2012 Fourth International Conference on Multimedia Information Networking and Security

BUFFER OVERFLOW EXPLOIT AND DEFENSIVE TECHNIQUES

Desheng Fu
Network Monitoring Engineering Center of Jiangsu Province
School of Computer & Software Nanjing University of Information Science & Technology
Nanjing, China
dsfu@vip.sina.com

Feiyue Shi
Network Monitoring Engineering Center of Jiangsu Province
School of Computer & Software Nanjing University of Information Science & Technology
Nanjing, China
maomaocheer@21cn.com

***Abstract*—Buffer overflow attack is most common and dangerous attack method at present. So the analysis is useful in studying the principle of buffer overflow and buffer overflow exploits. In the paper a didactic example is included to illustrate one method of buffer overflow exploits, and though adding a jmp esp instruction into the process space as a springboard, it makes the shellcode successfully to be executed. Finally, an overview for protecting and defending against buffer overflow is summarized.**

Keywords-buffer overflow; stack frame; exploits; shellcode; defensive techniques

I. INTRODUCTION

Over the course of the past 20 years, the buffer overflow vulnerability has become one of the main forms of computer system security vulnerabilities, buffer overflow vulnerability attacks accounted for the vast majority of remote network attacks, such attacks can make the worm quickly spread, so that it becomes a very serious security threat. In the first five months of 2010, the National Vulnerability Database (http://nvd.nist.gov) recorded 176 buffer overflow vulnerabilities, of which 136 had a high severity rating. Buffer overflow remains a major security hole today, ranking third on the Common Weakness Enumeration/SANS list of Top 25 Most Dangerous Software Errors. [1]Therefore, it is extremely important to analyze the principle and buffer overflow exploits and provide effectively protection and defensive techniques.

II. PRINCIPLE OF BUFFER OVERFLOW

When program runs, the system will allocate a section of the adjacent area of memory to store various types of data; this memory space is called a buffer. In the process of a large buffer of data copies to a small buffer, because of lacking of attention to the border of small buffer, "explode" the smaller of the buffer, which washed away the other data that next to a small buffer adjacent memory area, so that it causes memory problems. Successfully exploited the buffer overflow vulnerability can modify the value of the variable in memory, or even hijack the process, execute malicious code, ultimately, access to control of the host. [2]

From the function stack frame and the process of the function call and return shows that the memory allocation of function's local variables takes place in the stack frame, if the buffer variable is defined in a function, the memory space occupied by this buffer variable would be in the established stack frame when this function is called. Due to the operations of buffer (such as string copy) are always executing from low memory address to high memory address, and in the meantime, in memory the return address of function call is above this buffer (high address) - this is confirmed by the stack's characteristics , so that it provides the conditions to overwrite the return address. If there is a large amounts of data pushed into buffer, possibly buffer overflow would occur, the return address of function's stack frame could be modified, the flow of execution of the program would be altered, and then the pre-prepared code would be executed.

III. BUFFER OVERFLOW EXPLOITS

Filling the buffer with data casually what causes buffer overflow usually only shows the Segmentation Fault error, and it could not achieve the purpose of attacking and taking advantage of the buffer. The common method of buffer overflow exploits is to make the program run a user Shell, then through the shell to execute other commands. And if the program belongs to Root and has Suid permission, then an attacker would gain access to Shell Root, so that he can do any operation to the system.

To achieve the purpose usually needs to complete the two works: one is arranging the appropriate code in the program's address space and another is though initialization registers and memory properly, the program could jump to the scheduled address space to execute. It is relatively simple to arrange the appropriate code in the program's address space. If the code which is used to attack has already existed in the program, just transferring some parameters to the code, and then the program jumps into the target location could be completed. [3]

To carry out an exploit, attackers must find suitable code to attack and make program control jump to that location with the required data in memory and registers. Attackers glean information about the vulnerable program code and its runtime behavior from the program's documentation and source code, by disassembling a binary file, or by running the program in a debugger. In principle, program space which is used in buffer overflow can be any space. But as the

978-0-7695-4852-4/12 $26.00 © 2012 IEEE
DOI 10.1109/MINES.2012.81

positioning difference a variety mode of transfer are created, including the function activation record, pointer subterfuge, and longjump buffers.

A. Function activation record

When program execution calls a function, stack frame is allocated with function arguments, return address, the previous frame pointer, saved registers, and local variables. In the stack frame, the return address points to the next instruction for execution after the current function returns. Attackers can overflow a buffer on the stack beyond its allocated memory and modify the return address to change program control to a location of their choice.

B. Pointer subterfuge

Pointer subterfuge involves modifying pointer values, such as function, data, or virtual pointers; they also can modify exception handlers. Attackers can use pointer subterfuge in overruns of stacks, heaps, or objects containing embedded function pointers. This kind of attack is especially effective when the program uses methods for preventing return address modification because it does not change the saved return address.

C. Longjump buffers

Another method for hijacking program control uses longjump buffers. The C standard library provides setjmp/longjmp to perform nonlocal jumps. Function setjmp saves the calling function's environment into the jmp_buf type variable (which is an array type) for later use by longjmp, which restores the environment from the most recent invocation of the setjmp call. An attacker can overflow the jmp_buf with the address of the attacker's code; when the program calls longjmp, it will jump to the attacker's code.

D. An example of buffer overflow exploits

If the buffer contains the code we want to execute, fill the buffer with correct machine code, overlay the return address with the buffer's starting address, and then using the return address to control the program jump to the system stack to execute the code we entered, that is an example of buffer overflow exploits. Here is an example; the key code is as follows.

```
int password (char *password){
  int authenticated;
  char buffer[44];
  authenticated=strcmp(password,PASSWORD);
  strcpy(buffer,password);
  return authenticated;
}
main(){
  char password[1024];
FILE * fp;
  LoadLibrary("user32.dll");
  fscanf(fp,"%s",password);
  valid_flag = password(password);
  fclose(fp);
}
```

The program reads the contents of the file "password.txt", and then comparing the copied contents with the defined password. Different then displaying the error, the same displays correct. implanting binary machine code to the password.txt file, then though this machine code to call the windows API function "MessageBoxA", eventually pop-up an "overflow" message box on the desktop. To achieve the above results, we must complete the following works: First of all, analyze and debug the overflow code, get the offset of the return address. Secondly, get the starting address of the buffer and write the corresponding offset into the password.txt to wash the return address. Finally, write the executable machine code to pop up a message box. According to the analysis of the stack structure, we could get the statecharts of stack frame as shown in Figure 1:

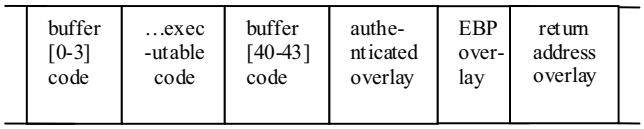

Figure 1. stack frame statecharts

If we write 44 characters into the password.txt, then the 45th hidden symbol "null" will wash away one low byte of authenticated, breaking through the limitations of password authentication. Dynamically debugging the program with OllyDbg, we could get the starting address of buffer "0x0012FB7C", write the address into password.txt from 53th to 56th, so that after the function return, the program would jump to execute the executable machine code.

To pop-up a message box you need to call the windows API function "MessageBoxA", the entrance of MessageBoxA address in memory is: 0x77D5058A in this experiment. The assembly code and instructions corresponding to the machine code as shown in Table 1:

TABLE 1 MACHINE CODE TABLE

Machine code(HEX)	Assembly instructions	Explanatory note
33 DB	XOR EBX,EBX	Push the string "overflow" into stack
53	PUSH EBX	
68 666C6F77	PUSH 776F6C66	
68 6F766572	PUSH 7265766F	
8B C4	MOV EAX,ESP	The value in EAX is string pointer
53	PUSH EBX	Push the four parameters into stack from right to left (0,overflow,overflow,0)
50	PUSH EAX	
50	PUSH EAX	
53	PUSH EBX	
B8 8A05D577	MOV EAX, 0x77D5058A	Call function "MessageBoxA"
FF D0	CALL EAX	

With hexadecimal editing tool UltraEdit writing the machine code into password.txt, filling bytes 53-56 with address "0x0012FB7C", and the rest of bytes with 0x90 (nop

instruction), the result of program running as shown in Figure 2:

Figure 2. result of program running

In practical exploits, however, because of the dynamic link libraries loading and uninstalling, the function of stack frame is likely to produce "shifting", so the method of overlaying return address does not possibly make the shellcode successfully to be executed. After analyzing the relationship between stack, register and code when overflowed, we could get the fact that when function returns, the ESP just points to the next position of the return address of stack frame. As the address of ESP register always points to the system stack and would not be disrupted by the overflowed data, so we could add a jmp esp instruction into the process space as a springboard. After filling the buffer with data, overlaying the return address with a jmp esp instruction, arranging the shellcode behind the function return address, so when the jmp esp instruction executed it will successfully jump into the shellcode.

In this experiment, we used a jmp esp instruction of user32.dll as a springboard, the address is 0x77D8408C. After successfully debugging in OllyDbg the shellcode as shown in Figure 3:

Figure 3. get shellcode

Fill the buffer with machine code first and then copy the above shellcode into password.txt, meanwhile, overlay the return address with 0x77D8408C. So the machine code in password.txt as shown Figure 4. Run the program, successfully pop-up the result of Figure 2.

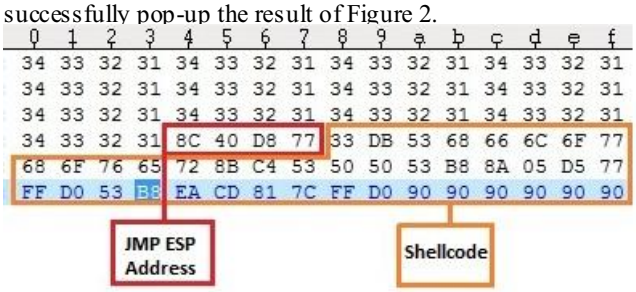

Figure 4. machine code in password.txt

IV. BUFFER OVERFLOW DEFENSIVE TECHNIQUES

Researchers have proposed various approaches to address buffer overflow problems, ranging from best practices in development to automated frameworks for recovering from attacks. The main four basic methods include writing secure code, performing bounds checking, runtime instrumentation, and static and dynamic code analysis. [4, 5]

A. Writing secure code

Writing secure code is the best solution for eliminating vulnerabilities, but at the same time it is very time-consuming and force- consuming. Programmers often pursue performance and ignore the correctness and security of the code. C and C++ provide no built-in protection for detecting out-of-bound memory accesses. Choosing programming languages like Java or environments like .NET that perform runtime bounds checking eliminates the problem. Standard C library functions including strcpy, strcat, and gets are unsafe because they do not perform bounds checking. Using safer versions of these functions such as strcpy_s and strcat_s is another good defensive coding practice. Both source and binary code analysis tools and network tools should be part of a programmer's arsenal for protecting against buffer-overflow attacks.

B. Performing bounds checking

Performing array bounds checking could prevent buffer overflow and exploit. As long as the array cannot be overflowed, the use of overflow attack will not happen. In order to achieve an array bounds checking, all the operations of reading and writing to the array should be checked to ensure that the operations are within the range of safety and right. The main methods of array bounds checking include compiler checking, memory access checking and using safety language. But it also decreases performance, programmers must employ heuristics to identify security-critical buffers and then apply bounds checking to those buffers.

C. Runtime instrumentation

Many runtime techniques for defending against attacks use return address modification to detect buffer overflows. Some proposals include obtaining information about buffer bounds estimates and instrumenting the code for runtime bounds checking.

Compile-time techniques like StackGuard [6] and Return Address Defender (RAD) [7] insert code to check for return address modification. StackGuard places a canary before the return address and checks the canary's value when the function returns. RAD creates a Return Address Repository global array and copies the return address to it in the function prologue. It then checks for modifications in the function epilogue. These approaches are not completely foolproof because attackers can alter the return address indirectly by using a pointer.

Solar Designer [6] and Pax [8] use a nonexecutable stack to combat buffer overflow. However, nonexecutable stack methods cannot defend against return-to-libc attacks and attacks on data segments; some instances also need an executable stack.

D. Static and dynamic code analysis

Static analysis of the program source code or disassembled binary code can identify buffer-overflow vulnerabilities. Although these techniques do not incur runtime overhead, they generate many false positives because they lack runtime information.

Other solutions use both static and dynamic analysis to detect buffer-overflow vulnerabilities. Researchers have proposed algorithms for selecting susceptible buffers, creating buffer overruns, and, based on the result, analyzing the application for susceptibility. This technique identifies locations that call unsafe library functions on local buffers and nonlibrary functions that read or copy user input. It then calculates the return address that attackers would overwrite to insert an attack string.

Microsoft has gradually increased protection to defend against buffer overflow vulnerabilities. Since XP SP2 Windows OS has already added the mechanism of PEB (Process Environment Block), SEH (Structure Exception Handling), VEH (Vector Exception Handling) and protection of cookie of the heap block structures technologies. The Windows 7 OS is not only inherited the protection of XP SP2, but also solves the shortage, has developed a set of comprehensive system for defending against buffer overflow. [9]

V. CONCLUSIONS

Buffer overflow attack is most common and dangerous attack method at present. It is extremely important to study buffer overflow for understanding the weaknesses in the system, mastering method of attack and protecting system security. This paper describes the principle of buffer overflow in section II, introduces buffer overflow exploits and analyses a buffer overflow instance in section III, finally, section IV provides an overview for protecting and defending against buffer overflow.

Although solutions and tools exist for flagging potential buffer overflow vulnerabilities, they are inadequate because of the wide scope of the problem and each approach's inherent limitations. Due to the diverse nature of the attacks, it is extremely difficult to prefabricate methods for defending against them. Therefore, we suggest exploring new methods that can defend against buffer exploits by acquiring knowledge from various sources. Such sources might include expert specifications and analysis of code involved in new attacks. Furthermore we suggest the integrated exploration of program analysis, pattern recognition, and data mining could be established.

REFERENCES

[1] http://cwe.mitre.org/top25/archive/2011/2011_cwe_sans_top25.pdf

[2] James C.Foster. Buffer Overflow Attacks-Detect, Exploit, Prevent [M]. TsingHua University Press, 2006,12.

[3] Jiang Tao. The Principle of Buffer Overflow and Protection [J]. Network and Computer Security, 2005, 6:56-59.

[4] Han Huiyan. The Exploits of Buffer Overflow in Windows OS [J]. Computer Development & Applications, 2008, 21(4):48-50.

[5] Wu Xueyang, Fan long, Chen Jingbo. The Research of Buffer Overflow in Windows OS [J]. Network Security Technology & Application, 2010, 12:71-74

[6] C. Cowan et al., "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade," Proc.Foundations Intrusion Tolerant Systems [Organically Assured and Survivable Information Systems] (OASIS 03), IEEE CS, 2003, pp. 227-237.

[7] J. Wilander ,M. Kamkar. A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Preven-tion. Proc. 10th Network and Distributed System Security Symp. (NDSS 03), Usenix, 2003, pp. 149-162.

[8] H. Ozdoganoglu et al., SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address. IEEE Trans. Computers, Oct. 2006, pp.1271-1285.

[9] Peng Jianshan, Wu Hao. Research of the Key Technology for the Windows Vista Memory Protection Mechanism [J]. Computer Engineering & Science, 2007, 29(12): 33-36.