IT Security 2024/2025
Exercise Sheet 5
– Applied Binary Exploitation –

Ben Swierzy

Publication: 07.11.2024
Deadline: 13.11.2024 10:00

Lightning Survey ⚡

Exercise 1 (Buffer Overflow, 2 points)**.** Consider the compiled binary `task1` and its source code `task1.c`. Exploit the buffer overflow in the program and create a file `/tmp/pwned` by extending the script `task1-exploit.py`. There are multiple ways to solve this task. As ASLR cannot be disabled on the pipeline runner, the binary provides some assistance in circumventing it.

Important: While you are technically not limited, you should only interact with the binary and only interact via basic IO. Other interactions with the system like reading the process memory map or directly creating the target file will lead to point deductions.

Exercise 2 (Shellcode, 3+1 points)**.** Consider the compiled binaries `task2{a,b}` and their source codes `task2{a,b}.c`.

a) Write shellcode that reads and prints the first 256 bytes of the file `/secret`. Submit your shellcode as `shellcode2a.bin` and your **commented** (!) assembly as `shellcode2a.s`.

b) Oh no! The program does not read arbitrary data anymore. Modify your shellcode so it passes the restrictions on the input. Submit the solution as `shellcode2b.bin` and your **commented** (!) assembly as `shellcode2b.s`.

The binary is compiled statically and is not relocatable, so you do not need to worry about ASLR here. However, you cannot be too sure about your location on the stack, so try to operate with relative addresses.

Exercise 3 (Return-oriented Programming, 4 points)**.** Extend the script `task3-exploit.py` by writing a ROP-chain to exploit the buffer overflow in `task3` to create the file `/tmp/pwned`. You find the source code in `task3.c`. Be again aware of ASLR.

Hint: An additional pipeline job calls ROPgadget on glibc for you, so you do not need to think about getting the correct version. Furthermore, another job fetches the offset of some symbols and strings.
Hint 2: When developing your exploit with gdb, you may want to set a breakpoint after the leak but before `gets`*. You can then write your exploit for the current try into a file and move it to stdin using the gdb command* `call (void)freopen("exploit", "r", stdin)`

Important: While you are technically not limited, you should only interact with the binary and only interact via basic IO. Other interactions with the system like reading the process memory map, libraries or directly creating the target file will lead to point deductions. This includes most of the ROP-specific tools shipped by pwntools.

Note: The pipeline image is based on the `ubuntu:jammy` *docker image. This might help you with debugging your solutions locally.*

Note 2: The pipelines for exercises 1 and 3 seem to sometimes fail without an obvious reason — even for the sample solutions. If you are pretty convinced your solution should work, you can try triggering the pipeline job again.