UP23 Lab04

Date: 2023-03-27

Simply Guess the Number

Remember we mentioned that `fork` copies almost everything from a parent process to its child process? This lab aims to understand how fork and stack frame works. You will need to use `gdb` to run and debug the challenge and read assembly codes. Your mission is to guess the number generated by our challenge server.

> Please read the instructions carefully before you implement this lab. You may solve the challenge locally on Apple chip-based machines, but the files you submit to the challenge server must be compiled for x86_64 architecture.

The Challenge Server

The challenge server can be accessed using the `nc` command:

```
nc up23.zoolab.org 10816
```

Upon connecting to the challenge server, you must first solve the Proof-of-Work challenge (ref: pow-solver (https://md.zoolab.org/s/EHSmQ0szV)). Then you can follow the instructions to submit (1) the shellcode to run on the server and (2) the magic payloads to solve the challenge. Since we have not officially introduced assembly language, we provide a simpler approach for retrieving binary codes directly from a compiled executable. See the instructions for more details.

Lab Instructions

Here are some hints for you. You can solve the challenge locally first and then verify your solution on the challenge server.

1. The binary and the source code of the challenge are available here - `remoteguess` (binary (https://up23.zoolab.org/up23/lab04/remoteguess) | view (https://up23.zoolab.org/code.html?file=up23/lab04/remoteguess.c) | download (https://up23.zoolab.org/up23/lab04/remoteguess.c)). Your mission is to ask the remote challenge server to print out `** Good Job!` and a `FLAG` stored on the server. Read the codes and think about how to solve them.

2. The `remoteguess` program optionally reads machine codes received from you. The machine codes are then executed in a sandboxed child process. We call the machine codes the shellcode.

3. The shellcode is called with a single parameter, which is the function pointer of the `printf` function. You can implement the shellcode in assembly. However, since we have not *officially* introduced assembly, you may use our sample code (view (https://up23.zoolab.org/code.html?file=up23/lab04/solver_sample.c) | download (https://up23.zoolab.org/up23/lab04/solver_sample.c)) to implement your codes in C and compile it to generate machine codes.

   Note that your C-based shellcode can only call the `fptr` function. No other function can be used in the shellcode.

4. To simplify the C-based shellcode submission process, you can use our provided `pwntools` python script to solve the pow and submit your shellcode. The submission script is available here (view (https://up23.zoolab.org/code.html?file=up23/lab04/submit.py) | download (https://up23.zoolab.org/up23/lab04/submit.py)). You have to place the `pow.py` file in the same directory and invoke the script by passing the path of your compiled solver executable as the first parameter to the submission script. Please do not strip the compiled executable because the submission script submits the entire executable to the server and tells the server to invoke the shellcode from the address of the `solver()` function.

   You can modify the sample codes and the submission script to fit your needs.

5. You may notice that there is a buffer overflow vulnerability implemented in the `guess()` function of the challenge. You can start by running and observing the behavior of this challenge. To debug at the assembly level, you may install the gef (https://github.com/hugsy/gef)

gdb extension. To dump assembly codes from the challenge or your solver, use the commands:

```
objdump -D remoteguess -M intel > remoteguess.s
objdump -D solver -M intel > solver.s
```

The codes will be stored in `remoteguess.s` and `solver.s` , respectively.

6. Each function call constructs a stack frame, which looks similar to the following structure (for x86_64 programs):

```
       |  .....                            |
       |  (previous func's stack frame)    |
       +-----------------------------------+
       |  additional parameters (if any)   | (higher address)
       |  .....                            |
       +-----------------------------------+
       |  return address (64-bit)          |
       +-----------------------------------+
rbp -> |  (pushed) rbp (64-bit)            |
       +-----------------------------------+
       |  canary (64-bit, if required)     |
       +-----------------------------------+
       |  local variables (if any)         |
rsp -> |  .....                            | (lower address)
       +-----------------------------------+
```

Additional Notes for Apple Chip Users

If you do not have a working x86_64 machine, you can still solve this challenge on an Apple chip Mac locally. Here are some hints for solving this challenge on Apple chip Macs.

1. You have to work in a Linux docker.
2. You must work with the x86_64 binary downloaded from this lab — Please ***do NOT*** compile the source codes into `aarch64` executables.
3. Please install `qemu-user-static` package, which allows you to run x86_64 executables on your Apple chip Macs.
4. The `sandbox()` feature may be incompatible with executables invoked with `qemu-user-static` package. Therefore, you can pass an additional `NO_SANDBOX` environment variable when invoking the `remoteguess` challenge.

```
NO_SANDBOX=1 ./remoteguess
-- or --
qemu-x86_64-static -E NO_SANDBOX=1 ./remoteguess
```

5. The solver program must be compiled with a x86_64 compiler. To do this, install the two additional packages `gcc-multilib-x86-64-linux-gnu` and `g++-multilib-x86-64-linux-gnu`, and replace the `gcc` (or `g++`) command with `x86_64-linux-gnu-gcc` (or `x86_64-linux-gnu-g++`). Sample commands for installing the packages and compiling `solver.c` is given below.

   ```
   apt install gcc-multilib-x86-64-linux-gnu g++-multilib-x86-64-linux-gnu
   x86_64-linux-gnu-gcc solver.c -o solver
   ```

Advanced Notes for Apple Chip Users

This section is only for students who want to solve this challenge natively on Apple chip Macs. The information provided here ***only works locally*** on Apple chip Macs. It is ***totally incompatible*** with our challenge server.

1. You can compile both the `remoteguess.c` program and your solver program `solver.c` natively on an Apple chip Mac (`aarch64` architecture), and then solve the challenge locally on a Mac.
2. The concept of solving this challenge on an `aarch64` machine is the same as what you have done on an `x86_64` machine. However, the calling conventions on the two architectures are different. Therefore, you may have to revise your solution to fit on `aarch64` architecture.
3. One critical difference on `aarch64` is that the `canary` cannot be easily obtained in your submitted shellcode. You may disable the stack protector feature on your compiled code to prevent your submitted shellcode from crashing.

   ```
   gcc solver.c -o solver -fno-stack-protector
   ```

4. Good luck and have fun!

Extra Notes for CTF Players

You may use return-oriented programming (ROP) (https://en.wikipedia.org/wiki/Return-oriented_programming) gadgets to get the shell from this challenge. But please don't break our system! No extra points even if you get the shell. 😆

Grading

- [87 pts] Your C-based solver works with the challenge server. That is, submitting your solver executable to the remote server can output both the `** Good Job!` message and the correct `FLAG` .
- [13 pts] You can implement a short shellcode directly in x86_64 assembly (<20 lines, each line contains only a single instruction). That is, embedding your assembly codes using the `asm()` feature of pwntools without loading an external solver executable.

> We have an execution time limit for your challenge. You have to solve the challenge within 60s.