UP23 Lab07

Date: 2023-05-08

One More Assembly Challenge: ROP Shell

This lab aims to practice writing ***more*** assembly codes. Your mission is to read FLAGs from various sources on the challenge server using the assembly language. However, before you can implement regular assembly codes, you have to bypass the constraints given by the server using ***return-oriented programming*** (ROP).

You have to access the FLAGs using different approaches on the challenge server. The three FLAGs can be obtained by:

1. Reading it from the `/FLAG` file.
2. Attaching to a shared memory of key `0x1337` and reading the first few bytes from memory. The shared memory is filled with zeros for unused spaces.
3. Connecting to a server running at `localhost:0x1337` and receiving the data from the server.

The Challenge Server

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

```
nc up23.zoolab.org 10494
```

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 from the server to solve the challenge. The server binary (download (https://up23.zoolab.org/up23/lab07/ropshell)) and source code (view (https://up23.zoolab.org/code.html?file=up23/lab07/ropshell.c) | download (https://up23.zoolab.org/up23/lab07/ropshell.c)) are available for your reference.

The server receives payloads from your solvers and runs the logic implemented on the server. After a received payload has been invoked, the server reports the termination status (`normal` , `signaled` , or `unknown` status) of the received payload. Read the codes carefully and find out how to solve the challenge!

Lab Hints

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

1. The content placed in the `code` array is predictable random bytes. This is because the bytes are generated using the pseudo-random number generator with a *known* timestamp-based seed.
2. If you plan to call libc functions from python, you may have a look at the `ctypes` package. If can be used to load libc in python using the following sample codes:

   ```
   import ctypes
   libc = ctypes.CDLL('libc.so.6')
   ```

3. The address of the `code` array is also available. Note that once the `code` array is filled with random bytes, the *write permission* is disabled.
4. In the default setting, you can only fill the content of the stack and see how it can change the behavior or settings of the process. The major challenge is: *how do you allow more code submissions to the challenge server and read the FLAGs?*

   > You may try to put your codes into the `code` array. It would much simply your implementation. However, it requires some tricks, and you have to think about how to do it.

5. Since the stack is writable but not executable, you can write ***return addresses*** and ***parameter values*** to the stack to manipulate the program workflow to achieve your goal. This is what we call ***return-oriented programming*** (ROP). Suppose you plan to invoke the `exit(0)` system call, the implementation in assembly would look like the following.

```
mov rax, 60
mov rdi, 0
syscall
```

You should set the values for register **rax** and **rdi**, and then invoke the **syscall** instruction. To achieve the goal using ROP, you can find the **gadgets** from the available program codes, fill in the required values in the stack, and then start invoking the gadgets using the `ret` instruction. A gadget is a few assemblyh instructions (usually `pop` instructions) followed by a `ret` instruction at the end. For example, suppose the gadgets are available at the following addresses:

```
                ...
0x7fa44d52511b: pop rax
                ret
                ...
0x7fa44d533631: pop rdi
                ret
                ...
0x7fa44d551050: syscall
                ret
                ...
```

You can fill the stack with the values:

```
       +------------------+
       |             ...  | (higher address)
       +------------------+
       |  0x7fa44d551050  |
       +------------------+
       |               0  |
       +------------------+
       |  0x7fa44d533631  |
       +------------------+
       |              60  |
       +------------------+
rsp -> |  0x7fa44d52511b  |
       +------------------+
       |             ...  | (lower address)
       +------------------+
```

When a `ret` instruction is invoked, it pops and jumps to the address `0x7fa44d52511b` , which pops values (60) into **rax** and jumps to the next address on the stack. By following the jumps (returns) between gadgets, you can finally achieve the goal of calling exit(0) system call.

6. To have more flexibilities on memory space usage and shellcode execution, you can use `mprotect` syscall to change the permission of known memory areas.
7. You may implement a two-stage shellcode. The first-stage shellcode sends a ROP-based shellcode to reset the permission of the target memory, receive the next-stage shellcode from users, and execute the received shellcode. The second-stage shellcode can be a typical shellcode implementation instead of ROP-base shellcode.
8. For accessing the shared memory, you have to ensure you attach the memory in *READ-ONLY* mode.
9. To simplify the code 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/lab07/submit_e2e8c248e0b11354c9197b5b8403ffc2.py) | download (https://up23.zoolab.org/up23/lab07/submit_e2e8c248e0b11354c9197b5b8403ffc2.py)). You have to place the `pow.py` file in the same directory and then invoke the script. You have to implement all the required payloads in the submission script.
10. The challenge server only accepts machine codes generated for Intel x86_64 CPU.

Grading

- [25 pts] The server can handle your payload and report a normal termination status of status code 37.
- [25 pts] You can show the FLAG read from the `/FLAG` file.
- [25 pts] You can show the FLAG stored in the shared memory. Note that you must remove all padded `null (\0)` bytes when displaying the FLAG.
- [25 pts] You can show the FLAG received from the internal network server.

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