Network Security

Exercise Sheet 01: System Security

Prof. Dr. Mathias Fischer, August See, Finn Sell

Summer semester 2024

Goals and Objectives For programming tasks, submit the code as an extra file. Please write for each (sub) task approximately how much time you need. You should reach at least 50%.

The goal is to perform the attacks on the remote binary. If this does not work, describe how you did it locally.

- Some hints to sovle the tasks are in the source files.
- Try it always locally first!
- Be nice to and on the remote server!
- Always hand in the proof if the server gives you one.

For the following tasks you will need:

- A linux system or something equivalent e.g., docker, wsl, ...
- The ability to run makefiles (basic compiler suite like gcc).

1 Stack overflows (20%)

Consider *simple-overflow*. This binary asks for a name and a password and prints a secret string (flag) if the password matches. The source code and a compiled *linux* binary are included for this task so you can try it out localy[1]. You will likely need a debugger like *GDB, pwndbg, IDA, Ghidra, ...* for some basic introspection of the running program. We also host a remote instance of the binary which is the actual target of your attack. You can connect to it on *tcp:195.37.209.19:9000* (e.g. using netcat *nc 195.37.209.19 9000*). Note that the remote server allows a maximum of 10 concurrent connections per IP.

[1]To compile it run *make*

1. Where is the buffer overflow vulnerability and how can it be fixed?
2. Which adjacent variables can be overflown? What can you do to e.g. the *secret* variable?
3. Use the buffer overflow vulnerability to force the execution of the binary to print the *Authenticated* line. How does it work? *Hint: Pay attention to the endianness and null termination of C style strings. Solve this only using the buffer overflow vulnerability and not with bruteforce.*

2 Binary exploitation

Consider *medium-format-string-rop*. The source code and a compiled binary are included just like Task 1. Similarly, the remote binary that should be your target is running at *tcp:195.37.209.10:9001*.

2.1 Information Leakage (20%)

1. Where are the format string vulnerabilities and how can they be fixed?
2. Try to leak the value of the secret. What is the secret on remote? What is the Proof?
   Hint: Use the marker to your advantage, consider up to 120 arguments on the stack (%120$p).

2.2 Altering binary execution (30%)

From this task onwards you will need some exploitation tooling like pwntools (`https://github.com/Gallopsled/pwntools`). This simplifies the creation of exploits, especially for format strings. Try this first locally and then on remote.
Note that the binary is position independent. Thus, the addresses are always different after each execution.

1. How do you obtain the address of the *win()* function?
2. How do you obtain the address of *(*main())'s return address on the stack? Why do you need this?
3. Call the win function by changing the return address of the main function to the address of win. What does win prints?
   Hint: Take a look at the slides about using format strings to write to values. See the example of pwnlib [2]

[2] `https://docs.pwntools.com/en/stable/fmtstr.html`

2.3 Remote code execution (30%)

If you manage to obtain RCE on the remote be nice!

1. How do you obtain the address of:
   a) The libc function *__libc_start_main*? The current execution offset in this function on the remote is *__libc_start_main + 0xF3*
   Hint: Take a look at the return address of main in a debugger or disassembler (Ghidra, IDA, Cutter.re) Pay attention the the function offset. Yours should be different then the offset on remote.
   b) The address of, e.g., the *system* function for the current execution of the binary.
   Hint: If you know the address of a libc function and which libc is used (for remote it is libc-remote.so), you can calculate an offset. Example is given in Listing 2. [3]
   c) The address of a string, e.g., */bin/sh* for the current execution of the binary.
   Hint: /bin/sh is present in the given libc_system.so file which is loaded on the remote.
   d) The address of a *pop rdi ret* and the adress for a *ret* gadget for the current execution of the binary (Used to pop /bin/sh into rdi so that it is a parameter to the system call. The ret gadget is used to align the stack [4]).

2. Which addresses on the stack would you overwrite with which found addresses and why (cf. Return Oriented Programming)?

```
return_addr = address of ???
return_addr +8 = address of ???
...
```

Listing 1: Overwriting return addresses

3. Get a shell on remote. What is the content of */exploitable/proof*?

```python
from pwn import *
# current_libc_some_func = address of some libc function of the current execution of
  the binary
  context.clear(arch = 'amd64')
  context.binary = exe = ELF('./medium-format-string-rop')
  libc = ELF('./libc-remote.so')
  addr_libc_some_func = libc.symbols['_some_func']
  libc_offset = current_libc_some_func - addr_libc_some_func # offset calculation
```

Listing 2: Calculation of addresses using pwntools

[3] https://docs.pwntools.com/en/stable/rop/rop.html
[4] https://stackoverflow.com/questions/54393105/libcs-system-when-the-stack-pointer-is-not-16-padded-causes-segmentation-faul