UP23 Lab03

Date: 2023-03-20

Function Call Shuffling

This lab aims to play with PLT/GOT table. Your mission is to ask our challenge server to output the lyrics for the well-known pop song *Keep the Faith* by *Michael Jackson* (Watch the Video (https://youtu.be/uNKfDyi0eoM)).

> 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 10281
```

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_LIBRARY_PATH=. LD_PRELOAD=./libsolver.so ./chals
```

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://up23.zoolab.org/code.html?file=up23/lab03/submit.py) | download (https://up23.zoolab.org/up23/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

This lab is a more complicated one (compared to previous ones), and, therefore, we provide a number of hints for you. We have prepared a modified sample challenge for you to solve locally, and then you can verify your solution on the challenge server. The directions of this lab are listed as follows.

> The local challenge only outputs a lot of "A" characters and verifies the checksum. It ***does not*** output the lyrics on the server.

1. The source code of the local challenge is available here - `chals` (view (https://up23.zoolab.org/code.html?file=up23/lab03/chals.c)) and the required runtime library `libpoem.so` (header (https://up23.zoolab.org/code.html?file=up23/lab03/libpoem.h) and source (https://up23.zoolab.org/code.html?file=up23/lab03/libpoem.c)). Alternatively, you can download everything from the zip (https://up23.zoolab.org/up23/lab03/lab03_pub.zip) file. You may have to install the `libunwind-dev` package manually to use the `Makefile` in the zip directly.
2. The `chals` program calls an `init()` function first and then calls the `code_*` functions in the library. Each `code_*` (or coding) function has a unique ID. It prints a short piece of a message, and combining all the pieces printed from the coding functions can form the whole message. At the end of the `chals`, it also verifies if the summation of checksum values returned from the coding functions is correct.
3. ***The implementation of the `chals` somehow calls incorrect coding functions. The objective of your solver is to ensure that the `chals` calls to the correct functions.*** For example, the first coding function called in `chals` is `code_498`, but all function calls to `code_498` should be `code_44`. To find the correct mappings of coding functions, please refer to the `ndat` array defined in shuffle.h (https://up23.zoolab.org/code.html?file=up23/lab03/shuffle.h). It is obvious that the ***index value*** of `498` in the `ndat` array is `44`.

4. It is intuitively that the preloaded solver may hijack some functions to solve this challenge. For example, you can implement `code_498` in your solver and let it actually call `code_44` instead. However, we have the following restrictions to prohibit you from doing this.
   - Your solver cannot have any `code_*` functions. It means that you cannot do what we just mentioned above.
   - The `code_*` functions must be called from the main function of `chals` .
5. To ensure a function call to `code_498` actually calls to `code_44` , one effective alternative approach is to ***fill*** the GOT table entry for `code_498` with the actual address of `code_44` . Since the server preloads your `libsolver.so` , you can implement the `init()` function, which is called by the `chals` at the beginning, to modify values filled in the GOT table.
6. Locating the address of the GOT table in the current running process is tricky. But since we have provided the binary of the `chals` in the zip (https://up23.zoolab.org/up23/lab03/lab03_pub.zip) file, you should be able to find the relative address of the `main` function and each `got` table entries from the binary. The relative addresses can be retrieved by `pwntools` using the script.

```
from pwn import *
elf = ELF('./chals')
print("main =", hex(elf.symbols['main']))
print("{:<12s} {:<8s} {:<8s}".format("Func", "GOT Offset", "Symbol Offset"))
for g in elf.got:
   if "code_" in g:
      print("{:<12s} {:<8x} {:<8x}".format(g, elf.got[g], elf.symbols[g]))
```

   Once you have the addresses, you can ***calculate*** the actual addresses of GOT table entries based on the runtime address of the process. The runtime address can be obtained from `/proc/self/maps` . You may simply read and parse the content of the file to obtain the addresses. One sample snapshot is shown below. Given that the relative address of the main function is `0x107a9` and the base address of the `/chals` is at `0x55c487240000` . We can calculate the actual address of the main function as `0x55c4872507a9` . The actual address of the GOT entries can also be obtained similarly.

```
** main = 0x55c4872507a9
55c487240000-55c48724b000 r--p 00000000 00:65 2630559  /chals
55c48724b000-55c487256000 r-xp 0000b000 00:65 2630559  /chals
55c487256000-55c487257000 r--p 00016000 00:65 2630559  /chals
55c487257000-55c487259000 r--p 00016000 00:65 2630559  /chals
55c487259000-55c48725a000 rw-p 00018000 00:65 2630559  /chals
```

> Note that the main function address and the got table addresses of the sample `chals` are exactly the same as the one running on the server.

7. If you have pwntools installed, you can use the command `checksec` to inspect the `chals` 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 coding 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.

8. For obtaining the actual addresses of coding functions, you may also consider using 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. ***Because the symbol offset on the challenge server may be different from your local one.***

9. Not sure if the point is useful for you, but if you are interested in how this challenge is designed, you may read the LZW (https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Welch) algorithm.

Additional Notes for Apple Chip Users

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

1. You have to work in a Linux docker.

2. The executables we packed in the zip file are compiled for x86_64. You can recompile them as `aarch64` binaries. Simply type `make` in the unpacked `lab03_pub` directory would work for you.

3. Once you have solved the challenge on your machine, you have to compile your solver implementation for x86_64 machines. 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 `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
```

4. Note that for the solver uploaded to the challenge server, the GOT table entry addresses should be retrieved from the x86_64 version of the `libpoem.so` binary, ***not your recompiled one***. You can always unpack the `libpoem.so` file from the zip file.

Grading

- [30 pts] Your solver works with the local sample challenge. That is, running the command locally would get the `** Good Job` message.

```
LD_LIBRARY_PATH=. LD_PRELOAD=./libsolver.so ./chals
```

- [70 pts] Your solver works with the challenge server. That is, submitting your solver implementation (`libsolver.so`) to the remote server can output the correct lyrics on the remote server.

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