	0000000000001000	".pdata"	例外資料	IMG	-R---	ERWC-
00007FF676157000	0000000000001000	".rsrc"	資源	IMG	-R---	ERWC-
00007FF676158000	0000000000001000	".reloc"	Base relocations	IMG	-R---	ERWC-
00007FFB29680000	0000000000001000	vcruntime140.dll		IMG	-R---	ERWC-
00007FFB29681000	00000000000010000	".text"	可執行代碼	IMG	ER---	ERWC-
00007FFB29691000	0000000000004000	".rdata"	唯讀的已初始化的資料	IMG	-R---	ERWC-
00007FFB29695000	0000000000001000	".data"	已初始化的資料	IMG	-RW--	ERWC-
00007FFB29696000	0000000000001000	".pdata"	例外資料	IMG	-R---	ERWC-
00007FFB29697000	0000000000001000	"._RDATA"		IMG	-R---	ERWC-
00007FFB29698000	0000000000001000	".rsrc"	資源	IMG	-R---	ERWC-
00007FFB29699000	0000000000001000	".reloc"	Base relocations	IMG	-R---	ERWC-

PE format

位址	大小	資訊	內容	類型	保護	初始保護
0000002DAA5A4000	0000000000005C000	Reserved (0000002DA		PRV		-RW--
0000002DAA600000	000000000000FC000	Reserved		PRV		-RW--
0000002DAA6FC000	0000000000004000			PRV	-RW-G	-RW--
000001FAD43A0000	00000000000010000			MAP	-RW--	-RW--
000001FAD43B0000	0000000000001000	\Device\HarddiskVo1		MAP	-R---	-R---
000001FAD43C0000	0000000000001D000			MAP	-R---	-R---
000001FAD43E0000	0000000000004000			MAP	-R---	-R---
000001FAD43F0000	0000000000001000			MAP	-R---	-R---
000001FAD4400000	0000000000002000			PRV	-RW--	-RW--
000001FAD4410000	000000000000C9000	\Device\HarddiskVo1		MAP	-R---	-R---
000001FAD44E0000	0000000000001000			MAP	-R---	-R---
000001FAD44F0000	00000000000010000			PRV	-RW--	-RW--
000001FAD4500000	000000000000F0000	Reserved (000001FAD		PRV		-RW--
000001FAD45F0000	0000000000001000			MAP	-R---	-R---
000001FAD4600000	0000000000001000			MAP	-R---	-R---
00007FF4F0140000	0000000000005000			MAP	-R---	-R---
00007FF4F0145000	000000000000FB000	Reserved (00007FF4F		MAP		-R---
00007FF4F0240000	0000000100020000	Reserved		PRV		-RW--
00007FF5F0260000	0000000002000000	Reserved		PRV		-RW--
00007FF5F2260000	0000000000001000			PRV	-RW--	-RW--
00007FF5F2270000	0000000000001000			MAP	-R---	-R---
00007FF5F2280000	0000000000033000			MAP	-R---	-R---
00007FF676150000	0000000000001000	hello.exe		IMG	-R---	ERWC-
00007FF676151000	0000000000002000	".text"	可執行代碼	IMG	ER---	ERWC-
00007FF676153000	0000000000002000	".rdata"	唯讀的已初始化的資料	IMG	-R---	ERWC-
00007FF676155000	0000000000001000	".data"	已初始化的資料	IMG	-RW--	ERWC-
00007FF676156000	0000000000001000	".pdata"	例外資料	IMG	-R---	ERWC-
00007FF676157000	0000000000001000	".rsrc"	資源	IMG	-R---	ERWC-
00007FF676158000	0000000000001000	".reloc"	Base relocations	IMG	-R---	ERWC-
00007FFB29680000	0000000000001000	vcruntime140.dll		IMG	-R---	ERWC-
00007FFB29681000	00000000000010000	".text"	可執行代碼	IMG	ER---	ERWC-
00007FFB29691000	0000000000004000	".rdata"	唯讀的已初始化的資料	IMG	-R---	ERWC-
00007FFB29695000	0000000000001000	".data"	已初始化的資料	IMG	-RW--	ERWC-
00007FFB29696000	0000000000001000	".pdata"	例外資料	IMG	-R---	ERWC-
00007FFB29697000	0000000000001000	"._RDATA"		IMG	-R---	ERWC-
00007FFB29698000	0000000000001000	".rsrc"	資源	IMG	-R---	ERWC-
00007FFB29699000	0000000000001000	".reloc"	Base relocations	IMG	-R---	ERWC-

VA RVA

- 解釋 RVA (Relative Virtual Address) 之前
- 先解釋什麼是 VA (Virtual Address)
- 做一下小實驗，如果執行兩個 hello.exe，記憶體位址分布長怎樣

VA RVA

- 兩個 process 的記憶體位址有重疊耶?!
- 如果改掉 A process 記憶體內容 (地址重疊的部分), B process 的內容也會被改嗎?
- 實驗一下, 答案是不會的
- 所以那個記憶體位址到底是啥

VA RVA

- 其實我們的程式所看到的記憶體位址，都是假的
- 都是虛擬記憶體位址 (Virtual Address)



VA RVA

- 那為什麼要這麼複雜?
- 如果程式都能直接碰到實體記憶體位址 PA (Physical Address)
- 你要怎麼知道這個 PA 有沒有被其他程式占用?
- 這個問題很難，但現代 OS 幫你搞定了這個問題
- OS 只給你 VA，實際上存取時，OS 有他的方式，能夠 VA <-> PA

VA RVA

- 正常狀況下 A process 的 0x55665566 VA
- 跟 B process 的 0x55665566 VA
- 不是對應到同一個 PA
- 解釋了剛剛的實驗結果

VA RVA

- 搞懂 VA 了，可以講 RVA 了
- 只是一個方便 PE 結構不用寫這麼多字的東西
- $VA = ImageBase + RVA$
- 第一條指令位址 $VA = ImageBase + \text{Entry point RVA}$

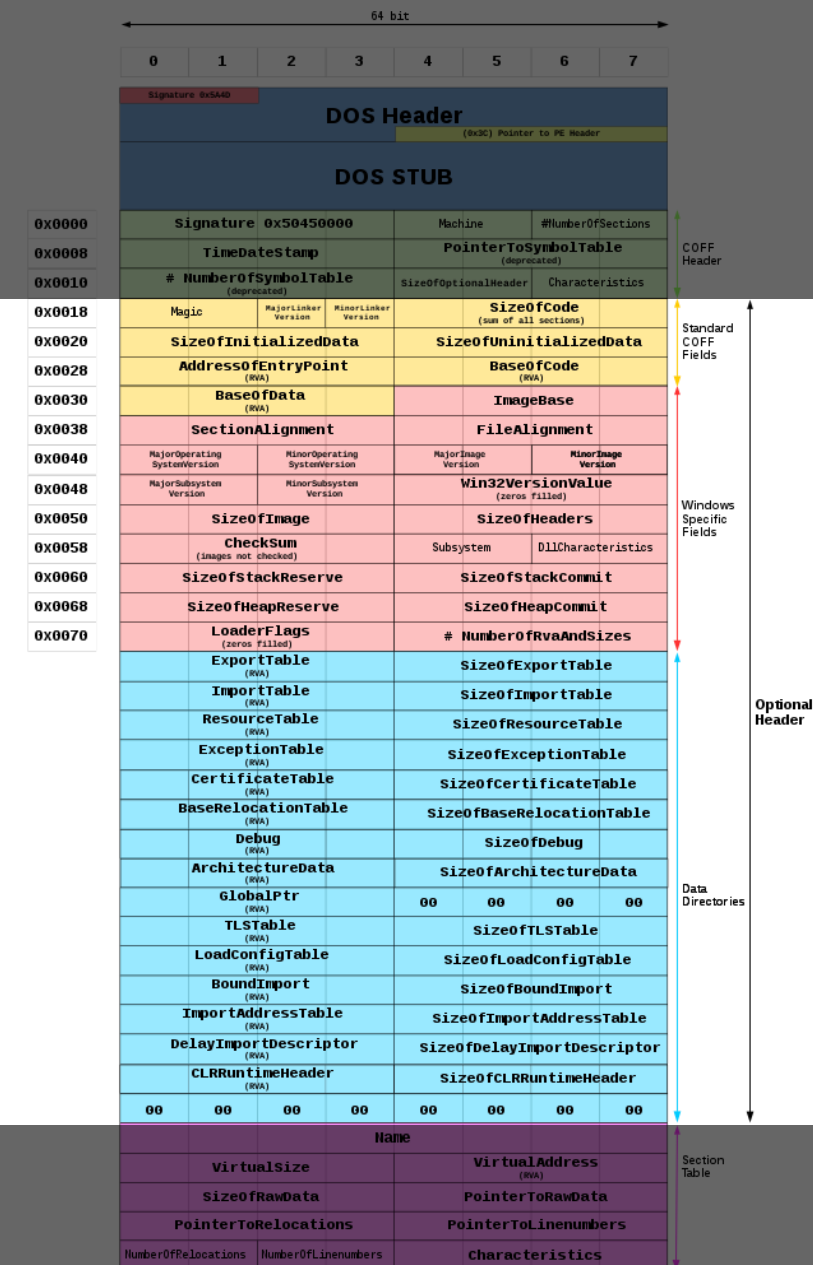
VA RVA

Disasm	General	DOS Hdr	Rich Hdr	File Hdr	Optional Hdr	Section Hdrs	Im
Offset	Name	Value	Value				
118	Magic	20B	NT64				
11A	Linker Ver. (Major)	E					
11B	Linker Ver. (Minor)	1D					
11C	Size of Code	1200					
120	Size of Initialized Data	2000					
124	Size of Uninitialized Data	0					
128	Entry Point	1580					
12C	Base of Code	1000					
130	Image Base	14000000					
138	Section Alignment	1000					
13C	File Alignment	200					
140	OS Ver						
142	OS Ver						
144	Image Ver. (Major)	0					
146	Image Ver. (Minor)	0					
148	Subsystem Ver. (Major)	6					
14A	Subsystem Ver. Minor)	0					
14C	Win32 Version Value	0					
150	Size of Image	9000					
154	Size of Headers	400					
158	Checksum	0					
15C	Subsystem	3	Windows console				
15E	DLL Characteristics	8160					
		40	DLL can move				
		100	Image is NX compatible				
		8000	TerminalServer aware				

程式進入點 RVA

程式基址

第一條指令位址 = 0x140001580



ASLR

- 可是實驗中, 我們的第一條指令位址顯然不是剛算的
- 原因是 ASLR (Address Space Layout Randomization)
- 記憶體位址每次執行時都是固定的話, 會有安全問題 (請看隔壁棚 Pwn 的課)
- 啟用了 ASLR, OS 就會隨機產生 ImageBase, 原本提供的 ImageBase 就被忽略了 QQ

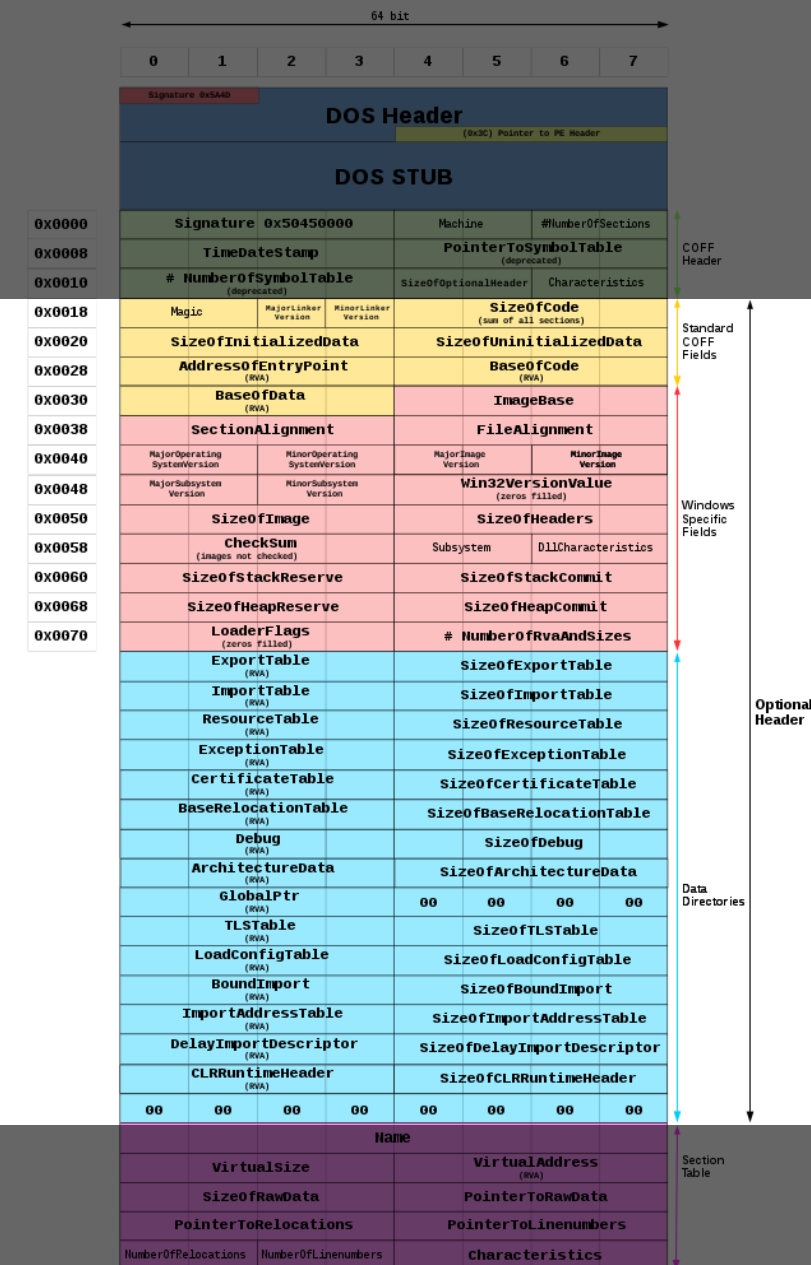
ASLR

- 第一條指令位址 $VA = \text{ASLR 隨機產生的 ImageBase} + \text{進入點 RVA}$
- 那麼要怎麼知道 ASLR 有沒有開啟呢?

ASLR

Disasm	General	DOS Hdr	Rich Hdr	File Hdr	Optional Hdr	Section Hdrs	Im
Offset	Name	Value	Value				
118	Magic	20B	NT64				
11A	Linker Ver. (Major)	E					
11B	Linker Ver. (Minor)	1D					
11C	Size of Code	1200					
120	Size of Initialized Data	2000					
124	Size of Uninitialized Data	0					
128	Entry Point	1580					
12C	Base of Code	1000					
130	Image Base	140000000					
138	Section Alignment	1000					
13C	File Alignment	200					
140	OS Ver. (Major)	6	Windows Vista / Server 2008				
142	OS Ver. (Minor)	0					
144	Image Ver. (Major)	0					
146	Image Ver. (Minor)	0					
148	Subsystem Ver. (Major)	6					
14A	Subsystem Ver. Minor)	0					
14C	Win32 Version Value	0					
150	Size of Image	9000					
154	Size of Headers	400					
158	Checksum	0					
15C	Subsystem	3	Windows console				
15E	DLL Characteristics	8160					
		40	DLL can move				
		100	Image is NX compatible				
		8000	TerminalServer aware				

若有 DLL can move 就是有啟用 ASLR

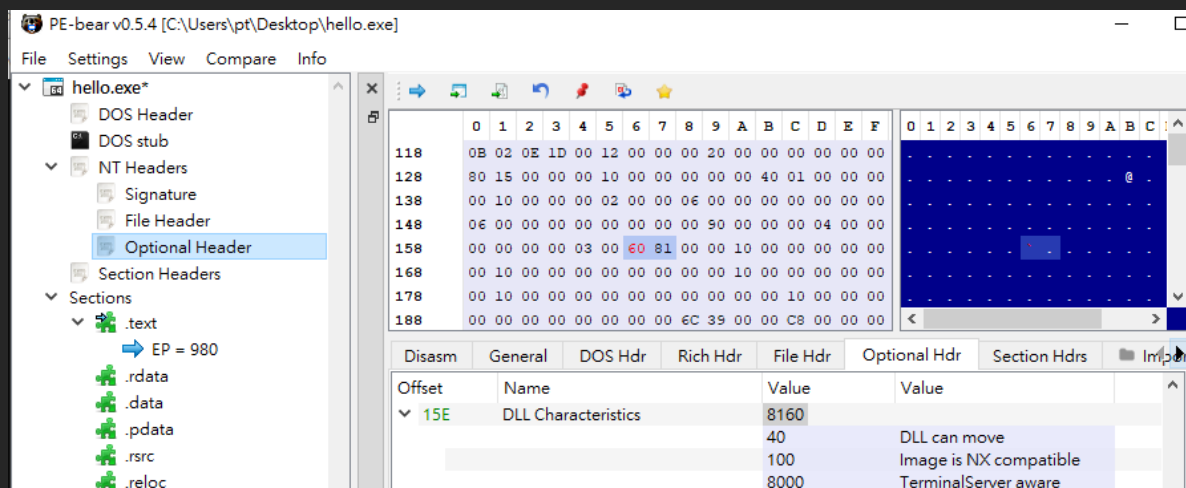


ASLR

- 如果把程式裡的這個 bit 拔掉, 就可以關掉 ASLR 了
- 來實驗一下!

ASLR

- DLL can move 是 0x40
- 減去 0x40 就是把它拔掉



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C
118	0B	02	0E	1D	00	12	00	00	00	20	00	00	00	00	00	00													
128	80	15	00	00	00	10	00	00	00	00	00	40	01	00	00	00													
138	00	10	00	00	00	02	00	00	06	00	00	00	00	00	00	00													
148	06	00	00	00	00	00	00	00	00	90	00	00	00	04	00	00													
158	00	00	00	00	03	00	20	81	00	00	10	00	00	00	00	00													
168	00	10	00	00	00	00	00	00	00	00	10	00	00	00	00	00													
178	00	10	00	00	00	00	00	00	00	00	00	00	00	10	00	00													
188	00	00	00	00	00	00	00	00	00	EC	39	00	00	C8	00	00													

Disasm	General	DOS Hdr	Rich Hdr	File Hdr	Optional Hdr	Section Hdrs	Imports
Offset	Name	Value	Value	Value	Value	Value	Value
15E	DLL Characteristics	8120					
		100			Image is NX compatible		
		8000			TerminalServer aware		

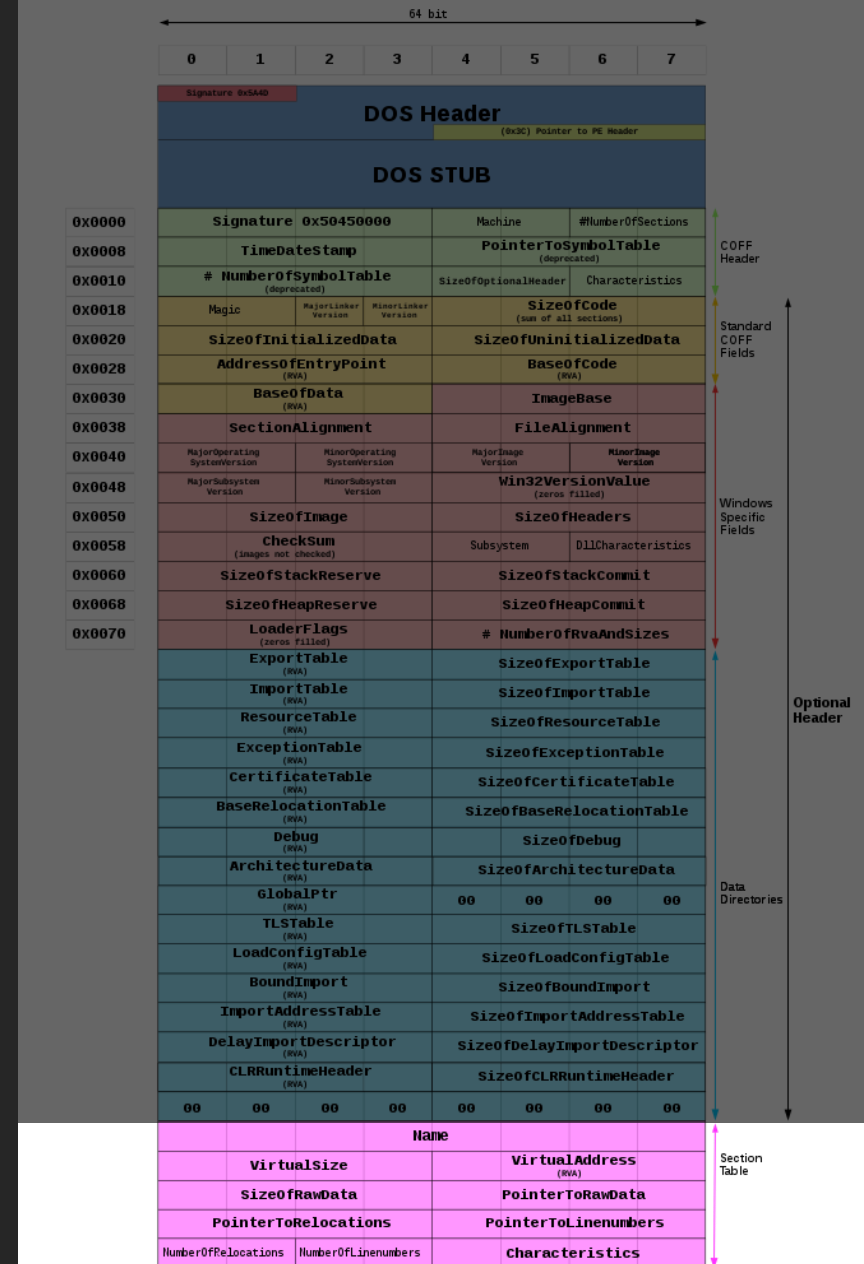
ASLR

- 記得另存一個新程式
- 再度執行看看

0000000140000000	00000000000001000	hello_noaslr.exe
0000000140001000	00000000000002000	".text"
0000000140003000	00000000000002000	".rdata"
0000000140005000	00000000000001000	".data"
0000000140006000	00000000000001000	".pdata"
0000000140007000	00000000000001000	".rsrc"
0000000140008000	00000000000001000	".reloc"

PE format

- Section Hdr
- 在右邊的圖是對應 Section Table
- 其實會有多個 Section Hdr
- 用 PE-Bear 來展示一下



PE format

PE-bear v0.5.4 [C:/Users/pt/Desktop/hello.exe]

File Settings View Compare Info

hello.exe

DOS Header

DOS stub

NT Headers

Signature

File Header

Optional Header

Section Headers

Sections

.text

EP = 980

.rdata

.data

.pdata

.rsrc

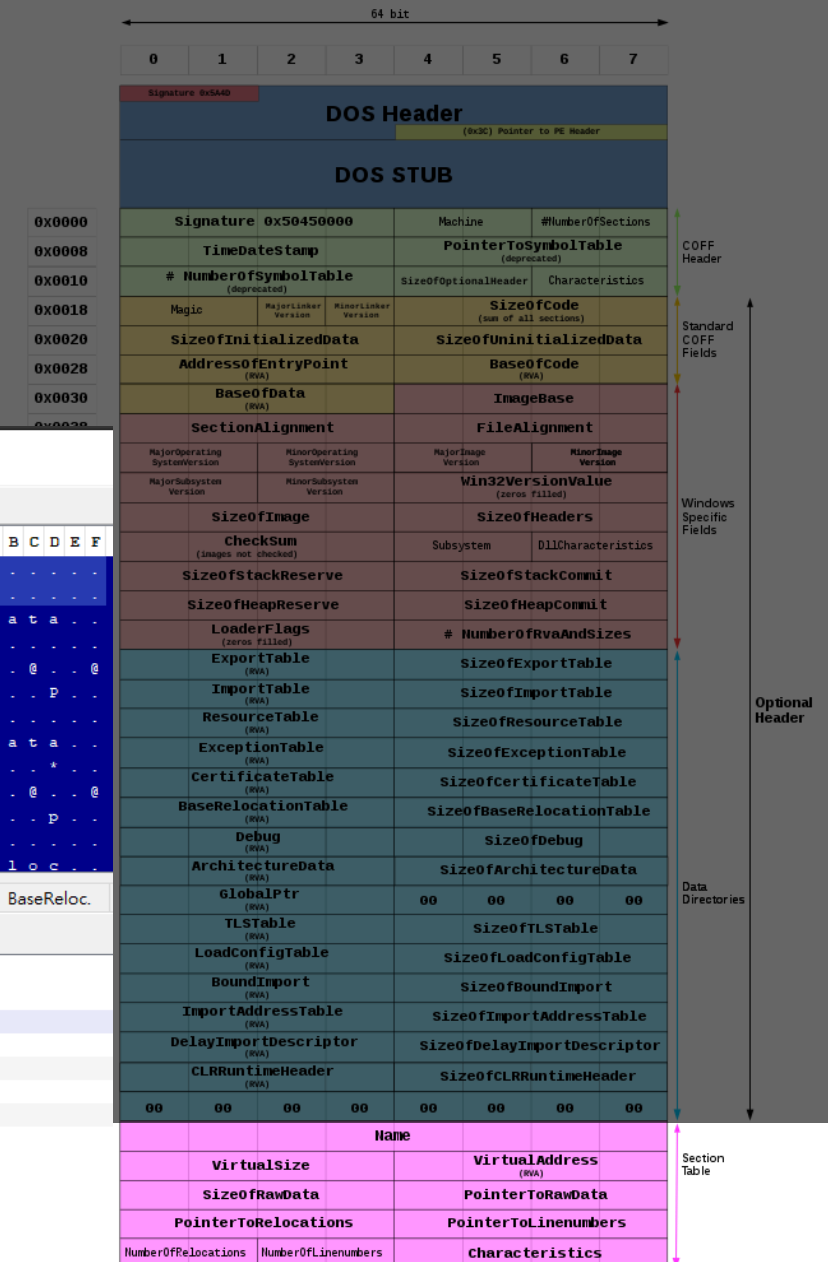
.reloc

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
208	2E	74	65	78	74	00	00	00	4C	10	00	00	00	10	00	00
218	00	12	00	00	00	04	00	00	00	00	00	00	00	00	00	00
228	00	00	00	00	20	00	00	60	2E	72	64	61	74	61	00	00
238	40	11	00	00	00	30	00	00	00	12	00	00	00	16	00	00
248	00	00	00	00	00	00	00	00	00	00	00	00	40	00	00	40
258	2E	64	61	74	61	00	00	00	48	06	00	00	00	50	00	00
268	00	02	00	00	00	28	00	00	00	00	00	00	00	00	00	00
278	00	00	00	00	40	00	00	C0	2E	70	64	61	74	61	00	00
288	98	01	00	00	00	60	00	00	00	02	00	00	00	2A	00	00
298	00	00	00	00	00	00	00	00	00	00	00	00	00	40	00	00
2A8	2E	72	73	72	63	00	00	00	E0	01	00	00	00	70	00	00
2B8	00	02	00	00	00	2C	00	00	00	00	00	00	00	00	00	00
2C8	00	00	00	00	40	00	00	40	2E	72	65	6C	6F	63	00	00

Disasm: Headers to [.text] General DOS Hdr Rich Hdr File Hdr Optional Hdr Section Hdrs Imports Resources Exception BaseReloc.

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
▼ .text	400	1200	1000	104C	60000020	0	0	0
> 1600	^	204C	^	r-x				
> .rdata	1600	1200	3000	1140	40000040	0	0	0
> .data	2800	200	5000	648	C0000040	0	0	0
> .pdata	2A00	200	6000	198	40000040	0	0	0
> .rsrc	2C00	200	7000	1E0	40000040	0	0	0
> .reloc	2E00	200	8000	2C	42000040	0	0	0

.text section hdr



PE format

PE-bear v0.5.4 [C:/Users/pt/Desktop/hello.exe]

File Settings View Compare Info

hello.exe

DOS Header
DOS stub
NT Headers
Signature
File Header
Optional Header
Section Headers

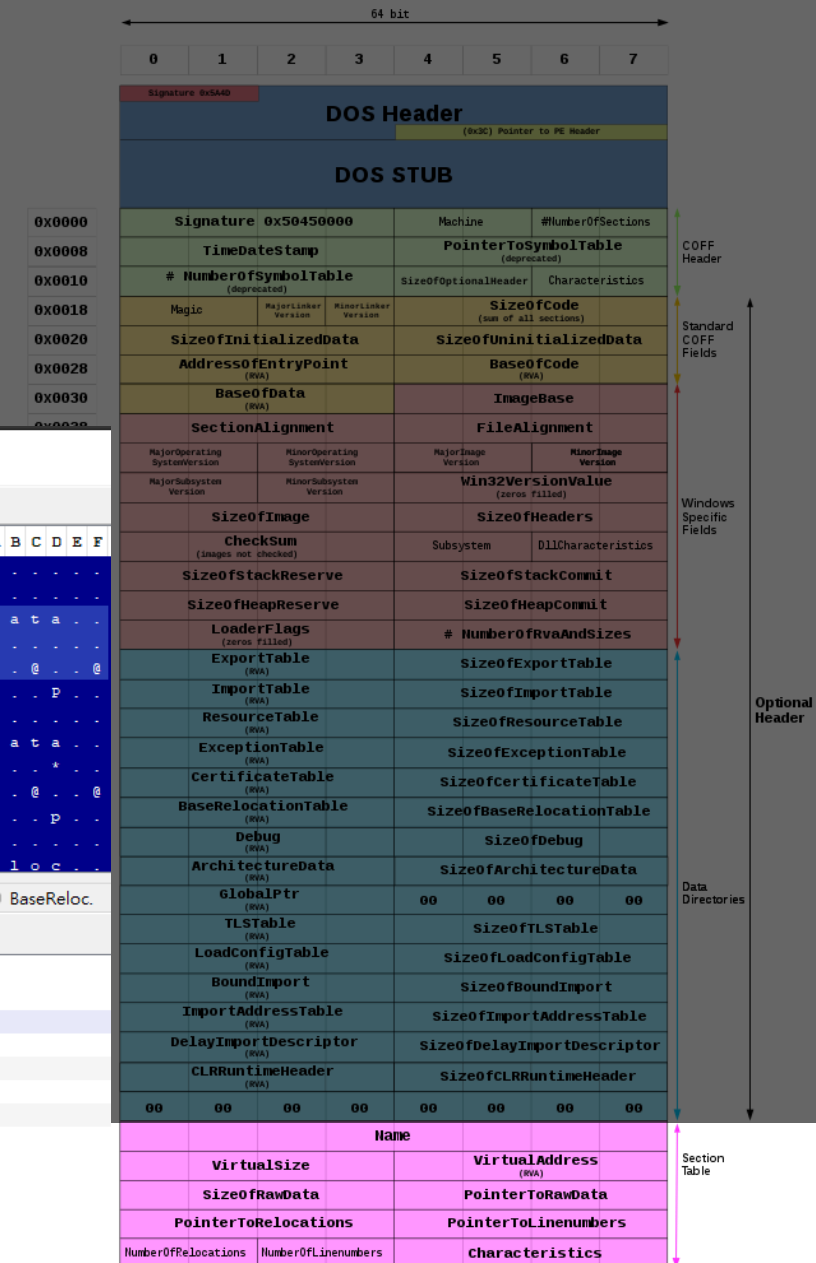
Sections
.text
EP = 980
.rdata
.data
.pdata
.rsrc
.reloc

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
208	2E	74	65	78	74	00	00	00	00	4C	10	00	00	00	10	00		.	t	e	x	t
218	00	12	00	00	00	04	00	00	00	00	00	00	00	00	00	00	
228	00	00	00	00	20	00	00	60	2E	72	64	61	74	61	00	00	
238	40	11	00	00	00	30	00	00	00	12	00	00	00	16	00	00	
248	00	00	00	00	00	00	00	00	00	00	00	00	00	40	00	00	40	
258	2E	64	61	74	61	00	00	00	48	06	00	00	00	50	00	00	
268	00	02	00	00	28	00	00	00	00	00	00	00	00	00	00	00	
278	00	00	00	00	40	00	00	C0	2E	70	64	61	74	61	00	00	
288	98	01	00	00	60	00	00	00	02	00	00	00	00	2A	00	00	
298	00	00	00	00	00	00	00	00	00	00	00	00	00	40	00	00	40	
2A8	2E	72	73	72	63	00	00	E0	01	00	00	00	00	70	00	00	
2B8	00	02	00	00	2C	00	00	00	00	00	00	00	00	00	00	00	
2C8	00	00	00	00	40	00	00	40	2E	72	65	6C	6F	63	00	00	

Disasm: Headers to [.text] General DOS Hdr Rich Hdr File Hdr Optional Hdr Section Hdrs Imports Resources Exception BaseReloc.

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
▼ .text	400	1200	1000	104C	60000020	0	0	0
> .rdata	1600	1200	204C	^	r-x			
> .data	2800	200	5000	648	C0000040	0	0	0
> .pdata	2A00	200	6000	198	40000040	0	0	0
> .rsrc	2C00	200	7000	1E0	40000040	0	0	0
> .reloc	2E00	200	8000	2C	42000040	0	0	0

.rdata section hdr



PE format

- 解釋一下 Section Hdr
- 程式檔案的 Raw Addr 開始的 Raw size 個 Bytes 會映射到 Virtual Addr 開始的 Virtual Size 個 Bytes
 - Virtual Addr 是 RVA
- Characteristics 設定了該區記憶體位址的權限

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
▼ .text	400	1200	1000	104C	60000020	0	0	0
> 1600	^	204C	^	r-x				
> .rdata	1600	1200	3000	1140	40000040	0	0	0
> .data	2800	200	5000	648	C0000040	0	0	0
> .pdata	2A00	200	6000	198	40000040	0	0	0
> .rsrc	2C00	200	7000	1E0	40000040	0	0	0
> .reloc	2E00	200	8000	2C	42000040	0	0	0

PE format

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
▼ .text	400	1200	1000	104C	60000020	0	0	0
> .rdata	1600	1200	3000	1140	40000040	0	0	0
> .data	2800	200	5000	648	C0000040	0	0	0
> .pdata	2A00	200	6000	198	40000040	0	0	0
> .src	2C00	200	7000	1E0	40000040	0	0	0
> .reloc	2E00	200	8000	2C	42000040	0	0	0

hello.exe x

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0400h:	48	8D	05	29	46	00	00	C3	CC	CC	CC	CC	CC	CC	CC	CC	H..)	F..	Ä	Ä	Ä	Ä	Ä	Ä	Ä	Ä	Ä	Ä	Ä	Ä	Ä	Ä
0410h:	48	8D	05	11	46	00	00	C3	CC	CC	CC	CC	CC	CC	CC	CC	H...F..	Ä	Ä	Ä	Ä	Ä	Ä	Ä	Ä	Ä	Ä	Ä	Ä	Ä	Ä	Ä
0420h:	48	89	4C	24	08	48	89	54	24	10	4C	89	44	24	18	4C	H%L\$.H%T\$.L%D\$.L															
0430h:	89	4C	24	20	53	56	57	48	83	EC	30	48	8B	F9	48	8D	%L\$ SVWHfìOH<ùH.															

位址	十六進位																ASCII													
00000000140001000	48	8D	05	29	46	00	00	C3	CC	CC	CC	CC	CC	CC	CC	CC	H..)	F..	Ä	Ä	Ä	Ä	Ä	Ä	Ä	Ä	Ä	Ä	Ä	Ä
00000000140001010	48	8D	05	11	46	00	00	C3	CC	CC	CC	CC	CC	CC	CC	CC	H...F..	Ä	Ä	Ä	Ä	Ä	Ä	Ä	Ä	Ä	Ä	Ä	Ä	Ä
00000000140001020	48	89	4C	24	08	48	89	54	24	10	4C	89	44	24	18	4C	H.L\$.H.T\$.L.D\$.L													
00000000140001030	89	4C	24	20	53	56	57	48	83	EC	30	48	8B	F9	48	8D	.L\$ SVWH.ìOH<ùH.													

PE format

- 但看了一下 .rdata, 好像不是這麼回事?

PE format

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
▼ .text	400	1200	1000	104C	60000020	0	0	0
> .rdata	1600	1200	3000	1140	40000040	0	0	0
> .data	2800	200	5000	648	C0000040	0	0	0
> .pdata	2A00	200	6000	198	40000040	0	0	0
> .rsrc	2C00	200	7000	1E0	40000040	0	0	0
> .reloc	2E00	200	8000	2C	42000040	0	0	0

hello.exe x

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1600h:	28	3C	00	00	00	00	00	00	34	3C	00	00	00	00	00	00
1610h:	44	3C	00	00	00	00	00	00	52	3C	00	00	00	00	00	00
1620h:	60	3C	00	00	00	00	00	00	18	41	00	00	00	00	00	00
1630h:	02	41	00	00	00	00	00	00	E8	40	00	00	00	00	00	00

位址	十六進位																ASCII
0000000140003000	D0	4F	0F	38	FB	7F	00	00	30	4F	0F	38	FB	7F	00	00	DO.8û...ON.8û...
0000000140003010	50	4B	0F	38	FB	7F	00	00	E0	48	0F	38	FB	7F	00	00	PK.8û...àH.8û...
0000000140003020	10	4E	0F	38	FB	7F	00	00	B0	01	0F	38	FB	7F	00	00	.N.8û...°.8û...
0000000140003030	A0	EA	51	39	FB	7F	00	00	80	7B	0E	38	FB	7F	00	00	êQ9û...{.8û...

PE format

- 但看了一下 .rdata, 好像不是這麼回事?
- 實際上是因為另一個機制, 改掉了這邊的資料
- 關鍵字: IMAGE_IMPORT_DESCRIPTOR、IAT (Import Address Table)
- 關於這個機制, 我們以後會專門做一期視頻給大家講解

逆向工程簡介

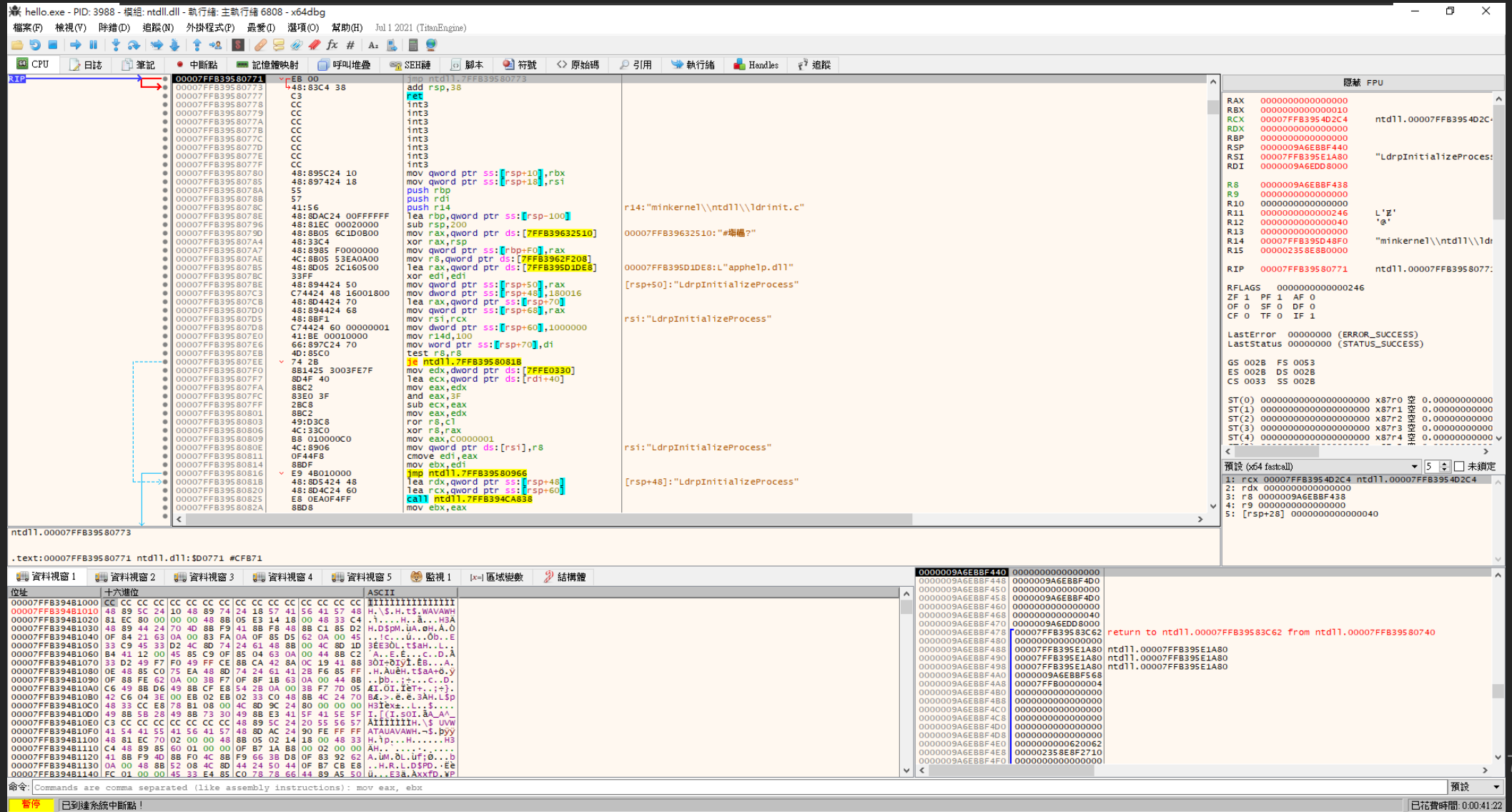
- 看完這個章節之後，你應該…
 - 知道為何要逆向工程
 - 因為沒有 source code QQ
 - 略懂逆向工程原理
 - 因為你知道了程式產生過程
 - 所以你知道了反組譯、反編譯這些技術為何可行
 - 略懂從哪開始進行逆向
 - 因為你知道了程式怎麼運作起來的

x86 組合語言

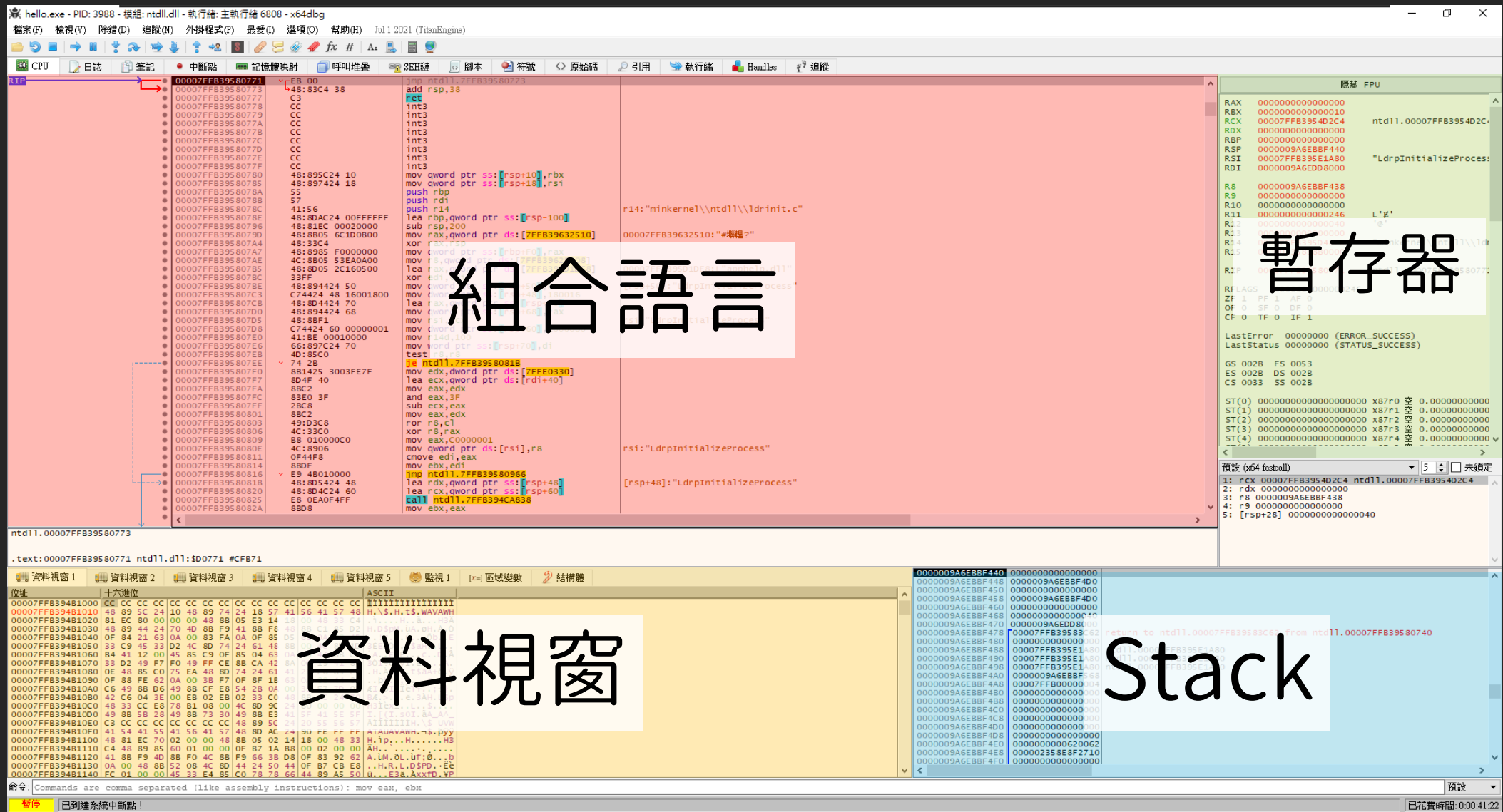
x86 組合語言

- 這個章節有點多東西要講
- 所以我決定通過講解 x64dbg 介面來聊 x86 組合語言

x64dbg



x64dbg



x64dbg

hello.exe - PID: 3988 - 模組: ntdll.dll - 執行緒: 主執行緒 6808 - x64dbg

檔案(F) 檢視(V) 除錯(D) 追蹤(N) 外掛程式(P) 最愛(I) 選項(O) 幫助(H) Jul 1 2021 (TitanEngine)

CPU 日誌 筆記 中斷點 記憶體映射 呼叫堆疊 SEH鏈 腳本 符號 原始碼 引用 執行緒 Handles 追蹤

組合語言

暫存器

資料視窗

Stack

命令: Commands are comma separated (like assembly instructions): mov eax, ebx

暫停 已到達系統中斷點!

已花費時間: 0:00:41.22

暫存器

- 通用暫存器 (General-Purpose Registers)

63	32	31	16	15	8	7	0	16-bit	32-bit	64-bit
			AH				AL	AX	EAX	RAX
			BH				BL	BX	EBX	RBX
			CH				CL	CX	ECX	RCX
			DH				DL	DX	EDX	RDY
			BP					BP	EBP	RBP
			SP					SP	ESP	RSP
			SI					SI	ESI	RSI
			DI					DI	EDI	RDI

Base Pointer

Stack Pointer

暫存器

- 指令暫存器
 - Instruction Pointer Register
 - 或稱 Program Counter
- 存放下一條指令的位址



16-bit	32-bit	64-bit
IP	EIP	RIP

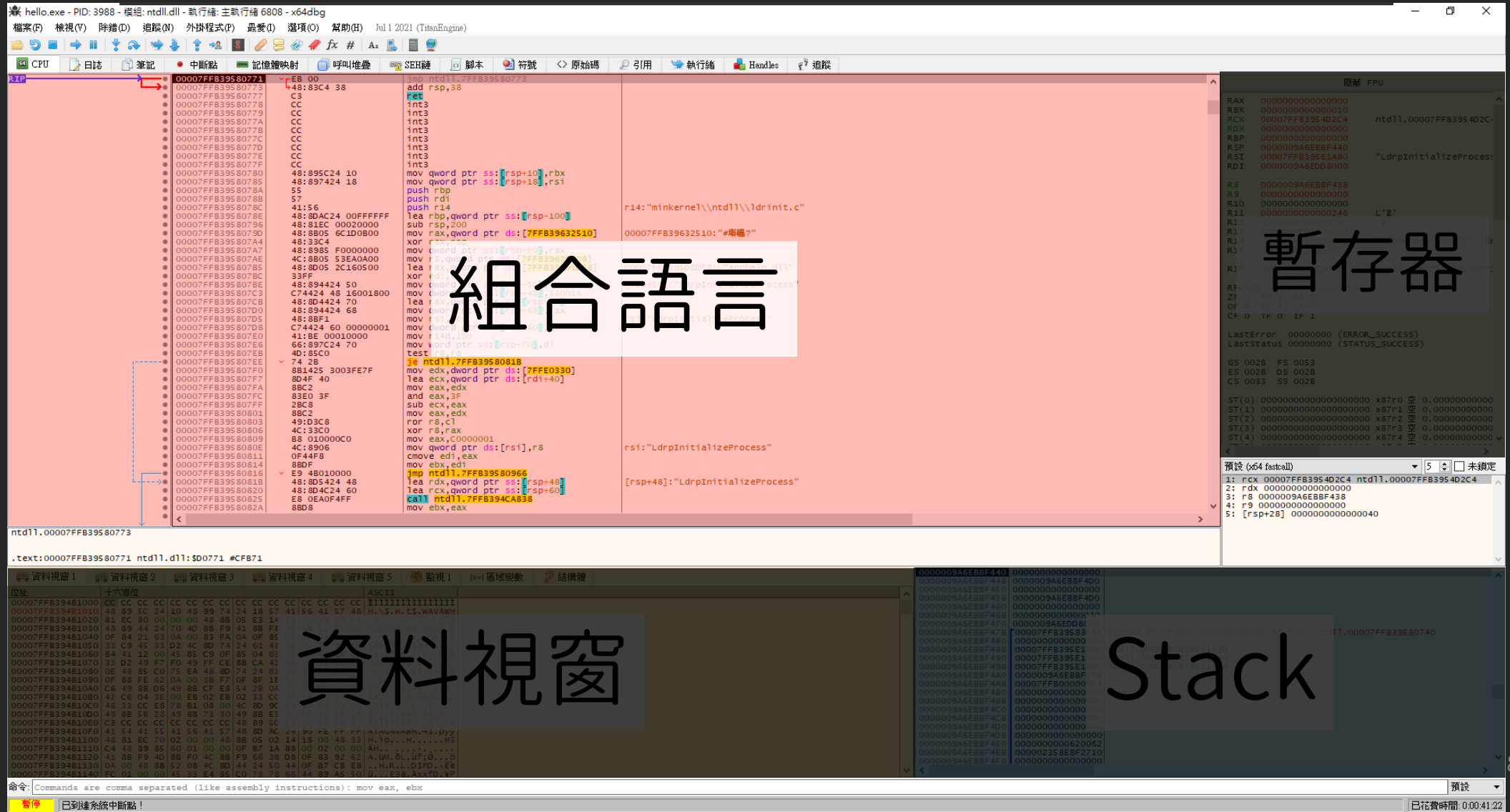
暫存器

- 還有更多的暫存器
- 關於其他暫存器，我們以後會專門做一期視頻給大家講解

暫存器

- 不同的指令集有不同的暫存器
- 這邊只講 x86 / x86-64 指令集
- 對其他指令集有興趣的人可以搜尋 ARM、MIPS

x64dbg



x86 組合語言

- 前面章節有提到組合語言, 現在就來學一下
- 直接舉例一條指令
- `mov rax, 0`
- 將 0 放進 RAX 暫存器裡面

x86 組合語言


- 除了 mov 搬移指令以外, 還有很多很多很多的指令
- 遇到沒看過的就菇狗關鍵字 “x86 <指令>”
- 加 add
- 減 sub
- 乘 mul
- 除 div
- 呼叫 call
- 返回 ret
- 跳 jmp



菇狗


x86 組合語言

- 單一條組語, 作用通常都很簡單



```
mov rax, 1
add rax, 5
mov rbx, 7
sub rbx, rax
inc rax
```

ASM



```
rax = 1
rax = rax + 5
rbx = 7
rbx = rbx - rax
rax++
```

C

x86 組合語言



真的很簡單

x86 組合語言

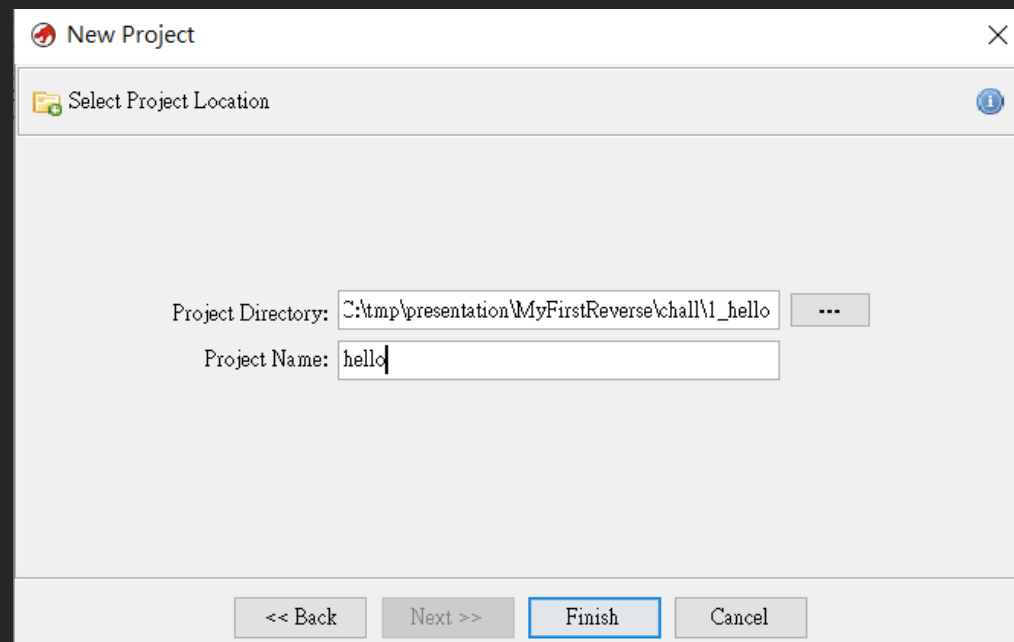
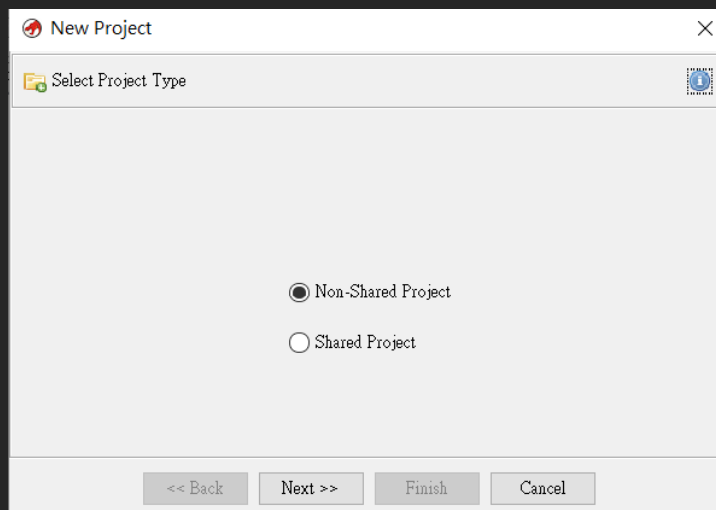
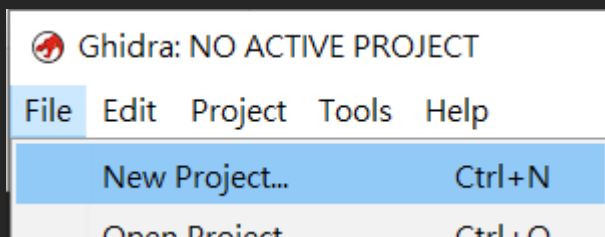


x86 組合語言

- 單一條組語, 作用通常都很簡單
- 一坨組語喇在一起, 就不是很容易看懂他在幹嘛了
- 用 ghidra 打開 chall/1_hello/hello.exe

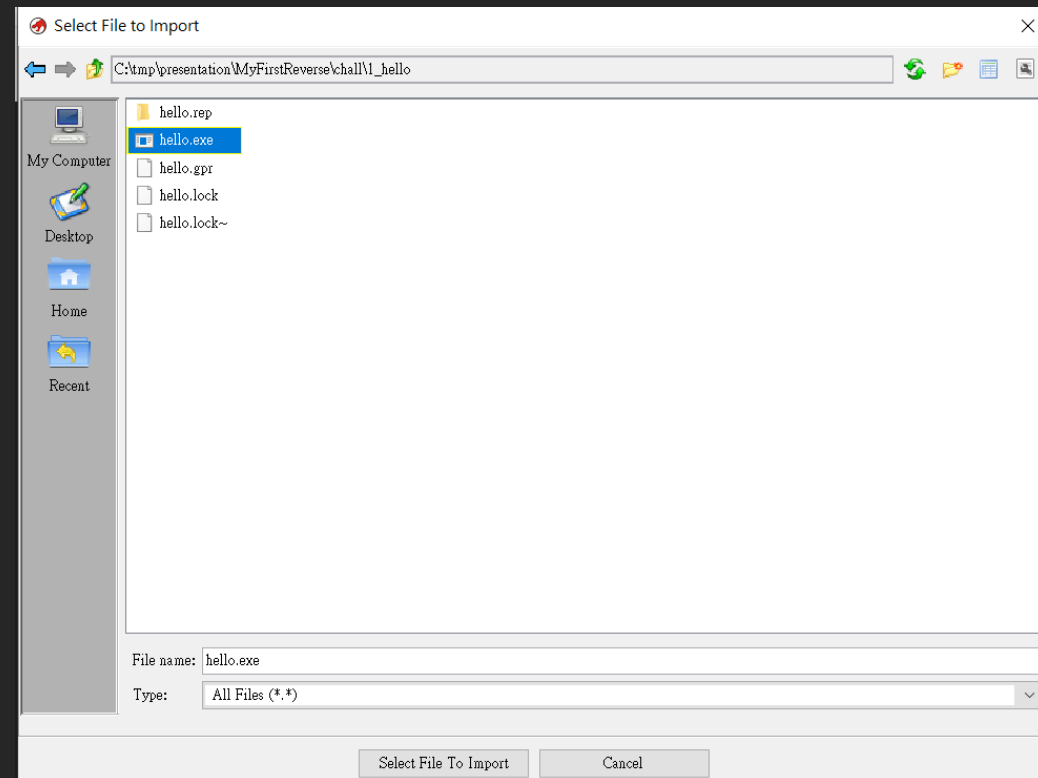
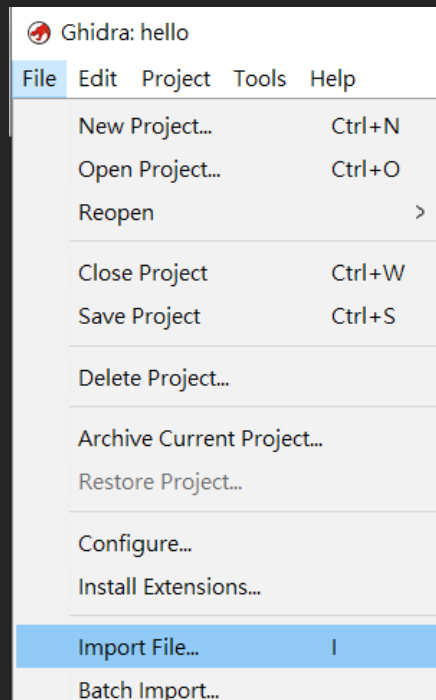
Ghidra

- 創一個新 project
- 自訂 Project 目錄以及名稱



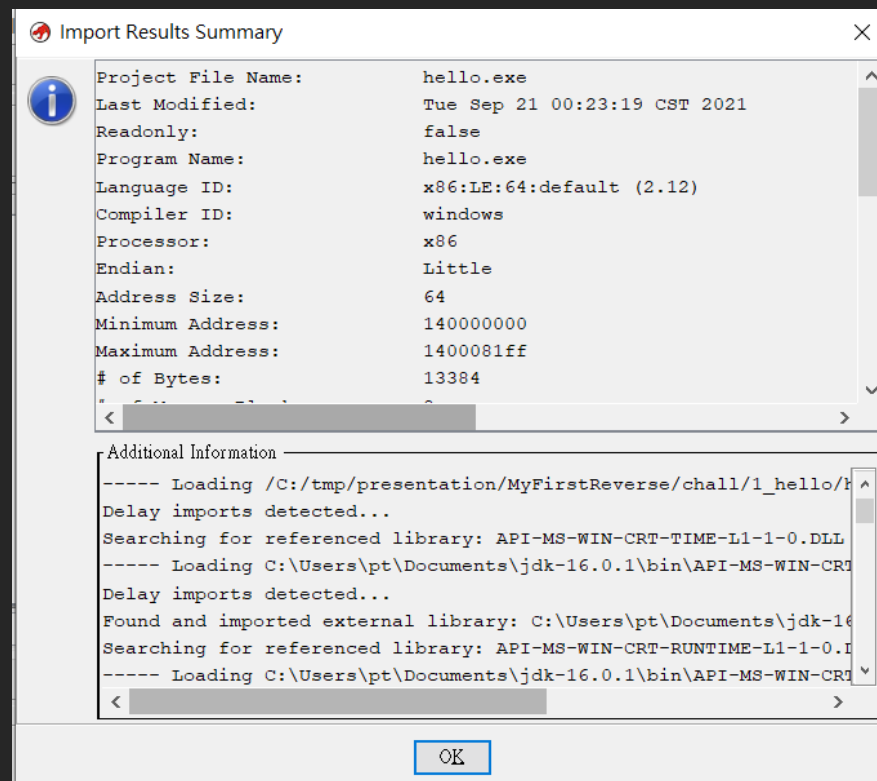
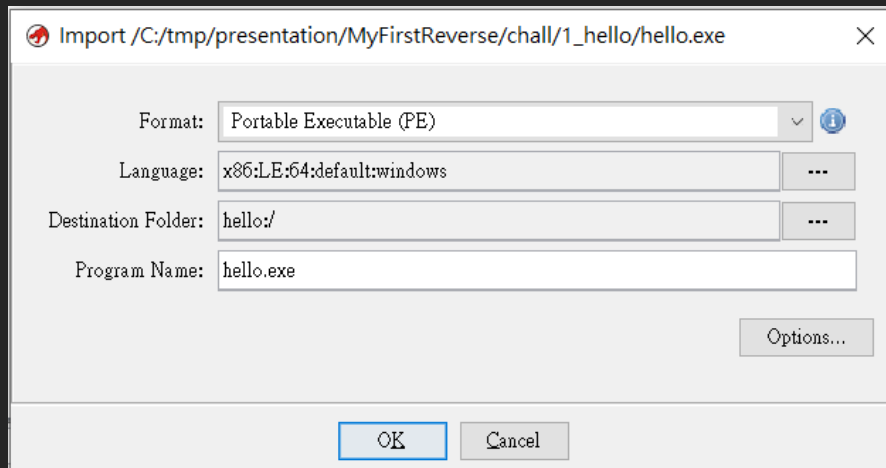
Ghidra

- 匯入我們要分析的程式



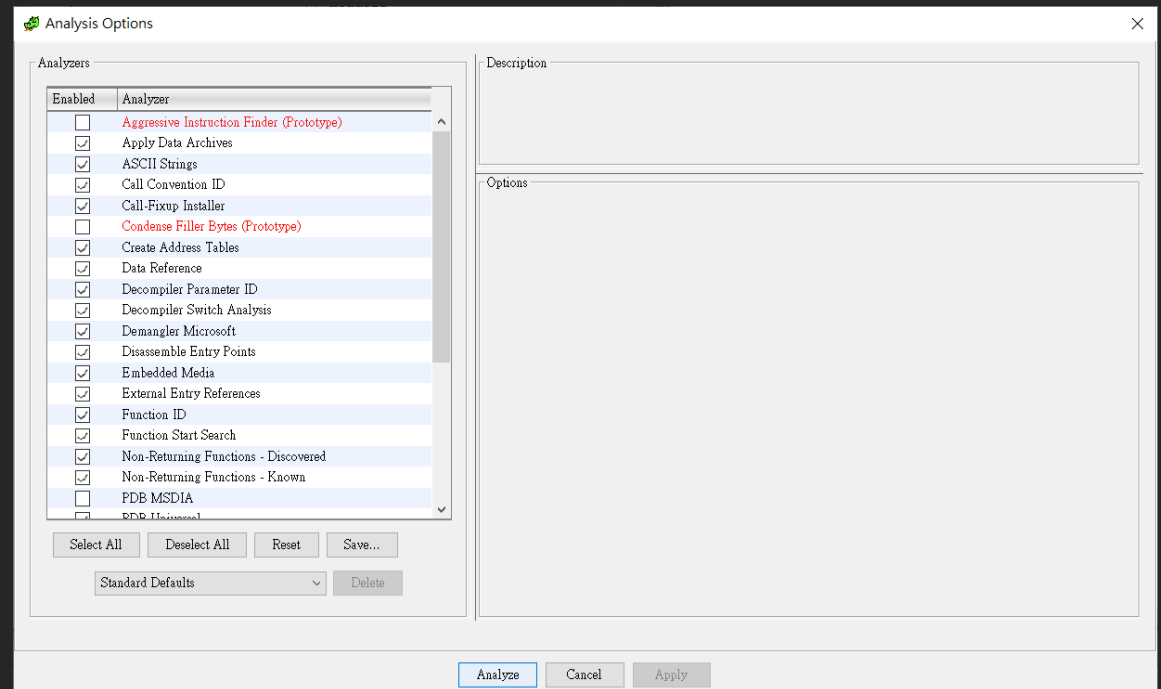
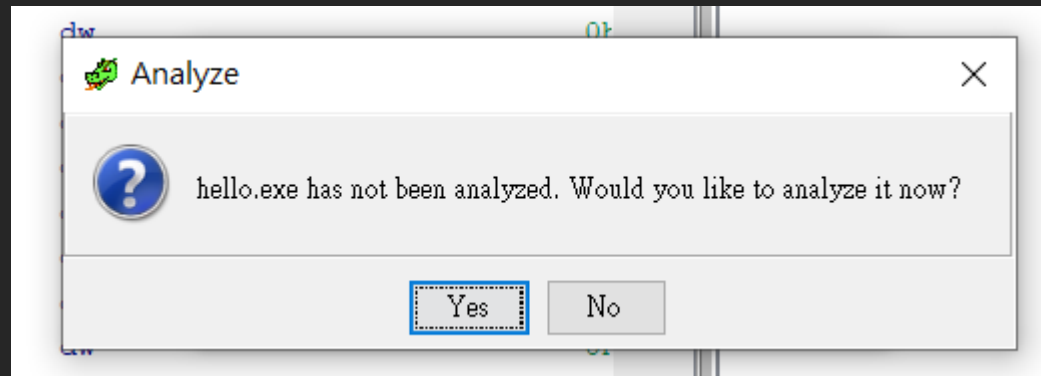
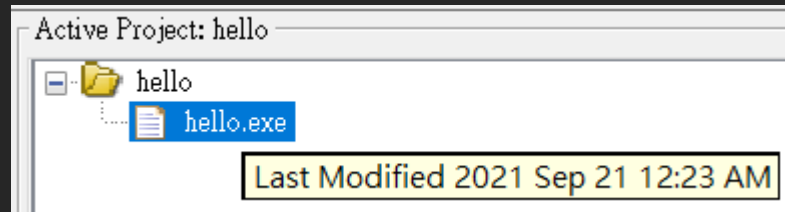
Ghidra

- 匯入我們要分析的程式



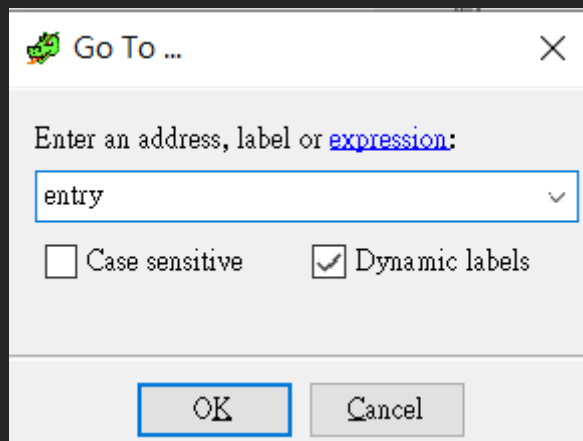
Ghidra

- 點兩下匯入好的程式, 打開分析畫面
- 讓他跑一些分析



Ghidra

- 最終蹦出分析畫面
- 按一下 g, 輸入 entry 直接跳到程式進入點



Ghidra

CodeBrowser: hello/hello.exe

File Edit Analysis Graph Navigation Search Select Tools Window Help

Program Trees

- hello.exe
 - Headers
 - .text
 - .rdata
 - .data
 - .pdata
 - .rsrc
 - .reloc
 - Debug Data

Symbol Tree

- Imports
- Exports
- Functions
- Labels
- Classes
- Namespaces

Filter:

Data Type Manager

- Data Types
 - Builtin Types
 - hello.exe
 - windows_vs12_32
 - windows_vs12_64

Filter:

Listing: hello.exe

LAB_140001570

```
140001570 8b cb MOV ECX,EBX
140001572 e8 91 09 CALL API-MS-WIN-CRT-RUNTIME-L1-1-0
00 00

-- Flow Override: CALL_RETURN (CALL_T...
```

uint __fastcall entry(void)

```
uint EAX:4 <RETURN>
undefined8 Stack[0x10]:8 local_res10
undefined8 Stack[0x8]:8 local_res8
undefined1 Stack[-0x18]:1 local_18
entry
```

140001580 48 83 ec 28 SUB RSP,0x28
140001584 e8 d7 03 CALL __security_init_cookie
00 00
140001589 48 83 c4 28 ADD RSP,0x28
14000158d e9 72 fe JMP LAB_140001404

Decompile: entry - (hello.exe)

```
1 /* WARNING: Function: __guard_dispatch_icall replaced with injection: guard_dispatch_icall */
2
3
4 uint entry(void)
5
6 {
7     undefined8 uVar1;
8     bool bVar2;
9     int iVar3;
10    ulonglong uVar4;
11    code **ppcVar5;
12    longlong *plVar6;
13    undefined8 uVar7;
14    undefined8 *puVar8;
15    uint *puVar9;
16    ulonglong uVar10;
17    uint unaff_EBX;
18    LPDWORD in_R9;
19
20    __security_init_cookie();
21    uVar4 = __srt_initialize_crt(1);
22    if ((char)uVar4 == '\0') {
23        FUN_140001a78(7);
24    }
25    else {
26        bVar2 = false;
27        uVar4 = __srt_acquire_startup_lock();
28        unaff_EBX = unaff_EBX & 0xffffffff00 | (uint)(uVar4 & 0xff);
29        if (DAT_1400055b0 != 1) {
30            if (DAT_1400055b0 == 0) {
```

Python - Interpreter

Python Interpreter for Ghidra
Based on Jython version 2.7.2 (v2.7.2:925a3cc3b49d, Mar 21 2020, 10:03:58)
[Java HotSpot(TM) 64-Bit Server VM (Oracle Corporation)]
Press 'F1' for usage instructions

Console | Bookmarks | Python

140001580 entry SUB RSP,0x28

Ghidra

The screenshot displays the Ghidra CodeBrowser interface for a file named `hello.exe`. The interface is divided into several panes:

- Program Trees:** Shows the file structure of `hello.exe`, including `Headers`, `.text`, `.rdata`, `.data`, `.pdata`, `.rsrc`, `.reloc`, and `Debug Data`.
- Symbol Tree:** Displays the symbol table, including `Imports`, `Exports`, `Functions`, `Labels`, and `Globals`. A large white box with the text "Symbol" is overlaid on this pane.
- Data Type Manager:** Shows the data types defined in the program, including `BuiltIn Types`, `hello.exe`, `windows_vs12_32`, and `windows_vs12_64`.
- Listing: hello.exe:** Displays the assembly code for the `entry` function. The code is shown in a table format with addresses, disassembled instructions, and comments. A large white box with the text "組合語言" (Assembly) is overlaid on this pane.
- Decompile: entry - (hello.exe):** Displays the decompiled C code for the `entry` function. A large white box with the text "反編譯" (Decompilation) is overlaid on this pane.
- Python - Interpreter:** Shows the Python interpreter output, indicating that the Python interpreter is running on the Ghidra platform.

The bottom status bar shows the current address (`140001580`), the current function (`entry`), and the current instruction (`SUB RSP,0x28`).

Ghidra

- 在進入 main 前會執行一些初始化工作
- 在反編譯視窗中找長得像是 `main(int argc, char *argv[], char *envp[])` 的 function call
- 點進去該 function (FUN_1400010e0)

```
53      uVar7 = _get_initial_narrow_environment();
54      puVar8 = (undefined8 *)__p__argv();
55      uVar1 = *puVar8;
56      puVar9 = (uint *)__p__argc();
57      uVar10 = (ulonglong)*puVar9;
58      unaff_EBX = FUN_1400010e0(uVar10,uVar1,uVar7,in_R9);
```

Ghidra

- 其實這個 function 就是我們原本的 main

```
int main(void)
{
    char name[32] = { 0 };
    int a, b, ans;

    srand(time(NULL));

    printf("What's your name?\n");

    scanf("%32s", name);

    printf("Hello, %s!\n", name);
```

```
for (int i = 0; i < 5; ++i) {
    a = (rand() % 10) + 1;
    b = (rand() % 10) + 1;

    printf("%d x %d = ?\n", a, b);

    scanf("%d", &ans);

    if (ans != a * b) {
        printf("Wrong!\n");
        evilcatboy();
    }
}

return 0;
```

Ghidra

- 嘗試看看 main 的組語
- 一坨組語喇在一起, 就不是很容易看懂他在幹嘛了

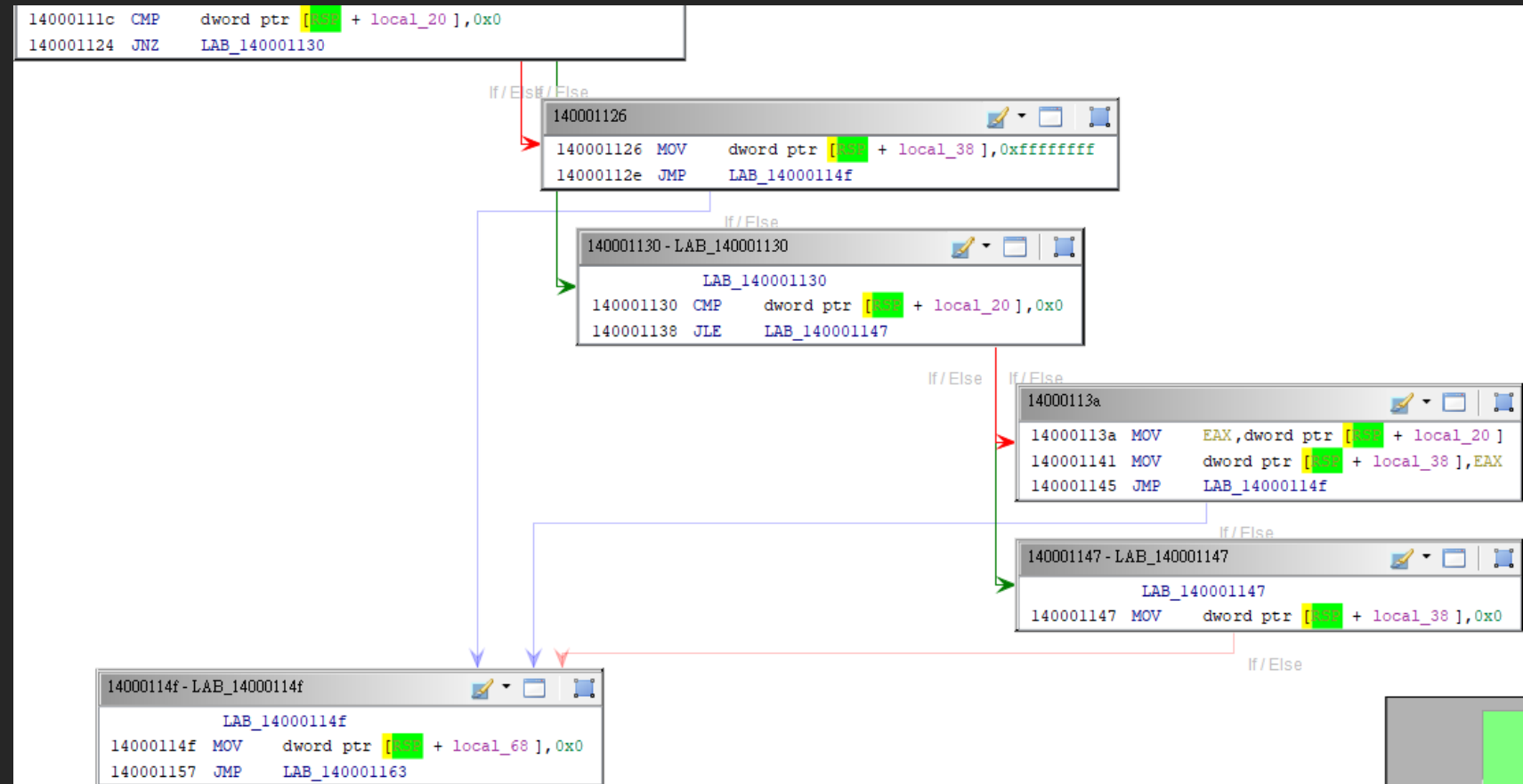
```
Function Graph [CodeBrowser: hello/hello.exe]
File Edit Navigation Search Select Help
Function Graph - FUN_1400010e0 - 9 vertices (hello.exe)
undefined2 Stack[-0x25c... local_25c
undefined4 Stack[-0x260... local_260
undefined1[16] Stack[-0x270... local_270
undefined1[16] Stack[-0x280... local_280
undefined1 Stack[-0x284... local_284
undefined4 Stack[-0x288... local_288
undefined8 Stack[-0x298... local_298
undefined4 Stack[-0x2a0... local_2a0
undefined8 Stack[-0x2a8... local_2a8
FUN_1400010e0
1400010e0 MOV qword ptr [RSP + local_res8], RBX
1400010e5 MOV qword ptr [RSP + local_res10], RBP
1400010ea MOV qword ptr [RSP + local_res18], RSI
1400010ef PUSH RDI
1400010f0 SUB RSP, 0x2c0
1400010f7 MOV RAX, qword ptr [DAT_140005008]
1400010fe XOR RAX, RSP
140001101 MOV qword ptr [RSP + local_18], RAX
140001109 XORPS XMM0, XMM0
14000110c XOR param_1, param_1
14000110e MOVUPS xmmword ptr [RSP + local_258[0]], XMM0
140001113 MOVUPS xmmword ptr [RSP + local_248[0]], XMM0
14000111b CALL qword ptr [__API-MS-WIN-CRT-TIME-L1-1-0.DLL:__time64]
140001121 MOV param_1, RAX
140001124 CALL qword ptr [__API-MS-WIN-CRT-UTILITY-L1-1-0.DLL::srand]
14000112a LEA param_1, [s_What's_your_name?_140003290]
140001131 CALL FUN_140001020
140001136 LEA param_2=local_258, [RSP + 0x70]
14000113b LEA param_1, [DAT_1400032a4]
140001142 CALL FUN_140001080
140001147 LEA param_2=local_258, [RSP + 0x70]
14000114c LEA param_1, [s_Hello,_%s!_1400032b0]
140001153 CALL FUN_140001020
140001158 XOR EBP, EBP
14000115a MOV ESI, 0x5
14000115f NOP
140001160 - LAB_140001160
LAB_140001160
140001160 CALL qword ptr [__API-MS-
140001166 MOV EDI, EAX
100
```

x86 組合語言

- 人類讀高階程式語言比直接讀組合語言快
- 若能知道對應的高階程式語言會變成什麼樣的組語
- 就能更快速的讀組語
- 以下來看看各種高階程式語言語法變成組語會長怎樣
- `demo/1_observe_c`

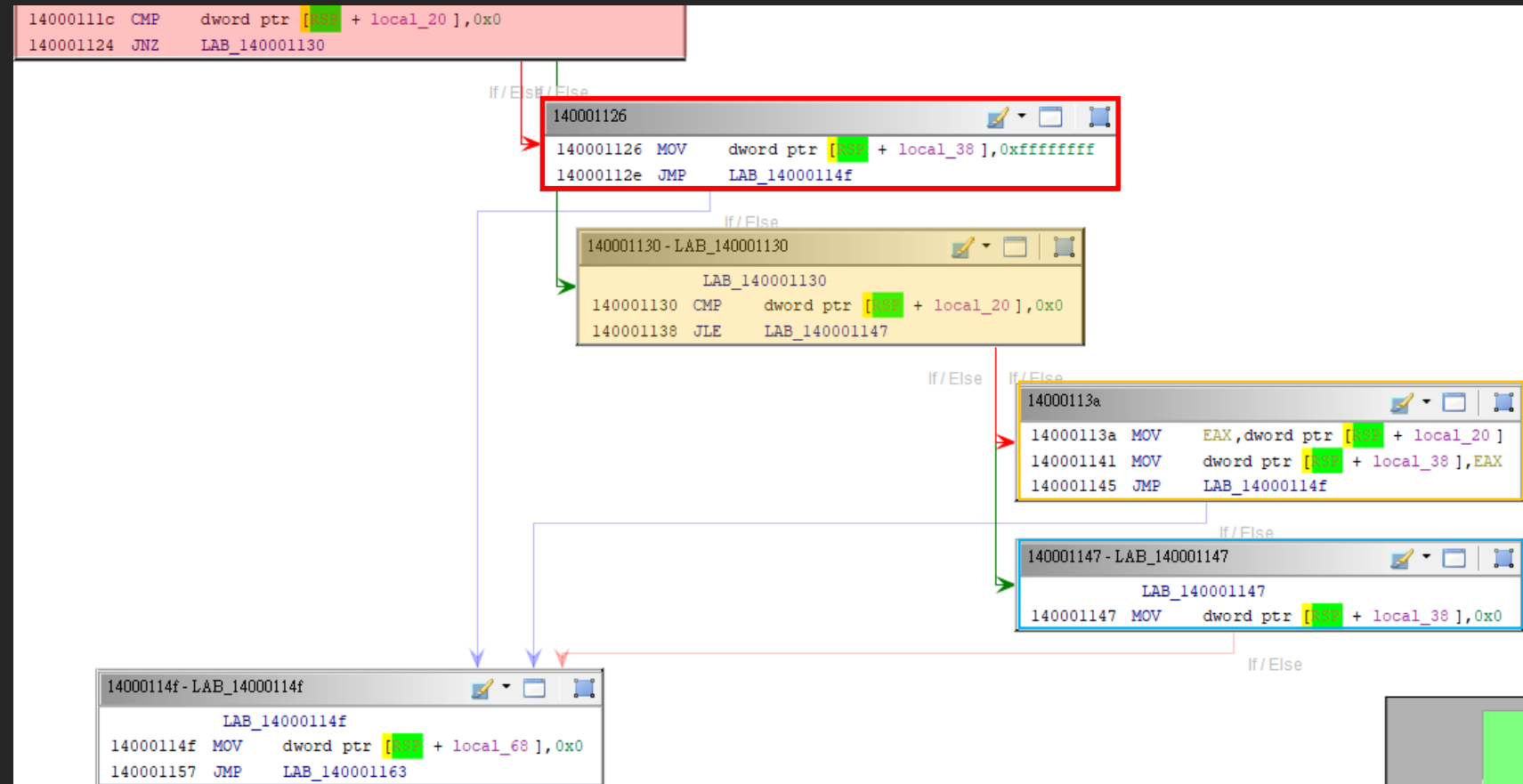
if

```
if (!local_id) {  
    local_info.id = -1;  
}  
else if (local_id > 0) {  
    local_info.id = local_id;  
}  
else {  
    local_info.id = 0;  
}
```



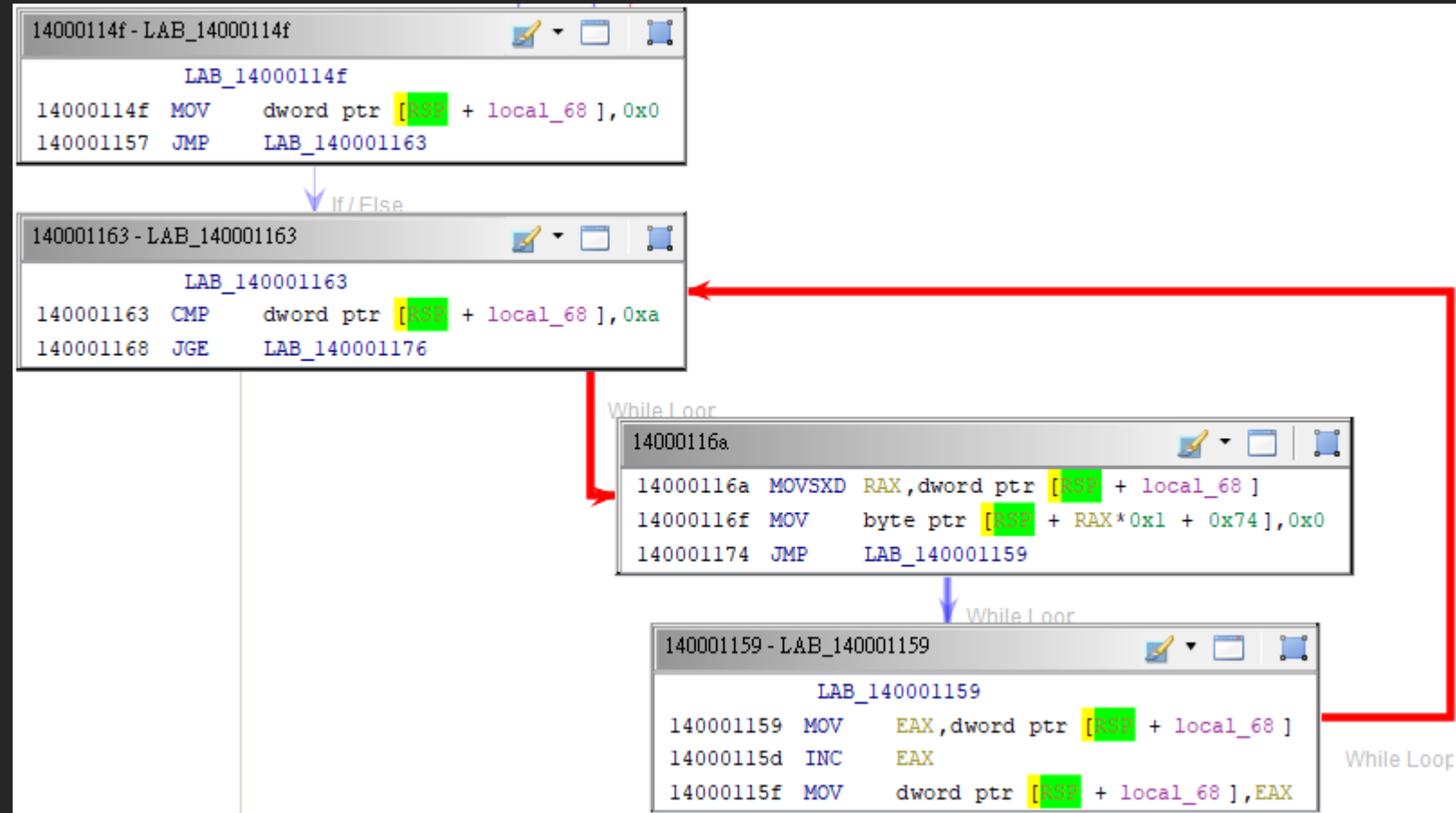
if

```
if (!local_id) {  
    local_info.id = -1;  
}  
else if (local_id > 0) {  
    local_info.id = local_id;  
}  
else {  
    local_info.id = 0;  
}
```



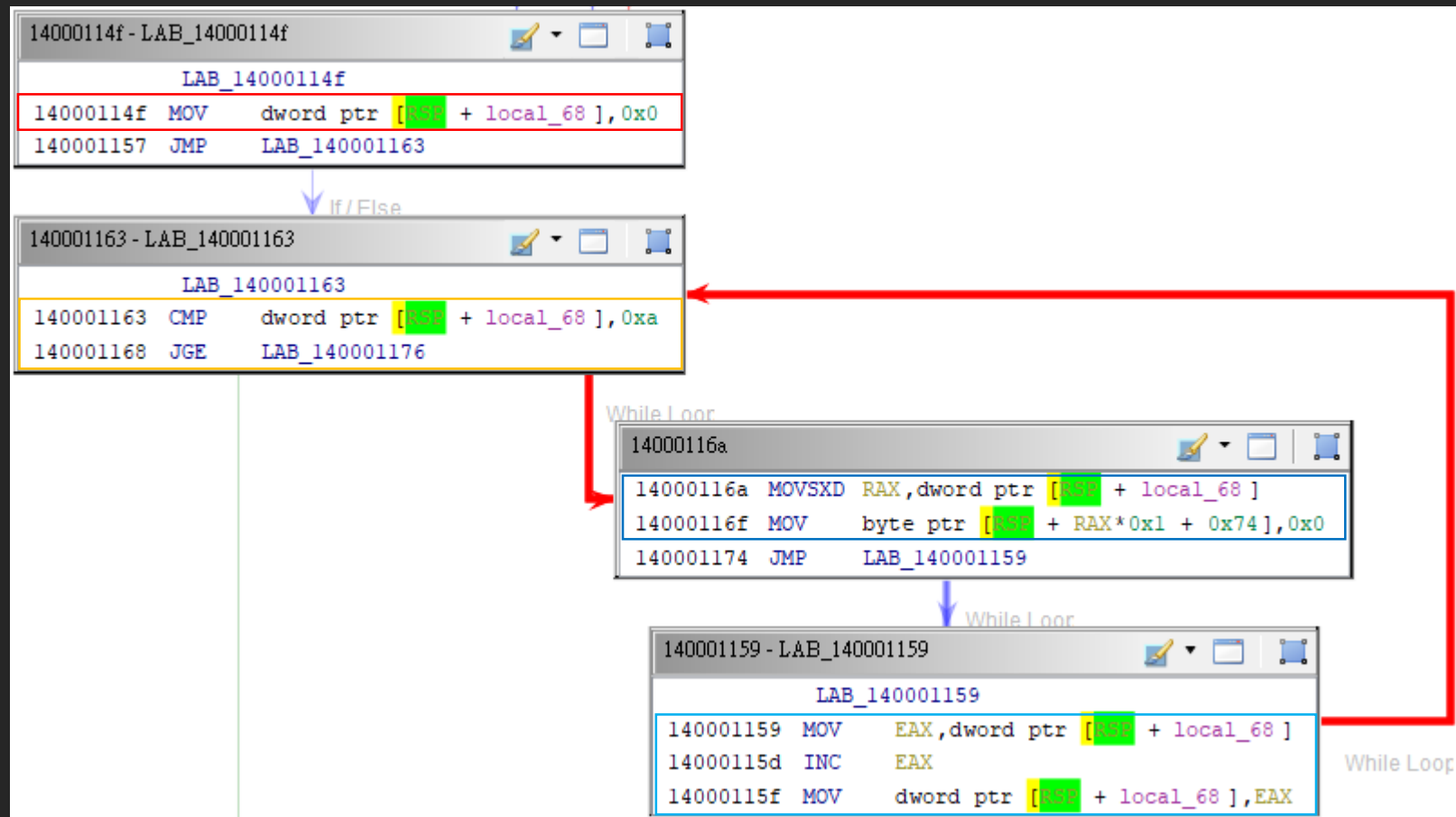
for

```
for (int i = 0; i < NAME_SIZE; ++i) {  
    local_info.name[i] = NULL;  
}
```



for

```
for (int i = 0; i < NAME_SIZE; ++i) {  
    local_info.name[i] = NULL;  
}
```



條件 jmp

- 配合 cmp / test
- 迴圈會往回跳 / If 只會往前跳

je / jz	相同 / 為 0
jne / jnz	不同 / 不為 0
jb / jl	無符號 / 有符號 小於
ja / jg	無符號 / 有符號 大於
jnb / jnl	無符號 / 有符號 不小於
Jna / jng	無符號 / 有符號 不大於

Array

```
int user[4];
```

```
user[0] = 0x10;  
user[1] = 0x11;  
user[2] = 0x12;  
user[3] = 0x13;  
user[4] = 0x14;
```

```
140001079 MOV     EAX, 0x4  
14000107e IMUL    RAX, RAX, 0x0  
140001082 MOV     dword ptr [RSP + RAX*0x1 + 0x50], 0x10  
14000108a MOV     EAX, 0x4  
14000108f IMUL    RAX, RAX, 0x1  
140001093 MOV     dword ptr [RSP + RAX*0x1 + 0x50], 0x11  
14000109b MOV     EAX, 0x4  
1400010a0 IMUL    RAX, RAX, 0x2  
1400010a4 MOV     dword ptr [RSP + RAX*0x1 + 0x50], 0x12  
1400010ac MOV     EAX, 0x4  
1400010b1 IMUL    RAX, RAX, 0x3  
1400010b5 MOV     dword ptr [RSP + RAX*0x1 + 0x50], 0x13  
1400010bd MOV     EAX, 0x4  
1400010c2 IMUL    RAX, RAX, 0x4  
1400010c6 MOV     dword ptr [RSP + RAX*0x1 + 0x50], 0x14
```

Array

一格多大

```
int user[4];
```

```
user[0] = 0x10;  
user[1] = 0x11;  
user[2] = 0x12;  
user[3] = 0x13;  
user[4] = 0x14;
```

第幾格

要寫什麼

一格多大

第幾格

要寫什麼

140001079	MOV	EAX, 0x4
14000107e	IMUL	RAX, RAX, 0x0
140001082	MOV	dword ptr [RSP + RAX*0x1 + 0x50], 0x10
14000108a	MOV	EAX, 0x4
14000108f	IMUL	RAX, RAX, 0x1
140001093	MOV	dword ptr [RSP + RAX*0x1 + 0x50], 0x11
14000109b	MOV	EAX, 0x4
1400010a0	IMUL	RAX, RAX, 0x2
1400010a4	MOV	dword ptr [RSP + RAX*0x1 + 0x50], 0x12
1400010ac	MOV	EAX, 0x4
1400010b1	IMUL	RAX, RAX, 0x3
1400010b5	MOV	dword ptr [RSP + RAX*0x1 + 0x50], 0x13
1400010bd	MOV	EAX, 0x4
1400010c2	IMUL	RAX, RAX, 0x4
1400010c6	MOV	dword ptr [RSP + RAX*0x1 + 0x50], 0x14

Array

```
int user[4];
```

```
user[0] = 0x10;
```

```
user[1] = 0x11;
```

```
user[2] = 0x12;
```

```
user[3] = 0x13;
```

```
user[4] = 0x14;
```

超出範圍照樣能編譯

```
140001079 MOV     EAX, 0x4
14000107e IMUL    RAX, RAX, 0x0
140001082 MOV     dword ptr [RSP + RAX*0x1 + 0x50], 0x10
14000108a MOV     EAX, 0x4
14000108f IMUL    RAX, RAX, 0x1
140001093 MOV     dword ptr [RSP + RAX*0x1 + 0x50], 0x11
14000109b MOV     EAX, 0x4
1400010a0 IMUL    RAX, RAX, 0x2
1400010a4 MOV     dword ptr [RSP + RAX*0x1 + 0x50], 0x12
1400010ac MOV     EAX, 0x4
1400010b1 IMUL    RAX, RAX, 0x3
1400010b5 MOV     dword ptr [RSP + RAX*0x1 + 0x50], 0x13
1400010bd MOV     EAX, 0x4
1400010c2 IMUL    RAX, RAX, 0x4
1400010c6 MOV     dword ptr [RSP + RAX*0x1 + 0x50], 0x14
```

Struct

```
#define NAME_SIZE 10
```

```
typedef struct {  
    int id;  
    char name[NAME_SIZE];  
    char *data;  
} info_struct;
```

```
info_struct local_info;
```

```
local_info.id = 0;  
local_info.name[0] = NULL;  
local_info.data = malloc(0x100);
```

```
1400010f3 MOV     dword ptr [RSP + local_0x70], 0x0  
1400010fb MOV     EAX, 0x1  
140001100 IMUL    RAX, RAX, 0x0  
140001104 MOV     byte ptr [RSP + RAX*0x1 + 0x74], 0x0  
140001109 MOV     param_1, 0x100  
14000110e CALL    qword ptr [->API-MS-WIN-CRT-HEAP-L1-1-0.DLL::malloc]  
140001114 MOV     qword ptr [RSP + local_0x80], RAX
```

Struct

```
#define NAME_SIZE 10
```

```
typedef struct {  
    int id;  
    char name[NAME_SIZE];  
    char *data;  
} info_struct;
```

```
info_struct local_info;
```

```
local_info.id = 0;
```

```
local_info.name[0] = NULL;
```

```
local_info.data = malloc(0x100);
```

1400010f3	MOV	dword ptr [RSP + local_0x70], 0x0
1400010fb	MOV	EAX, 0x1
140001100	IMUL	RAX, RAX, 0x0
140001104	MOV	byte ptr [RSP + RAX*0x1 + 0x74], 0x0
140001109	MOV	param_1, 0x100
14000110e	CALL	qword ptr [->API-MS-WIN-CRT-HEAP-L1-1-0.DLL::malloc]
140001114	MOV	qword ptr [RSP + local_0x80], RAX

Struct

```
#define NAME_SIZE 10
```

```
typedef struct {  
    int id;  
    char name[NAME_SIZE];  
    char *data;  
} info_struct;
```

```
info_struct local_info;
```

```
local_info.id = 0;
```

```
local_info.name[0] = NULL;
```

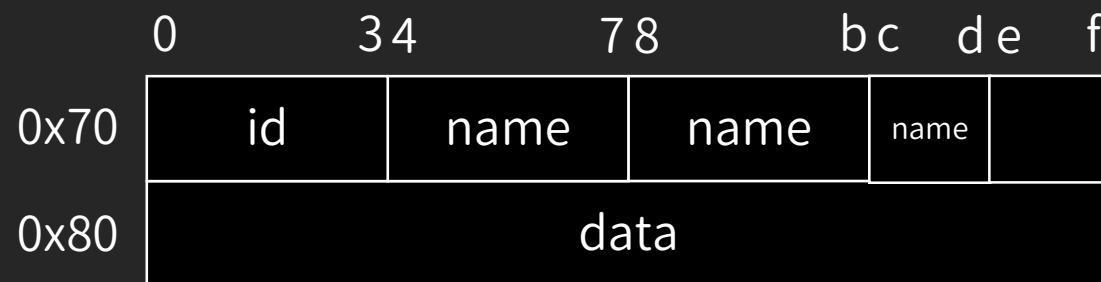
```
local_info.data = malloc(0x100);
```

```
1400010f3 MOV    dword ptr [RSP + local_0x70], 0x0  
1400010fb MOV    EAX, 0x1  
140001100 IMUL   RAX, RAX, 0x0  
140001104 MOV    byte ptr [RSP + RAX*0x1 + 0x74], 0x0  
140001109 MOV    param_1, 0x100  
14000110e CALL    qword ptr [->API-MS-WIN-CRT-HEAP-L1-1-0.DLL::malloc]  
140001114 MOV    qword ptr [RSP + local_0x80], RAX
```

可以觀察到 compiler 將 RSP + 0x70 的位址當 id

RSP + 0x74 的位址當 name 起始位址

將 RSP + 0x80 當 data



Struct

```
#define NAME_SIZE 10
```

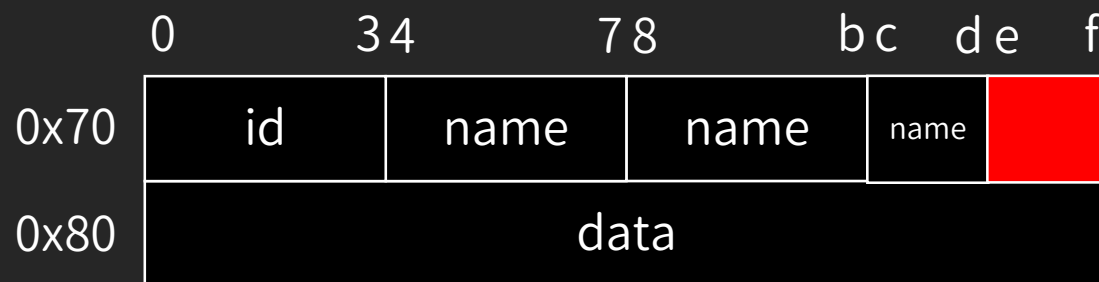
```
typedef struct {  
    int id;  
    char name[NAME_SIZE];  
    char *data;  
} info_struct;
```

```
info_struct local_info;
```

```
local_info.id = 0;  
local_info.name[0] = NULL;  
local_info.data = malloc(0x100);
```

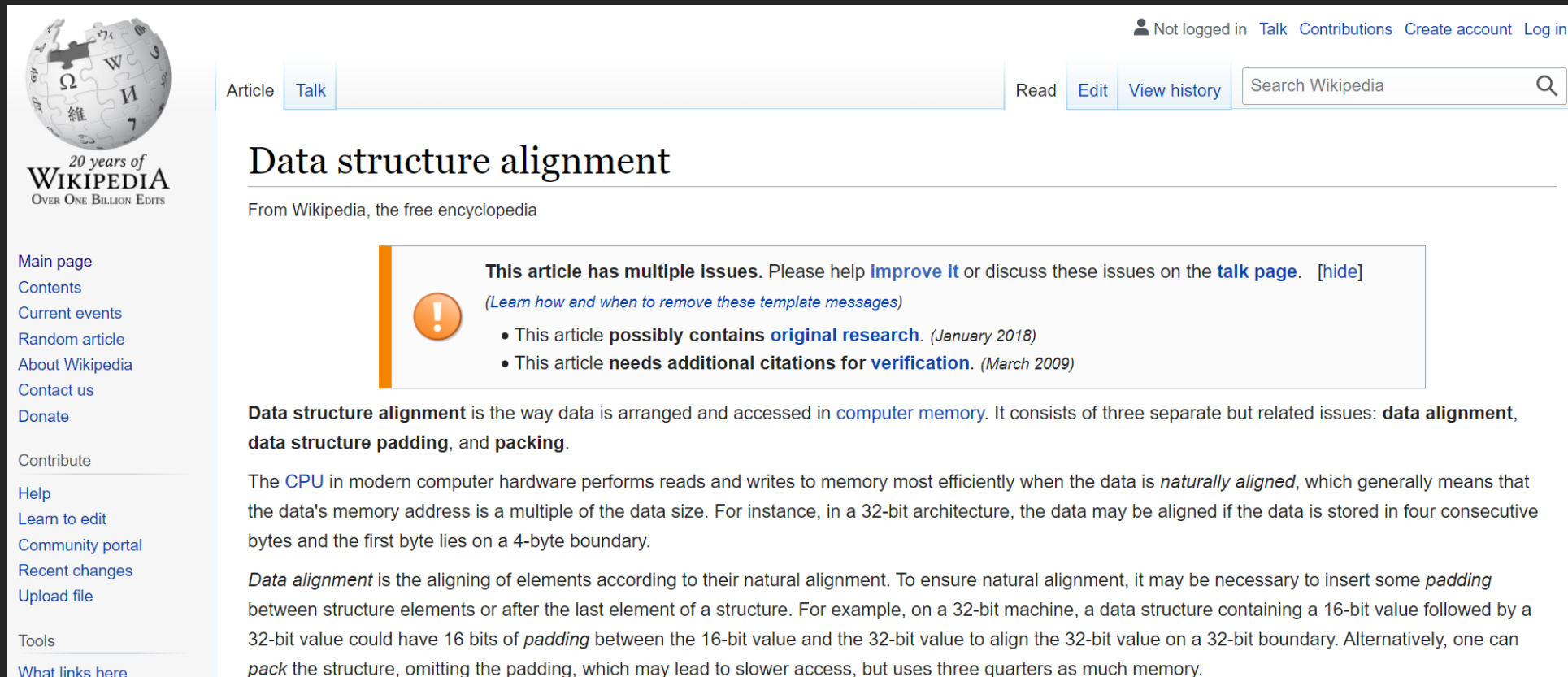
```
1400010f3 MOV     dword ptr [RSP + local_0x70], 0x0  
1400010fb MOV     EAX, 0x1  
140001100 IMUL    RAX, RAX, 0x0  
140001104 MOV     byte ptr [RSP + RAX*0x1 + 0x74], 0x0  
140001109 MOV     param_1, 0x100  
14000110e CALL    qword ptr [->API-MS-WIN-CRT-HEAP-L1-1-0.DLL::malloc]  
140001114 MOV     qword ptr [RSP + local_0x80], RAX
```

有兩 Byte 沒有用到



Struct

- Struct alignment



The screenshot shows the Wikipedia article for "Data structure alignment". At the top left is the Wikipedia logo with the text "20 years of WIKIPEDIA OVER ONE BILLION EDITS". To the right of the logo is a sidebar with links: Main page, Contents, Current events, Random article, About Wikipedia, Contact us, Donate, Contribute, Help, Learn to edit, Community portal, Recent changes, Upload file, Tools, and What links here. The main content area has a header with "Article" and "Talk" tabs, and a search bar. The title "Data structure alignment" is prominently displayed. Below the title is a sub-header "From Wikipedia, the free encyclopedia". A warning box with an orange exclamation mark icon contains the text: "This article has multiple issues. Please help [improve it](#) or discuss these issues on the [talk page](#). [hide] (Learn how and when to remove these template messages)". Below this, two bullet points are listed: "• This article **possibly contains original research**. (January 2018)" and "• This article **needs additional citations for verification**. (March 2009)". The main text of the article begins with "Data structure alignment is the way data is arranged and accessed in [computer memory](#). It consists of three separate but related issues: **data alignment**, **data structure padding**, and **packing**." The text continues: "The [CPU](#) in modern computer hardware performs reads and writes to memory most efficiently when the data is *naturally aligned*, which generally means that the data's memory address is a multiple of the data size. For instance, in a 32-bit architecture, the data may be aligned if the data is stored in four consecutive bytes and the first byte lies on a 4-byte boundary." The article then defines "Data alignment" as the aligning of elements according to their natural alignment. To ensure natural alignment, it may be necessary to insert some *padding* between structure elements or after the last element of a structure. For example, on a 32-bit machine, a data structure containing a 16-bit value followed by a 32-bit value could have 16 bits of *padding* between the 16-bit value and the 32-bit value to align the 32-bit value on a 32-bit boundary. Alternatively, one can *pack* the structure, omitting the padding, which may lead to slower access, but uses three quarters as much memory.

Call function

```
printf("local_id : %d\n"  
      "&local_id : %p\n"  
      "p : %p\n"  
      "&p : %p\n"  
      "pp : %p\n"  
      "name : %s\n", local_id, &local_id, p, &p, pp, local_info.name);
```

undefined8	RCX:8	param_1
undefined8	RDX:8	param_2
undefined8	R8:8	param_3
undefined8	R9:8	param_4

```
1400011f4 LEA RAX=>local_34,[RSP + 0x74]  
1400011f9 MOV qword ptr [RSP + local_78],RAX  
1400011fe MOV RAX,qword ptr [RSP + local_60]  
140001203 MOV qword ptr [RSP + local_80],RAX  
140001208 LEA RAX=>local_40,[RSP + 0x68]  
14000120d MOV qword ptr [RSP + local_88],RAX  
140001212 MOV param_4,qword ptr [RSP + local_40]  
140001217 LEA param_3=>local_20,[RSP + 0x88]  
14000121f MOV param_2,dword ptr [RSP + local_20]  
140001226 LEA param_1,[s_local_id:_%d&local_id:_%p_p:_140004030]  
14000122d CALL FUN_140001290
```

x86 Calling Convention

- 規定了呼叫函數時如何傳遞參數
- Windows
 - `Function(rcx, rdx, r8, r9)`
- Linux
 - `Function(rdi, rsi, rdx, rcx, r8, r9)`
- 多的參數會放到 stack 上

區域變數 vs 全域變數

```
global_id = 100;  
local_id = 1;
```

```
1400010de MOV     dword ptr [DAT_1400046b8],0x64  
1400010e8 MOV     dword ptr [RSP + local_20],0x1
```

區域變數 vs 全域變數

- 全域變數: 直接寫到特定位址
- 區域變數: 以 RSP 來定位

```
global_id = 100;  
local_id = 1;
```

```
1400010de MOV     dword ptr [DAT_1400046b8],0x64  
1400010e8 MOV     dword ptr [RSP + local_20],0x1
```

x64dbg

hello.exe - PID: 3988 - 模組: ntdll.dll - 執行緒: 主執行緒 6808 - x64dbg

檔案(F) 檢視(V) 除錯(D) 追蹤(N) 外掛程式(P) 最愛(I) 選項(O) 幫助(H) Jul 1 2021 (TitanEngine)

CPU 日誌 筆記 中斷點 記憶體映射 呼叫堆疊 SEH鏈 腳本 符號 原始碼 引用 執行緒 Handles 追蹤

組合語言

暫存器

資料視窗

Stack

19

命令: Commands are comma separated (like assembly instructions): mov eax, ebx

暫停 已到達系統中斷點!

已花費時間: 0.00:41.22

Endian

- Byte 的順序
- 一個整數 0x12345678，兩種儲存方式

0	1	2	3
0x78	0x56	0x34	0x12

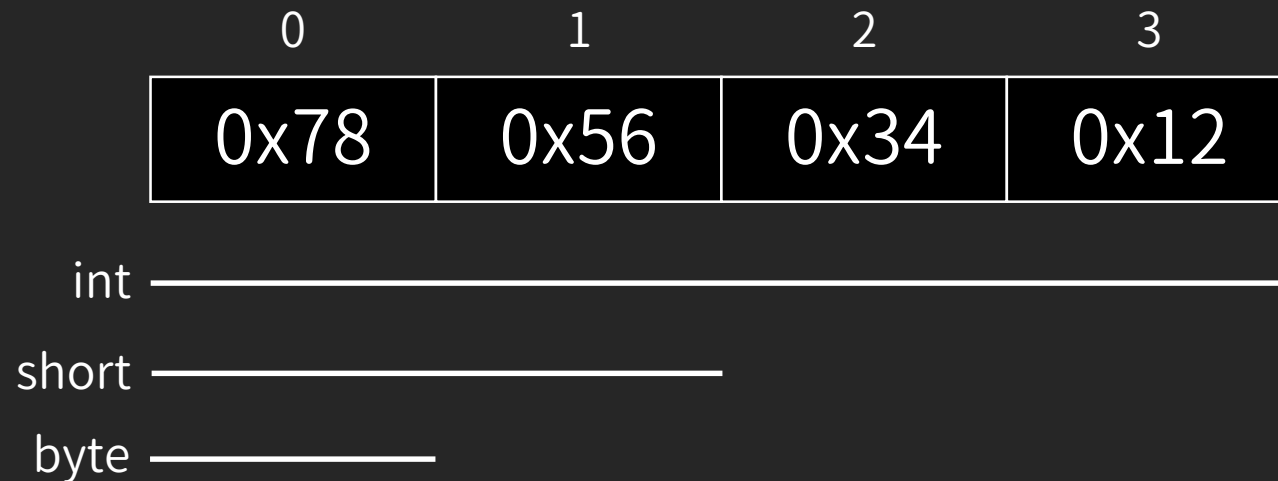
Little Endian

0	1	2	3
0x12	0x34	0x56	0x78

Big Endian

Endian

- 常見是用 Little Endian
- 將 int 0x12345678 轉成 short 0x5678, 起始位址不用改變



Little Endian

x64dbg

hello.exe - PID: 3988 - 模組: ntdll.dll - 執行緒: 主執行緒 6808 - x64dbg

檔案(F) 檢視(V) 除錯(D) 追蹤(N) 外掛程式(P) 最愛(I) 選項(O) 幫助(H) Jul 1 2021 (TitanEngine)

CPU 日誌 筆記 中斷點 記憶體映射 呼叫堆疊 SEH鏈 腳本 符號 原始碼 引用 執行緒 Handles 追蹤

組合語言

暫存器

資料視窗

Stack

命令: Commands are comma separated (like assembly instructions): mov eax, ebx

已花費時間: 0:00:41.22

Stack Frame

- Q1: 函數都是以 RSP 或 RBP 來定位區域變數，那怎麼區別不同函數的區域變數？
- Q2: 呼叫函數後，RIP 就從 A 函數跑到 B 函數了，要怎麼 return 回 A 函數？
- 如果不知道答案，那你就需要看一下這章

Stack Frame

- 不同區域會有不同的 Stack Frame
 - 裡面存放著區域變數
- 在 Function 的頭部和尾部, 有一些用來處理 Stack Frame 的指令
 - 頭部: Prologue
 - 尾部: Epilogue

main

```
push rbp  
mov rbp, rsp
```

...

```
leave  
ret
```

Stack Frame

main

```
push rbp
mov rbp, rsp
sub rsp, 20h
...
call function1
leave
ret
```

function1

```
push rbp
mov rbp, rsp
sub rsp, 30h
...
leave
ret
```

0x00007fffffffe5c8

RSP

Stack

Stack Frame

main

```
push rbp
mov rbp, rsp
sub rsp, 20h
...
call function1
leave
ret
```

function1

```
push rbp
mov rbp, rsp
sub rsp, 30h
...
leave
ret
```

0x00007fffffff5c8

RSP

Stack

Stack Frame

main

```
push rbp
mov rbp, rsp
sub rsp, 20h
...
call function1
leave
ret
```

function1

```
push rbp
mov rbp, rsp
sub rsp, 30h
...
leave
ret
```

0x00007fffffff5c0

0x00007fffffff5c8

RBP 原本的值

RSP

Stack

Stack Frame

main

```
push rbp
mov rbp, rsp
sub rsp, 20h
...
call function1
leave
ret
```

function1

```
push rbp
mov rbp, rsp
sub rsp, 30h
...
leave
ret
```

0x00007fffffff5c0

0x00007fffffff5c8

RBP 原本的值

RSP RBP

Stack

Stack Frame

main

```
push rbp
mov rbp, rsp
sub rsp, 20h
```

...

```
call function1
leave
ret
```

function1

```
push rbp
mov rbp, rsp
sub rsp, 30h
...
```

```
leave
ret
```

0x00007fffffff5a0

RSP

Main Stack Frame

0x00007fffffff5c0

RBP

RBP 原本的值

0x00007fffffff5c8

Stack

Stack Frame

main

```
push rbp
mov rbp, rsp
sub rsp, 20h
...
```

call function1

0x401234 leave
ret

function1

```
push rbp
mov rbp, rsp
sub rsp, 30h
...
```

```
leave
ret
```

0x00007fffffff5a0

RSP

0x00007fffffff5c0

0x00007fffffff5c8

Main Stack Frame

RBP 原本的值

RBP

Stack

Stack Frame

main

```
push rbp
mov rbp, rsp
sub rsp, 20h
...
call function1
leave
ret
```

function1

```
push rbp
mov rbp, rsp
sub rsp, 30h
...
leave
ret
```

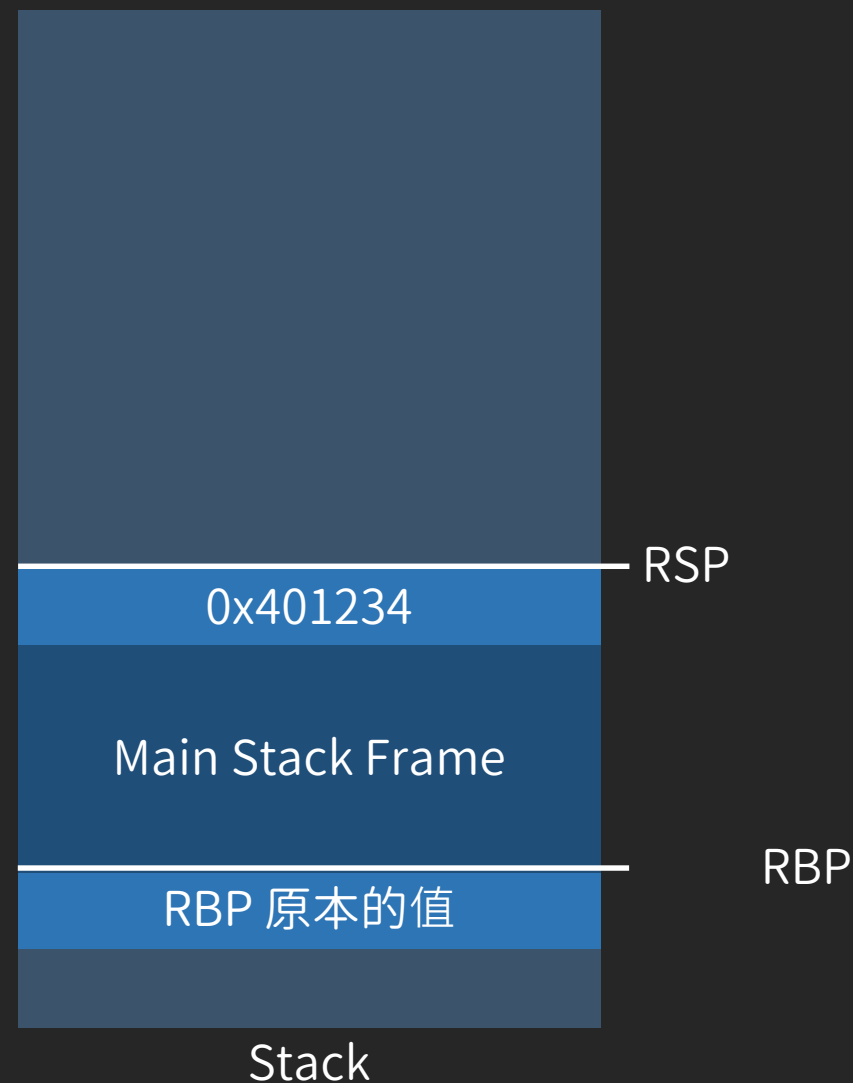
0x401234

0x00007fffffff598

0x00007fffffff5a0

0x00007fffffff5c0

0x00007fffffff5c8



Stack Frame

main

```
push rbp
mov rbp, rsp
sub rsp, 20h
...
call function1
leave
ret
```

function1

```
push rbp
mov rbp, rsp
sub rsp, 30h
...
leave
ret
```

0x401234

0x00007fffffff590

0x00007fffffff598

0x00007fffffff5a0

0x00007fffffff5c0

0x00007fffffff5c8

0x00007fffffff5c0

0x401234

Main Stack Frame

RBP 原本的值

Stack

RSP

RBP

Stack Frame

main

```
push rbp
mov rbp, rsp
sub rsp, 20h
...
call function1
leave
ret
```

function1

```
push rbp
mov rbp, rsp
sub rsp, 30h
...
leave
ret
```

0x401234

0x00007fffffff590

0x00007fffffff598

0x00007fffffff5a0

0x00007fffffff5c0

0x00007fffffff5c8

0x00007fffffff5c0

0x401234

Main Stack Frame

RBP 原本的值

Stack

RSP RBP

Stack Frame

main

```
push rbp
mov rbp, rsp
sub rsp, 20h
...
call function1
leave
ret
```

function1

```
push rbp
mov rbp, rsp
sub rsp, 30h
...
leave
ret
```

0x00007fffffffef560

RSP

Function1
Stack Frame

0x00007fffffffef590

RBP

0x00007fffffffef5c0

0x00007fffffffef598

0x401234

0x00007fffffffef5a0

Main Stack Frame

0x00007fffffffef5c0

RBP 原本的值

0x00007fffffffef5c8

Stack

0x401234

Stack Frame

main

```
push rbp
mov rbp, rsp
sub rsp, 20h
...
call function1
leave
ret
```

0x401234

function1

```
push rbp
mov rbp, rsp
sub rsp, 30h
...
leave
ret
```

```
leave
=
mov rsp, rbp
pop rbp
```

0x00007fffffffef560

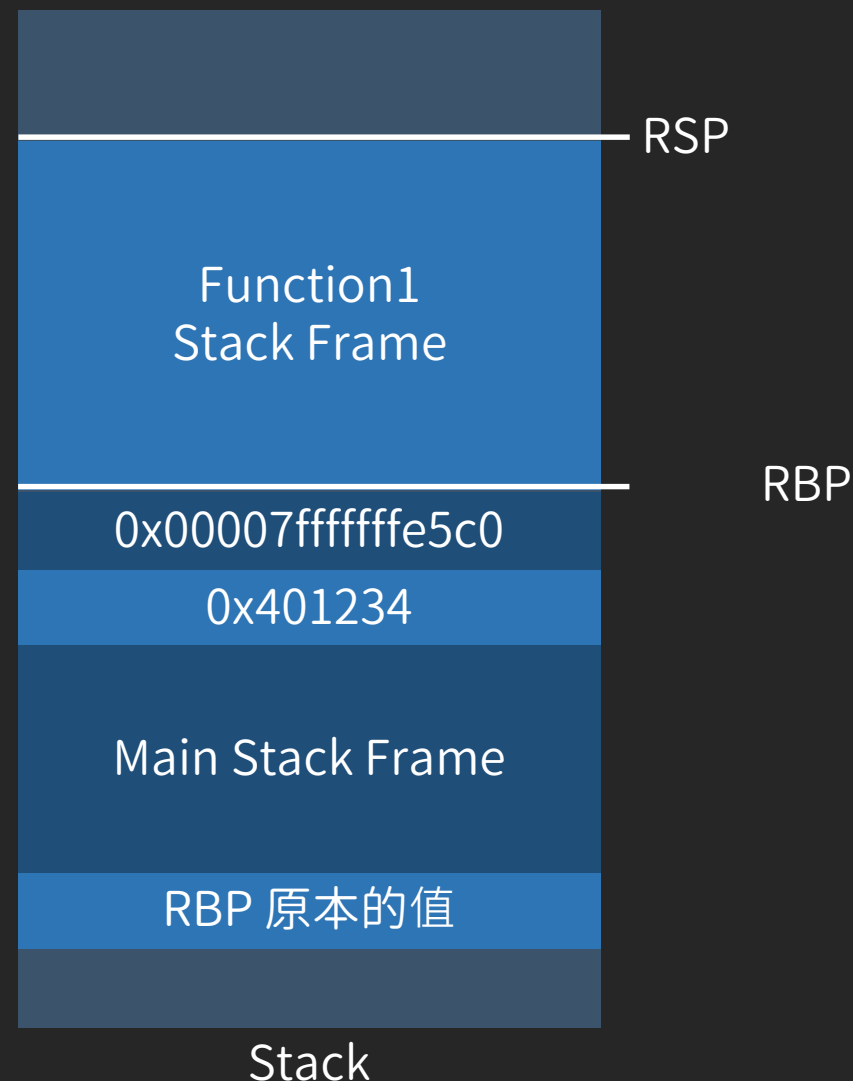
0x00007fffffffef590

0x00007fffffffef598

0x00007fffffffef5a0

0x00007fffffffef5c0

0x00007fffffffef5c8



Stack Frame

main

```
push rbp
mov rbp, rsp
sub rsp, 20h
...
call function1
leave
ret
```

function1

```
push rbp
mov rbp, rsp
sub rsp, 30h
...
leave
ret
```

```
leave
=
mov rsp, rbp
pop rbp
```

0x00007fffffffef560

0x00007fffffffef590

0x00007fffffffef598

0x00007fffffffef5a0

0x00007fffffffef5c0

0x00007fffffffef5c8

Function1
Stack Frame

0x00007fffffffef5c0

0x401234

Main Stack Frame

RBP 原本的值

RSP

RBP

Stack

0x401234

Stack Frame

main

```
push rbp
mov rbp, rsp
sub rsp, 20h
...
call function1
leave
ret
```

function1

```
push rbp
mov rbp, rsp
sub rsp, 30h
...
leave
ret
```

```
leave
=
mov rsp, rbp
pop rbp
```

0x00007fffffffef560

Function1
Stack Frame

0x00007fffffffef590

0x00007fffffffef598

0x00007fffffffef5a0

0x00007fffffffef5c0

0x401234

RSP

Main Stack Frame

0x00007fffffffef5c0

0x00007fffffffef5c8

RBP 原本的值

RBP

Stack

Stack Frame

main

```
push rbp
mov rbp, rsp
sub rsp, 20h
...
call function1
leave
ret
```

function1

```
push rbp
mov rbp, rsp
sub rsp, 30h
...
leave
ret
```

```
leave
=
mov rsp, rbp
pop rbp
```

0x00007fffffffef560

Function1
Stack Frame

0x00007fffffffef590

0x00007fffffffef598

0x00007fffffffef5a0

0x00007fffffffef5c0

0x401234

Main Stack Frame

0x00007fffffffef5c0

0x00007fffffffef5c8

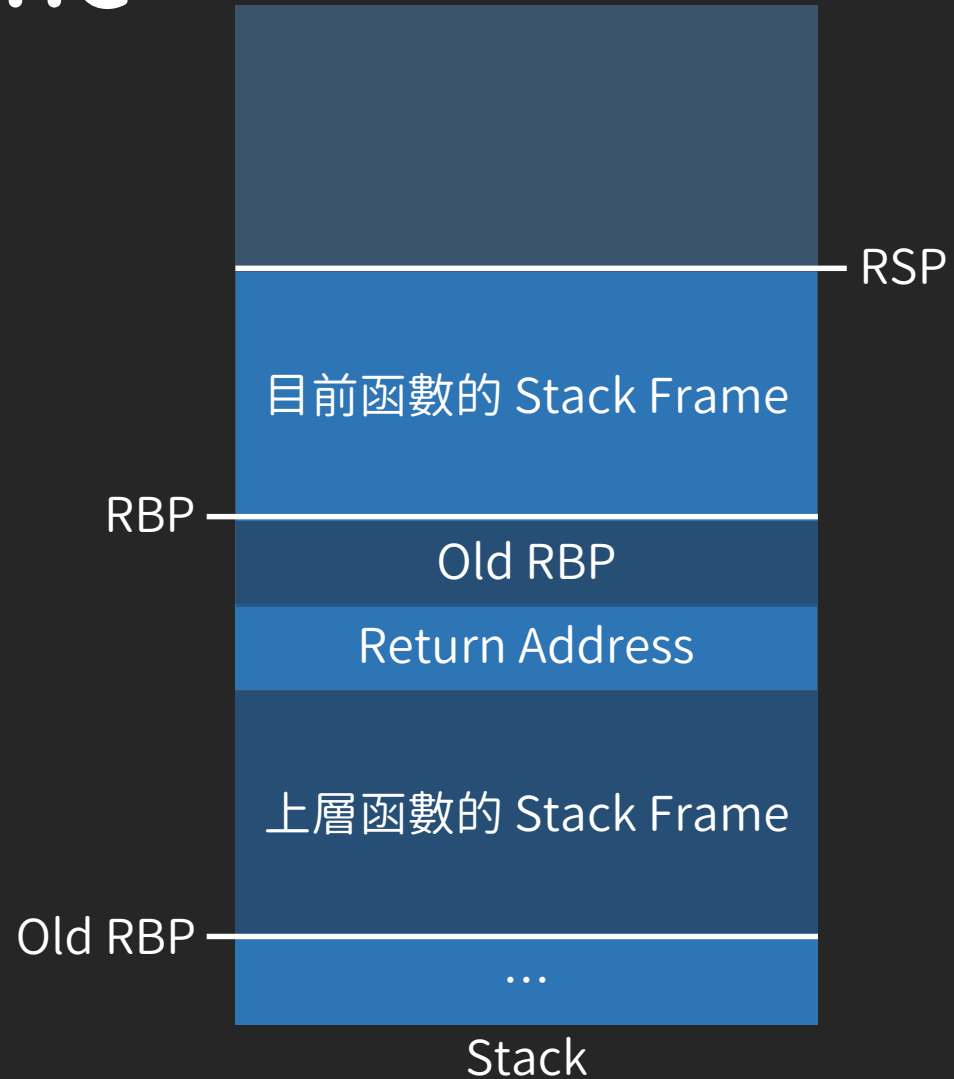
RBP 原本的值

RSP

Stack

Stack Frame

- 統整一下



Stack Frame

- Q1: 函數都是以 RSP 或 RBP 來定位區域變數，那怎麼區別不同函數的區域變數？
 - A1: 想辦法讓不同函數的 stack 區域不同
- Q2: 呼叫函數後, RIP 就從 A 函數跑到 B 函數了，要怎麼 return 回 A 函數？
 - A2: 在呼叫 B 函數前把下一條指令 push 進 stack
B 函數執行 ret
把 A 函數下一條指令從 stack pop 回 rip
進而回到 A 函數

x86 組合語言

- 看完這個章節之後，你應該…
 - 略懂暫存器
 - 略懂 x86 組合語言
 - 略懂 endian
 - 略懂 stack frame
 - 會用 Ghidra 找到 main 函數

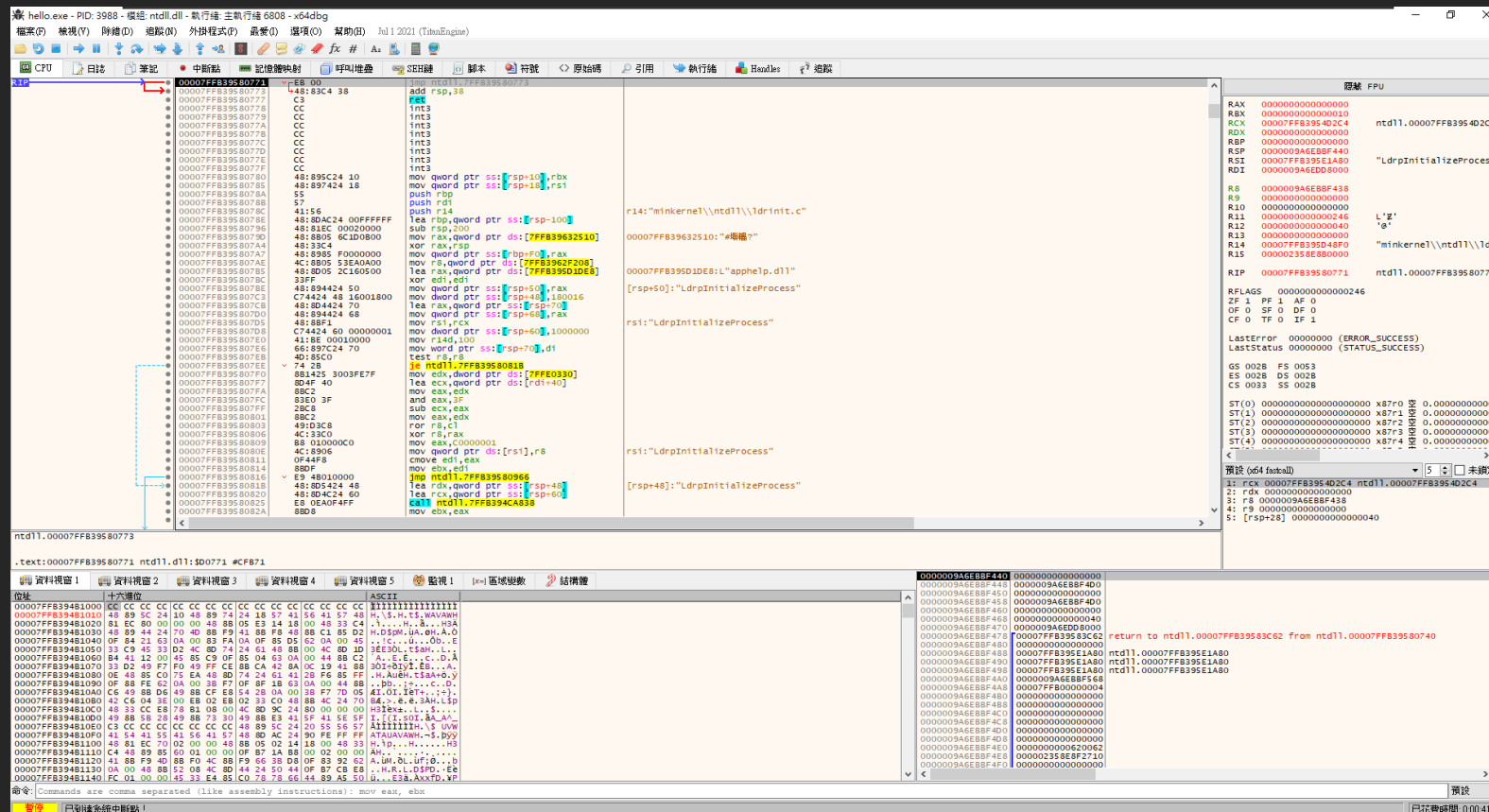
分析方法

分析方法

- 分析方法分成動態 / 靜態分析

分析方法

- 如果你用工具來讓程序跑跑停停,在這過程中分析程序內部邏輯,這叫做動態分析



分析方法

- 如果你只看反組譯/反編譯的 code 來分析這個叫做靜態分析



Function Graph [CodeBrowser: hello/hello.exe]

File Edit Navigation Search Select Help

Function Graph - FUN_1400010e0 - 9 vertices (hello.exe)

```
undefined1[16] Stack[-0x270... local_270
undefined1[16] Stack[-0x280... local_280
undefined1 Stack[-0x284... local_284
undefined4 Stack[-0x288... local_288
undefined8 Stack[-0x298... local_298
undefined4 Stack[-0x2a0... local_2a0
undefined8 Stack[-0x2a8... local_2a8

FUN_1400010e0
1400010e0 MOV     qword ptr [RSP + local_res8],RDX
1400010e5 MOV     qword ptr [RSP + local_res10],RBP
1400010ea MOV     qword ptr [RSP + local_res18],RSI
1400010ef PUSH    RDI
1400010f0 SUB     RSP,0x2c0
1400010f7 MOV     RAX,qword ptr [DAT_140005008]
1400010fe XOR     RAX,RSP
140001101 MOV     qword ptr [RSP + local_18],RAX
140001109 XORPS   XMM0,XMM0
14000110c XOR     param_1,param_1
14000110e MOVUPS  xmmword ptr [RSP + local_258[0]],XMM0
140001113 MOVUPS  xmmword ptr [RSP + local_248[0]],XMM0
14000111b CALL    qword ptr [->API-MS-WIN-CRT-TIME-L1-1-0.DLL:_time64]
140001121 MOV     param_1,RAX
140001124 CALL    qword ptr [->API-MS-WIN-CRT-UTILITY-L1-1-0.DLL:_srand]
14000112a LEA     param_1,[s_What's_your_name?_140003290]
140001131 CALL    FUN_140001020
140001136 LEA     param_2=>local_258,[RSP + 0x70]
14000113b LEA     param_1,[DAT_1400032a4]
140001142 CALL    FUN_140001080
140001147 LEA     param_2=>local_258,[RSP + 0x70]
14000114c LEA     param_1,[s_Hello,_%s!_1400032b0]
140001153 CALL    FUN_140001020
140001158 XOR     EBP,EBP
14000115a MOV     ESI,0x5
14000115f NOP
```

140001160 - LAB_140001160

```
LAB_140001160
140001160 CALL    qword ptr [->API-MS-WIN-CRT-TIME-L1-1-0.DLL:_time64]
140001166 MOV     EDI,EAX
140001168 MOV     EAX,0x66666667
```

分析方法

- 實務上為兩者交叉使用,看個人喜好方式
- 靜態分析工具 (反組譯, 反編譯, 分析函數, 重新命名函數/變數)
 - IDA
 - Ghidra
- 動態分析工具 (設定中斷點, 觀察暫存器、記憶體內容, 觀察位址空間)
 - x64dbg
 - windbg
 - gdb

分析方法

- 接下來示範一下動靜態交叉分析
 - 動態分析工具選用 x64dbg
 - 靜態分析工具選用 Ghidra
- 前面的章節我們已經用 Ghidra 找到 hello.exe 的 main 了
- 接續著介紹其他功能

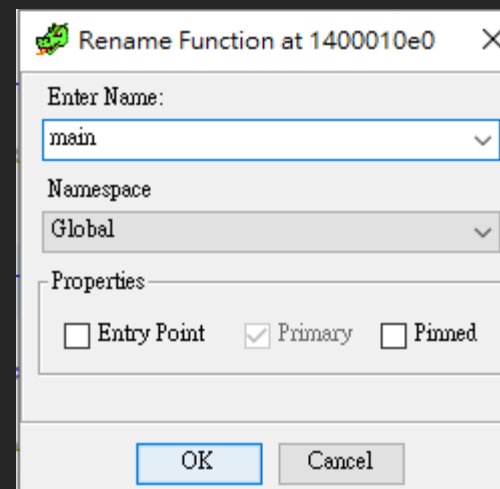
Ghidra

- 各種快捷鍵 : Ghidra cheatsheet

Key			Markup			Cycle Integer Types		
Action Context	Mods + Key	Menu → Path	Undo	Ctrl+Z	Edit → Undo		❖ → Data → Cycle → byte, word, dword, qword	
The action may only be available in the given context.			Redo	Ctrl+Shift+Z	Edit → Redo		❖ → Data → Cycle → char, string, unicode	
❖ indicates the context menu, i.e., right-click.			Save Program	Ctrl+S	File → Save <i>program name</i>		❖ → Data → Cycle → float, double	
The Ctrl key is replaced by the command key on Macintosh.			Disassemble	D	❖ → Disassemble		❖ → Data → Create Array	
Load Project/Program			Clear Code/Data	C	❖ → Clear Code Bytes		❖ → Data → pointer	
New Project	Ctrl+N	File → New Project	Add Label	L	❖ → Add Label		❖ → Data → Create Structure	
Open Project	Ctrl+O	File → Open Project	Edit Label	L	❖ → Edit Label		❖ → New → Structure	
Close Project ¹	Ctrl+W	File → Close Project	Rename Function	L	❖ → Function → Rename Function		File → Parse C Source	
Save Project ¹	Ctrl+S	File → Save Project	Remove Label	Del	❖ → Remove Label		❖ → References → Show References to <i>context</i>	
Import File ¹	I	File → Import File	Remove Function	Del	❖ → Function → Delete Function		² When possible, arrays and pointers are created of the data type currently applied.	
Export Program	O	File → Export Program	Define Data	T	❖ → Data → Choose Data Type ❖ → Data → <i>type</i>		Miscellaneous	
Open File System ¹	Ctrl+I	File → Open File System	Repeat Define Data	Y	❖ → Data → Last Used: <i>type</i>		Select	Select → <i>what</i>
¹ These actions are only available if there is an active project. Create or open a project first.			Rename Variable	L	❖ → Rename Variable		Program Differences	² Tools → Program Differences
Help/Customize/Info							Rerun Script	Ctrl+Shift+R
Ghidra Help	F1	Help → Contents					Assemble	Ctrl+Shift+G
About Ghidra		Help → About Ghidra						❖ → Patch Instruction

Ghidra

- 函數改名
- 對我們剛找到的 main 函數按 L 改名
- 已經點進去了想退回來? 按 Alt + ←



Ghidra

- 變數改名
- 接著也幫 main 的參數改名一下, 依序改成 argc, argv, envp

```
Decompile: main - (hello.exe)
1
2 void main(undefined8 argc,undefined8 argv,undefined8 envp,LPDWORD param_4)
```

Ghidra

- 變數改名
- 改掉裡面一些區域變數的名稱, 比如說...

```
time0 = _time64((__time64_t *)0x0);  
srand((uint)time0);
```

Ghidra

- 看到一個不知道在幹嘛的函數, 你有兩個選項
- 一, 點進去乖乖把他逆完
- 二, 用動態分析工具直接跑他, 觀察發生什麼事情來推斷這函數在幹嘛

```
FUN_140001020("What\'s your name?\n", argv, envp, param_4);  
FUN_140001080(&DAT_1400032a4, local_258, envp, param_4);  
FUN_140001020("Hello, %s!\n", local_258, envp, param_4);
```


Ghidra

- 來示範一下第二個選項
- 先看一下他的記憶體位址
- 在 0x140001131 呼叫了 FUN_140001020

```
FUN_140001020("What\'s your name?\n",argv,envp,param_4);  
FUN_140001080(&DAT_1400032a4,local_258,envp,param_4);  
FUN_140001020("Hello, %s!\n",local_258,envp,param_4);
```

14000112a	LEA	argc, [s_What's_your_name?_140003290]
140001131	CALL	FUN_140001020

Ghidra

- 在 0x140001131 呼叫了 FUN_140001020
- 用 PE-Bear 看 ImageBase 多少: 0x140000000
- $RVA = VA - ImageBase$
= 0x140001131 - 0x140000000
= 0x1131

Disasm: .rdata	General	DOS Hdr	Rich Hdr	File Hdr	Optional Hdr
Offset	Name	Value		Value	
130	Image Base	140000000			

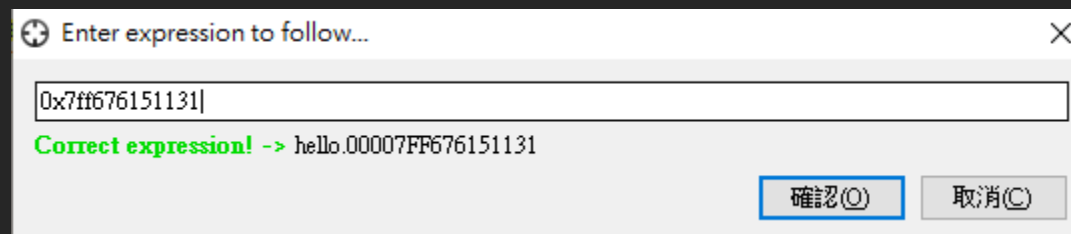
x64dbg

- 將 hello.exe 拖進去 x64dbg
- 先找到這次 ASLR 隨機產生的 ImageBase: 0x7ff676150000
- $VA = ImageBase + RVA$
 $= 0x7ff676150000 + 0x1131 = 0x7ff676151131$

64 CPU 日誌 筆記 中斷點 記憶體映射		
00007FF676150000	0000000000001000	hello.exe
00007FF676151000	0000000000002000	".text"
00007FF676153000	0000000000002000	".rdata"
00007FF676155000	0000000000001000	".data"
00007FF676156000	0000000000001000	".pdata"
00007FF676157000	0000000000001000	".rsrc"
00007FF676158000	0000000000001000	".reloc"

x64dbg

- 按 ctrl+g 後, 輸入想去的位址, 這邊我們想去 0x7ff676151131



x64dbg

- 我們就到了執行時期中, 我們想觀察的函數的位址了
- 點一下那條 `call hello.7ff676151020` 指令
- 按 F2 設定中斷點

●	00007FF67615112A	48:8D0D 5F210000	<code>lea rcx,qword ptr ds:[7FF676153290]</code>	00007FF676153290:"What's your name?\n"
●	00007FF676151131	E8 EAFEFFFF	<code>call hello.7FF676151020</code>	

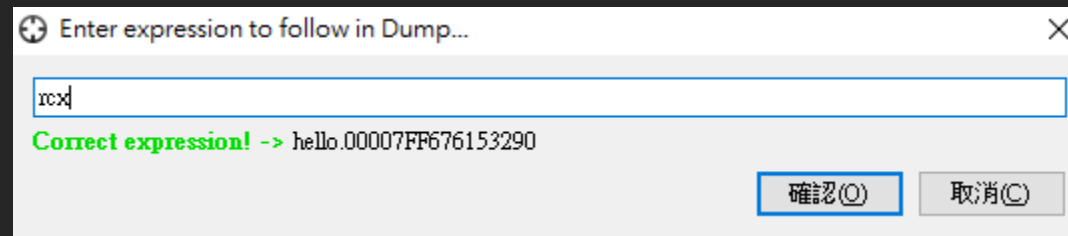
x64dbg

- 設定好中斷點後, 就能按 f9 讓他繼續執行
- 再多按幾次 f9, 讓程式跑到我們設定的中斷點

RIP	00007FF67615112A	48:8D0D 5F210000	lea rcx,qword ptr ds:[7FF676153290]	00007FF676153290:"What's your name?\n"
	00007FF676151131	E8 EAFEFFFF	call hello.7FF676151020	
	00007FF676151136	48:8D5424 70	lea rdx,qword ptr ss:[rsp+70]	

x64dbg

- 這時候觀察一下該 function call 之前設定了哪些參數
- 只有設定了 rcx, 看一下 rcx 內容
- 點一下資料視窗中間, 然後按 ctrl+g, 輸入 rcx



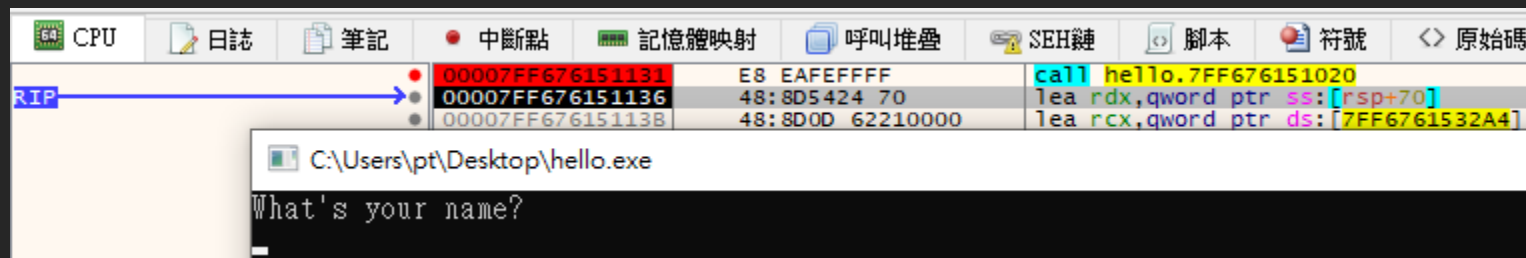
x64dbg

- 看到了 rcx 指向著一段文字
- 文字以 \x00 結尾, 因此這串字串是 “What’s your name?\n”

資料視窗 1	資料視窗 2	資料視窗 3	資料視窗 4	資料視窗 5	監視 1
位址	十六進位				ASCII
00007FF676153290	57 68 61 74	27 73 20 79	6F 75 72 20	6E 61 6D 65	What's your name
00007FF6761532A0	3F 0A 00 00	25 33 32 73	00 00 00 00	00 00 00 00	?...%32s.....
00007FF6761532B0	48 65 6C 6C	6F 2C 20 25	73 21 0A 00	00 00 00 00	Hello, %s!.....
00007FF6761532C0	25 64 20 78	20 25 64 20	3D 20 3F 0A	00 00 00 00	%d x %d = ?

x64dbg

- 觀察完參數後, 按下 f8 步過此函數
 - f7 是步入, 會走進函數裡面
 - f8 是步過, 會走完這個函數
- 可以發現視窗輸出了剛剛第一個參數指向的字串
- 所以我們姑且先判斷此函數是 puts 或 printf 之類的函數



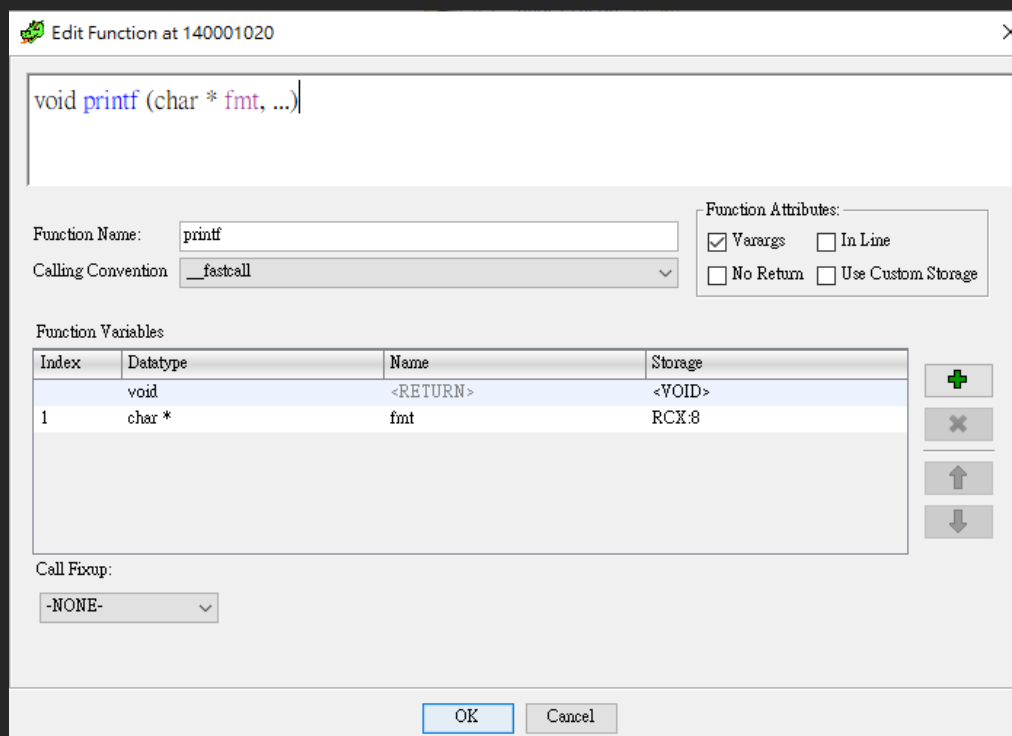
Ghidra

- 在 0x140001131 呼叫了 FUN_140001020
- 回來 Ghidra, 點進去 FUN_140001020 裡面看
- 發現他實際上還會呼叫 __stdio_common_vfprintf

```
2 void FUN_140001020(undefined8 param_1,undefined8 param_2,undefined8 param_3,undefined8 param_4)
3
4 {
5
6
7
8
9
10
11 local_res10 = param_2;
12 local_res18 = param_3;
13 local_res20 = param_4;
14 uVar1 = __acrt_iob_func(1);
15 puVar2 = (undefined8 *)FUN_140001000();
16 __stdio_common_vfprintf(*puVar2,uVar1,param_1,0,&local_res10);
17 return;
18 }
```

Ghidra

- 綜合以上觀察, 判斷他是 printf, 把他改名
- 改一下他的參數型態, 對著函數名稱右鍵選擇 Edit Function Signature



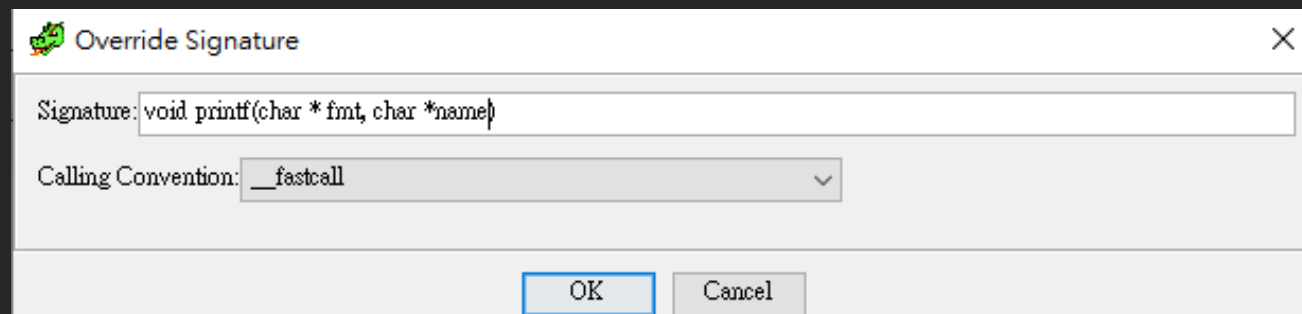
Ghidra

- 但 Ghidra 目前對於參數數量不一定的函數, 支援度比較差
- 可以看到第二個 printf 應該是要兩個參數的
- 目前解法是對第二個 printf 右鍵選擇 Override Signature

```
printf("What\'s your name?\n");  
FUN_140001080(&DAT_1400032a4, local_258, envp, param_4);  
printf("Hello, %s!\n");
```

Ghidra

- 只能先手動校正



```
printf("What\'s your name?\n");  
FUN_140001080(&DAT_1400032a4, local_258, envp, param_4);  
printf("Hello, %s!\n", local_258);
```

Ghidra

- 那個 FUN_140001080 是什麼?
- 看一下他第一個參數, 對他點兩下
- “%32s”

DAT_1400032a4			
1400032a4	??	25h	%
1400032a5	??	33h	3
1400032a6	??	32h	2
1400032a7	??	73h	s
1400032a8	??	00h	

Ghidra

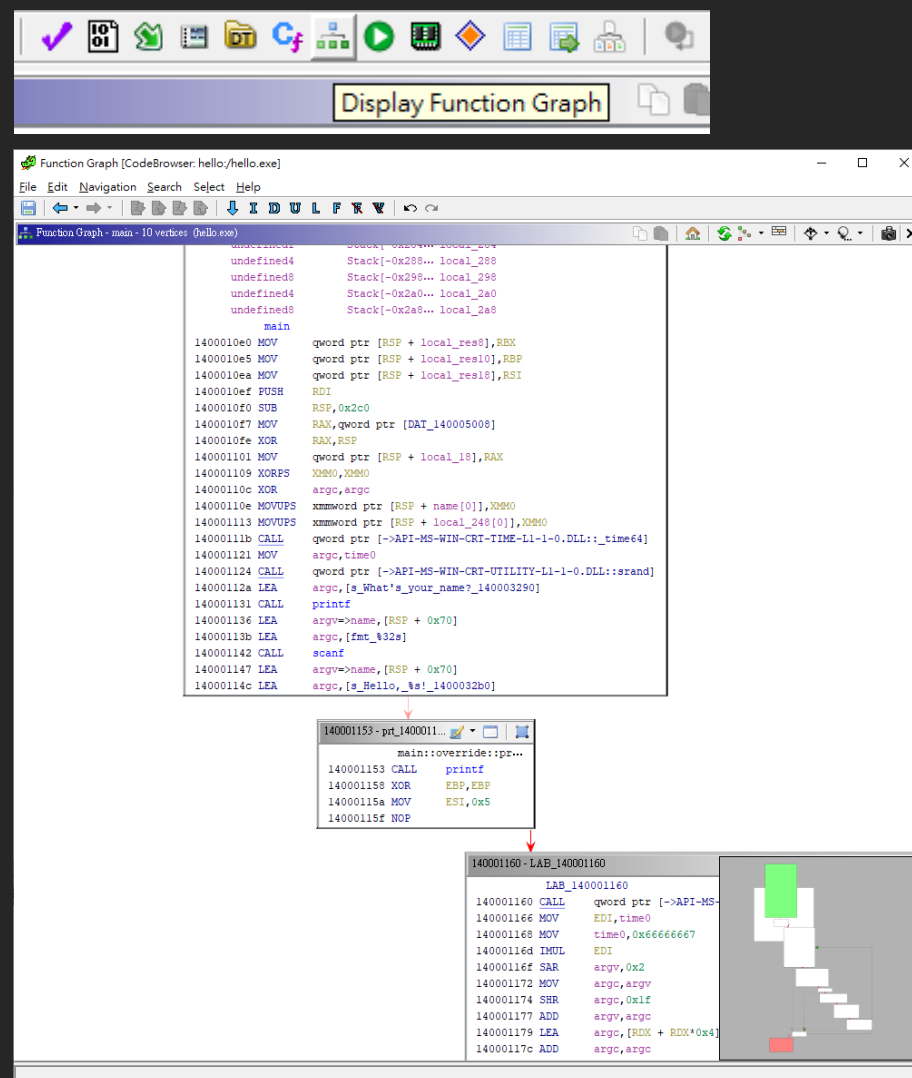
- 前面問你叫什麼名字
- 接下來呼叫 FUN_140001080, 帶著 “%32s” 跟 local_258 呼叫
- 接下來跟你說 Hello
- 實驗一下就知道 FUN_140001080 是 scanf
- 而 local_258 也能順著程式的含意, 將他命名為 names

```
printf("What\'s your name?\n");  
FUN_140001080(&DAT_1400032a4, local_258, envp, param_4);  
printf("Hello, %s!\n", local_258);
```

```
printf("What\'s your name?\n");  
scanf(&fmt_%32s, name, envp, param_4);  
printf("Hello, %s!\n", name);
```

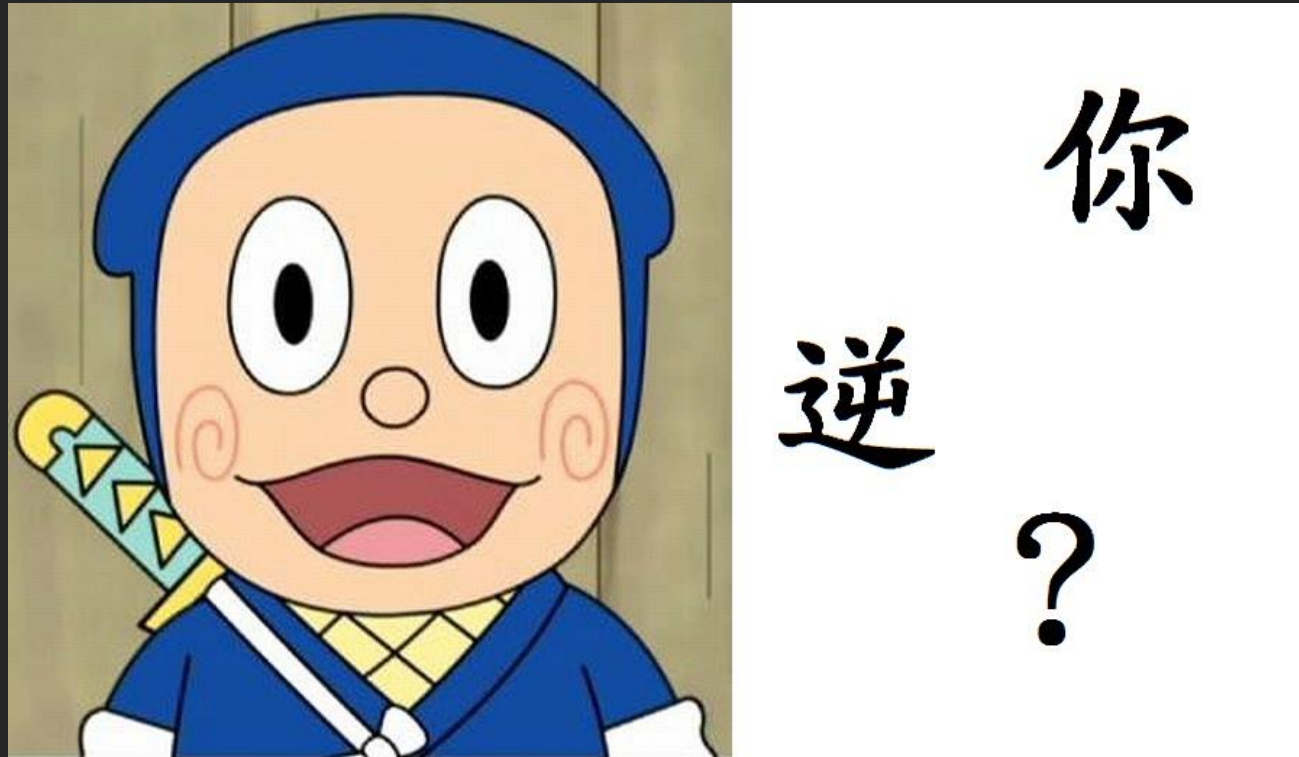
Ghidra

- 這程式反編譯的算不錯
- 有些程式反編譯根本不能看, 只能老老實實看組語
- 點上面的 Display Function Graph
- 這個介面看組語友善多了



Ghidra

- 剩下交給你逆了, hello.exe 其實會寫一個檔案, 試著找出裡面究竟寫了什麼吧



ELF 逆向工程

ELF 逆向工程

- 上面的章節都是以 Windows exe 來舉例, 目的是讓大家先從熟悉的环境開始
- 接著來帶一下很基礎的 ELF 逆向

ELF 逆向工程

- 前面有提到不同 OS 是如何載入程式的方式大同小異
- 頭部結構
 - Windows: PE (Portable Executable) Header
 - Linux: ELF (Executable and Linkable Format)
- 載入後，就從程式進入點開始執行
- 基於篇幅的原因, 這裡就只講 ELF 的程式進入點之類的怎麼看

ELF 逆向工程

```
⚡ XAYB readelf -a XAYB
ELF Header:
  Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class: ELF64
  Data: 2's complement, little endian
  Version: 1 (current)
  OS/ABI: UNIX - System V
  ABI Version: 0
  Type: DYN (Shared object file)
  Machine: Advanced Micro Devices X86-64
  Version: 0x1
  Entry point address: 0x10b0
  Start of program headers: 64 (bytes into file)
  Start of section headers: 15144 (bytes into file)
  Flags: 0x0
  Size of this header: 64 (bytes)
  Size of program headers: 56 (bytes)
  Number of program headers: 11
  Size of section headers: 64 (bytes)
  Number of section headers: 30
  Section header string table index: 29
```

ELF 檔案開頭固定為 \x7fELF

DYN 表示其啟用 PIE

程式進入點 offset

PIE

- PIE (Position Independent Executable)
- 與 ASLR 類似
- 隨機化程式起始的位址

關閉 PIE / ASLR ?

- Q: Windows 的 PE format 可以將 ASLR 關閉, 那 Linux 的 ELF format 能不能關掉 ASLR 跟剛剛說的 PIE?
- ASLR 不是 ELF format 決定要不要開, 是 kernel 的設定, 要關 ASLR 要去設定 kernel
- PIE 理論上是可以關閉, 但要做的修改不像改 PE 的 ASLR 去掉某個 bit 這麼簡單, 實務上至少我沒有找到工具可以關 PIE, 但其實不關也沒關係

ELF 逆向工程

- 在 Linux 中, 不知道某個檔案是什麼, 可以先用 file 指令看一下

```
✂ XAYB file XAYB
XAYB: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=825727001154a3ce5d456150e5b313f3db4b55da, for GN
U/Linux 3.2.0, not stripped
```


ELF 逆向工程

- ELF x64 檔案

```
⚡ XAYB file XAYB
XAYB: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=825727001154a3ce5d456150e5b313f3db4b55da, for GN
U/Linux 3.2.0, not stripped
```

ELF 逆向工程

- 啟用 PIE

```
⚡ XAYB file XAYB
XAYB: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=825727001154a3ce5d456150e5b313f3db4b55da, for GN
U/Linux 3.2.0, not stripped
```

ELF 逆向工程

- 有連結到其他 library

```
⚡ XAYB file XAYB
XAYB: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=825727001154a3ce5d456150e5b313f3db4b55da, for GN
U/Linux 3.2.0, not stripped
```

ELF 逆向工程

- Debug symbol 有留下來

```
⚡ XAYB file XAYB
XAYB: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=825727001154a3ce5d456150e5b313f3db4b55da, for GN
U/Linux 3.2.0, not stripped
```

ELF 逆向工程

- 組語的部分都跟前面講的一樣
- 注意 calling convention 與 windows 不同

分析方式

- 靜態的部分一樣能使用 Ghidra 進行
- 動態的部分這邊介紹使用 gdb 進行
- 實務上也是交叉使用, 就只是換了動態 debug 的工具

gdb

- 下斷點
- breakpoint
- b

gdb

- 執行
- run
- r

gdb

- 繼續執行
- continue
- C

gdb

- 步過
- ni

gdb

- 步入
- si

gdb

- 秀出記憶體內容
- x/<幾個><格式><尺寸> <記憶體位址>
- e.g.
- x/16xg 0x1000
- 秀出從 0x1000 開始的 16 個以十六進制 (x) 格式表示的 8 Bytes(g)

逆向工程技巧

逆向工程技巧

- 逆向工程的做法你可以
 1. 整支程式從頭逆向到尾
 2. 只挑重點逆向, 挑你感興趣的部分逆向
- 第二個做法是什麼意思???

逆向工程技巧

- 想像一下, 你要開發一支後門程式, 勢必要連線到你的後門
- 連線要用到跟網路相關的 Windows API / Library Function
- 這時候你就可以從程式在哪裡呼叫到這些函數開始逆向

逆向工程技巧

- 想像一下一支正常給使用者用的程式, 多多少少要告訴使用者一些資訊吧, 就會有一些固定的字串
- 比如說“登入失敗”、“密碼:”
- 就可以從程式在哪邊用到這些字串開始逆向

逆向工程技巧

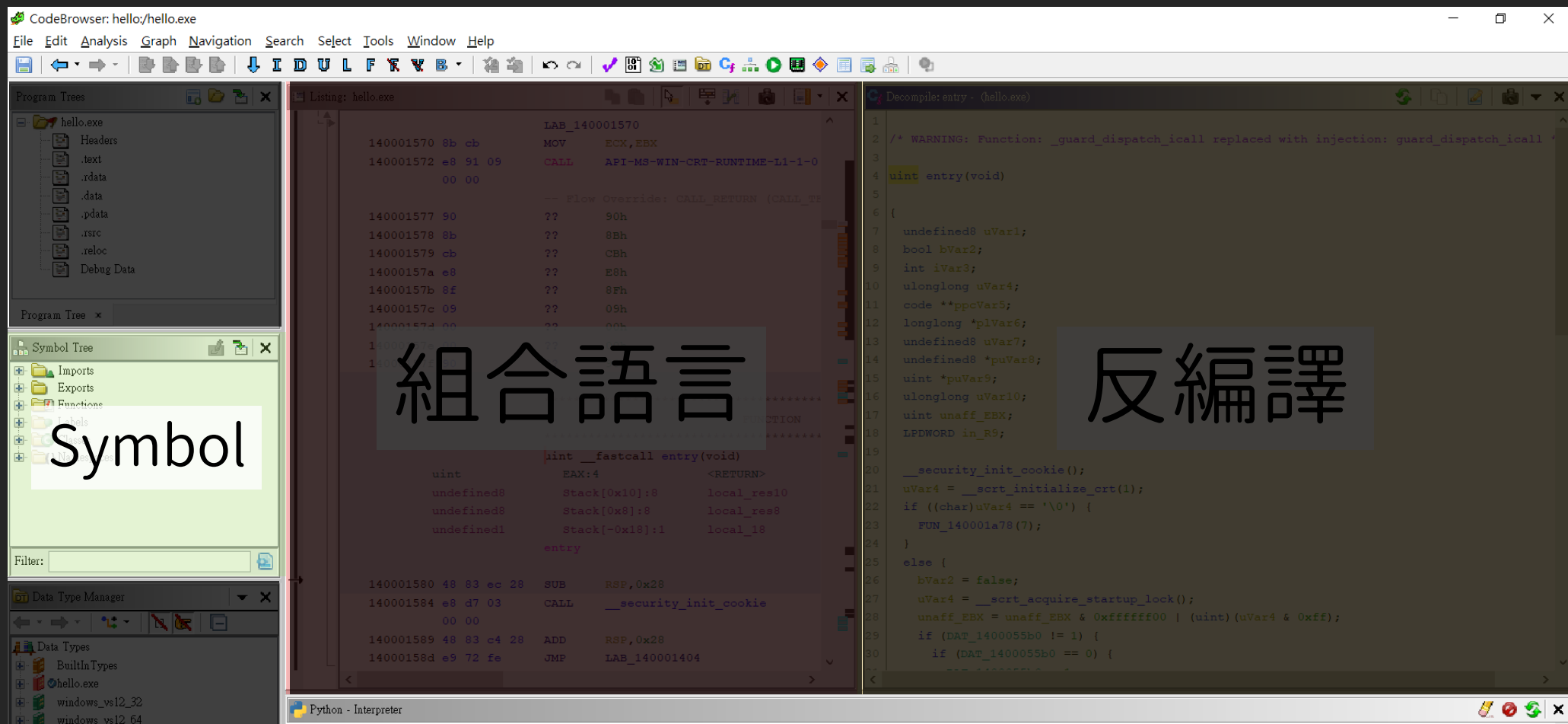
- 總之方法很多
- 想辦法找到一支程式你感興趣的程式碼區間
- 但有的時候很難

逆向工程技巧

- 另外有的時候, 你不用老老實實的逆完整個函數
- 直接從輸入進去的東西是什麼, 輸出的東西是什麼, 來直接猜函數在幹嘛
- 就像前幾個章節中, 我們猜出 scanf 的過程
- 簡單來說就是 通靈

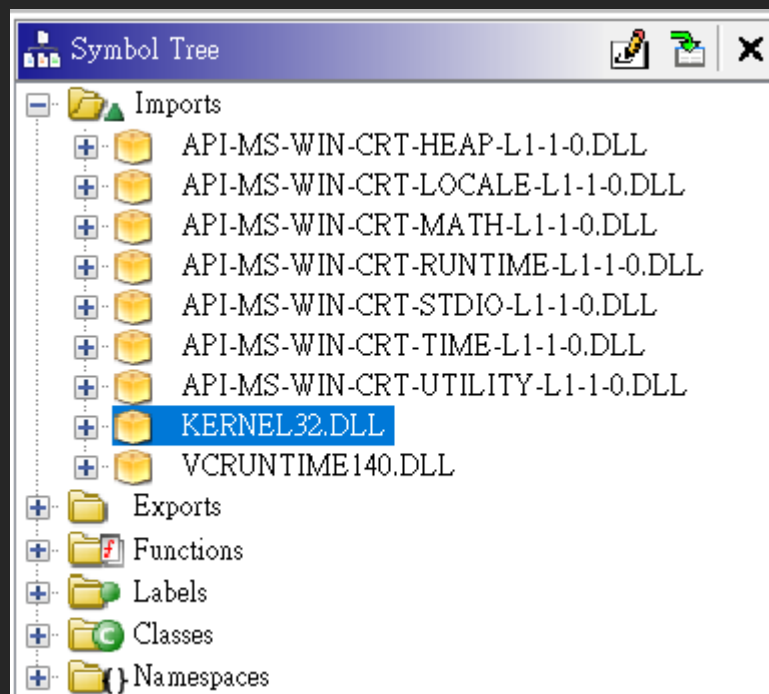
逆向工程技巧

- 找特別函數



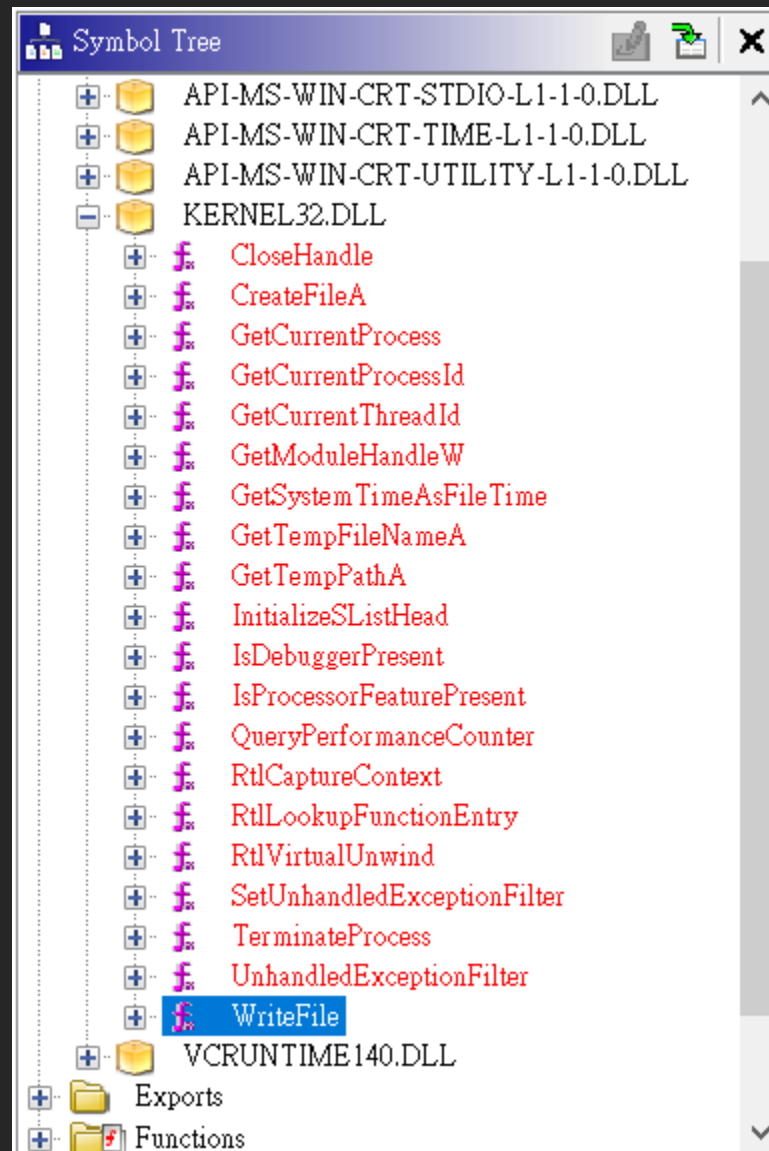
逆向工程技巧

- 找特別函數
- 看這支函數連結哪些函數庫
- Kernel32.dll 是 windows 中提供很多函數的函數庫



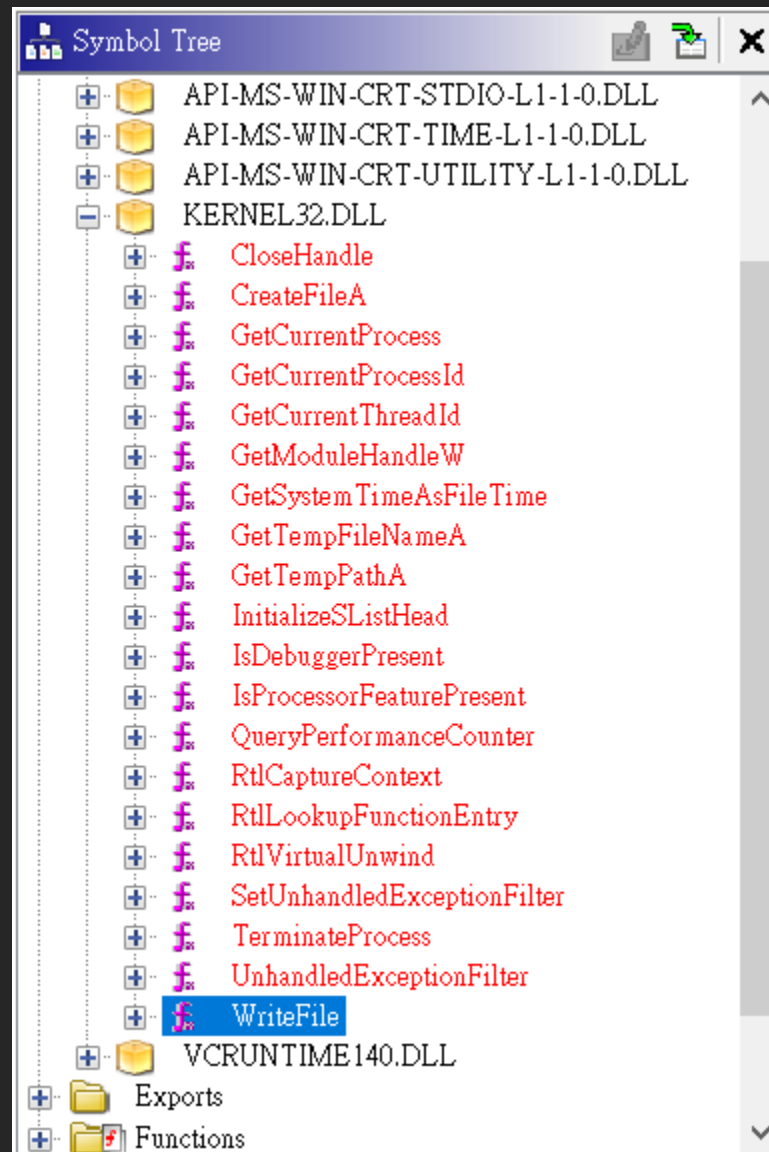
逆向工程技巧

- 這支程式用到 WriteFile
- 點兩下



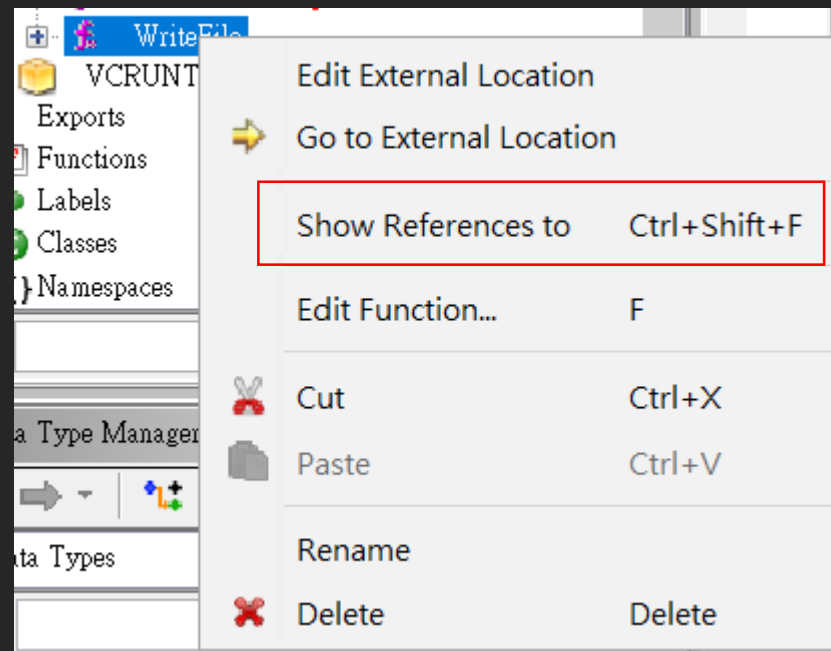
逆向工程技巧

- 這支程式用到 WriteFile
- 要怎麼找到用到 WriteFile 的位址?



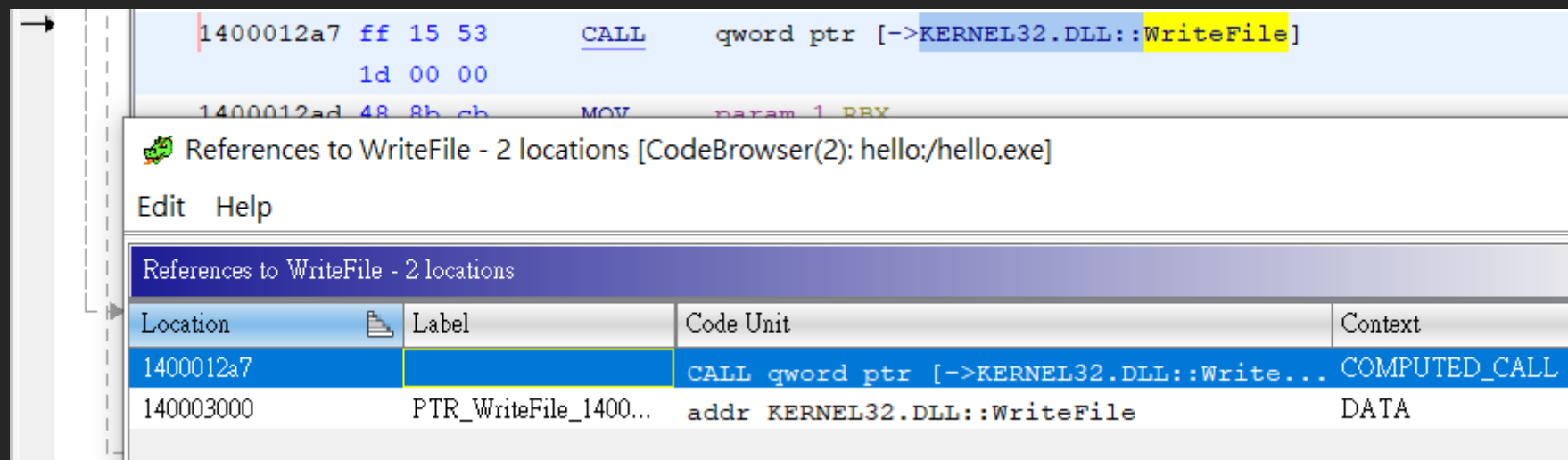
逆向工程技巧

- 對他按右鍵
- Show References to



逆向工程技巧

- 就能找到程式中呼叫到 WriteFile 的位址了



The screenshot shows a debugger window with assembly code. The instruction at address 1400012a7 is a CALL to WriteFile in KERNEL32.DLL. Below the assembly view, a window titled 'References to WriteFile - 2 locations [CodeBrowser(2): hello:/hello.exe]' is open, displaying a table of references.

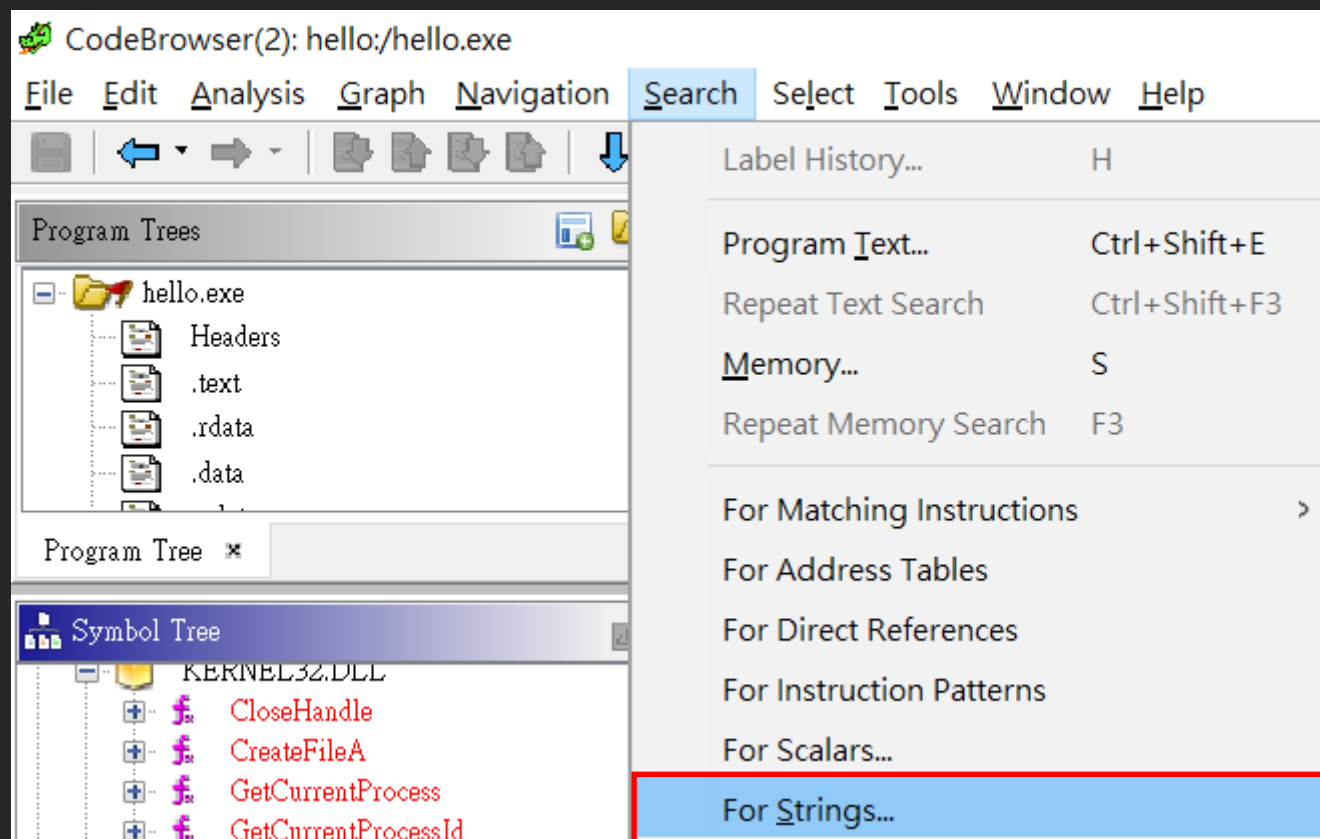
Location	Label	Code Unit	Context
1400012a7		CALL qword ptr [->KERNEL32.DLL::Write...	COMPUTED_CALL
140003000	PTR_WriteFile_1400...	addr KERNEL32.DLL::WriteFile	DATA

逆向工程技巧

- 實務上會刻意留意的函數很多
- 可以參考 reference 連結

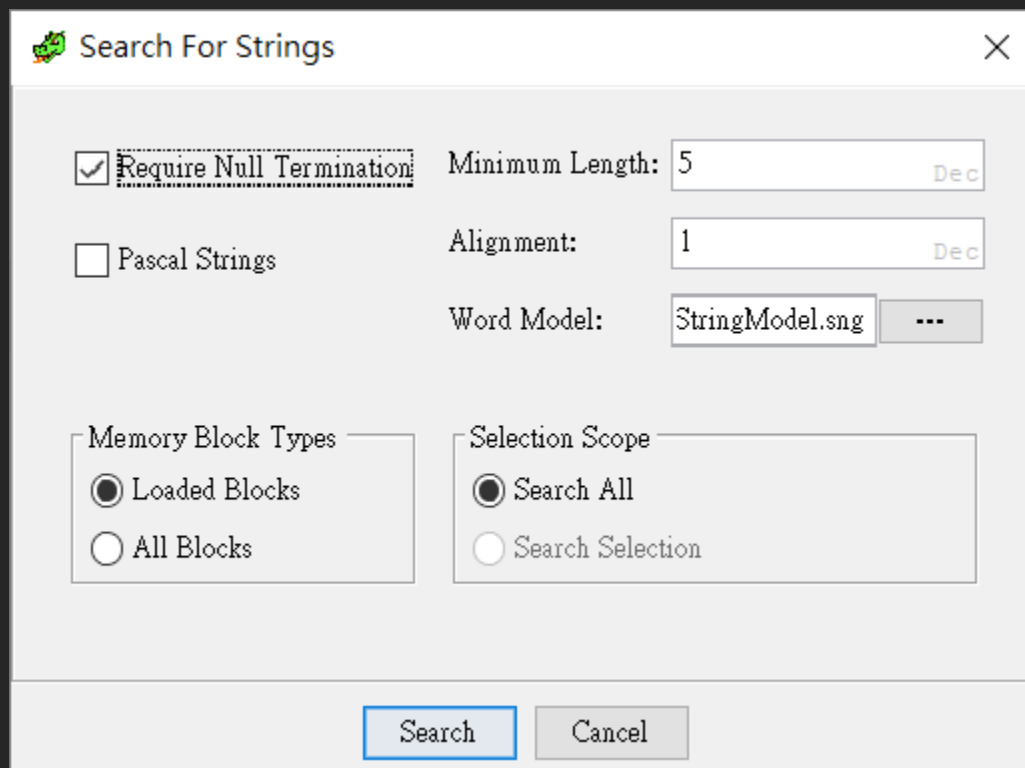
逆向工程技巧

- 找字符串



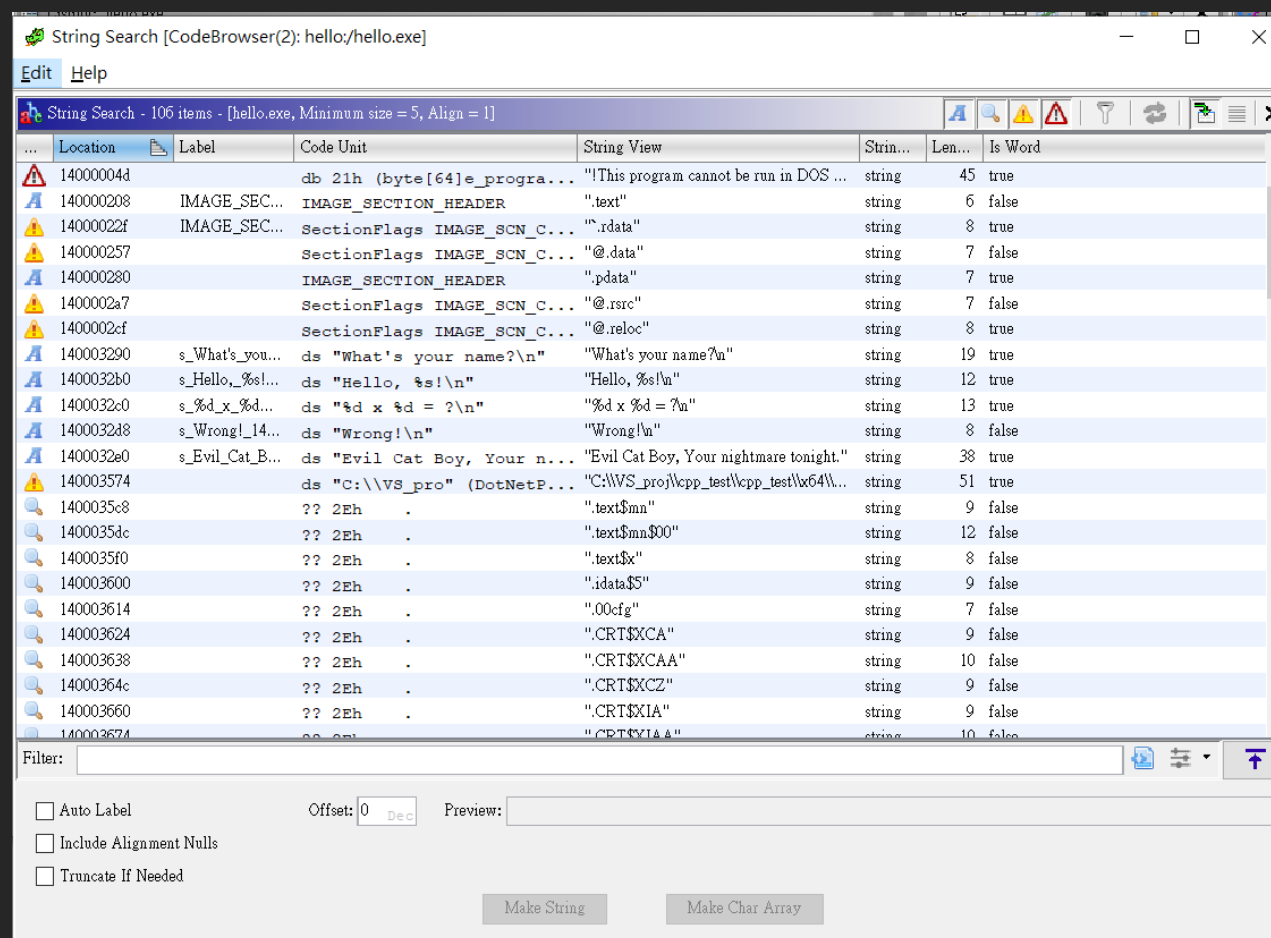
逆向工程技巧

- 設定找怎麼樣的字串



逆向工程技巧

- 結果出爐



Location	Label	Code Unit	String View	String	Len...	Is Word
14000004d		db 21h (byte[64]e_progra...	"!This program cannot be run in DOS ...	string	45	true
140000208	IMAGE_SEC...	IMAGE_SECTION_HEADER	".text"	string	6	false
14000022f	IMAGE_SEC...	SectionFlags IMAGE_SCN_C...	".rdata"	string	8	true
140000257		SectionFlags IMAGE_SCN_C...	".data"	string	7	false
140000280		IMAGE_SECTION_HEADER	".pdata"	string	7	true
1400002a7		SectionFlags IMAGE_SCN_C...	".rsrc"	string	7	false
1400002cf		SectionFlags IMAGE_SCN_C...	".reloc"	string	8	true
140003290	s_What's_you...	ds "What's your name?\n"	"What's your name?\n"	string	19	true
1400032b0	s_Hello,_%s!	ds "Hello, %s!\n"	"Hello, %s!\n"	string	12	true
1400032c0	s_%d x %d = ?\n"	ds "%d x %d = ?\n"	"%d x %d = ?\n"	string	13	true
1400032d8	s_Wrong!_14...	ds "Wrong!\n"	"Wrong!\n"	string	8	false
1400032e0	s_Evil_Cat_B...	ds "Evil Cat Boy, Your n...	"Evil Cat Boy, Your nightmare tonight."	string	38	true
140003574		ds "C:\\VS_pro" (DotNetP...	"C:\\VS_proj\\cpp_test\\cpp_test\\x64\\...	string	51	true
1400035c8		?? 2Eh .	".text\$mn"	string	9	false
1400035dc		?? 2Eh .	".text\$mn\$00"	string	12	false
1400035f0		?? 2Eh .	".text\$fx"	string	8	false
140003600		?? 2Eh .	".idata\$5"	string	9	false
140003614		?? 2Eh .	".00cfg"	string	7	false
140003624		?? 2Eh .	".CRT\$XCA"	string	9	false
140003638		?? 2Eh .	".CRT\$XCAA"	string	10	false
14000364c		?? 2Eh .	".CRT\$XCZ"	string	9	false
140003660		?? 2Eh .	".CRT\$XIA"	string	9	false
140003674		?? 2Eh .	".CRT\$XIAA"	string	10	false

Filter: []

☐ Auto Label Offset: 0 Dec Preview: []

☐ Include Alignment Nulls

☐ Truncate If Needed

Make String Make Char Array

逆向工程技巧

- 找字符串在 Linux 中還可以用指令 strings 簡單完成

```
< XAYB strings XAYB
/lib64/ld-linux-x86-64.so.2
]EaP
srand
__isoc99_scanf
puts
time
putchar
printf
__cxa_finalize
__libc_start_main
libc.so.6
GLIBC_2.7
GLIBC_2.2.5
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
u/UH
gfffH
gfffH
gfffH
gfffH
[]A\A]A^A_
The Lucky 5 digits number is ... >
Enter something to start this game ...
%100s
Generating answer ...
You have %d chances to guess the answer. I'm so kind :)
```

Q & A

感謝收聽



疫情期間 少出門 勤洗手