ASGN02

Matteo Di Fabio: 264339
Luca Genovese: 264364

1. Buffer Overflow

A buffer overflow condition occurs when a user or process tries to put more data than previously allocated into a buffer of a fixed size. This leads to an overflow which ends up overwriting portions of memory destined for other instructions. Normally a buffer overflow would lead to a segmentation error but can be exploited to allow hackers to execute arbitrary code or commands on a target system, often with root privileges. Such errors occur when accessing the limits of arrays or strings, when copying a string to a buffer that is too small or when a pointer points to an incorrect location. During execution, the functions of a program must store the data necessary for processing on the stack. However, it may happen that there is not enough space and that, if the function does not notice the error, the data is stored the same but overwriting and corrupting the stack. This activity related to buffer attacks is called smashing the stack and aims to overwrite the return address of the running program.

2. Secure programming

The best countermeasure for buffer overflow vulnerabilities is secure programming practices. Measures may be taken to minimize attack conditions of this type.
The concept of secure programming is generally defined in as:
”The practice of developing computer software so that it is protected from all kinds of vulnerabilities, attacks or anything that can cause harm to the software or the system using it.” .
Software security must be considered at all stages of software development: design, development, test and maintenance. Often this takes second place and the programs are written quickly to meet deadlines. Although it is impossible to write a bug-free program, it is easier and cheaper to capture design problems already in the design phase and fix them before they become security problems with serious consequences.
The key points to follow are:

- Learn from mistakes: when you find a security problem in software, you need to recognize and study it in order to understand how it occurred and how to prevent it in the future.
- Minimize the attack surface: limiting the modules, services and interfaces enabled to those required and used.
- Use defense in depth: take multiple countermeasures in a stratified or gradual manner to achieve security goals at every level of your application.
- Assume that external systems are insecure: consider any data received from a system that is out of your control as insecure and as a possible source of attack.

Threat modeling should be performed on all system components as changes are made, to understand the expected confidence limits and how to mitigate potential exploits.

Secure programming practices include:

- Do not trust the user: validate all input data from the user, including the verification of limits for each variable, especially for environment variables. The user’s input must always be sanitized and verified before being used, for example, by filtering or deleting special characters.
- Keep It Secure and Simple (KISS): simple is easier to protect. Keep your projects simple and you can demonstrate the security of a system. Unfortunately, some developers overestimate

design, finding themselves a complex colossus that is hard to understand and even harder to protect. Only when all parts of the application are clearly understood can they be revised to encode security defects and made safe.
- Use good cryptographic processes to mitigate damage if the worst happens. For example, using good algorithms known as AES-256 and encrypting data using keys not stored in the same place of data are just two of the basic rules to follow.
- Use more secure routines such as *fgets(), strncpy(), strncat()* and verify system call return codes.
- Reduce the executed code with root privileges.
- Apply all security patches provided by the manufacturer.

Probably the most common source of vulnerabilities is bugs that allow attackers to corrupt memory and use it to execute other code. Two well-known examples are:
- Toyota unintended acceleration cases: the occurrence of any degree of acceleration that the driver of the vehicle does not want to intentionally cause is only one of the manifestations of software errors that endanger safety. The bug originated from a stack overflow that may corrupt critical variables of the operating system.
- Heartbleed bug in OpenSSL: vulnerability in the Openssl encryption library that allows you to read the memory of systems protected by SSL/TLS encryption, used to exchange data securely. The emission of malformed inputs allows to read more data than should be permissible. This compromises the secret keys used, the names and passwords of users and their data.

3. Other countermeasures

Now that we know the threats that could alter the behavior of the firmware and allow the attacker to gain control of the system, we examine the possible countermeasures. Each of these has advantages and disadvantages in terms of cost and performance.
The best countermeasure is provided by secure programming techniques just discussed and a non-executable stack.
In addition, the programmer must perform tests and auditing of each program. To do this, you can use several tools available, such as using tools that automatically detect memory management bugs or enable compiler optimization.
GCC or other similar compilers provide optional options that alert developers when potentially dangerous *printf()* family implementations are detected during compilation and may detect illegal access to the stack.
These techniques are a good starting point for a defensive approach but are not enough to minimize the impact of this type of attack.
Other types of countermeasures available are:
- Non eXecutable stack
- Address Space Layout Randomization
- Stack canaries
- Control Flow Integrity

3.1 Non eXecutable stack

To counter code injection attacks, memory areas can be marked as non-executable and you can check the configuration of a given section with the readelf command. The use of the Non eXecutable stack (NX) allows to mitigate the code injection but does not affect other forms of Arbitrary Code Execution (ACE), such as return-to-libc, since the malicious code is not stored in the stack.

3.2 Address Space Layout Randomization

Address Space Layout Randomization (ASLR) is a technique that is based on randomization of address space and that involves the base address of an executable and the location of heap libraries and stacks in a process address space outlined randomly each time this is created. The hacker will be forced to guess or conduct a brute force attack on the key memory addresses. Depending on the size of the key space and the entropy level, it may prove to be an impossible task. However, like stack execution prevention checks, this technique is not infallible. You can bypass the ASLR with brute force attacks, repeating different addresses until success, or collecting useful information via ROP or JOP attacks.

3.3 Stack canaries

This method involves generating a random secret number that is pushed into the stack under the return address each time the program is executed. Before a function return, the canary stack integrity is checked against the original value. If no alteration is detected, execution resumes normally. If an alteration is detected, the execution of the program ends immediately. It seems an efficient mechanism to mitigate any stack smashing. However, there are two techniques that can be used to bypass them:

- Stack Canary Leaking: if an attacker can read the data in the canary stack, this value can be used to create a corrupted stack but that contains the right canary value so no alarm is generated.
- Bruteforcing attack: the canary value is determined when the program starts for the first time. If the program forks, it maintains the same values in child processes and this is used by the attacker to replicate the brute force attack on all child processes.

3.4 Control Flow Integrity

To check the correct execution of a program, the flow of the program is calculated before the runtime phase and, through additional instructions, the correct execution of the program is verified. The integrity is checked by comparing the jumps and return addresses of the functions with the licit ones present in the Control Flow Graph. It represents the best solution against arbitrary code execution but it is also very expensive and difficult to implement.

4. Examples

4.1 Corrupting function return address

This attack exploit the buffer overflow to execute any other function in our program!
Note: we will consider binaries in 32-bit architecture.
Let us consider the following code:

```
void highSecurityFunction() {
    printf("You have executed a function with high security level!");
}

void lowSecurityFunction() {
    char buffer[20];

    printf("Enter some text:\n");
    scanf("%s",buffer);
    printf("You entered: %s\n");
}

int main(int argc, char **argv)
{
    lowSecurityFunction();
    return 0;
}
```

Reading the presented code it is noticeable that the *lowSecurityFunction()* is accessible to standard users, instead, the *highSecurityFunction()* is not normally accessible. In fact, only *lowSecurityFunction()* is invoked. Due to the fact that the read string could exceed the buffer size a code

vulnerability is present. At this point, using buffer overflow vulnerability, it is possible to execute *highSecurityFunction()*.

We can disassemble the binary file to extract info with the *objdump* instruction, this allows us to collect information about our program such as:

- The address of *highSecurityFunction()*
- The amount of bytes reserved for local variables of *lowSecurityFunction()*
- The (relative) address of buffer

For this example we know that:

- 28 bytes have been reserved for buffer;
- buffer is allocated right next to %ebp (the Base pointer to main function)
- 4 bytes are used to store %ebp
- the next 4 bytes are used to store the return address

Knowing all these, now we can execute the function *highSecurityFunction()* using an input with:

- 32 bytes of any random characters
- 4 bytes with the address of highSecurityFunction()

This can be done with the following code:

python -c 'print("a"*32 + "\x8b\x84\x04\x08)' | ./return

In our case, output is as follows:

```
CC> python -c 'print("a"*32 + "\x8b\x84\x04\x08")' | ./return
Enter some text:
You entered: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa??
You have executed a function with high security level!
Segmentation fault (core dumped)
```

4.2 Code injection

Attackers exploit software flaws to introduce malicious code into a vulnerable computer program, to do that they inject some instructions exploiting one of the input sections and the program flow is redirected to them.

```c
#include <stdio.h>
#include <string.h>

void welcome(char *name)
{
    char buf[10];
    strcpy(buf, name);
    printf("Welcome %s\n", buf);
}

int main(int argc, char *argv[])
{
    welcome(argv[1]);
    return 0;
}
```

An example of code injection attack is shellcode injection. This kind of code injection aimed at executing a shell command, it is often used to attack remote servers. The injected payload is just a system call invoking the terminal (located at /bin/sh in Unix systems) that once obtained a shell, any command can be issued to the system, any rogue file can be created, any information can be easily stolen, etc.
Let us consider the following code.
This code contains a clear vulnerability: we have not any guarantee that *buf* is large enough to contain name, this may cause a buffer overflow.

To exploit this vulnerability an attacker can:

- overwrite value of register *eip* to refer to a memory area that he/she controls;
- fill that area with a shellcode, namely a set of assembly instructions that spawn a shell.

4.3 Code Reuse Attacks

Code-reuse attacks are software exploits in which an attacker directs control flow through existing code with a malicious result. Examples of such attacks are:

- Return-to-libc
- Return Oriented Programming (ROP)
- Jump Oriented Programming (JOP)

Our case study is return-to-lib, so called because one of functions in the C Standard Library (libc) is used. It is a technique that, by using a buffer overflow, replaces a return address with the one of another function in the process memory.
Let us consider again the following vulnerable code:

```c
#include <stdio.h>
#include <string.h>

void welcome(char *name)
{
    char buf[10];
    strcpy(buf, name);
    printf("Welcome %s\n", buf);
}

int main(int argc, char *argv[])
{
    welcome(argv[1]);
    return 0;
}
```

By performing a return-to-libc attack, we can execute a shell with the same access rights of the executed program.

The basic steps of the attack are:

- Find the address of system() libc function; → can be found using gdb properly.
- Find the address of string "/bin/sh"; → can be found using gdb properly.
- Corrupt the stack and let system() be called with parameter /bin/sh.

To perform an attack we have to pass to a vulnerable program the following inputs:

- A sequence of bytes long enough to trigger the buffer overflow.
- The address of function *system().*
- A sequence of 4 bytes for a return address.
- The address of function *exit()* can be used.
- The address of /bin/sh.