Reverse Engineering - 3

2021/12/03

Presented by LJP

whoami

- LJP / LJP-TW
- SQLab @ NYCU 碩一
- CTF @ 10sec / TSJ
- Pwner

Outline

- Sysinternals
- TLS Callback
- TEB
- Exception
- Packer
- Anti-Reverse

Sysinternals

Sysinternals

- 除了直接逆向逆起來，還可以先觀察程式跑起來時會做什麼

- e.g.
  - 創 child process
  - 讀寫檔案
  - 網路連線

Sysinternals

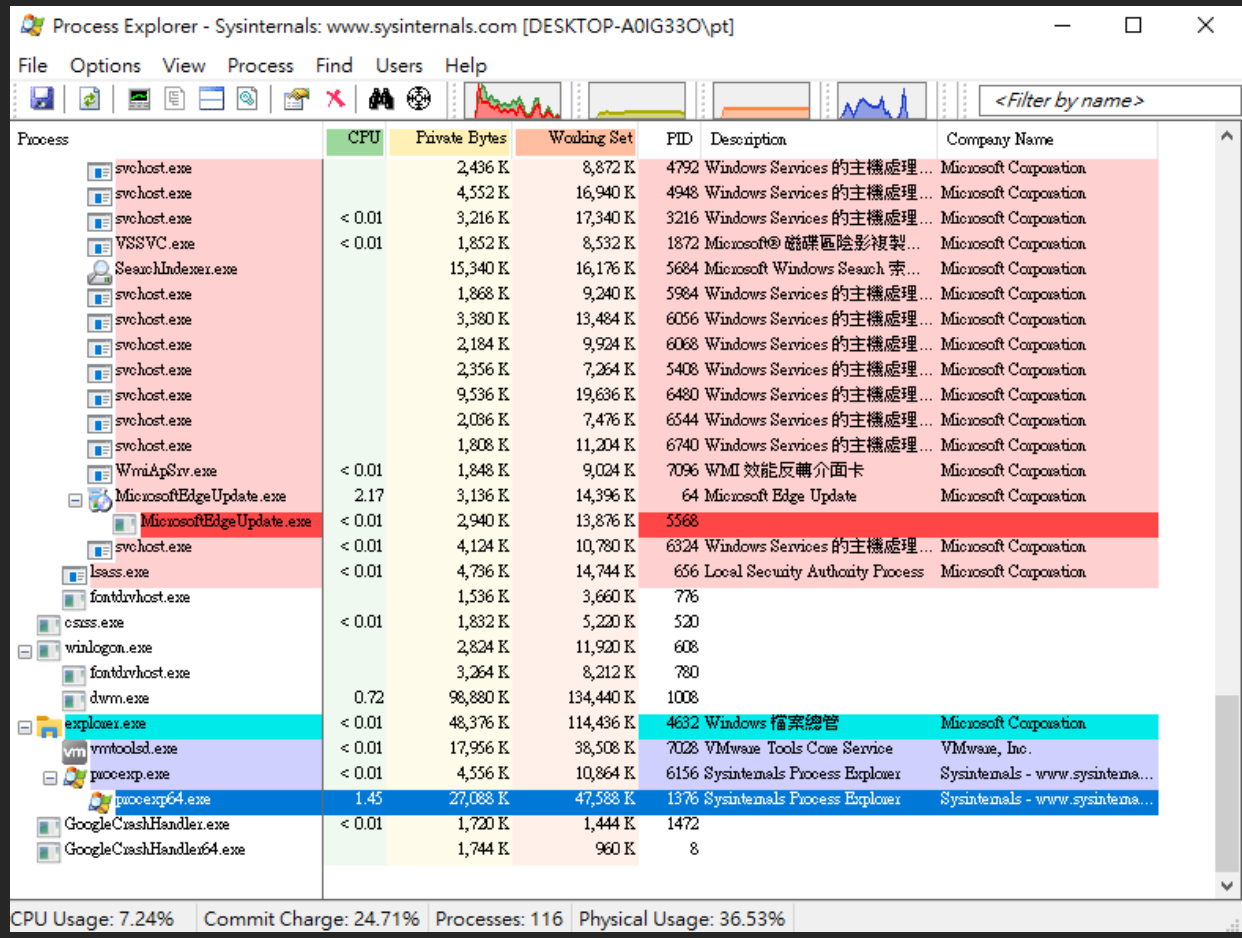
- Sysinternals 集成了許多工具
- 介紹以下兩個工具, 其他工具可以自行摸索
- Procexp
- Procmon

Ref: https://docs.microsoft.com/en-us/sysinternals/downloads/sysinternals-suite

Sysinternals
Procexp

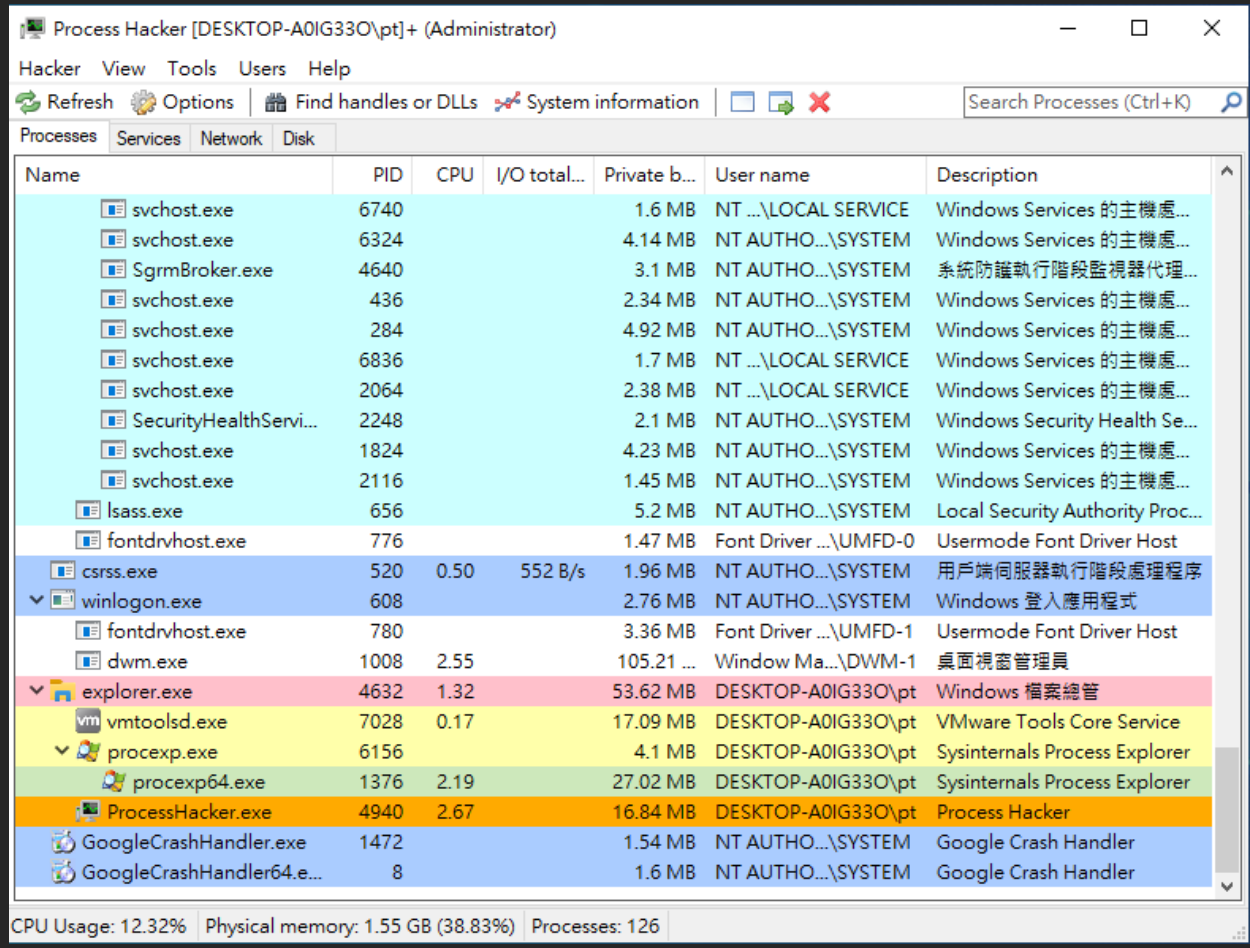
- 好看版的工作管理員

- 這邊另外推薦與之類似的工具
  - Process Hacker



Process Hacker

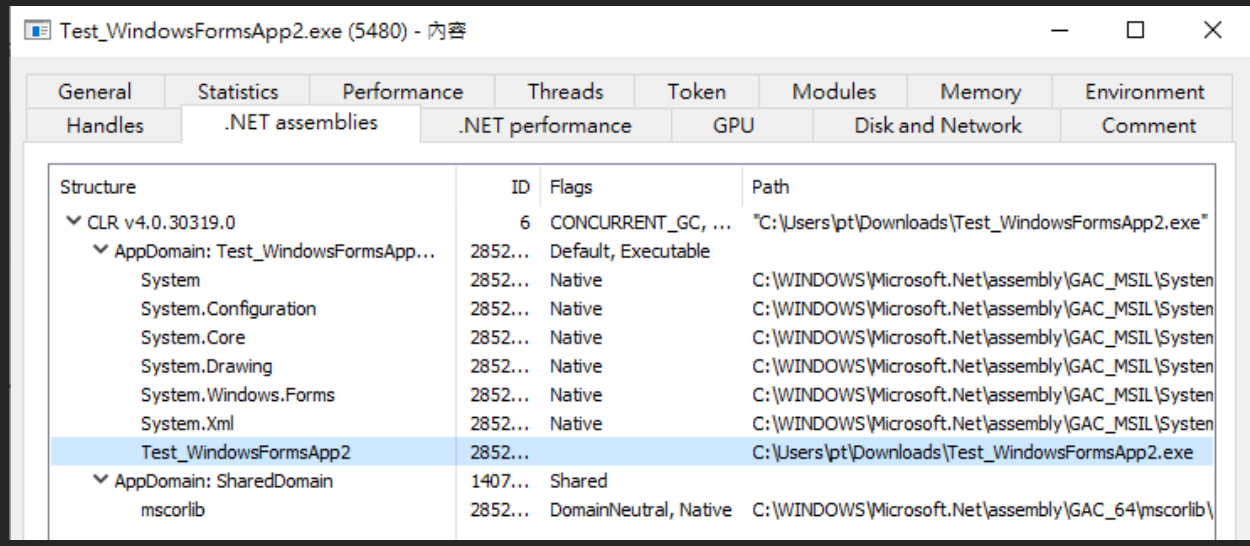
- 相較於 Procexp, 提供更多訊息



Ref: https://github.com/processhacker/processhacker

Process Hacker

- 相較於 Procexp, 提供更多訊息
- e.g. .NET assemblies

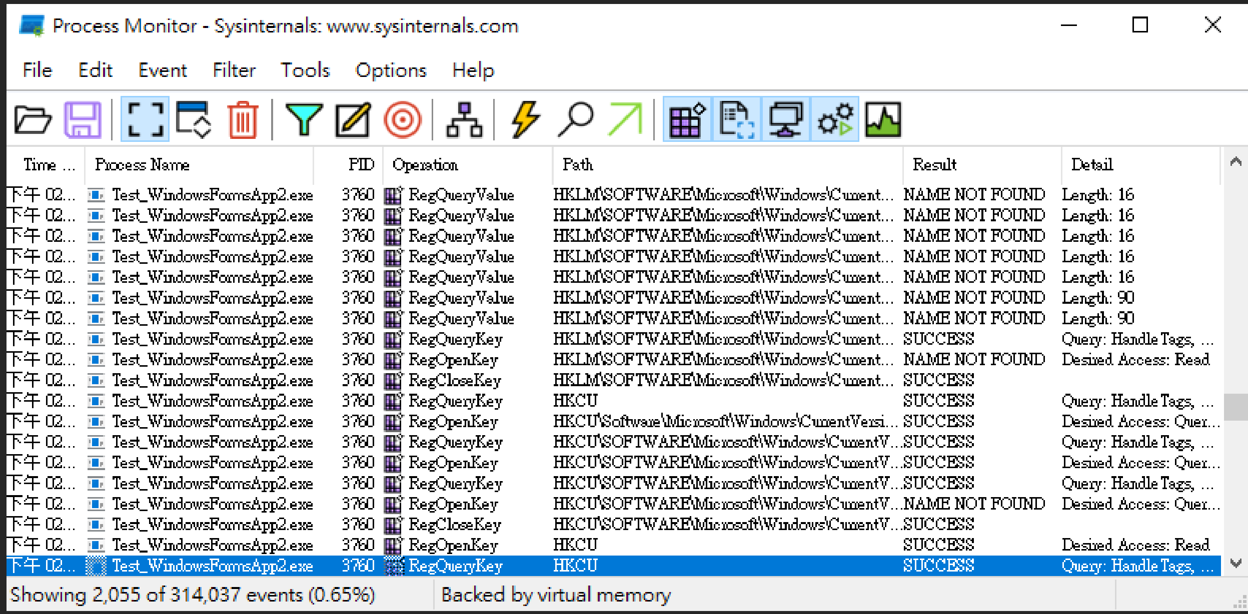


Ref: https://github.com/processhacker/processhacker

Sysinternals

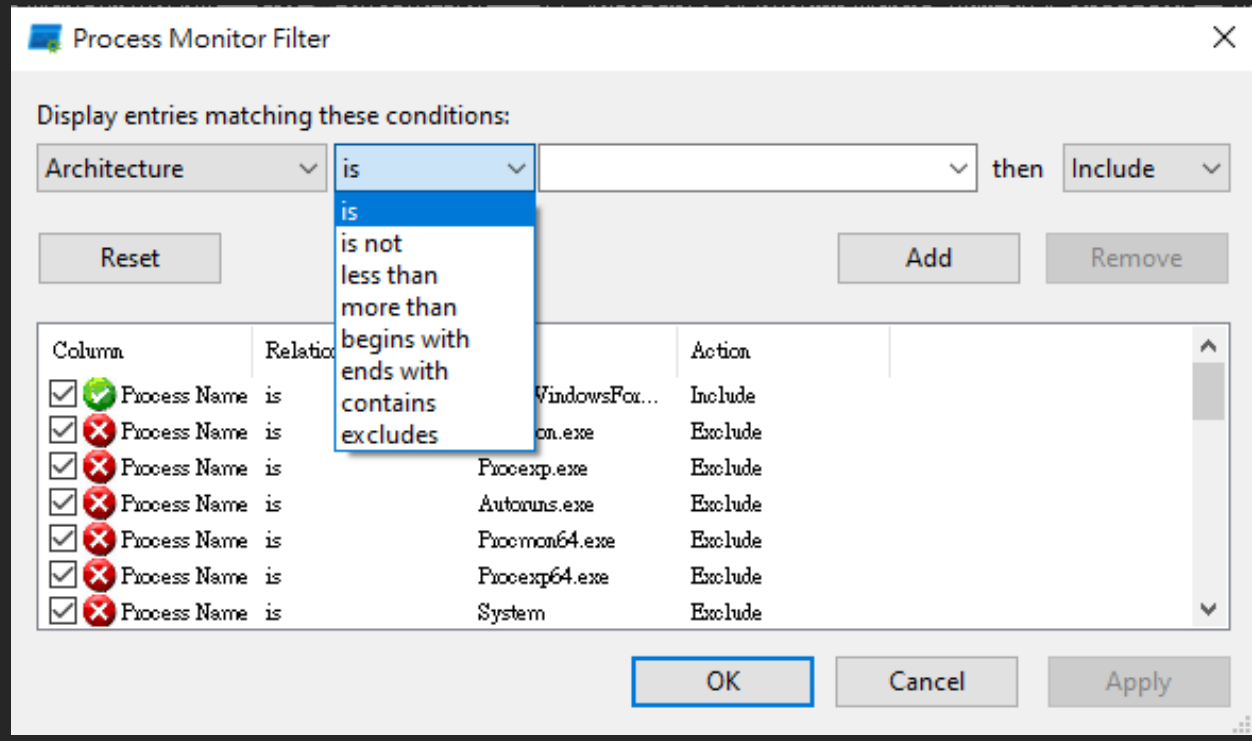
Procmon

- 監控程序行為
  - Registry
  - File system
  - Network
  - Process/Thread



Sysinternals
Procmon

- 訊息量過大, 請善用 Filter
- 可以直接在顯示面板對資料右鍵快速篩進/篩掉想要的資料



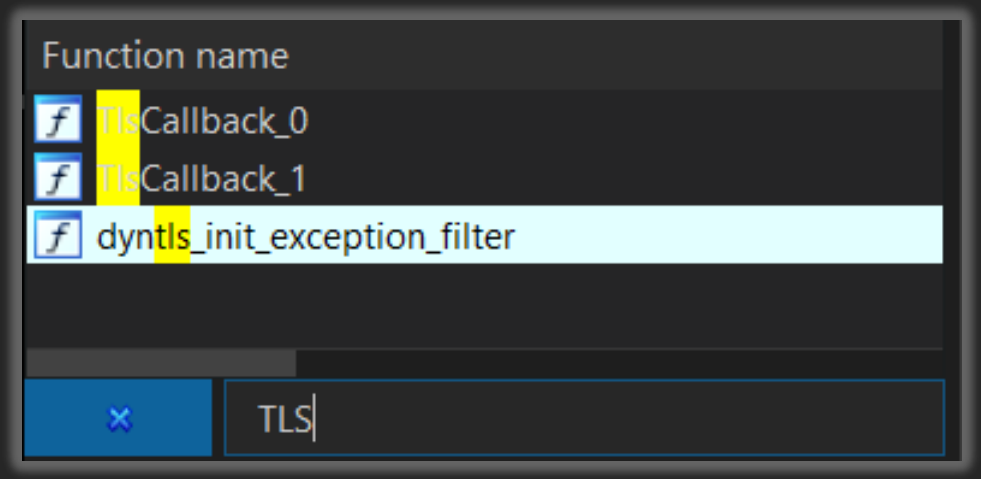
TLS callback

TLS Callback

- Process/Thread 的開始/結束時都會自動呼叫到 TLS Callback
- 跟之前討論過的 init/fini 不同
- 呼叫時機
  - TLS Cb -> Entry Point -> init -> main -> fini -> TLS Cb

TLS Callback

- IDA 能自動辨識出 TLS Callback
- How?



TLS Callback

- TLS Directory

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

Offset	Name	Value	Value
1A8	Exception Directory	20000	105C
1B0	Security Directory	0	0
1B8	Base Relocation Table	24000	66C
1C0	Debug Directory	1B700	70
1C8	Architecture Specific Data	0	0
1D0	RVA of GlobalPtr	0	0
1D8	TLS Directory	1B900	28

TLS

Offset	Name	Value
1A900	StartAddressOfRawData	14001BD40
1A908	EndAddressOfRawData	14001BD41
1A910	AddressOfIndex	14001EA90
1A918	AddressOfCallBacks	1400132C8
1A920	SizeOfZeroFill	0
1A924	Characteristics	100000

TLS Callbacks [2 entries]

Offset	Callback
122C8	140001070
122D0	140001090

TLS Callback

- TLS Directory
- AddressOfCallbacks

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
122C8	70	10	00	40	01	00	00	00	90	10	00	40	01	00	00	00
122D8	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

TLS

Offset	Name	Value
1A900	StartAddressOfRawData	14001BD40
1A908	EndAddressOfRawData	14001BD41
1A910	AddressOfIndex	14001EA90
1A918	AddressOfCallBacks	1400132C8
1A920	SizeOfZeroFill	0
1A924	Characteristics	100000

TLS Callbacks [2 entries]

Offset	Callback
122C8	140001070
122D0	140001090

Lab 1

TEB

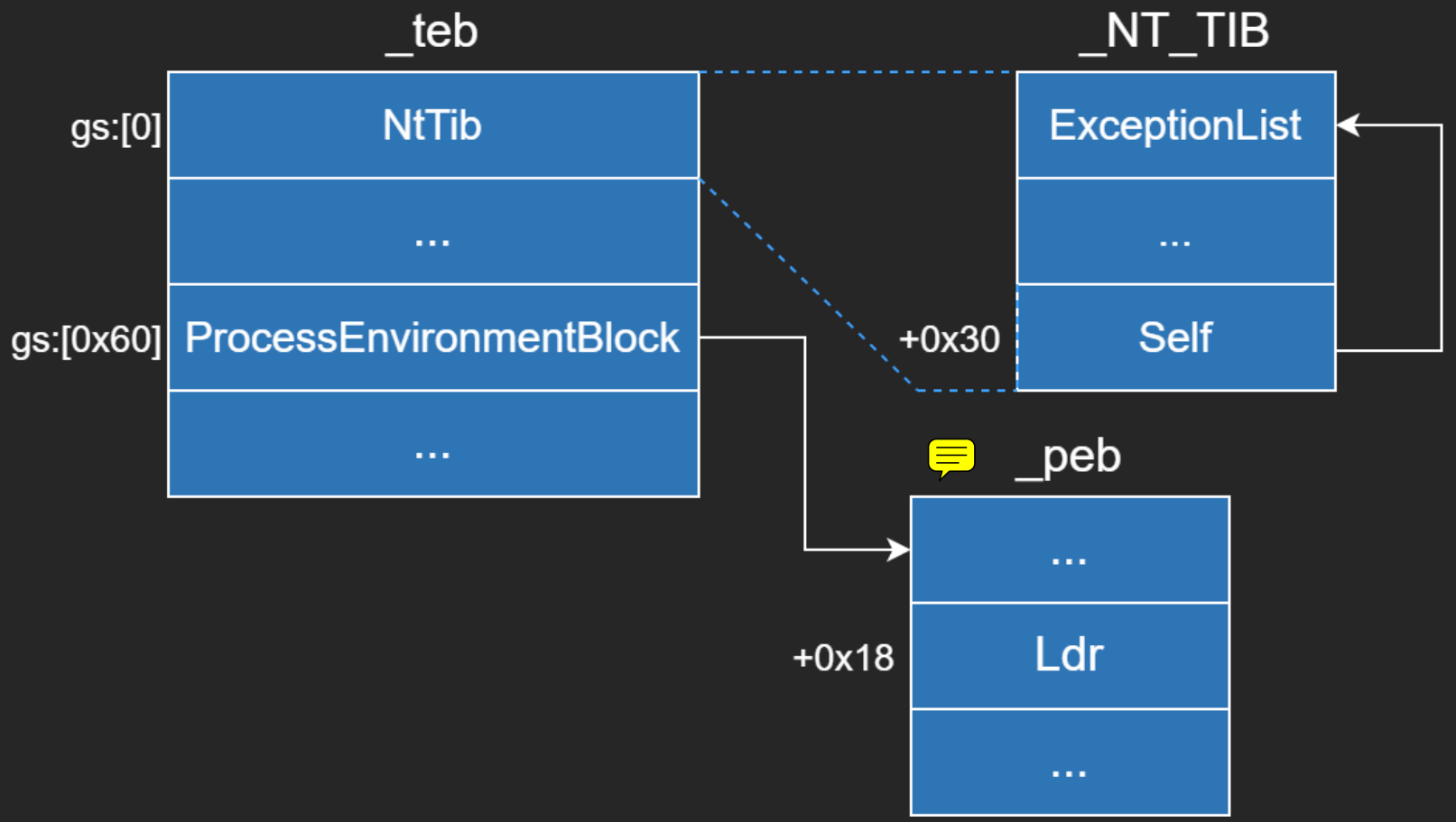
Thread Environment Block

TEB

- 前面我們講 PEB 時其實跳過了 TEB
- FS/GS 就是存放 TEB

- 取得 PEB
- 32bit
  - FS:[0x30]
- 64bit
  - GS:[0x60]

TEB (64-bit)


_teb
gs:[0]
NtTib
...
gs:[0x60]
ProcessEnvironmentBlock
...
_NT_TIB
ExceptionList
...
+0x30
Self
_peb
...
+0x18
Ldr
...

TEB

	ExceptionList	Self	PEB
32bit FS	[0]	[0x18]	[0x30]
64bit GS	[0]	[0x30]	[0x60]

TEB

所以那個 fs / gs 到底是啥

TEB

- 接下來的部分，你不知道也是可以繼續分析程式
- Segment Register
  - CS、DS、SS、ES、FS、GS
- FS:[0x30]??　GS:[0x60]??
- 這些咚咚有特別的記憶體算法

TEB

- FS:[0x30]
- 實際算法為 base address + 0x30
- Base address 怎麼來的?

TEB

- 如果是在 Compatibility mode …

TEB

- 如果是在 Compatibility mode …
- 等等 mode 是啥?!?!

TEB

- 各種模式

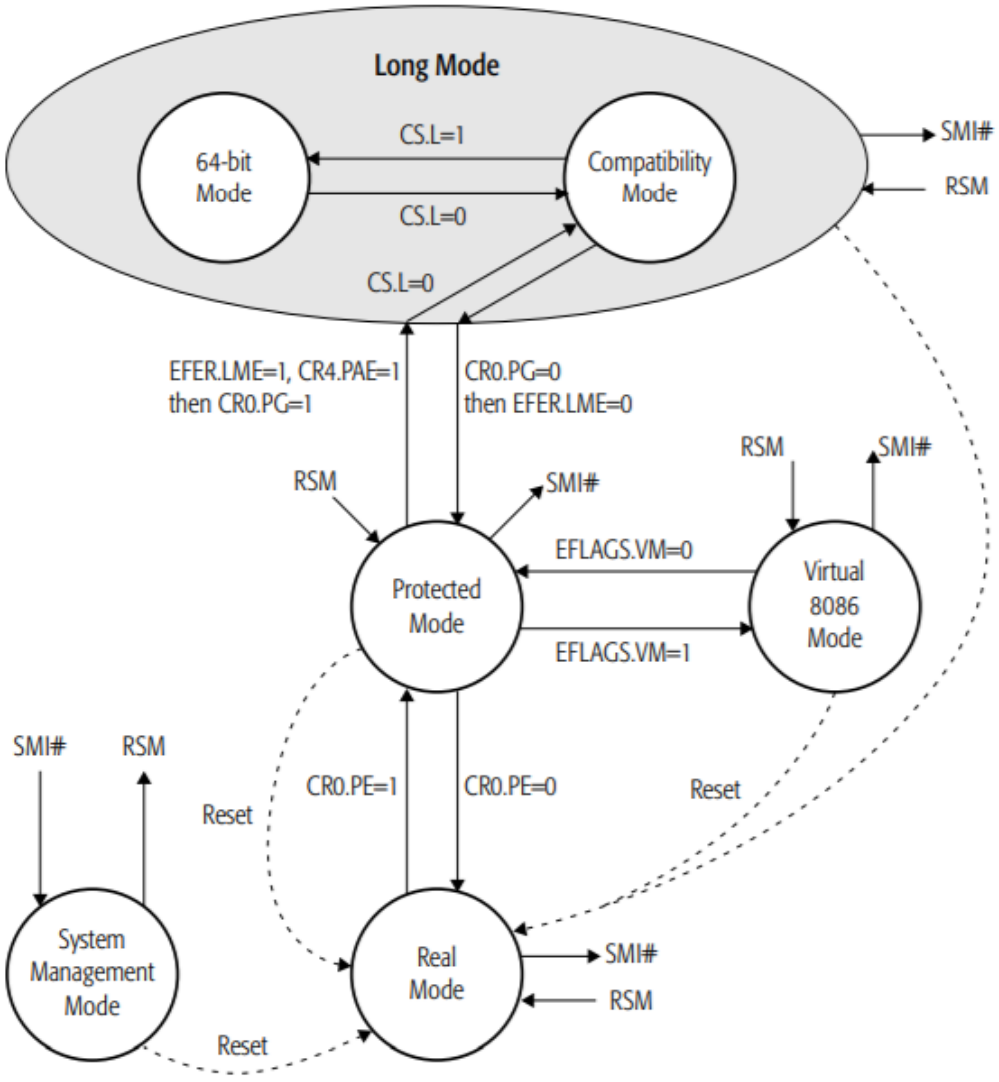


Figure 1-6. Operating Modes of the AMD64 Architecture

Ref: AMD64 Architecture Programmer's Manual Volume 2: System Programming

TEB

- 各種模式
- 實驗一下，如果 x64 windows 運行 x32 程式會在什麼模式

TEB

- CS.L = 0

```
32.1: kd> dg @cs
                                                    P Si Gr Pr Lo
Sel        Base              Limit          Type    l ze an es ng Flags
---- ---------------- ---------------- ---------- - -- -- -- -- --------
0023 00000000`00000000 00000000`ffffffff Code RE Ac 3 Bg Pg P  Nl 00000cfb
32.1: kd> .formats @cs
Evaluate expression:
  Hex:     00000000`00000023
  Decimal: 35
  Octal:   0000000000000000000043
  Binary:  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00100011
  Chars:   .......#
  Time:    Thu Jan  1 08:00:35 1970
  Float:   low 4.90454e-044 high 0
  Double:  1.72923e-322
```

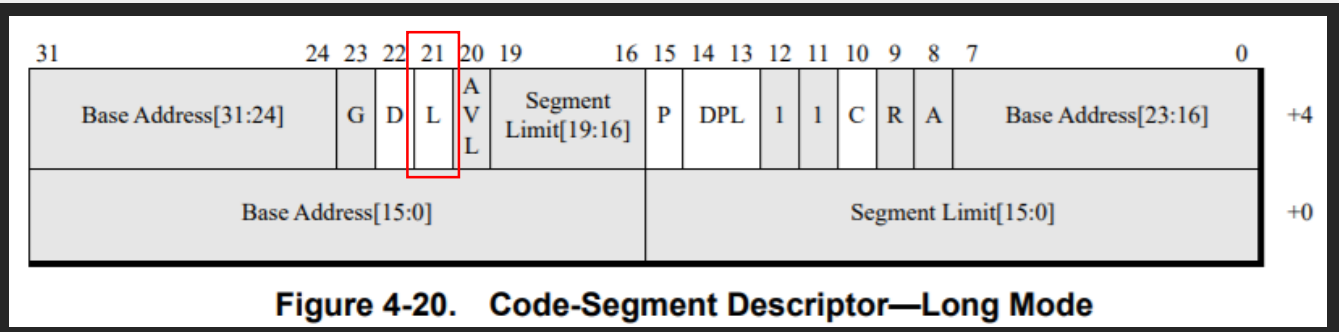


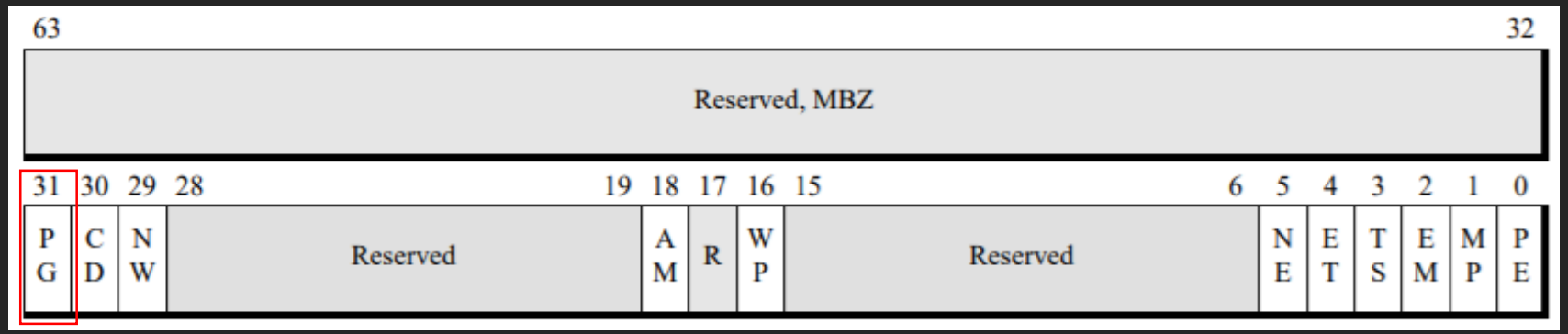
Figure 4-20. Code-Segment Descriptor—Long Mode

Ref: AMD64 Architecture Programmer's Manual Volume 2: System Programming

TEB

- CR0.PG = 1

```
32.1: kd> .formats @cr0
Evaluate expression:
  Hex:     00000000`80050033
  Decimal: 2147811379
  Octal:   00000000002000120063
  Binary:  00000000 00000000 00000000 00000000 10000000 00000101 00000000 00110011
```



Ref: AMD64 Architecture Programmer's Manual Volume 2: System Programming

TEB

- 各種模式
- CS.L = 0
- CR0.PG = 1
- x64 windows 運行 x32 程式會在 Compatibility Mode

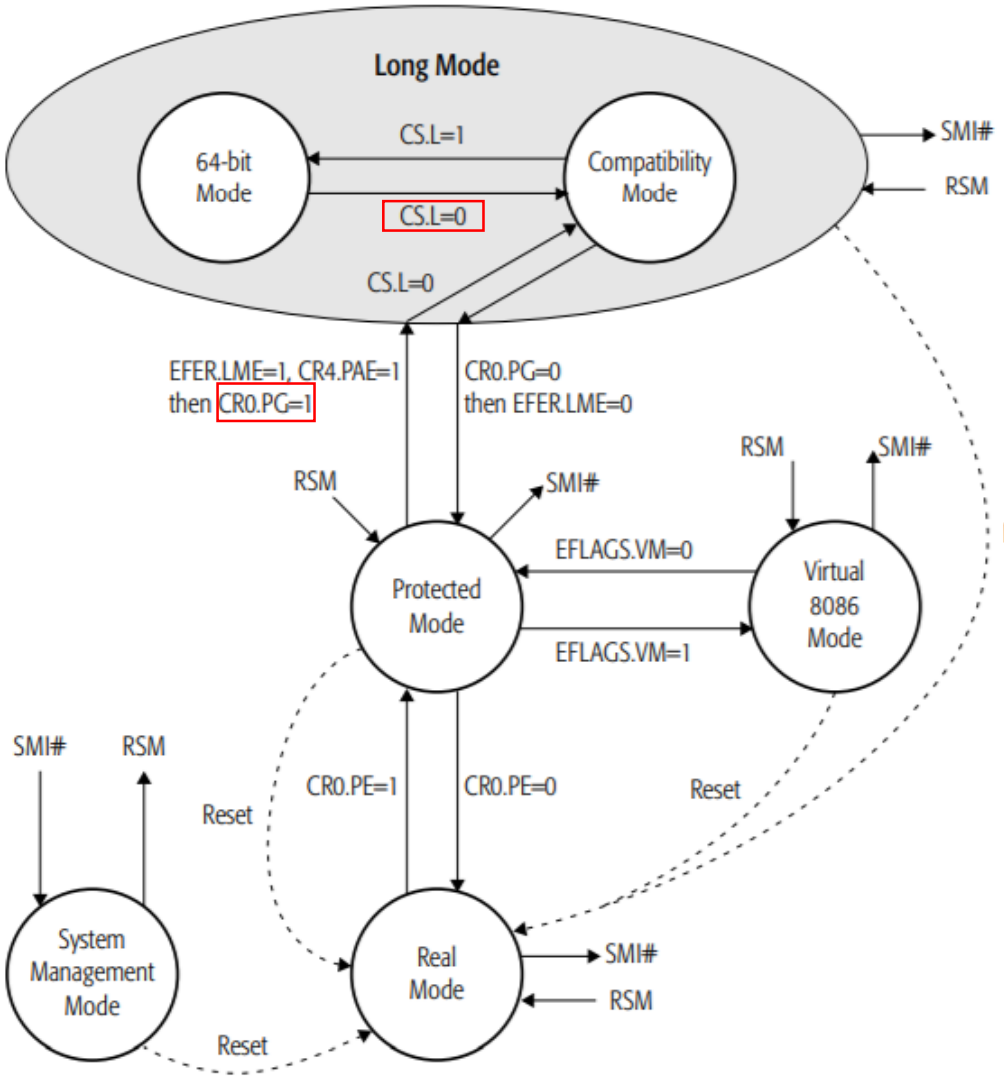


Figure 1-6. Operating Modes of the AMD64 Architecture

Ref: AMD64 Architecture Programmer's Manual Volume 2: System Programming

TEB

- 在 Compatibility mode 中, base address 是這樣來的…
- Segment Register 結構如下

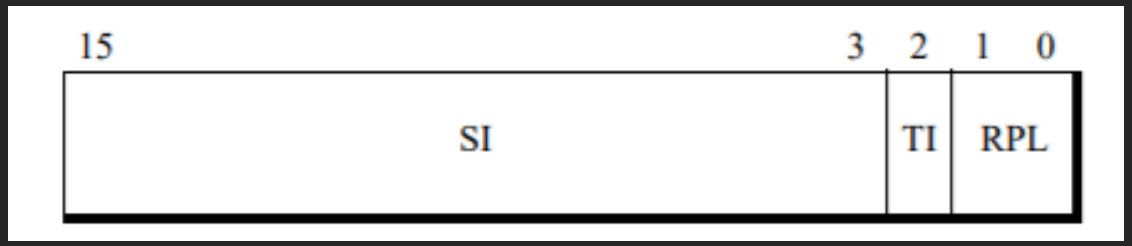
Bits	Mnemonic	Description	R/W
15:3	SI	Selector Index	R/W
2	TI	Table Indicator	R/W
1:0	RPL	Requestor Privilege Level	R/W

Figure 4-3. Segment Selector

TEB

- 看一下 FS
- FS = 0x53
- FS.TI = 0
- FS.SI = 1010 (bin) = 10 (dec)

```
32.1: kd> .formats fs
Evaluate expression:
  Hex:     00000000`00000053
  Decimal: 83
  Octal:   0000000000000000000123
  Binary:  00000000 00000000 00000000 00000000 00000000 00000000 00000000 01010011
```

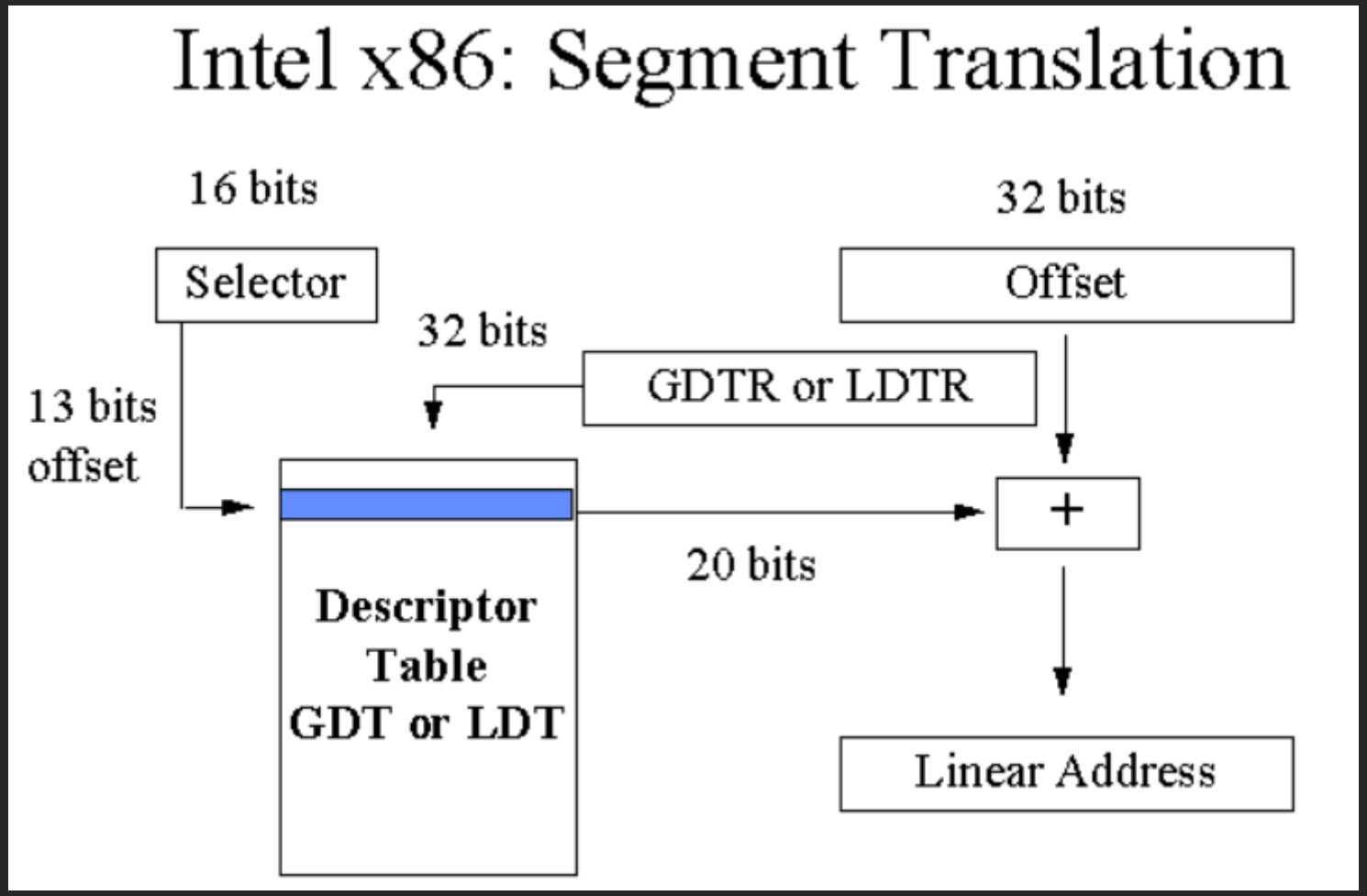


TEB

- 看一下 FS
- FS = 0x53
- FS.TI = 0
- FS.SI = 1010 (bin) = 10 (dec)
- 若 TI 為 0, 則用以下式子算 Segment Descriptor 位址
  - GDT + SI * 8
    - GDT: Global Descriptor Table
    - GDTR: GDT Register, 存放 GDT 值的暫存器

```
32.0: kd> dq (@gdtr + 0xa * 8) L1
002b:fffff805`15297000  0040f327`10003c00
```

TEB



Ref: https://www.cs.umd.edu/~meesh/cmsc411/website/saltz/cs412/lect3.html

TEB

- Segment Descriptor 結構如下

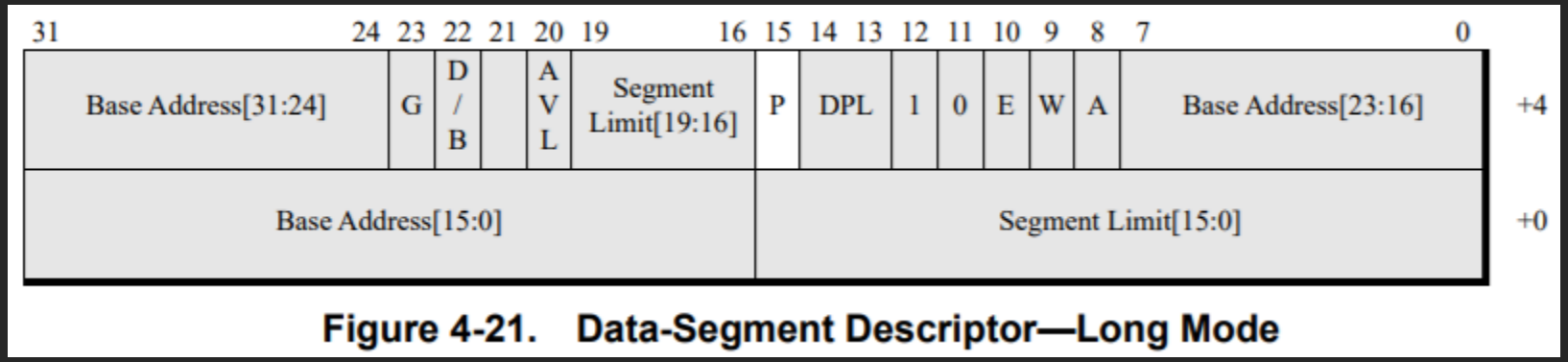


Figure 4-21. Data-Segment Descriptor—Long Mode

```
32.0: kd> dq (@gdtr + 0xa * 8) L1
002b:fffff805`15297000  0040f327`10003c00
```

TEB

- Segment Descriptor 結構如下
- 組合一下 Base Address
- 算出 FS 值為 0x53 時, 從 GDT 爬出 Segment Descriptor, 得到 Base address = 0x00271000

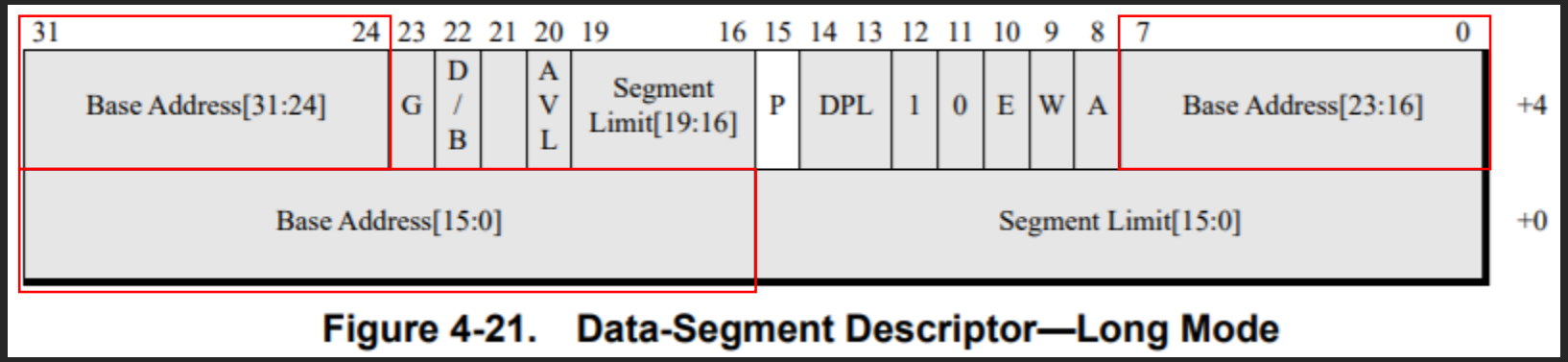


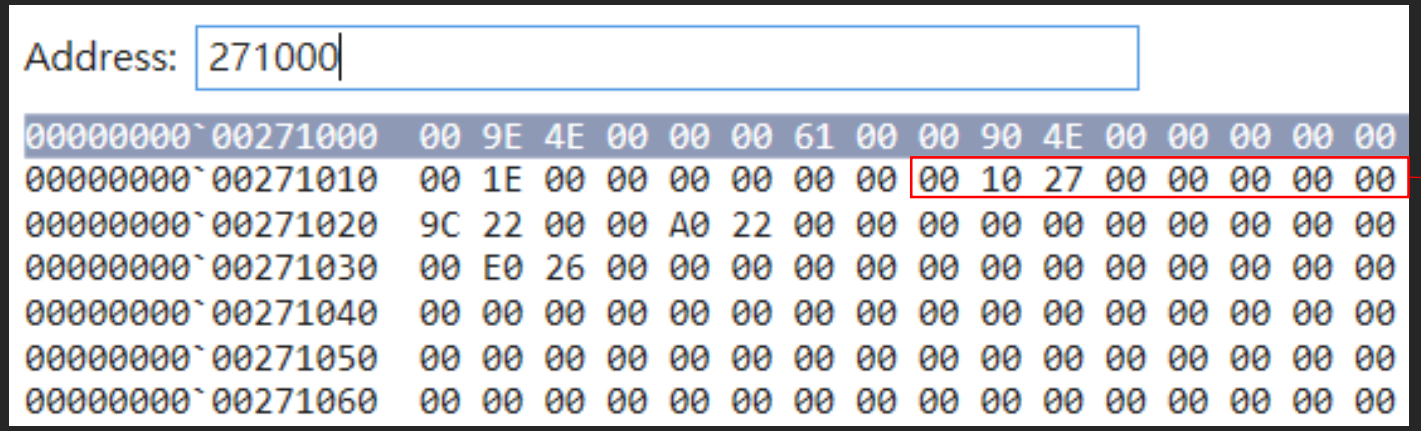
Figure 4-21. Data-Segment Descriptor—Long Mode

```
32.0: kd> dq (@gdtr + 0xa * 8) L1
002b:fffff805`15297000  0040f327`10003c00
```

TEB

- Segment Descriptor 結構如下
- 組合一下 Base Address
- 算出 FS 值為 0x53 時, 從 GDT 爬出 Segment Descriptor, 爬到 Base address = 0x00271000

- TEB 位址: 0x00271000



TEB

- 如果是在 64-bit mode …
- x64 windows 跑 x64 程式
- 跳過驗證 CS.L = 1 的部分

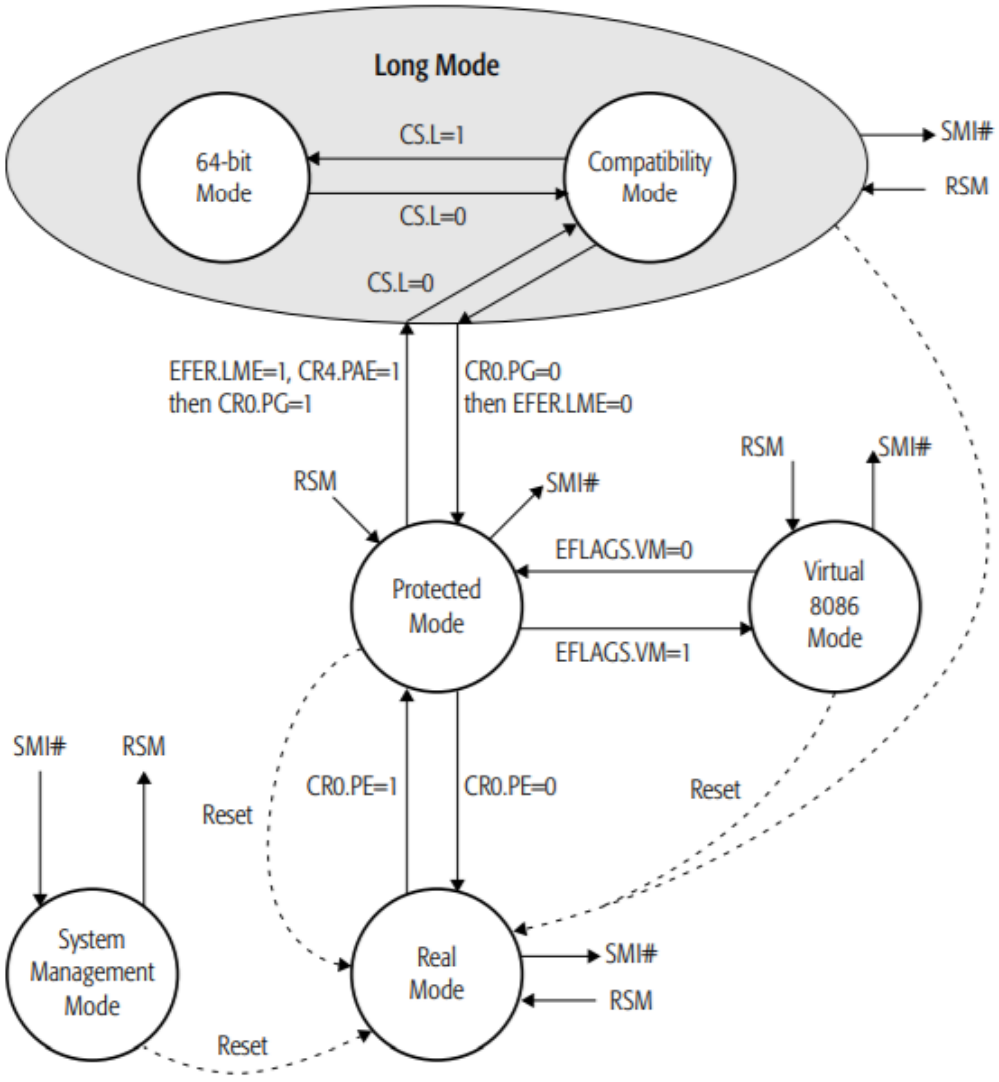


Figure 1-6. Operating Modes of the AMD64 Architecture

TEB

- GS 的 base 就是 MSR GS.Base
- GS.Base 的 MSR Address 為 0xc0000101
- 進到 Kernel 後, GS.Base 會跟另一個 MSR KernelGSBase 互換
- KernelGSBase 的 MSR Address 為 0xc0000102

TEB

- 直接就是 TEB, 不用爬 Descriptor

```
1: kd> rdmsr 0xC0000102
msr[c0000102] = 00000000`0021f000
```

Address: 21f000

```
00000000`0021F000  00 00 00 00 00 00 00 00 00 00 61 00 00 00 00 00
00000000`0021F010  00 D0 60 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000`0021F020  00 1E 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000`0021F030  00 F0 21 00 00 00 00 00 00 00 00 00 00 00 00 00
00000000`0021F040  84 11 00 00 00 00 00 00 38 0D 00 00 00 00 00 00
00000000`0021F050  00 00 00 00 00 00 00 00 F0 2B 7F 00 00 00 00 00
00000000`0021F060  00 E0 21 00 00 00 00 00 00 00 00 00 00 00 00 00
```

TEB

TEB

- 總之, 結論是…

	ExceptionList	Self	PEB
32bit FS	[0]	[0x18]	[0x30]
64bit GS	[0]	[0x30]	[0x60]

Exception

Exception

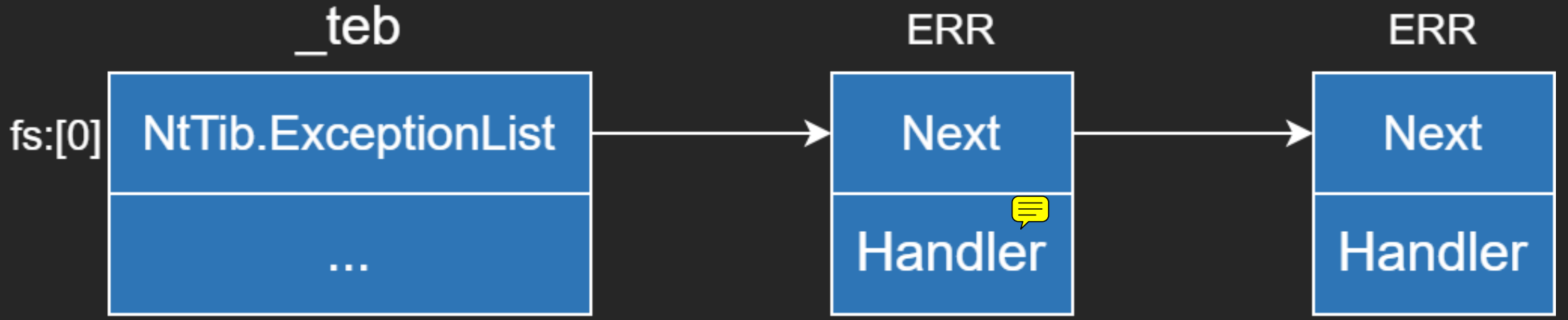
- Visual C++
  - Structured Exception Handling (SEH)
  - C++ Exception Handling (EH)
- GCC
  - RTTI
  - SjLj exceptions
  - Zero-cost (table based)

Ref: https://www.hexblog.com/wp-content/uploads/2012/06/Recon-2012-Skochinsky-Compiler-Internals.pdf

SEH

- Structured Exception Handling
- Windows 的機制
- 32 bit 與 64bit 機制不同
- Try, catch, finally 可以利用此機制實作

SEH (32-bit)



☆ ERR = _EXCEPTION_REGISTRATION_RECORD

SEH

- Handler 為 function pointer

```
1  EXCEPTION_DISPOSITION
2  __cdecl _except_handler (
3      struct _EXCEPTION_RECORD *ExceptionRecord,
4      void * EstablisherFrame,
5      struct _CONTEXT *ContextRecord,
6      void * DispatcherContext
7  );
```

SEH

- 舉個例子

```
68  int func(void *ExceptionRecord, void *EstablisherFrame, struct _CONTEXT *ContextRecord, void *DispatcherContext)
69  {
70      printf("YOYOYOYO\n");
71      printf("eip: %x\n", ContextRecord->Eip);
72      ContextRecord->Eip += 4;
73      return 0;
74  }
75
76  int main(int argc, char **argv)
77  {
78      asm ("push %[func]\n\t"
79           "mov %%fs:0, %%eax\n\t"
80           "push %%eax\n\t"
81           "mov %%esp, %%fs:0\n\t"
82           :
83           : [func] "i" (func));
84
85      printf("GOGO!\n");
86
87      if (argc < 2) {
88          return 0;
89      }
90
91      int i = 1;
92      int zero = atoi(argv[1]);
93
94      printf("%d / %d = %d\n", i, zero, i / zero);
95  }
```

```
68  int func(void *ExceptionRecord, void *EstablisherFrame, struct _CONTEXT *ContextRecord, void *DispatcherContext)
69  {
70      printf("YOYOYOYO\n");
71      printf("eip: %x\n", ContextRecord->Eip);
72      ContextRecord->Eip += 4;
73      return 0;
74  }
75
76  int main(int argc, char **argv)
77  {
78      asm ("push %[func]\n\t"
79           "mov %%fs:0, %%eax\n\t"
80           "push %%eax\n\t"
81           "mov %%esp, %%fs:0\n\t"
82           :
83           : [func] "i" (func));
84
85      printf("GOGO!\n");
86
87      if (argc < 2) {
88          return 0;
89      }
90
91      int i = 1;
92      int zero = atoi(argv[1]);
93
94      printf("%d / %d = %d\n", i, zero, i / zero);
95  }
```

在 stack 創一個 ERR
Handler 指向自製 handler func

```c
int func(void *ExceptionRecord, void *EstablisherFrame, struct _CONTEXT *ContextRecord, void *DispatcherContext)
{
    printf("YOYOYOYO\n");
    printf("eip: %x\n", ContextRecord->Eip);
    ContextRecord->Eip += 4;
    return 0;
}

int main(int argc, char **argv)
{
    asm ("push %[func]\n\t"
         "mov %%fs:0, %%eax\n\t"
         "push %%eax\n\t"
         "mov %%esp, %%fs:0\n\t"
         :
         : [func] "i" (func));

    printf("GOGO!\n");

    if (argc < 2) {
        return 0;
    }

    int i = 1;
    int zero = atoi(argv[1]);

    printf("%d / %d = %d\n", i, zero, i / zero);
}
```

在 stack 創一個 ERR Handler 指向自製 handler func

Zero 給 0, 製造 exception

```
68  int func(void *ExceptionRecord, void *EstablisherFrame, struct _CONTEXT *ContextRecord, void *DispatcherContext)
69  {
70      printf("YOYOYOYO\n");
71      printf("eip: %x\n", ContextRecord->Eip);
72      ContextRecord->Eip += 4;
73      return 0;
74  }
75
76  int main(int argc, char **argv)
77  {
78      asm ("push %[func]\n\t"
79           "mov %%fs:0, %%eax\n\t"
80           "push %%eax\n\t"
81           "mov %%esp, %%fs:0\n\t"
82           :
83           : [func] "i" (func));
84
85      printf("GOGO!\n");
86
87      if (argc < 2) {
88          return 0;
89      }
90
91      int i = 1;
92      int zero = atoi(argv[1]);
93
94      printf("%d / %d = %d\n", i, zero, i / zero);
95  }
```

導致 exception 的指令 idiv 為 4 Bytes
+4 跳過 idiv 指令

```
.text:004014ED F7 7C 24 18                    idiv    [esp+28h+var_10]
.text:004014F1 89 44 24 0C                    mov     [esp+28h+var_1C], eax
```

在 stack 創一個 ERR
Handler 指向自製 handler func

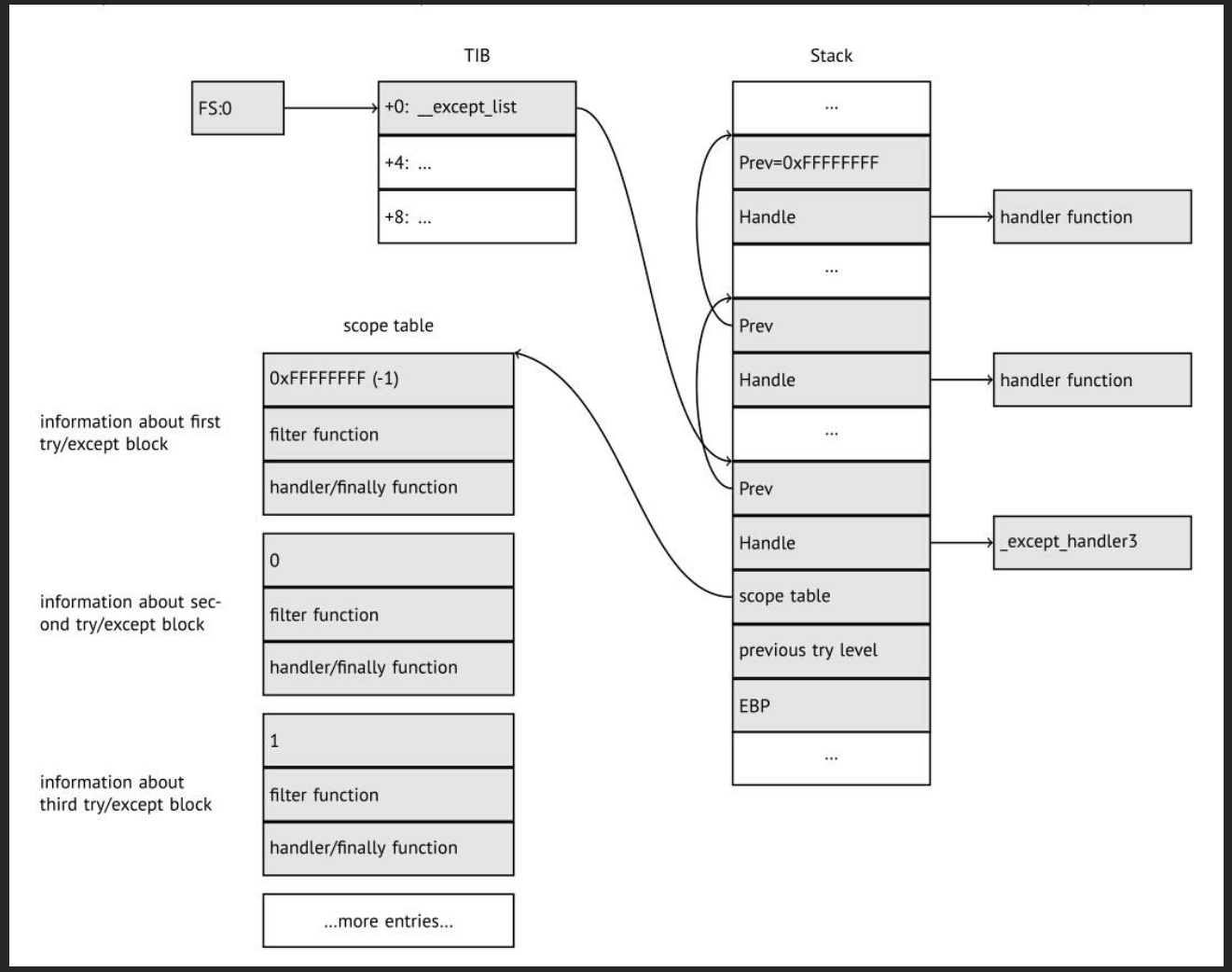
Zero 給 0, 製造 exception

```
PS C:\tmp\presentation\secure programming\code\LAB_9> .\except.exe 0
GOGO!
YOYOYOYO
eip: 401502
1 / 0 = 1
PS C:\tmp\presentation\secure programming\code\LAB_9>
```

SEH

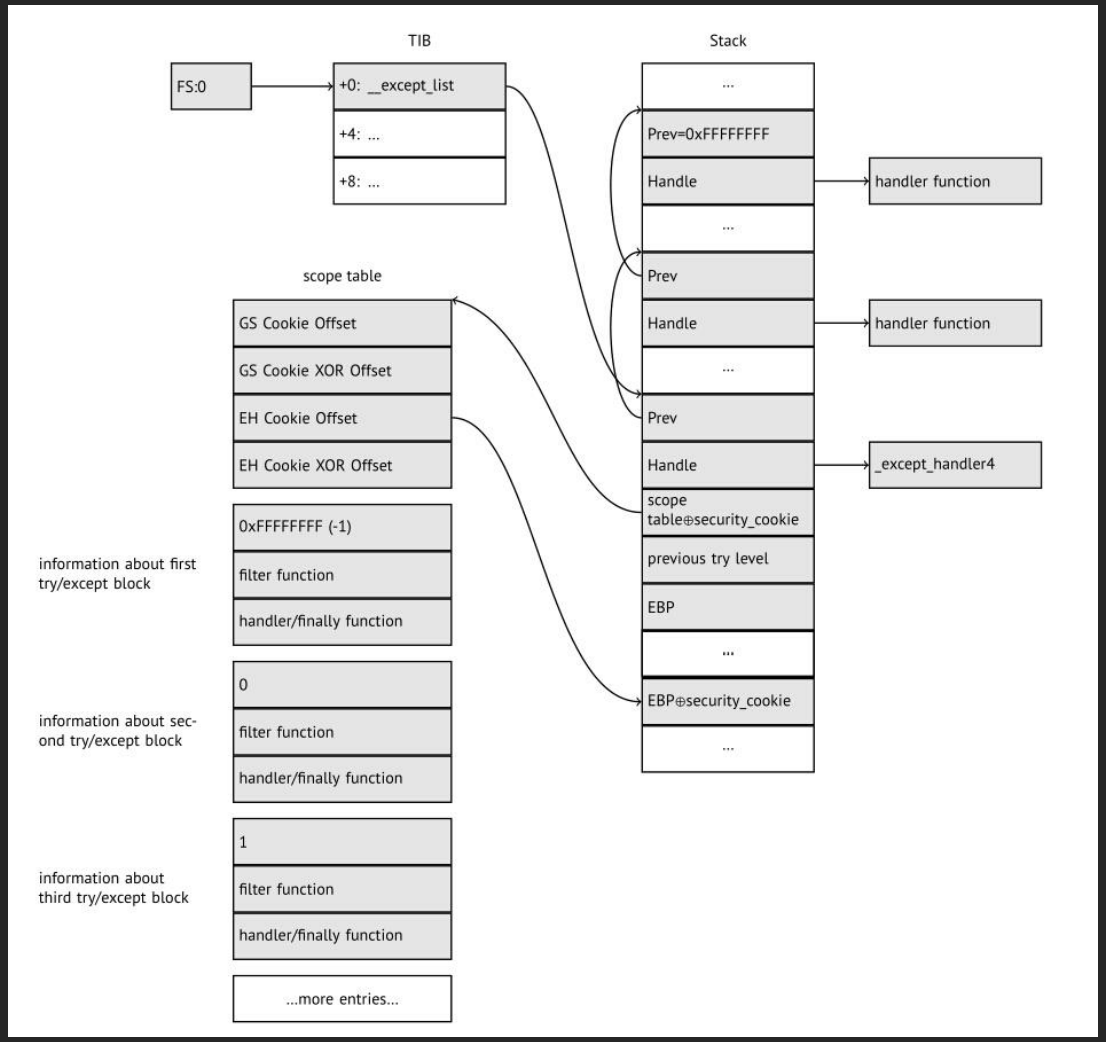
- 逆向方式就是逆 handler
- 不同 compiler 的 handler 實作都不同
- VS 用的 MSVC，handler 的實作…

_except_handler3



Ref: https://www.bookstack.cn/read/reverse-engineering-for-beginners-zh/spilt.16.spilt.35.book.md

_except_handler4



Ref: https://www.bookstack.cn/read/reverse-engineering-for-beginners-zh/spilt.16.spilt.35.book.md

SEH

- 逆向方式就是逆 handler
- 不同 compiler 的 handler 實作都不同
- VS 用的 MSVC，handler 的實作…
- 事情變得很複雜, 但總之就是在 handler 裡面折騰就對ㄌ

SEH

- 回顧一下
- 若 function 需要自行新增 ERR (可能函數內有 try-catch)
- 此 function 需為了增加此 ERR 而在 prolog/epilog 加 code
- 但 exception 又是較少跑到的
- 加的那些 code 很常是跑心酸的

SEH (64-bit)

- 64 bit, SEH 不用鏈表了, 改成 table-based
- 什麼 table? 請看 Exception Directory

Disasm: .text | General | DOS Hdr | Rich Hdr | File Hdr | Optional Hdr | Section Hd

Offset	Name	Value	Value
	Data Directory	Address	Size
180	Export Directory	0	0
188	Import Directory	3F16C	28
190	Resource Directory	47000	1E0
198	Exception Directory	43000	2904
1A0	Security Directory	0	0

SEH (64-bit)

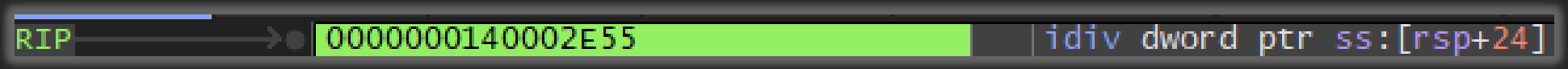
- Exception Directory

Exception

Offset	BeginAddress	EndAddress	UnwindInfoAddress
40400	1000	1017	3CE98
4040C	1020	1037	3CE98
40418	1038	1058	3CE98
40424	1058	1088	3CE98
40430	1088	10D8	3D234
4043C	10E4	1104	3CE98
40448	111C	113C	3CE98
40454	1180	11A4	3CE28

SEH (64-bit)

- 怎麼找 handler?
- 首先看你在哪裡丟出 exception
- 0x140002e55 除 0



SEH (64-bit)

- 0x140002e55 除 0
- 將 0x140002e55 換算回 RVA: 0x2e55
- 查表

Exception

Offset	BeginAddress	EndAddress	UnwindInfoAddress
40638	2DB0	2DE6	3CDF8
40644	2DF0	2EE4	3CED8

SEH (64-bit)

- 查看 UnwindInfoAddress

RVA 換算回 Raw Offset

3C2D8	19 0D 01 00 0D 62 00 00 AC D2 00 00 E8 CE 03 00
3C2E8	18 F5 CE 03 00 F8 CE 03 00 06 CF 03 00 04 08 10
3C2F8	02 00 00 02 00 CF 03 00 02 00 20 C6 02 00 06 00

Exception

Offset	BeginAddress	EndAddress	UnwindInfoAddress
40638	2DB0	2DE6	3CDF8
40644	2DF0	2EE4	3CED8

SEH (64-bit)

- 查看 UnwindInfoAddress
- 對應其結構

3C2D8	19	0D	01	00	0D	62	00	00	AC	D2	00	00	E8	CE	03	00
3C2E8	18	F5	CE	03	00	F8	CE	03	00	06	CF	03	00	04	08	10
3C2F8	02	00	00	02	00	CF	03	00	02	00	20	C6	02	00	06	00

```
typedef struct _UNWIND_INFO {
    UBYTE Version       : 3;
    UBYTE Flags         : 5;
    UBYTE SizeOfProlog;
    UBYTE CountOfCodes;
    UBYTE FrameRegister : 4;
    UBYTE FrameOffset   : 4;
    UNWIND_CODE UnwindCode[1];
/*  UNWIND_CODE MoreUnwindCode[((CountOfCodes + 1) & ~1) - 1];
*   union {
*       OPTIONAL ULONG ExceptionHandler;
*       OPTIONAL ULONG FunctionEntry;
*   };
*   OPTIONAL ULONG ExceptionData[]; */
} UNWIND_INFO, *PUNWIND_INFO;
```

Ref: https://docs.microsoft.com/en-us/cpp/build/exception-handling-x64?view=msvc-170

SEH (64-bit)

- 查看 UnwindInfoAddress
- 對應其結構

```
3C2D8   19 0D 01 00 0D 62 00 00 AC D2 00 00 E8 CE 03 00
3C2E8   18 F5 CE 03 00 F8 CE 03 00 06 CF 03 00 04 08 10
3C2F8   02 00 00 02 00 CF 03 00 02 00 20 C6 02 00 06 00
```

CountOfCodes = 1

```
typedef struct _UNWIND_INFO {
    UBYTE Version       : 3;
    UBYTE Flags         : 5;
    UBYTE SizeOfProlog;
    UBYTE CountOfCodes;
    UBYTE FrameRegister : 4;
    UBYTE FrameOffset   : 4;
    UNWIND_CODE UnwindCode[1];
/*  UNWIND_CODE MoreUnwindCode[((CountOfCodes + 1) & ~1) - 1];
*   union {
*       OPTIONAL ULONG ExceptionHandler;
*       OPTIONAL ULONG FunctionEntry;
*   };
*   OPTIONAL ULONG ExceptionData[]; */
} UNWIND_INFO, *PUNWIND_INFO;
```

Ref: https://docs.microsoft.com/en-us/cpp/build/exception-handling-x64?view=msvc-170

SEH (64-bit)

- 查看 UnwindInfoAddress
- 對應其結構

```
3C2D8   19 0D 01 00 0D 62 00 00 AC D2 00 00 E8 CE 03 00
3C2E8   18 F5 CE 03 00 F8 CE 03 00 06 CF 03 00 04 08 10
3C2F8   02 00 00 02 00 CF 03 00 02 00 20 C6 02 00 06 00
```

CountOfCodes = 1

UnwindCode 陣列

```
typedef struct _UNWIND_INFO {
    UBYTE Version       : 3;
    UBYTE Flags
    UBYTE SizeOfProlo
    UBYTE CountOfCode
    UBYTE FrameRegiste
    UBYTE FrameOffset   : 4;
    UNWIND_CODE UnwindCode[1];
/*  UNWIND_CODE MoreUnwindCode[((CountOfCodes + 1) & ~1) - 1];
*   union {
*       OPTIONAL ULONG ExceptionHandler;
*       OPTIONAL ULONG FunctionEntry;
*   };
*   OPTIONAL ULONG ExceptionData[]; */
} UNWIND_INFO, *PUNWIND_INFO;
```

UNWIND_CODE 大小為 2 Bytes
陣列長度為 CountOfCodes

Ref: https://docs.microsoft.com/en-us/cpp/build/exception-handling-x64?view=msvc-170

SEH (64-bit)

- 查看 UnwindInfoAddress
- 對應其結構

```
3C2D8   19 0D 01 00 0D 62 00 00 AC D2 00 00 E8 CE 03 00
3C2E8   18 F5 CE 03 00 F8 CE 03 00 06 CF 03 00 04 08 10
3C2F8   02 00 00 02 00 CF 03 00 02 00 20 C6 02 00 06 00
```

CountOfCodes = 1

ExceptionHandler

UnwindCode 陣列

```c
typedef struct _UNWIND_INFO {
    UBYTE Version       : 3;
    UBYTE Flags         : 5;
    UBYTE SizeOfProlog;
    UBYTE CountOfCodes;
    UBYTE FrameRegister : 4;
    UBYTE FrameOffset   : 4;
    UNWIND_CODE UnwindCode[1];
/*  UNWIND_CODE MoreUnwindCode[((CountOfCodes + 1) & ~1) - 1];
*   union {
*       OPTIONAL ULONG ExceptionHandler;
*       OPTIONAL ULONG FunctionEntry;
*   };
*   OPTIONAL ULONG ExceptionData[]; */
} UNWIND_INFO, *PUNWIND_INFO;
```

Ref: https://docs.microsoft.com/en-us/cpp/build/exception-handling-x64?view=msvc-170

SEH (64-bit)

- 查看 UnwindInfoAddress
- 對應其結構
- 找到 handler 開逆

```
3C2D8   19 0D 01 00 0D 62 00 00 AC D2 00 00 E8 CE 03 00
3C2E8   18 F5 CE 03 00 F8 CE 03 00 06 CF 03 00 04 08 10
3C2F8   02 00 00 02 00 CF 03 00 02 00 20 C6 02 00 06 00
```

CountOfCodes = 1

ExceptionHandler

UnwindCode 陣列

```
.text:000000014000D2AC __CxxFrameHandler4 proc near             ; CO
.text:000000014000D2AC                                          ; DA
.text:000000014000D2AC
.text:000000014000D2AC rawIP2StateRVA  = byte ptr -48h
.text:000000014000D2AC var_40          = dword ptr -40h
.text:000000014000D2AC var_38          = qword ptr -38h
.text:000000014000D2AC recursive       = byte ptr -30h
.text:000000014000D2AC FuncInfoDe      = FH4::FuncInfo4 ptr -28h
.text:000000014000D2AC var_8           = byte ptr -8
.text:000000014000D2AC pRN             = qword ptr  8
.text:000000014000D2AC
.text:000000014000D2AC                 mov     rax, rsp
.text:000000014000D2AF                 mov     [rax+10h], rbx
.text:000000014000D2B3                 mov     [rax+18h], rbp
.text:000000014000D2B7                 mov     [rax+20h], rsi
```

SEH (64-bit)

- IDA pro 很 pro, 都爬好了
- 但 freeware 沒有很 pro, 沒有爬 QQ

```
.text:0000000140002DF0 main            proc near               ; CODE XI
.text:0000000140002DF0                                         ; DATA XI
.text:0000000140002DF0
.text:0000000140002DF0 _Val            = dword ptr -18h
.text:0000000140002DF0 var_14          = dword ptr -14h
.text:0000000140002DF0 var_10          = dword ptr -10h
.text:0000000140002DF0 arg_0           = dword ptr  8
.text:0000000140002DF0 arg_8           = qword ptr  10h
.text:0000000140002DF0
.text:0000000140002DF0 ; __unwind { // __CxxFrameHandler4
.text:0000000140002DF0                 mov     [rsp+arg_8], rdx
.text:0000000140002DF5                 mov     [rsp+arg_0], ecx
.text:0000000140002DF9                 sub     rsp, 38h
.text:0000000140002DFD                 lea     rdx, _Val       ; "gogo"
```

Lab 2

Packer

Packer

- 目的是將程式變得難逆
- Packer 中文稱為加殼器
- 常見的殼類型分為
  - 壓縮殼
  - VM 殼

Packer

- 壓縮殼
  - 把 code 壓縮起來
  - 在執行時才把 code 解壓縮回來, 並且執行
  - UPX

- VM 殼
  - 實作另一套 VM
  - 把原始 code 變成給 VM 跑的 code
  - 想逆? 請直接逆完 VM
  - VMProtect

Packer

- UPX

```
PS C:\tmp\presentation\secure programming\code\lab_9> upx
                       Ultimate Packer for eXecutables
          Copyright (C) 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004
UPX 1.25w           Markus F.X.J. Oberhumer & Laszlo Molnar           Jun 29th 2004

Usage: upx [-123456789dlthVL] [-qvfk] [-o file] file..

Commands:
  -1     compress faster                   -9    compress better
  -d     decompress                        -l    list compressed file
  -t     test compressed file              -V    display version number
  -h     give more help                    -L    display software license
Options:
  -q     be quiet                          -v    be verbose
  -oFILE write output to `FILE'
  -f     force compression of suspicious files
  -k     keep backup files
  file.. executables to (de)compress

This version supports: dos/exe, dos/com, dos/sys, djgpp2/coff, watcom/le,
                       win32/pe, rtm32/pe, tmt/adam, atari/tos, linux/386

UPX comes with ABSOLUTELY NO WARRANTY; for details type `upx -L'.
```

Ref: https://upx.github.io/

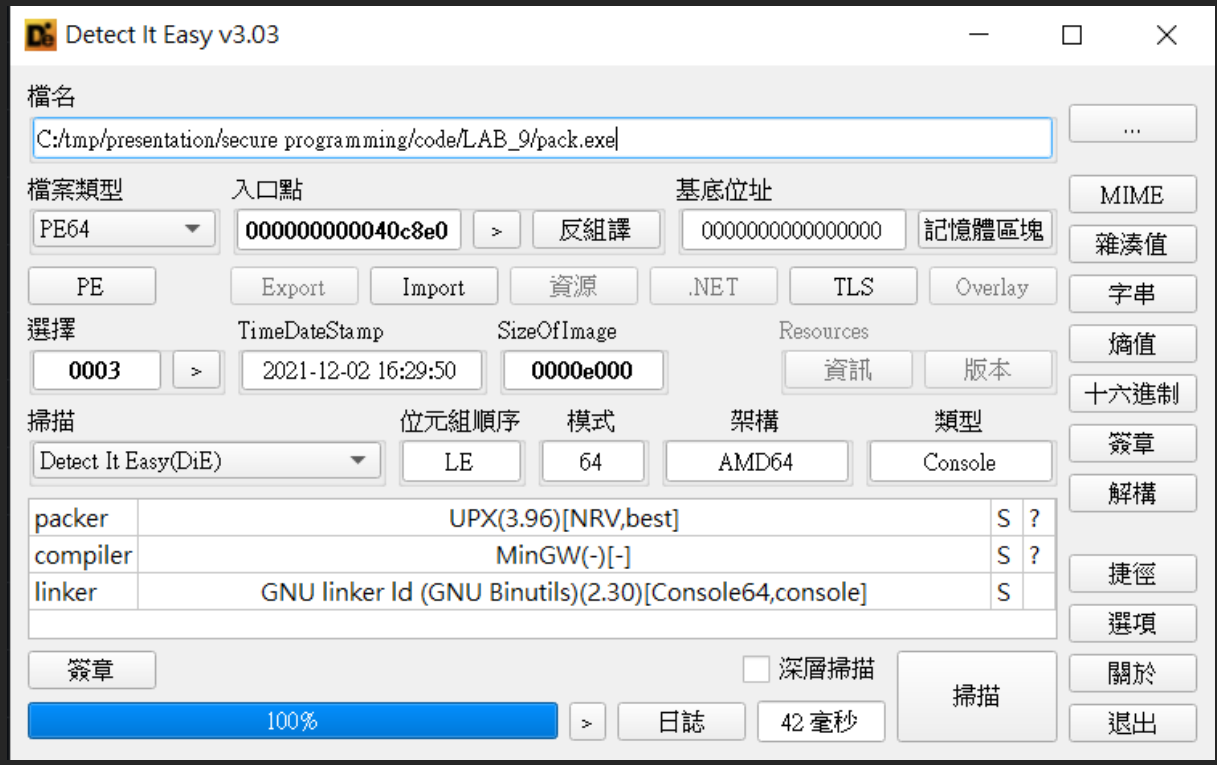
Packer

- UPX

```
PS C:\tmp\presentation\secure programming\code\LAB_9> upx64 .\unpackme.exe -o pack.exe
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2020
UPX 3.96w       Markus Oberhumer, Laszlo Molnar & John Reiser   Jan 23rd 2020

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
     15360 ->      8192   53.33%    win64/pe     pack.exe

Packed 1 file.
PS C:\tmp\presentation\secure programming\code\LAB_9> ls


    目錄: C:\tmp\presentation\secure programming\code\LAB_9


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        2021/12/2  下午 03:59            132 Makefile
-a----        2021/12/2  下午 04:29           8192 pack.exe
-a----        2021/12/2  下午 03:58             64 unpackme.c
-a----        2021/12/2  下午 04:29          15360 unpackme.exe
```

Packer

- DIE (Detect It Easy)
- 查殼的工具
- 查到殼後再上網找脫殼器
- 再不行才自己脫殼



Ref: https://github.com/horsicq/Detect-It-Easy

Anti-Reverse

Anti-Reverse

- 蠻多花招可以反逆向工程
- 其實前面講的幾個點就是在反逆向工程
- 太多招了, 可以參考 Reference 連結
- 這個章節舉幾個例子

Ref: https://github.com/LordNoteworthy/al-khaser

Anti-Disassembly

- 想一下怎麼實作反組譯器

- Linear disassembly
  - 一行一行的反組譯下去

- Flow-oriented disassembly
  - 如果反組譯到 jmp, 則順著執行流程反組譯
  - IDA

Anti-Disassembly

- 舉個例子

Anti-Disassembly

```
3  int evil_func()
4  {
5      printf("Bye~\n");
6
7      asm ("call SELF\n\t"
8           "SELF:\n\t"
9           "pop %%rax\n\t"
10          "add $0xd, %%rax\n\t"
11          "jmp *%%rax\n\t"
12          :
13          :
14          :
15     );
16
17     asm ("1: .byte 0xe9, 0x80, 0x87, 0x55, 0x66, 0x01 \n\t":::);
18
19     printf("YO!\n");
20 }
```

Anti-Disassembly

```
3  int evil_func()
4  {
5      printf("Bye~\n");
6
7      asm ("call SELF\n\t"
8           "SELF:\n\t"
9           "pop %%rax\n\t"
10          "add $0xd, %%rax\n\t"
11          "jmp *%%rax\n\t"
12          :
13          :
14          :
15     );
16
17     asm ("1: .byte 0xe9, 0x80, 0x87, 0x55, 0x66, 0x01 \n\t":::);
18
19     printf("YO!\n");
20 }
```

取得當前指令位址, 加上 offset 後跳過去

Anti-Disassembly

```
3  int evil_func()
4  {
5      printf("Bye~\n");
6
7      asm ("call SELF\n\t"
8           "SELF:\n\t"
9           "pop %%rax\n\t"
10          "add $0xd, %%rax\n\t"
11          "jmp *%%rax\n\t"
12          :
13          :
14          :
15     );
16
17     asm ("1: .byte 0xe9, 0x80, 0x87, 0x55, 0x66, 0x01 \n\t":::);
18
19     printf("YO!\n");
20 }
```

取得當前指令位址, 加上 offset 後跳過去

實際上就是跳到這邊

Anti-Disassembly

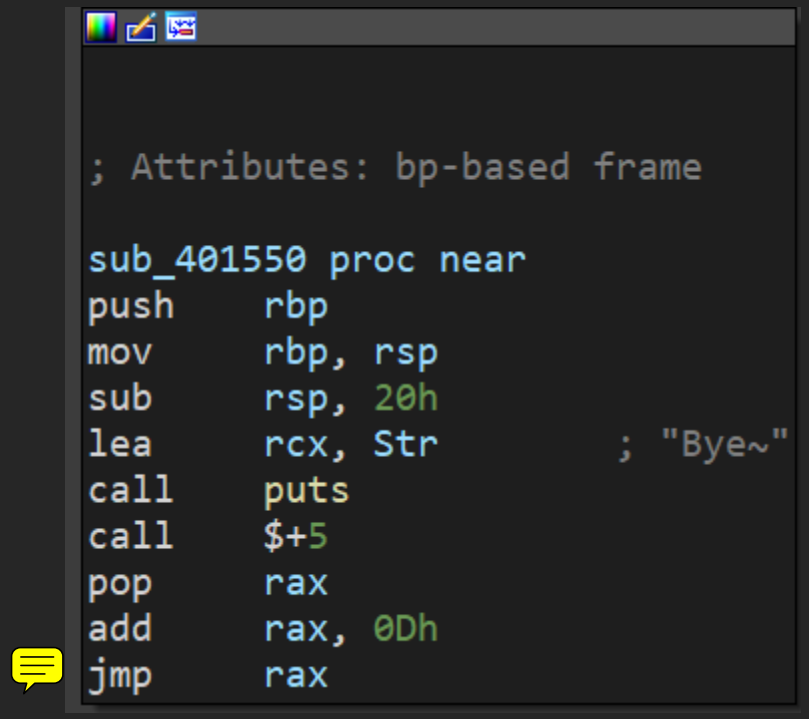
```
3  int evil_func()
4  {
5      printf("Bye~\n");
6
7      asm ("call SELF\n\t"
8           "SELF:\n\t"
9           "pop %%rax\n\t"
10          "add $0xd, %%rax\n\t"
11          "jmp *%%rax\n\t"
12          :
13          :
14          :
15     );
16
17     asm ("1: .byte 0xe9, 0x80, 0x87, 0x55, 0x66, 0x01 \n\t":::);
18
19     printf("YO!\n");
20 }
```

取得當前指令位址, 加上 offset 後跳過去

在這之間塞一坨垃圾

實際上就是跳到這邊

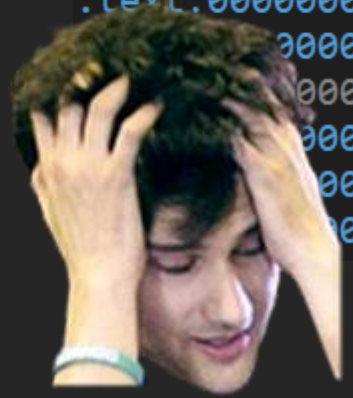
Anti-Disassembly



Graph View 不行? 換 Text View?

Anti-Disassembly

```
.text:0000000000401550 sub_401550      proc near               ; CODE XREF: main+19↓p
.text:0000000000401550                                         ; DATA XREF: .pdata:000000000040506C↓o
.text:0000000000401550                 push    rbp
.text:0000000000401551                 mov     rbp, rsp
.text:0000000000401554                 sub     rsp, 20h
.text:0000000000401558                 lea     rcx, Str        ; "Bye~"
.text:000000000040155F                 call    puts
.text:0000000000401564                 call    $+5
.text:0000000000401569                 pop     rax
.text:000000000040156A                 add     rax, 0Dh
.text:000000000040156E                 jmp     rax
.text:000000000040156E ; ---------------------------------------------------------------------------
.text:0000000000401570                 dq 8D480166558780E9h, 151EE800002A880Dh, 5D20C48348900000h
.text:0000000000401588 ; ---------------------------------------------------------------------------
.text:0000000000401588                 retn
.text:0000000000401588 sub_401550      endp
```



不好意思, IDA 不知道哪邊是 code, 請手動定義

Anti-Disassembly

```
00401550    PUSH    RBP
00401551    MOV     RBP,RSP
00401554    SUB     RSP,0x20
00401558    LEA     RCX,[DAT_00404000]
0040155f    CALL    MSVCRT.DLL::puts
00401564    CALL    LAB_00401569

                    LAB_00401569
00401569    POP     RAX
0040156a    ADD     RAX,0xd
0040156e    JMP     RAX=>LAB_00401576
00401570    ??      E9h
00401571    ??      80h
00401572    ??      87h
00401573    ??      55h     U
00401574    ??      66h     f
00401575    ??      01h

                    LAB_00401576
00401576    LEA     RCX,[DAT_00404005]
0040157d    CALL    MSVCRT.DLL::puts
00401582    NOP
00401583    ADD     RSP,0x20
00401587    POP     RBP
00401588    RET
```

但 Ghidra 解的出來, 太神啦

Anti-Debug

- 偵測是不是正在被 debug

- IsDebuggerPresent
- CheckRemoteDebuggerPresent
- …

- x64dbg 可以用 ScyllaHide 來反制

Ref: https://github.com/x64dbg/ScyllaHide

Anti-Debug

- Debugger 是怎麼達到 “設定中斷點” 這件事情的?
- x64dbg 預設方式是用 int 3, opcode 為 0xcc
- 把設斷點的位址內容改成 0xcc
- 執行到 int 3 時會觸發 exception_breakpoint
- Debugger 接收此 exception, 並且把原本指令填回去

Anti-Debug

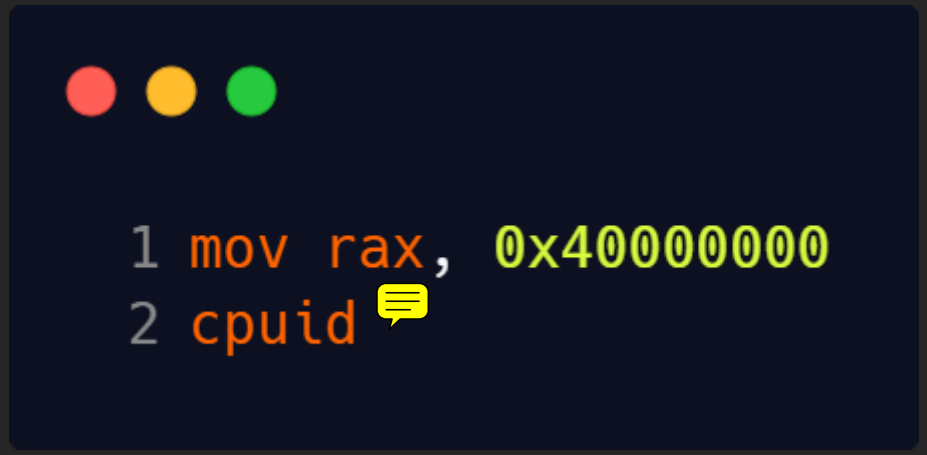
- 直接掃 code 段記憶體是否有 0xcc
- 就知道有沒有被設中斷點
- 就知道有沒有 debugger

Anti-VM

- 分析者通常是把惡意程式丟進 VM 裡面動態分析
- 各種 VM 會有自己特別的檔案 / Registry / 行為 / 裝置 / 程序
- 用這些資訊來判斷自己是不是在 VM 裡面

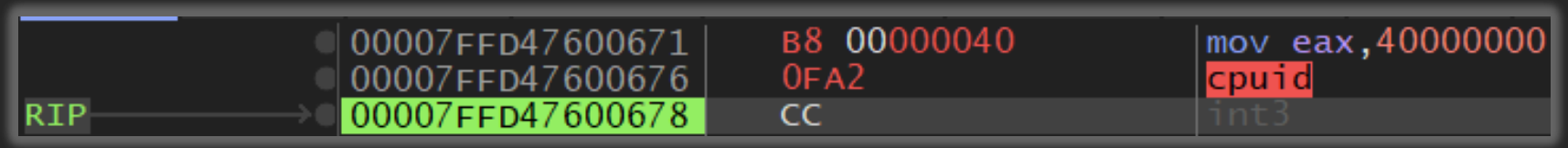
Anti-VM

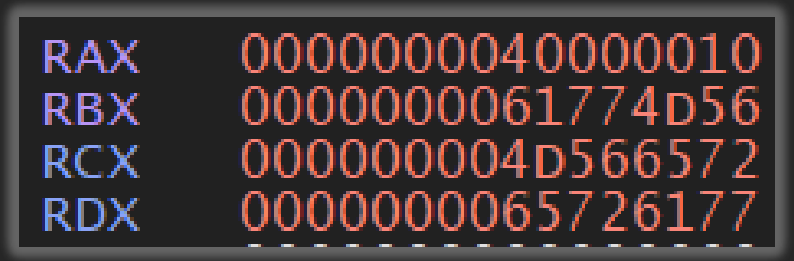
- 舉個例子: cpuid



- 執行完後, ebx ecx edx 的值會有特徵

Anti-VM






```
>>> bytes.fromhex('61774D56')[::-1]
b'VMwa'
>>> bytes.fromhex('4D566572')[::-1]
b'reVM'
>>> bytes.fromhex('65726177')[::-1]
b'ware'
```

Q & A

下課囉 \(. _ .)>