EXPLOITATION OF SOFTWARE VULNERABILITIES, BASED ON BUFFER OVERFLOW ERRORS

Autor: Sveatoslav PERSIANOV
Coordonator: Mihail KULEV

Universitatea Tehnică a Moldovei

Abstract: *In present, there are a lot of companies which create software solutions, for different purposes, but only few of them are doing this well. A software application is a system of components, which are interconnected and working together, and as any complex system it has vulnerabilities (bugs). Most common vulnerabilities are based on buffer overflow errors. This paper describes the whole process of software exploitation, from identifying the bug to writing the exploit. Also it contains some techniques which will defend our applications from this type of errors.*

Cuvinte cheie: *Buffer overflow (BOF) errors, exploit, stack, processor's registers, exploitation.*

I. Introduction

Exploitation is the process of taking the computer's code or set of rules and change them so the computer does what you want it to do. It is one of the most fundamental concepts when talking about computers security.

Buffer overflow errors. Definition. A BOF occurs when a process or program tries to store more data into a temporary storage area (buffer) than it was intended to hold. Since buffers are created to store a finite amount of information, the extra information can overflow into adjacent buffers, overwriting existent data in them.

Types of buffer overflow errors. There two types of BOF errors: Stack based and Heap based.

Stack BOF errors, also known as stack smashing. The stack is a LIFO mechanism that computers use to pass arguments to functions as well as to refer to the local variables. Stack smashing occurs when a program writes to a memory address on the program's call stack outside of the intended data structure, usually a fixed length buffer. Main goal in exploitation of this type of errors is obtaining control of program's extended instruction pointer (EIP). This pointer contains next instruction address which should be executed by the processor.

Heap BOF errors. The heap is an area of memory utilized by an application and is allocated dynamically at the runtime. These errors are exploited in a different manner than stack-based overflows. Usually this area of memory contains program data. Exploitation is performed by computing this data in specific ways to cause the application to overwrite internal structures, such as linked lists pointers. The most danger consequence is that the overflow may result in data corruption or unexpected behavior by any process which uses the affected memory area.

II. Searching vulnerabilities. Buffer properties

At the beginning of the whole process we need to set up the work environment. We need a debugger (which will pop up when test application crashes and display registers state), fuzzing tools (used in black box testing), one of the scripting language interpreters (example: python, perl, etc.).

Figure 1. Typical crash error of Windows application.

Step 1. Testing. Software testing is an investigation conducted to gain information about the quality of the program or service under test. Test techniques include the process of executing a program or application with the intent of finding software bugs.

There are 2 testing methods: White-box and Black-box. These two approaches are used to describe the point of view that a tester takes when designing a test case.

Generally the differences between white- and black-box methods are that in first case tester has the information about internal structure of the application (it can be source code, diagram, flow charts, etc.). Black-box method is oriented in testing the application providing different types of inputs (fuzzing technique).

Step 2. Check overflowed buffers. During the testing period, tester obtains a list of errors. Now we need to check if that errors can be exploited or not. For example, we have an application which reads some information from a file and when that file has a size greater than value X, the application crashes. Debugger's window pop's up and we can see that the address value from Instruction Pointer is overwritten. That means, the error is exploitable. See picture below:

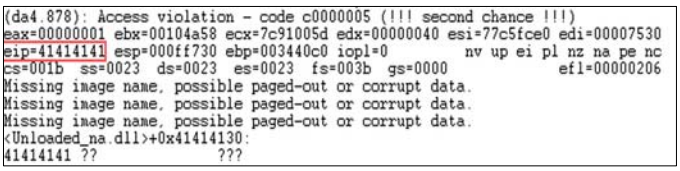

Figure 2. EIP register overwritten.

EIP value "41414141" is equivalent with "AAAA" in hex (input file of size X bytes was filled with As).

Step 3. Find the offset of the EIP. We have an input buffer, of size X bytes, which overwrites the application buffer, then EIP. In order to change the flow of the program, we need to put in EIP the first address of the memory area where we want to redirect it. To do this we need to determine the EIP register offset (or the size of overflowed buffer). This can be done using a unique pattern for input buffer and when the EIP overwriting occurs, the register will contain a unique sequence of bytes.

So, the sequence will be represented in inversed order (little-endian, there is also big-endian). Big-endian and little-endian are terms that describe the order in which a sequence of bytes are stored in computer memory. Big-endian is the order in which the "big end" (most significant value sequence) is stored first (at the lowest storage address). Little-endian is the order in which the "little end" (least significant value in the sequence) is stored first. For example, in a big-endian computer, the two bytes required for the hexadecimal number 3C0F would be stored as 3C0F in storage (3C is stored at address 1000, 0F will be at address 1001). In a little-endian system, it would be stored as 0F3C (0F at address 1000, 3C at 1001).

Now we have the necessary information about the vulnerable buffer.

III. Exploitation

Exploit. Definition. Types. Examples. An exploit is a piece of software, a chunk of data, or sequence of commands that takes advantage of a bug, glitch or vulnerability in order to cause unintended or unanticipated behavior to occur on computer software, hardware or something electronic. Such behavior frequently includes things as gaining control of a computer system or allowing privilege escalation or a denial-of-service attack. Generally exploits are categorized by three criteria:

- The type of vulnerability they exploit;
- Whether they need to be run on the same machine as the program that has the vulnerability (local exploits) or can be run on one machine to attack a program to attack a program running on another machine (remote exploit);
- The result of running the exploit (Elevation of Privileges, Denial of Service, Spoofing, etc.);

An exploit can execute different tasks, from running a program to opening a TCP/IP connection with a host on the Internet. Example below is a shellcode which executes the calc.exe application on Windows machine:

xdb¥xc0¥x31¥xc9¥xbf¥x7c¥x16¥x70¥xcc¥xd9¥x74¥x24¥xf4¥xb1¥x1e¥x58¥x31¥x78¥x18¥x83¥xe8¥xfc¥x03¥x78¥x68¥xf4¥x85
¥x30¥x78¥xbc¥x65¥xc9¥x78¥xb6¥x23¥xf5¥xf3¥xb4¥xae¥x7d¥x02¥xaa¥x3a¥x32¥x1c¥xbf¥x62¥xed¥x1d¥x54¥xd5¥x66¥x29¥x2

1¥xe7¥x96¥x60¥xf5¥x71¥xca¥x06¥x35¥xf5¥x14¥xc7¥x7c¥xfb¥x1b¥x05¥x6b¥xf0¥x27¥xdd¥x48¥xfd¥x22¥x38¥x1b¥xa2¥xe8¥xc3¥xf7¥x3b¥x7a¥xcf¥x4c¥x4f¥x23¥xd3¥x53¥xa4¥x57¥xf7¥xd8¥x3b¥x83¥x8e¥x83¥x1f¥x57¥x53¥x64¥x51¥xa1¥x33¥xcd¥xf5¥xc6¥xf5¥xc1¥x7e¥x98¥xf5¥xaa¥xf1¥x05¥xa8¥x26¥x99¥x3d¥x3b¥xc0¥xd9¥xfe¥x51¥x61¥xb6¥x0e¥x2f¥x85¥x19¥x87¥xb7¥x78¥x2f¥x59¥x90¥x7b¥xd7¥x05¥x7f¥xe8¥x7b¥xca

Introduction to shellcoding. A shellcode is a small piece of code used as a payload in the exploitation. It is called "shellcode" because it typically starts a command-shell from which the attacker can control the compromised machine, but any piece of code that is used as payload in exploitation is called shellcode. Usually it is written in machine code.

There are some differences between writing shellcodes for Linux and for Windows machines. In Linux we have direct way to interface with the kernel through the int 0x80 interface. Windows, on the other hand, does not have a direct kernel interface. The system must be interfaced by loading the address of the function that needs to be executed from a DLL (Dynamic Link Library).

In most cases, shellcode should be of a small size, because of the limited sizes of the overflowed buffers, so they usually are written in Assembly language.

Below is shown an application which puts a thread into sleep for 5 seconds.

```
;sleep.asm
[SECTION .text]

global _start
_start:
       xor eax,eax
       mov ebx, 0x77e61bea    ;address of Sleep
       mov ax, 5000           ;pause for 5000ms
       push eax
       call ebx                       ;Sleep(ms);
```

We load the address of function Sleep, 0x77e61bea. Then we need to compile the code:

```
nasm -f elf sleep.asm
ld -o sleep sleep.o; objdump -d sleep
```

After we run these commands we obtain the following output:

```
Disassembly of section .text:

08048080 <_start>:
8048080: 31 c0                xor %eax,%eax
8048082: bb ea 1b e6 77   mov $0x77e61bea,%ebx
8048087: 66 b8 88 13      mov $0x1388,%ax
804808b: 50                   push %eax
804808c: ff d3            call *%ebx
```

The shellcode will be: "¥x31¥xc0¥xbb¥xea¥x1b¥xea¥x77¥x66¥xb8¥x88¥x13¥x50¥xff¥xd3".

Executing the shellcode. At this step we have our shellcode and the ESP (Extended Stack Pointer) register which points to it. The reasoning behind the overwriting EIP with the address of ESP was that we want the application to jump to ESP and run the shellcode. Jumping to ESP is a very common thing in Windows applications. In fact, Windows applications use one or more dll's, and these dll's contains a lot of code instructions.

```
ModLoad: 7c900000 7c9b2000   C:\WINDOWS\system32\ntdll.dll
ModLoad: 7c800000 7c8f6000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 78050000 78120000   C:\WINDOWS\system32\WININET.dll
ModLoad: 77c10000 77c68000   C:\WINDOWS\system32\msvcrt.dll
ModLoad: 77f60000 77fd6000   C:\WINDOWS\system32\SHLWAPI.dll
ModLoad: 77dd0000 77e6b000   C:\WINDOWS\system32\ADVAPI32.dll
ModLoad: 77e70000 77f02000   C:\WINDOWS\system32\RPCRT4.dll
ModLoad: 77fe0000 77ff1000   C:\WINDOWS\system32\Secur32.dll
ModLoad: 77f10000 77f59000   C:\WINDOWS\system32\GDI32.dll
ModLoad: 7e410000 7e4a1000   C:\WINDOWS\system32\USER32.dll
```

Figure 3. Some system Dynamic Link Libraries.

Furthermore, the addresses used by these dll's are pretty static. So, we need to find a dll that contains the instruction to jump to the ESP, and if we could overwrite EIP with the address of that instruction in that dll, then our shellcode should be executed.

First of all we need the opcode of the "jmp esp" instruction, which is "ff e4". Now we need to search this opcode in libraries loaded by the target application (using the debugger attached to the application process).

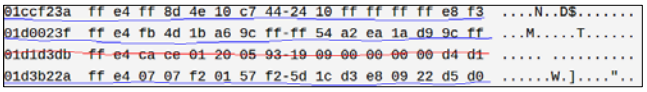

Figure 4. List of addresses of "jmp esp" instruction.

Third line can not be used, because it contains null bytes, which in our case will stop the execution of the program. First, second and fourth addresses are best choice.

So if we overwrite the EIP register with one of that three address, our shellcode will be executed (in this case execution of calc.exe application).

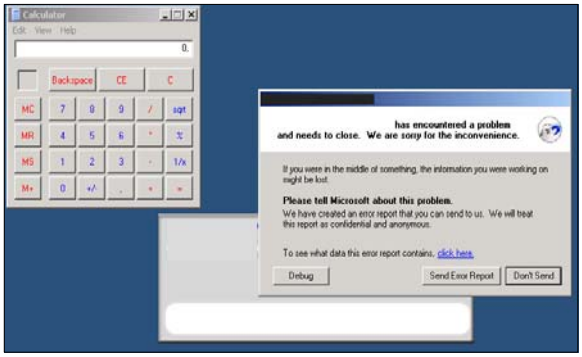

Figure 5. Shell code executed, while the application crashes.

Of course instead of opening an instance of calc.exe we can write a shellcode which will create a TCP/IP connection with some host on the Internet and return a shell. In this case we'll have much more possibilities to use the vulnerability.

IV. Conclusion

In this paper I presented the process of applications exploitation, which is pretty straight forward and easy to do. I think, software companies should pay much more attention to the security of their products, because as a result the users are exposed to attacks.

Nowadays we use a lot of applications which were written in C programming language and there are some functions which are exposed to buffer overflow errors, like: strcat(), strcpy(), sprintf(), vsptrinf(), bcopy(), gets() and scanf().

Also we need to use some tools like stackguard, Immunix and vulnerability scanners to secure our systems.

Big corporations like Google and Facebook are paying people for exploits in their software products (from 500$ → 60.000$). I think this fact shows the importance of writing and maintaining a secure application, service or system.

V. References

[1] CEH (Certified Ethical Hacker): http://www.eccouncil.org/courses/certified_ethical_hacker.aspx
[2] Glenford J. Myers, *"The Art of Software Testing"*, John Wiley & Sons, 2004 , 224 pages
[3] Jon Erikson, *"The Art of Exploitation" 2nd edition*, 2003, 241 pages
[4] Comptia Security+ : http://www.vtc.com/products/CompTIA-SecurityPlus-2011-Objectives-Tutorials.htm