Scribe: Cryptography and Network Security (Class.7.1.B)

Chandan Kumar

18-Oct-2020

1 Topics Covered

1.1 System Security: Memory Integrity attacks and defences

1. What does memory integrity mean?
2. Control hijacking attacks
   (a) Stack smashing attacks/ buffer overflow
   (b) Integer overflow attacks
3. Defences

1.2 Memory integrity

Let us first understand the basic definition of integrity, memory integrity, and attacks related to memory integrity.

Integrity: This integrity is same as the integrity defined in confidentiality, integrity, and availability (CIA model). It means that system and data should not get altered by any unauthorized parties.

Memory integrity: The memory component of a computer like RAM, caches, etc. should not get altered by unauthorized parties. Generally, the memory component stores code, data, passwords, secrets, etc. whose unauthorised access/alteration may lead to huge financial and livelihood loss. As for example, the browser running in our system have codes and data executing in machine after getting loaded in RAM so an attacked must not be able to alter codes/data for browser loaded in RAM.

Memory integrity attack: Suppose an attacker is able to send a malicious packets to a target machine. The target machine's component like web server

is then taken over by him. He will then execute arbitrary code on target by hijacking an applications *control flow*. Now, he can change the execution order of instructions.

1.3 Control hijacking attacks:

There are two types of control hijacking attacks, 1. Stack smashing attacks/buffer overflow, and 2.Integer overflow attacks.

1.3.1 Stack smashing attacks/ buffer overflow

Ingredients for the attack: Understanding a memory integrity attack requires knowledge of c functions and stack memory access through codes.The *exec system call* which is used to execute a file residing in active process. How can an attacker exploit *exec syscall* to execute an arbitrary file? The knowledge of processor and firmware specifications help a lot to an attacker to launch an attack. Information about stack frame structure, stack growth direction, the little endian vs big endian specification about firmware.
Let us have a look at a basic c code and understand how it's implementation is vulnerable to memory integrity.

```
void foo(char *s){
        char buf[10];
        strcpy(buf, s);
        printf("buf is %s\n",s);
}


foo("kjhjhhdkjhddhjhj");
```

Looking at the above code snippet, it is very obvious that *sanity checking* has not been performed on the size of string s. Whenever we use *strcpy* function, the content of string s is directly copied into *buf* without taking into account the size of *buf*. So an attacker can always pass some arbitrary code through such function and compromise the complete system. The idea of *buffer overflow* has been utilised in the code snippet above by inputting a very large string. Sometimes, for a firmware with sloppy code, it becomes easy to crash the software by overflowing a buffer (SEGV typically). But sometimes it'e possible to actually make the server do whatever is intended by an attacker (instead of crashing).
Stack is a memory location which grows downward (from high addresses to low addresses) once a new function call has been made. Pushing something on the stack moves the top of stack towards address 0x00000000. For example in the given code snippet, the top of stack stores the *return address* of the foo function call, after that *parameters* passed to function are stored, then *stack frame pointer* is saved and finally the *local variables*.
The general idea of buffer overflow is built upon following three points:-

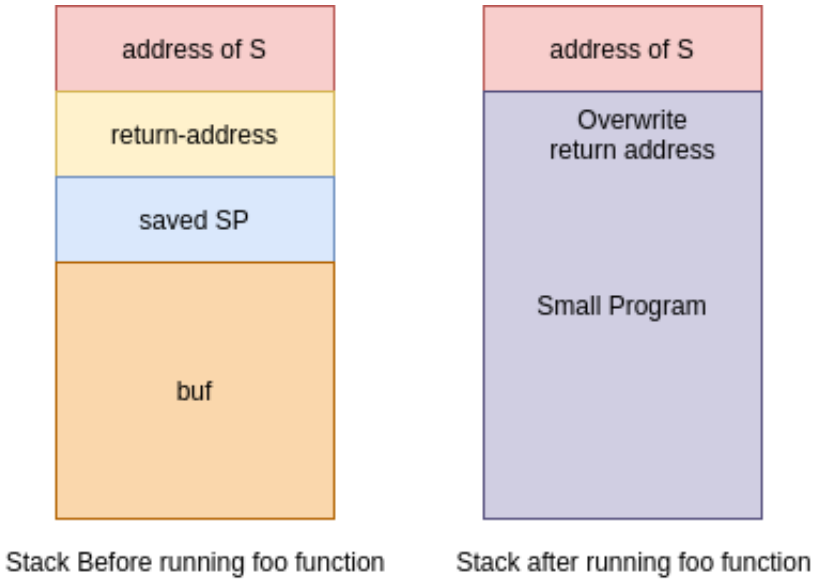

Figure 1: Figure showing buffer flow attack

1. Overflow buffer so that it overwrites the return address.
2. When the function has completed its execution, the control jump to whatever address is on the stack.
3. Put code in the buffer & set the return address to point to it.

It is evident from figure 1 that how execution of foo function has led to overwriting of return address which in turn helps an attacker to introduce a small malicious program whose execution might lead to severe damage of system as a whole.

1.3.2 Two major issues

Even after executing the code successfully, an attacker is still left with following two issues to implement buffer flow attack.

1. How do one know what value should the new "return address" be?
   (a) It's the address of the buffer, but how do we know the buffer on the stack?
2. How do we build the "small program" and put it in a long string ?
   (a) There can not be a null character.
   (b) Once OS encounters a null character, it stops execution.

1.3.3 Guessing addresses

Generally, an attacker need the source code of an app so he can estimate the address of both the buffer and and the return-address. If it is assumed that stack address starts from 0 then reverse engineering the closed source apps will work good. Secondly, an estimate is often regarded as good enough if an attacker can guess the million times and find out the exact address.

1.3.4 Building the small program

The most important functionality of this small program stuffed into the buffer is do *exec()*. An *exec()* is a functionality of an operating system that runs an executable file in the context of an already existing process, replacing the previous executables. So sometimes it helps alot in changing the password of database or other files.

1.3.5 Generating a String

An attacker can write an exec() function and implement the functionality which he is intended to perform and he will generate a machine code corresponding to that exec() function. Now he will copy down the individual byte values and build a string. The machine code of a simple exec() requires only less than 100 bytes so it is easier to incorporate this byte code inside the string.
Some important point that an attacker must take care:-
The small program should be position-independent and able to run at any memory. Secondly, it should be limited by code length so that both machine code as well as return address can be easily accommodated on the stack. In addition to this, the attacker must put a bunch of No-Operations (NOPs) instruction ahead of small program so that as long as the new return -address points to a NOP, it will okay but as soon as it executes the small program code, the action of execution will start.

2 Conclusion

This scibe briefly discusses the memory integrity, buffer overflow attack particurly in detail. The buffer overflow has been depicted through the a vulnerable c program. *strcpy* is the basic string copy function which has been exploited by the attacker to get his own small malicious program run on the victims system. Many other such c functions includes *strcat()*, *gets()*, *scanf()*, etc which is vulnerable unless proper flag are utilised while programming.

References

Class lectures