Saarland University
Systems Security
Prof. Dr. Thorsten Holz

CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Systems Security

WiSe 2023/2024

Assignment 4 / December 19th, 2023
Due January 8th, 2024, 23:59
Carefully read "Assignments" in CISPA CMS before working on the tasks.

Please note that you need to achieve 50% of the 240 possible points in all seven exercise sheets to be admitted to the exam; points above 50% count as bonus points for your final grade.

Task 1: Code Injection (7 points)

In this task, you will exploit a buffer overflow in a more complex program, `rate_my_name`. Inject your shellcode to spawn a `/bin/sh` shell and read the flag. Write and submit a *commented* script that writes your exploit to `stdout`, such that the output can be used as the argument for the target program. Before starting, we recommend you investigate which C standard library functions are used by the program and familiarize yourself with interesting ones. If in doubt, always read the manpages.

Taskname for remote: `code_injection`

Task 2: Stack Canaries (8 points)

Following the lecture, one of your fellow students noticed their program is susceptible to a buffer overflow vulnerability. Luckily, he learned how stack canaries can help to fix that! Since he does not trust the existing implementations, he decides to implement his own version to ensure that the canaries really work. You have managed to steal parts of the source code, but unfortunately the valuable canary values are missing. Can you still exploit the program?

Write and submit a *commented* script that writes your exploit to `stdout`, such that the output can be used as the argument for the target program. After spawning a `/bin/sh` shell, you should be able to retrieve the `flag` located within the same directory. Since the implementation of the countermeasure is non-deterministic, you might have to run `task submit` multiple times.

Taskname for remote: `fake_canary`

- Don't think too complicated: This task uses only a simple, conceptual imitation of stack canaries.
- Note that there are various "bad characters" (besides nullbytes)
  http://blog.disects.com/2014/04/exploitation-identifying-bad-characters.html

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>

int stack_canaries[10] = ...
int r;

int overflow(char *password) {
    int canary = stack_canaries[r];
    char password_buffer[16];
    strcpy(password_buffer, password);
    if (canary != stack_canaries[r]) {
        printf("Someone tampered with my stack :(\n");
```

```
        printf("I'm out!\n");
        exit(1);
    }
    return 0;
}
int main(int argc, char *argv[]) {
    if(argc < 2) {
        printf("Usage: %s <password>\n", argv[0]);
        exit(0);
    }
    srand(time(NULL));
    r = rand() % 10;
    overflow(argv[1]);
}
```

Task 3: Address Space Layout Randomization (ASLR) (10 points)

a) **ASLR Bypass (8 Points)** *Address Space Layout Randomization* (ASLR) randomizes the memory layout for each program execution. This means, it places stack, heap, code, linked libraries etc. at random addresses. As a consequence, even if an attacker is able to hijack the control flow, they might not know *where* they should re-direct the control flow to (for example, the address at which their injected shellcode is placed). In general, this defense mechanism can be circumvented by adding an initial leakage stage to the exploit: first a so-called *information leak* (more or less any way to extract run-specific addresses) is used in order to reconstruct (parts of) the current memory layout. Often, this is done by leaking one specific address and then calculating offsets relative to this address. An attacker can then dynamically adapt their exploit to work for the current memory layout.

Your task is to write and submit a *commented* script that exploits the vulnerability in the target binary. Other than previous tasks, your submission must call the target binary, leak its memory layout, and then exploit the vulnerability to spawn `/bin/sh`. To do so, use the provided `pwntools` template and submit both your exploit and the value of the `flag`. Note that the randomization may place program parts at addresses that contain bad bytes (such that your exploit fails). If this happens, you can run `task submit` again. Before working on the exploit itself, we recommend that you research and answer the following questions:

b) **Additional Questions (2 Points)** `pwntools` allows you to print the addresses of functions after spawning a process.

1. How would you print the address of the function `system` given a process object `p`?
2. What would be the implications for a `setuid` binary if this was possible?

Username for remote: `aslr`

- Use the template `solution_template.py` to (automatically) interact with the target program.
- The flag is located at `/home/user/flag`.
- In the lecture, we discussed that ASLR can sometimes be defeated by a brute-force approach (especially on 32-bit systems). This is *not* the solution here and not acceptable.

Task 4: Return-Oriented Programming (ROP) (12 points)

Return-Oriented Programming (ROP) is a generalization of return-to-libc attacks and allows to perform arbitrary computations ("Turing completness") without injecting new code. The basic idea is to *chain* small code snippets—so-called ROP gadgets—together that each fulfill a specific task (e. g., adding two registers).

a) **Chaining of Gadgets (4 Points)**

First, we want to investigate how ROP gadgets can be chained to spawn a shell. To this end, your task is to translate the shellcode from Figure 1 to a *semantically* equivalent ROP chain. Assume that the ROP chain can be copied with `strcpy` into the process' memory due to some implementation flaw.

Prepare and submit the stack in Figure 1 such that during execution of the ROP chain the shell `/bin/sh` is spawned via `execve`.

Hints:

- Some gadgets may have undesired side effects. Mind the order in which you place the gadgets on the stack.
- Do not try to translate the reference shellcode line by line but preserve its functionality. Also, remember that gadgets can be used multiple times if needed but you may not change them.

b) **ROP-based Exploitation (8 Points)**

Your task is to exploit the program `rop` using Return-Oriented Programming. Such an attack starts with locating suitable gadgets that can be chained to a ROP chain. Finding these gadgets manually, however, is a tedious task. Therefore, typically automated tools such as `ropper` are used. For this task, you may use these tools but only to find suitable gadgets.

Write and submit a *commented* script that writes your exploit to `stdout`, such that the output can be used as the argument for the target program. This exploit should call—via a ROP chain—`execve` to spawn an interactive python3 process (i. e., `/usr/bin/python3`) in the context of the targeted program. After spawning python3, you should be able to retrieve the `flag` located within the same directory.

Taskname for remote: `rop`

- To increase the number of gadgets, consider linked libraries such as `libc`. Mind that the addresses provided by `ropper` for gadgets in `libc` are relative to the base of `libc`, since the library is built as position independent code (PIC).
- If you struggle with rebasing the gadgets, consider drawing the memory layout of `rop` in order to understand the steps required to calculate the correct addresses.
- You can export environment variables with custom strings if you need to.
- Within `python3` you can use `print(open("flag").read().strip())` to read the flag.

Total points on sheet: 37

Shellcode

```
1   xor eax, eax
2   push eax
3   ; / b i n / / s h
4   ; 2f 62 69 6e  2f 2f 73 68
5   ; push in inverse order
6   push 0x68732f2f
7   push 0x6e69622f
8   ; prepare execve call
10  mov ebx, esp
11  mov ecx, eax
12  mov edx, eax
13  mov  al, 0xb
14  int 0x80
```

ROP-Gadgets

```
0xc0de1111:
  pop esi
  mov ebx, 0xA
  ret
```

```
0xc0de2222:
  lea eax, [eax]
  ret
```

```
0xc0de3333:
  mov ecx, 0
  ret
```

```
0xc0de4444:
  int 0x80
  ret
```

```
0xc0de5555:
  inc eax
  pop edx
  jmp edx
```

```
0xc0de6666:
  mov eax, esp
  mov dword [esp+0x8], 0
  add esp, 0xC
  ret
```

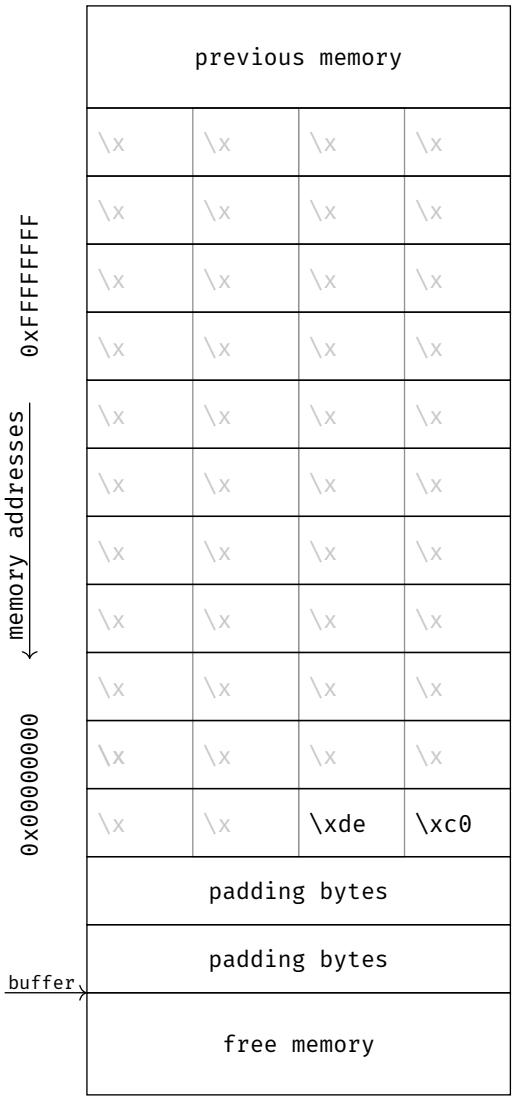
```
0xc0de7777:
  xor eax, ebx
  xor ebx, eax
  xor eax, ebx
  ret
```

```
0xc0de8888:
  mov ecx, 0x42
  inc ecx
  cdq
  ret
```

```
0xc0de9999:
  mov ecx, 0
  lea esp, [esp]
  pop ecx
  ret
```

```
0xc0deaaaa:
  mov eax, 0xB
  pop ebx
  push 0
  ret
```

Stack with ROP-Chain



Description

Figure 1: ROP Chain