Buffer Overflow Attacks on Linux Principles Analyzing and Protection

Zhimin Gu Jiandong Yao Jun Qin
Department of Computer Science, Beijing Institute of Technology (Beijing 100081)

Abstract

In order to attack and get the remote root privilege, using buffer overflow and *suid* program has become the commonly used method for hackers. In this paper, the attacking principles using buffer overflow on Linux have been analyzed, and the corresponding protection strategies have been put forwarded, too.

Keyword: buffer overflow, stack, EBP, libsafe

1 Introduction

With the increasing of accessing Internet, the network security problems have become the public's focus. Based on the safety report of CERT, buffer overflow attacks, which accounting for about half of all, have become the most common ones among all sorts of attacking methods. The C programming language has vulnerability, i.e., it does not automatically bounds-check array and pointer reference. By writing data whose length is more than the actual size of the buffer, the specific memory was modified. The attacking goal, changing the program executing sequence, was achieved. Buffer overflow attack exploits this very characteristic.

2 The Reasons about Buffer Overflow

The programs written in C often were plagued with buffer overflows. Two main reasons were described below: 1.The C does not do bounds check for array and pointer reference. 2. The standard C library, such as *strcpy*、*strcat*, is not safe [1].

When a C program runs on Linux, the mapping of the process in memory was divided into three parts: code segment、data segment and stack segment. The stack segment was used for allocating space for auto variables and saving the arguments and return address when function calling occurs. The function calling was realized through pushing data into stack or popping data out from stack. So, when you writing data whose length is more than the size of buffer to the destination, the neighboring place will be overwritten, resulting into buffer overflow. Through changing the overflowing memory address, the hacker can get the logon privileges of remote system.

If the exploited program has suit bit, the hacker will get the root privilege.

So, bound-checking, should be done by the programmer, actually often be omitted, which results into the security problems to the applications written in C.

3 The Principles about Buffer Overflow Attacks

3.1 C Function Calling

When function calling occurred, the saved data (including return address) on stack was called Stack Frame. A new stack frame was created for each function call. For the convenience of referring the local variables and arguments, many a CPU using a so called Frame Pointer (FP) or

local basic register (LB) to point to one specific place of a stack frame [2]. For the CPU of Intel, BP (EBP) is used for achieving this function. When referring to the local variables or arguments, only positive or negative offset to EBP was enough. Because the stack increasing to lower address, the positive and negative offset to EBP is needed to access local variables and arguments, respectively.

The function calling in C is compiled to call instruction. Its executing procedure follows up:

1. The function entry arguments were pushed onto stack by opposite order;
2. Push IP (Instruction Pointer) onto stack (done by call instruction), i.e., the return address — RET;
3. Push the previous EBP onto stack so that it can be restored when the function exits. Let current EBP points this address;
4. Push the local variables onto stack;
5. Execute the function;
6. Leave the function (return).

When the function returns, copy EBP to SP, pop the stack, restore the former EBP、IP and clear the stack. The calling and exiting procedures are listed as below:

Calling: *pushl %ebp*
movl %esp %ebp
Return: *movl %ebp %esp*
popl %ebp

ENTER and LEAVE instructions of Intel can well be used for this purpose — the context saving when function calling and the context restoring when function exiting.

3.2 Buffer Overflow

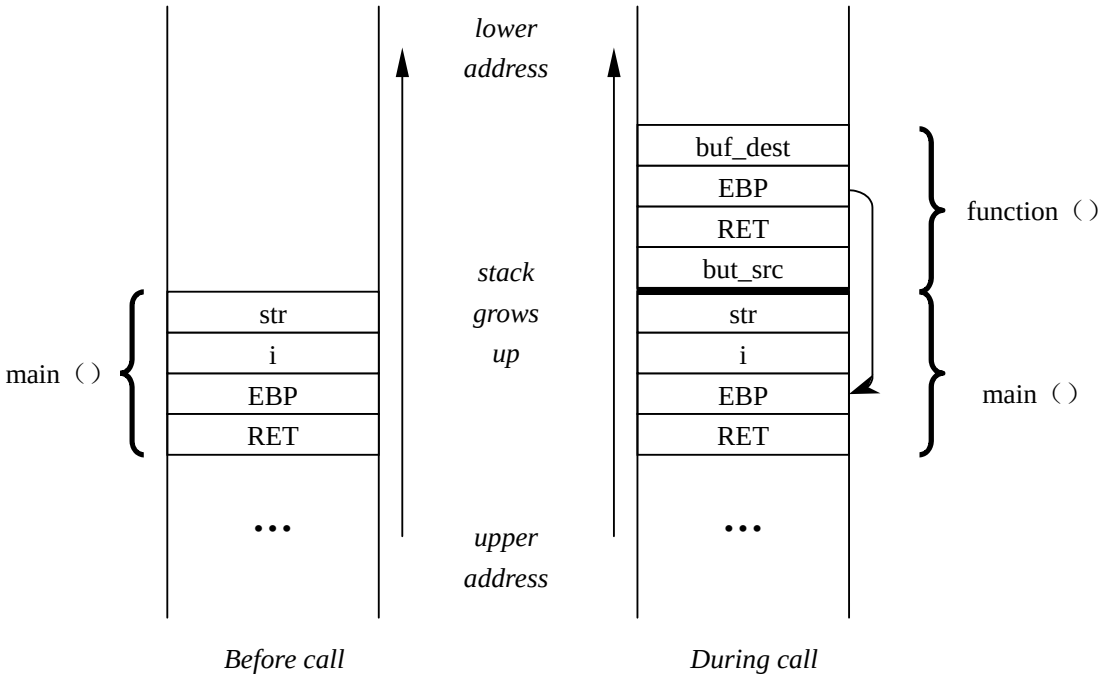
Now let's explain how buffer overflow can happen by one example program.

One example is listed as follow:

```
function(char *buf_src)
{
    char buf_dest[16];
    strcpy(buf_dest, buf_src);
}
/* main function */
main()
{
    int i;
    char str[256];
    for(i=0; i<256; i++) str[i] = 'a';
    function(str);
}
```

We can see obviously from the program that the size of array *str* (256 bytes) exceeds greatly the length of *buf_dest* (16 bytes). Buffer overflow occurred.

The stack using conditions before and during the function calling were showed in the following figure:



We can see from this figure that the content of array *str*(256-character ‘a’s, i.e., 0x616161…) has already overwritten all the former contents from *buf_dest* to buf_dest+256, including the EBP and RET saved when function calling. So when the function exits, it will return to 0x61616161, the segment fault occurred.

Buffer overflow can change the executing sequence of program. If we can write a starting address of an elaborate attack code to RET, the system can be hijacked. If the attacked program having the *suid* bit and the attacking code getting an interactive Shell, the hacker will get the root privilege. This is the main method used by hackers.

3.3 Shellcode

From the previous section, if you can give array *str* with specific content, you can let the program return to execute an special block of code so that you can attack the system. In order to get an interactive shell when the function returns, usually execute code: execve(“/bin/sh”, “/bin/sh”,NULL). This code, having something to do with the specific assembler and machine hardware, was called *shellcode*. For example:

```
char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";   [2]
```

This *shellcode* was acquired by disassembling. You can get one shell through overwriting only RET with the starting address of *shellcode*. The following program can achieve this function:

```
/* global variables */
char shellcode[] =      （see to the above discription）
char large_string[128];
/* main function */
void main() {
```

```
    char buffer[96];
    int i;
    long *long_ptr = (long *) large_string;
    for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer;        /* to stuff large_string with the beginning address */
                                           /* of buffer */
    for (i = 0; i < strlen(shellcode); i++)
    large_string[i] = shellcode[i];    /* put shellcode to the foremost part of arge_string*/
    strcpy(buffer，large_string);
}
```

In the above program, when returning from the function *strcpy*, its RET has already been modified as the beginning address of *large_string*, where *shellcode* lies. So *shellcode* will execute and an interactive shell will be obtained successfully. This was the result of executing *strcpy* without bound checking.

The above illustrates how to attack the program of our own. But in the reality, we cannot know the code of attacked program, even the address of buffer. How can we attack the unknown program? It is involving how to construct *large_string*. An effective method is to put *shellcode* at the middle of *large_string* and fill its former part with NOPs. And then set it to an environment variable. By executing the program with the buffer overflow vulnerability using this environment variable as an argument, you can get a shell. Filling the former part with NOPs is for increasing the probability of returning to *shellcode* when exiting. If the return address points to any NOP, *shellcode* will be execute eventually, and shell will be got.

4 The Protection of Buffer Overflow

The attacks using stack overflow listed above actually make use of the executable attribute of the stack, i.e., the codes in the stack can be executed. We can eliminate this trouble by making the stack non-executable. However, there are some fatal shortcomings of this method [1]:

1. In order to set the stack non-executable, everyone must make patch and recompile the kernel, but this is unpractical.
2. *Nested function calls* and *trampoline functions* can be executed successfully depending on the executable attribute of stack.
3. Setting the stack non-executable is unfit for another attack *return-into-libc* (pointing the program control flow to the shared library).

Libsafe 2.0, developed in Bell lab, can protect effectively the attacks by buffer overflow. The basic idea is: before calling the function with buffer overflow vulnerability, such as *strcpy*, intercepting this calling, check the distance from the destination memory address of writing to the places of RET and saved EBP in stack, comparing it with the length of writing content, if exceed, then overflow occurred, exit the function call.

Another method is avoiding the attacks using safe C library. For example, Richard Jones and Paul Kelly have developed *gcc/egcs* with bound checking, which can check most part of potential overflow problems [3]. For windows, we can use the bound checking program made by NuMega. Its function is same to *gcc* with bound checking [3].

5 Conclusions

The C programming language, without the bound checking, has resulting in many programs with the problems of buffer overflow vulnerabilities. Hackers mainly overwrite the function return address by overflowing the variables in stack, and then coping a block of attack code to attain their aim. Libsafe can protect this kind of attacks effectively. We can also using the safe C library to programming, to make the programs free of buffer overflow vulnerabilities.

References

[1] Arash Baratloo, Timothy Tsai, and Navjot Singh. Libsafe: Protecting Critical Element of Stacks. White Paper. December 25, 1999

[2] Aleph One. Smashing The Stack For Fun And Profit. www.smth.org. Oct 1997

[3] Matt Conover (a.k.a. Shok) & w00w00 Security . w00w00 on heap Overflows. http://www.w00w00.org/files/articles/heaptut.txt. January 1999