Workshop - Introduction to software exploitation

Aleknight

Setup

The tools we will use:

- gdb with pwndbg
- make
- A C compiler
- ropper / ROPgadget

We need to disable the ASLR, so for the that you need to run the following command:

```
echo '0' | sudo tee /proc/sys/kernel/randomize_va_space
```

To build the examples, you have to go in the folder *src/* and type the command

```
make
```

Program context

2.1 A general overview

During the program execution, we can find different memory regions with different purpose

- Code : contains data we want execute
- Data : contains information useful from the program execution (global variables, constants)
- Stack : contains information about the current state of the program (local variable, current function)

You can observe this different regions with pwndbg:

```
gdb ./program
b main
run
vmmap
```

This informations can be also found in the file */proc/<pid>/maps*

2.2 Some gdb tricks

- `&var` is used to get the address of the variable `var`
- `*addr` is used to get the value stored at the address `addr`

Function context

The system should store some information in order to help the execution of functions :

- A space to store local variable
- An address that refers to the current dedicated space for local variable
- What to do when the function ends

You can discover a bit more the function context if use the program `play`.

```
gdb -q ./src/play
pwndbg> break f
pwndbg> run
```

This will stop the program at the beginning of the function `f`.

A first exploitation - Integer overflow

In this part, we will play with the program `addition`.

0. Look at the source of this program in `src/addition.c` and try some inputs with the program.

1. Enter a number that will be interpreted as an another by the program. Explain what happend.

2. Enter two legal numbers and get a bad result for the addition. Explain what happend.

3. Find a way to be "admin" in the program `login` with the associated source `src/login.c` (You have to find an other way that using the password in source :-D). Explain how you do that.

Stack buffer overflow

As seen previously, the stack stores local variables such as array, but also the address of the next instruction to execute after the function. Now, we will try to discover how we can execute something else at the end of the function.

We will use the program `repeat` in this part.

0. Look at the source in `src/repeat.c` and play with the program.

1. Find an input such that `repeat` crash. What happend ?

2. Now, try to replay the `main` function of `repeat`.

5.1 Get a shell

Now, we want to be able to execute any program with `repeat`, for that we will try to have a shell. We will use a shellcode

3. `repeat` allows to execute some code in its stack, at which address we are writting when we do the call to the `gets` function.

5.1.1 Writing a shellcode

4. In order to get shell, we should do a call to the `execve` system call. Now you have to write some lines of assembly. If you are lazy, you can find it on the Internet

In order to provide the wanted system call, we need to have:

- rax = 0x3b

- rdi = the address of the start of '/bin/sh'
- rsi = 0
- rdx = 0

The reference of all instructions can be found here: https://www.felixcloutier.com/x86/

Returned Oriented Programming

Now, you can't use a shellcode to get a shell, so we will try to use an other method. So we will use the only part of program that is executable: the original code of the program. We will use the fact that the `ret` instruction just `pop` an address from the stack and jump to this one. As this is the end, I let you search a bit :-D