UP24 Lab03

Date: 2024-03-25

GOT Maze Challenge

This lab aims to play with LD_PRELOAD and GOT table. Your mission is to ask our challenge server to solve the maze, i.e., walk from the start position to the end position.

> Please read the instructions carefully before you implement this lab. You may implement the codes to solve the challenge on an Apple chip-based machine, 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 up.zoolab.org 10385
```

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 upload your ***solver*** implementation, which must be compiled as a ***shared object*** (`.so`) file. Our challenge server will use `LD_PRELOAD` to load your uploaded solver along with the challenge. Therefore, the behavior of the challenge can be controlled by your solver.

Suppose your solver is named `libsolver.so` . Once your solver has been uploaded to the server, it will run your solver in a clean Linux runtime environment using the following command.

```
LD_PRELOAD=/libsolver.so /maze
```

To simplify the uploading process, you can use our provided `pwntools` python script to solve the pow and upload your solver binary executable. The upload script is available here (view (https://up.zoolab.org/code.html?file=unixprog/lab03/submit.py) | download (https://up.zoolab.org/unixprog/lab03/submit.py)). You have to place the `pow.py` file in the same directory and invoke the script by passing the path of your solver as the first parameter to the submission script.

Lab Instructions

We provide a number of hints for you to solve the challenge. The directions for this lab are listed below. You may download all the relevant distfiles in a single package from here (https://up.zoolab.org/unixprog/lab03_dist.tbz).

1. A shared library `/libmaze.so` is available on the challenge server. You may read libmaze.h (https://up.zoolab.org/code.html?file=unixprog/lab03/libmaze.h) first to see what functions and features are available in the library. A simplified source code of `/libmaze.so` is also available here (libmaze_dummy.c) (https://up.zoolab.org/unixprog/lab03/libmaze_dummy.c) (view (https://up.zoolab.org/code.html?file=unixprog/lab03/libmaze_dummy.c)).

2. Note that we did not provide the compiled shared library file for you. However, you can call the functions in the library by locating the function addresses in the library using the dlopen(3) (https://man7.org/linux/man-pages/man3/dlopen.3.html) and dlsym(3) (https://man7.org/linux/man-pages/man3/dlsym.3.html) functions. Note that you *cannot call `move_*` functions directly in your solver when it is running on the remote challenge server*, but it's OK to do that if you solve the challenge in your local machine (for testing purposes). Also, note that the two functions (`dlopen` and `dlsym`) only work for functions exported from a shared object.

3. The source code of the challenge is available here - `maze` (maze.c (https://up.zoolab.org/code.html?file=unixprog/lab03/maze.c) and moves.c (https://up.zoolab.org/code.html?file=unixprog/lab03/moves.c)). The main program (maze.c) registers the address of its main function, initializes the library, and loads an existing maze from `/maze.txt` . It then calls `move_NNN` functions in a fixed order to perform random walks in the maze. Obviously, the random walk process cannot solve the maze.

4. A sample `/maze.txt` is as follows. You may load and parse it by yourself or reuse the library functions to load it for you. The content of the file will be different every time you connect to the challenge server.

```
7 5
1 1
5 3
1 1 1 1 1 1 1
1 0 0 0 0 0 1
1 0 0 0 0 0 1
1 0 0 0 0 0 1
1 1 1 1 1 1 1
```

In the example, the maze has a dimension of `7x5` (width x height), a start position at `(x=1, y=1)`, and an end position at `(x=5, y=3)`, followed by the content of the maze. To walk to the end position, you may need to walk toward right four times and walk downward two times.

5. To solve the maze correctly, you should control the main program to call the correct movement functions, e.g., `move_up`, `move_down`, `move_left`, or `move_right`, instead of calling the random movement functions `move_NNN`.

6. It is intuitively that the preloaded solver may hijack some functions to solve this challenge. For example, you can implement `maze_init` function in your solver and let it perform anything before or after you call the actual `maze_init` function.

7. Since the `move_NNN` functions are all implemented in the shared library, it is feasible that you can hijack the function calls from the `main` function to the `move_NNN` functions by modifying the GOT table of the corresponding functions. For example, making function calls to `move_1`, `move_2`, and `move_3` can be altered and become calling `move_right`, `move_down`, and `move_right`, respectively.

   > Note: You are not allowed to hijack `move_*` functions using `LD_PRELOAD` on the challenge server. Please hijack it using the GOT table.

8. Locating the *runtime* address of the GOT table in a running process could be tricky. But since we have provided a special function `maze_get_ptr`, you can obtain the real address of the `main` function in runtime. We also provide the binary file of the `maze` (https://up.zoolab.org/unixprog/lab03/maze) executable. You should be able to find the relative address of the `main` function and each `GOT` table entry from the binary. The relative addresses can be retrieved by `pwntools` using the script.

```
from pwn import *
elf = ELF('./maze')
print("main =", hex(elf.symbols['main']))
print("{:<12s} {:<10s} {:<10s}".format("Func", "GOT Offset", "Symbol Offs
for s in [ f"move_{i}" for i in range(1200)]:
   if s in elf.got:
      print("{:<12s} {:<10x} {:<10x}".format(s, elf.got[s], elf.symbols[s
```

Once you have the addresses, you can ***calculate*** the actual addresses of GOT table entries based on the runtime address of the `main` function. One sample snapshot is shown below. Given that the relative address of the `main` function is `0x1b7a9` and the GOT offset of the `move_1` function is `0x231b0` . Suppose the real address of the `main` function is at `0x55f6edc857a9` . The actual address of the GOT entry for `move_1` can be obtained by `0x55f6edc857a9 - 0x1b7a9 + 0x231b0` .

```
main = 0x1b7a9
Func         GOT Offset Symbol Offset
move_1       231b0      19a74
move_2       21ea8      17464
```

9. If you have pwntools installed, you can use the command `checksec` to inspect the `maze` program. The output should be

```
Arch:     amd64-64-little
RELRO:    Full RELRO
Stack:    No canary found
NX:       NX enabled
PIE:      PIE enabled
```

Note the `Full RELRO` message, which means that the address of movement functions will be resolved upon the execution of the challenge. Therefore, your solver may have to make the region *writable* by using the mprotect(2) (https://man7.org/linux/man-pages/man2/mprotect.2.html) function before you modify the values in the GOT table. Note that the address passed to the mprotect function may need to be multiples of 4KB (page size).

Additional Notes for Apple Chip Users

If you do not have a working x86_64 machine, you can still solve this challenge. However, you have to work in a Linux docker to perform cross-compilation. You may consider using the `crossbuild` docker images mentioned in Lab02 Pre-Lab Announcement (https://md.zoolab.org/6H8ogpJHTjKI9BApnhNygA). The quick start command is pasted below for your

reference.

```
docker run -it --rm --user "$UID:$GID" -v "`pwd`:/build" -w /build -e PS1="bu
```

To compile your solver implementation for x86_64 machines, 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 `libsolver.c` is given below.

```
apt install gcc-multilib-x86-64-linux-gnu g++-multilib-x86-64-linux-gnu
x86_64-linux-gnu-gcc -o libsolver.so -shared -fPIC libsolver.c
```

Grading

1. [10 pts] Write a `Makefile` to compile, link, and generate `libmaze.so` (from `libmaze_dummy.c`) and `maze` (from `maze.c`). You may simply start by unpacking the `lab03_dist.tbz` file. Simply run the `make` command in the working directory, and it should produce the two required files.

2. [*10+10 pts*] (Part A - 10pts) Implement a solver that can solve the challenge, i.e., walk from the start to the end position, in your **local** machine. You can work with the `libmaze.so` and `maze` file generated from the previous grading item.

   You must use the following maze to test your solver. Place the content of the maze in `/maze.txt` and run the testcase:

```
15 15
1 1
13 13
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1 1 0 1
1 0 1 0 0 0 0 0 0 0 0 0 1 0 1
1 0 1 0 1 0 1 1 1 1 1 1 1 0 1
1 0 0 0 1 0 0 0 0 0 1 0 0 0 1
1 0 1 1 1 1 1 1 1 0 1 0 1 1 1
1 0 0 0 0 0 1 0 1 0 0 0 1 0 1
1 1 1 1 1 0 1 0 1 1 1 1 1 0 1
1 0 0 0 1 0 1 0 0 0 1 0 0 0 1
1 1 1 0 1 0 1 0 1 0 1 1 1 0 1
1 0 0 0 1 0 0 0 1 0 1 0 0 0 1
1 0 1 1 1 1 1 1 1 0 1 0 1 0 1
1 0 0 0 0 0 0 0 0 0 0 0 1 0 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

(Part B - 10pts) Similar to the Part A of this scoring item, but we have to enforce the constraints requested in this lab.

- You cannot modify `maze.c` and `libmaze_dummy.c` - as those are not allowed on the server.
- Your solver must be implemented as a shared object and preloaded using `LD_PRELOAD` - the definition of *solver* defined in the beginning of this lab.
- Your solver can only call the `move_*` functions to walk in the maze, or modify the GOT table - the same requirement as running on the server.
- You cannot print out the `Bingo` message by yourself - the same requirement as running on the server.

The above grading items can be done on your own desktop/laptop. It doesn't matter if you are working on either an Intel or Apple chip-based machine.

3. [10 pts] You can produce an `x86-64` solver shared object and submit it to our challenge server. The shared object should print out a message `UP112_GOT_MAZE_CHALLENGE` on the server.
   - This must be implemented in your solver C codes. You have to upload the compiled shared object to the server.
4. [10 pts] Use the pwntool scripts to retrieve the GOT addresses of the `move_*` functions from our provided `maze` executable.

    - This is done in your local desktop / laptop.

5. [20 pts] Your solver can obtain the main function address via the `maze_get_ptr` function. Once you get the main function address, print it out in the form of `SOLVER: _main = <the-address-you-obtained>` .

    - This must be implemented in your solver C codes. You have to upload the compiled shared object to the server.

6. [30 pts] Your solver can solve the maze on the remote challenge server. A few shell commands will be printed out from the challenge server once you have solved the challenge successfully. Run the shell commands and you should get a `Signature Verified Successfully` message from your console.

   The public key displayed in the shell commands should be

```
-----BEGIN PUBLIC KEY-----
MCowBQYDK2VwAyEAXmNQRmUKoJVMEBz2vhqqtoFsh/iMOroPZagFl9ia6IU=
-----END PUBLIC KEY-----
```

You have to ensure your working environment has `openssl` installed.

We have an execution time limit for your challenge. You have to solve the challenge within about 90s.