The War in the Stack

Blekinge Institute of Technology

Ge Zhang

Index

1 ABSTRACT 3

2 INTRODUCTION OF THE STACK 4

2.1 What Is Buffer Overflow and Stack 4

2.2 What Is the Damage of Buffer Overflow 4

2.3 The Guide of Shell Code 5

2.4 An Example of buffer overflow exploit 5

3 STACK PROTECTION 7

3.1 Two Ways 7

3.2 The Memory Structure in Windows NT 7

3.3 Use Hook API to Prevent shellcode execution 7

4 ANTI-(STACK PROTECTION) 10

4.1 User mode attacking 10

4.2 Kernel mode attacking 10

4.2.1 Fragment shellcode 10

4.2.2 Forgery stack frames 11

5 CONCLUSION 12

6 REFERENCE 12

1 Abstract

This paper mainly focuses on the stack which is always interested by hackers and security researchers. I will introduce that what is the stack, why stack overflow will cause problem, how to protect buffer overflow by avoiding shellcode executing and the technique that can let hackers to bypass the buffer overflow protection mechanism. I will also take some knowledge of hook API, Memory structure of Microsoft Windows as foundation.

The debug environment:
O.S.: Windows XP Pro, SP2
IDE: Visual C++ 6.0

Key Words: Buffer Overflow, Hook API, Stack

2 Introduction of The stack

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 [1].For example:

Code 2.1

OS: windows XP professional SP2

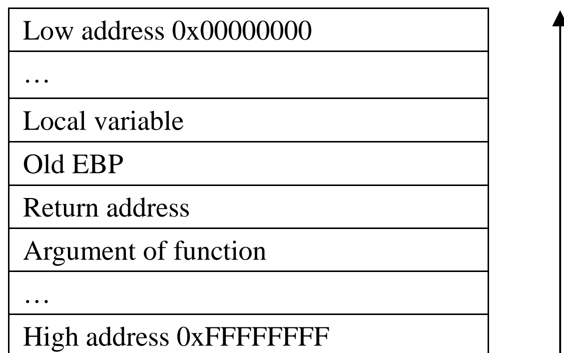
Compiler: VC 6.0

```
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:



Low address 0x00000000
…
Local variable
Old EBP
Return address
Argument of function
…
High address 0xFFFFFFFF

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.

2.3 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 [2].
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:

2.4 An Example of buffer overflow exploit

Code 2.2
OS: windows XP professional SP2
Compiler: VC 6.0

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


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 test();
int main(int argc, char* argv[])
{
   char a[200];
   int i;
   for(i=0;i<125;i++) a[i]=shellcode[i];//inject CMD code to stack
   a[125]='\0';
   test();
```

```
    return 0;
}

void test()
{

    char buf[126];
    char* j;
    int i;
    for(i=0;i<132;i++) buf[i]='x';    //to reach the return address
    buf[132]=0xac;                    //revise return address (a)
    buf[133]=0xfe;
    buf[134]=0x12;
    buf[135]=0x00;
}
```

The function of shellcode is to startup a new console (CMD shell) on the screen. At first, I inject the shellcode into the char array a, and then, call function test. In the array buf of this function, I let it buffer overflow and return to the address of array a. As the result, the system begins to execute the shellcode in a. Therefore, a intruder can do everything in a system with some buffer overflow vulnerability if he uses different shellcodes.

3 Stack Protection

Because so many problems of information security are caused by buffer overflow in recent years, more and more organizations have begun to find the way of how to protect system from buffer overflow.

3.1 Two Ways

One way is to set a random check value between return address and other values when the program is being compiled. If we try to use buffer overflow to revise the return address, the check value will also be changed. They call this check value canary, and the program will alarm if the canary was different with before.
Another way is to make the stack non-executable. In most cases, the injected attack code will run from the stack (just as the example before, I copy the shellcode to a buf a, and then call the address of it).Furthermore, in windows system, the attribute of code segment is executable but non-writable, while the attribute of stack is both executable and writeable. Therefore, we can make use of the difference of code segment and stack and trace the return address and check the kind of attributes that the page has since in most cases, programs are not running in stack [3]. In this article, we focus on the second technique.

3.2 The Memory Structure in Windows NT

In Windows NT, the virtual address space of the system is divided into two parts: the low 2GB user address space and the high 2 GB system space. When the CPU is running in user mode, only pages of the user address space are accessible, so applications cannot interfere with the operating system components that are accessible only in kernel mode. All the user processes share the high 2 GB system space. When a user-mode application calls API, it first calls into a sub-system DLL. The subsystem DLL API translates the documented function to an undocumented one in the native API set as part of NTDLL.DLL. When necessary, the native API calls the Windows NT executive, and the processor is switched to kernel mode [3]. Therefore, is it possible to extend some functions into a certain API to monitor the content of register and memory page of the program?

3.3 Use Hook API to Prevent shellcode execution

At first, let us have a look at the structure of windows PE file.

The signature of DOS (“ME”)
…The signature of PE (“PE”)
.text segment
.data initialization data
.idata import table
.edata export table
Debugger information

When we write PE programs, we call the API functions by name, not by the addresses of them. They will connect to specified API by IAT (import address table) until running time. That means, we can revise the IAT to substitute some addresses of important functions with the addresses of ourselves.

Code 3.1
OS: windows XP professional SP2
Compiler: VC 6.0

```
SDLLHook ProcessHook =
{
    "Kernel32.DLL",
    false, NULL,     // Default hook disabled, NULL function pointer.
    {
    {"GetProcAddress", MyGetProcAddress},
    {"CreateProcessW", MyCreateProcessW},
    {"CreateProcessA", MyCreateProcessA},
    {NULL,NULL}
    }
};
```

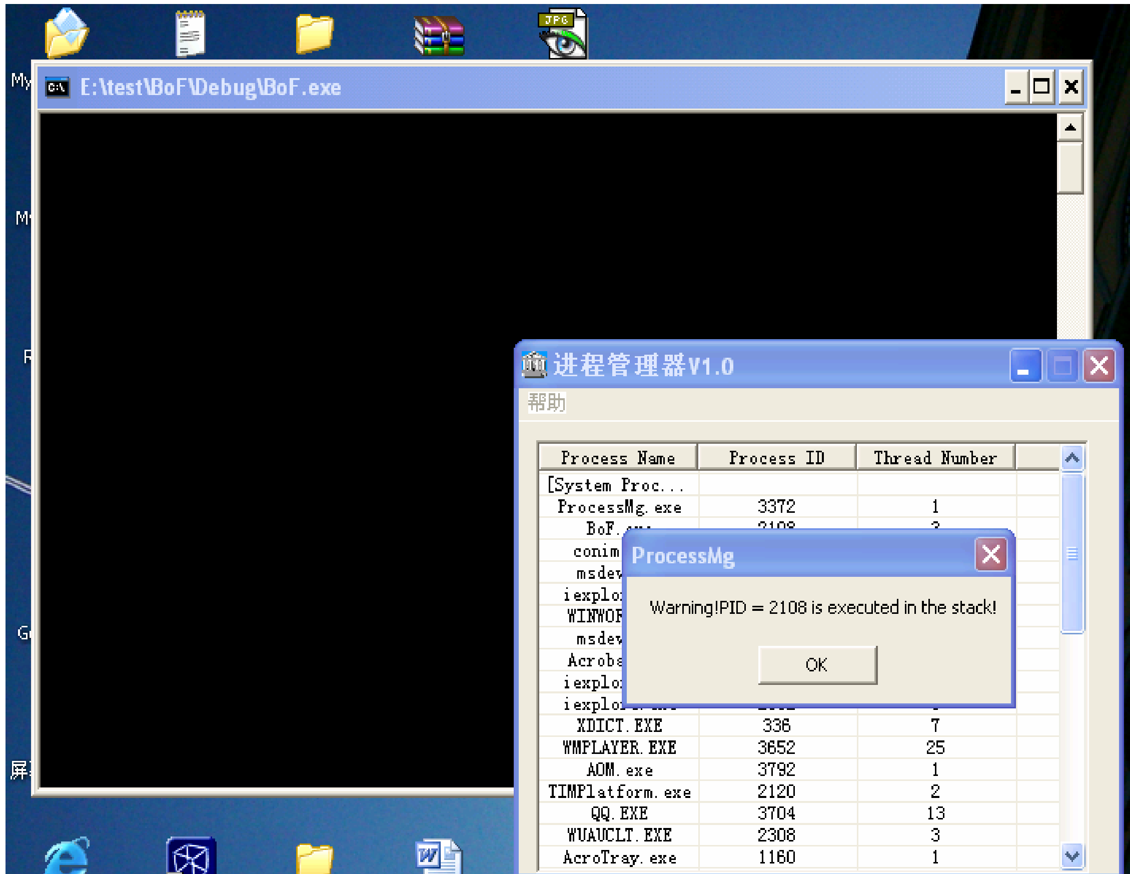
In code 3.1, I substituted GetProcAddress(),CreateProcessW (),CreateProcessA () with MyGetProcAddress(), MyCreateProcessW(), MyCreateProcessA(). After I startup the Hook API program, every program will call my MyCreateProcess() rather than CreateProcess() in Kernel32.DLL. So, I can easily get the register information from other programs.

In MyCreateProcess() function, we add some codes to detect and prevent executing code in a stack, and return CreateProcess() of Kernel32.DLL so that it will not affect normal programs.

Code 3.2
OS: windows XP professional SP2
Compiler: VC 6.0

```
    CONTEXT ct;
    ct.ContextFlags = CONTEXT_FULL;
    HANDLE hThread = GetCurrentThread();
    GetThreadContext(hThread, &ct);
    //Get return address
    ReadProcessMemory(m_hDebug, (void*)ct.Esp, &buf, sizeof(buf),
```

```
&dwRead);
pCallBackAdd = (void*)buf;
if((DWORD)pCallBackAdd < 0x00200000)
{
    MessageBox("Alarm: The program may be buffer overflowed");
    return 0;
}
return  CreateProcess(.....)
```



4 Anti-(Stack Protection)

4.1 User mode attacking

- Some unexperienced programmer maybe neglect that in windows NT system, many API have two versions: Unicode and ANSI. If we only hook ANSI version API, the attackers may call another version to bypass our detection. For example, for CreateProcess() function, we should capture not only CreateProcessA() but also CreateProcessW().
- The relationship between high level API library and low level API library is also ignored easily because many of low level API are undocumented. If the attacker is very familiar with kernel API, he can use low kernel API instead of user API, just like use NTDLL.DLL instead of kernel.dll. The result is our hook program failed to check his process.

4.2 Kernel mode attacking

4.2.1 Fragment shellcode

Attackers can divide their shellcode into different pieces and transfer from stack by dint of some codes in NTDLL.DLL.
This is some code in NTDLL.DLL

Code 4.1
OS: windows XP professional SP2
Source：NTDLL.DLL

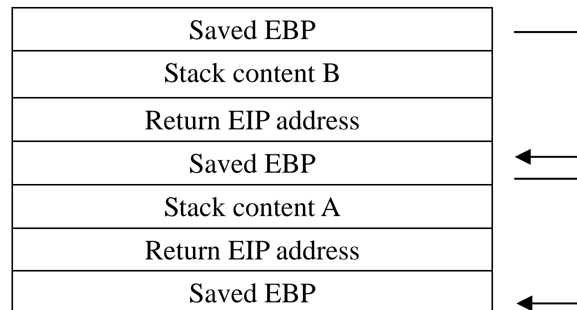
```
.78462FDF: AB stosd
.78462FE0: 5F pop edi
.78462FE1: C20400 retn 00004
...
.784635EC: 8BC6 mov eax,esi
.784635EE: 5F pop edi
.784635EF: 5E pop esi
.784635F0: C3 retn
```

Firstly, the attacker can smash the stack and revise the return address to 0x7863EC, and then the program will jump to 0x7863EC and execute the codes, furthermore, if the attack has already prepared two number pushed into the stack, these two number will be poped to EDI and ESI (see 0x784635EE and 0x784635EF). If the next value in the stack is an address, the program will jump to anther place to execute (see 0x784635F0). That will repeat several times to copy the shellcode

to another place and no need to execute shellcode in the stack. You will find it likes a springboard!

4.2.2 Forgery stack frames

Stack backtracing is a technique to ransack stack frames and find the return address. It is also called “return into libc” check. That is a good way, but this mechanism depends on the EBP value saved in the stack.



As the chart, the function of EBP is to let the stack point (ESP) find the way back. If the saved EBP is changed, it is difficult to check that a call or jmp actually points to the API being called. Most important, it is difficult to check return addresses beyond the last valid EBP frame pointer. It cannot stack backtrace any further).

5 Conclusion

In the current information security world, attacking and defending is a well balanced system. They develop equally, on the other side, they restrict each other. New techniques always appear one by one. It is really difficult to predict which side will win at last. That is an endless war.

6 Reference

1. Wikipedia. Buffer Overflow. See http://en.wikipedia.org/wiki/Buffer_overflow , access 11/2/2006
2. 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
3. Peter Szor. "The Art of Computer --Virus Research and Defense". Symantec Press. Addison-Wesley U.S. 1st edition, 2005
4. A Buffer Overflow Study-Attacks & Defenses Pierre-Alain FAYOLLE, Vincent GLAUME, 2002
5. Hook API source code, see http://www.codeproject.com/system/Paladin.asp, access 25/2/2006
6. JEFFREY RICHTER , "Programming Applications for Microsoft Windows Fourth Edition" Microsoft pub 2000