Saarland University
Systems Security
Prof. Dr. Thorsten Holz

CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Systems Security

WiSe 2023/2024

Assignment 3 / December 5th, 2023
Due December 18th, 2023, 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: Analysis of Programs (10 points)

Your task is to analyze the following program and especially its subroutines `f` and `g`.

```
function main:
  ...
  .text_0000059E  push    3
  .text_000005A0  push    2
  .text_000005A2  push    1
  .text_000005A4  call    f
  .text_000005A9  add     esp, 0Ch
  ...
```

```
function g:
  .text_0000052D    push    ebp
  .text_0000052E    mov     ebp, esp
  .text_00000530    sub     esp, 4h
  .text_00000533    mov     dword ptr [ebp-4], 0
  .text_0000053A    jmp     short loc_546

  .text_0000053C  loc_53C:
  .text_0000053C    mov     eax, [ebp+0Ch]
  .text_0000053F    add     [ebp+8], eax
  .text_00000542    add     dword ptr [ebp-4], 1

  .text_00000546  loc_546:
  .text_00000546    mov     eax, [ebp-4]
  .text_00000549    cmp     eax, [ebp+0Ch]
  .text_0000054C    jl      short loc_53C
  .text_0000054E    mov     eax, [ebp+8]
  .text_00000551    leave
  .text_00000552    retn
```

```
function f:
  .text_00000553    push    ebp
  .text_00000554    mov     ebp, esp
  .text_00000556    sub     esp, 8h
  .text_00000559    cmp     dword ptr [ebp+8], 0
  .text_0000055D    jnz     short loc_56C
  .text_0000055F    mov     edx, [ebp+8]
  .text_00000562    mov     eax, [ebp+0Ch]
  .text_00000565    add     eax, edx
  .text_00000567    mov     [ebp-4], eax
  .text_0000056A    jmp     short loc_57D

  .text_0000056C  loc_56C:
  .text_0000056C    push    dword ptr [ebp+0Ch]
  .text_0000056F    push    dword ptr [ebp+8]
  .text_00000572    call    g
  .text_00000577    add     esp, 8
  .text_0000057A    mov     [ebp-4], eax

  .text_0000057D  loc_57D:
  .text_0000057D    mov     edx, [ebp-4]
  .text_00000580    mov     eax, [ebp+10h]
  .text_00000583    add     eax, edx
  .text_00000585    mov     [ebp-8], eax
  .text_00000588    mov     eax, [ebp-8]
  .text_0000058B    leave
  .text_0000058C    retn
```

a) **Reconstruction of C code (3 Points)**

Translate functions `f` and `g` to equivalent high-level code. A notation close to C is sufficient. What is the return value of the function `f` if it is called with parameters (`1, 2, 3`)?

b) **Calling Conventions (2 Point)**

Which *calling convention* is used? Briefly explain your answer. Additionally, indicate the order in which the parameters are pushed onto the stack and indicate at which offsets (relative to the base pointer `ebp`) the callee’s first three parameters are placed (e. g., first parameter’s offset at `[ebp-100]`).

c) **Stack frames (5 Points)**

Investigate how stack frames and memory change over time during execution. Fill empty cells in the template on the last page such that your solution includes the following:

- Description of what is stored in memory (left most stack frame).
- Content of memory at each of the four points in time. To this end, enter the concrete values stored during execution (also for the instruction pointer `eip`!). Leave all cells empty for which the value is not known. For the saved `ebp`, you may use the placeholder `saved ebp`. You can ignore the endianness for this task.
- Position of the stack pointer `esp` and base pointer `ebp` for Snapshot 2 to Snapshot 4.

Task 2: Data-Only Attack (7 points)

a) **Simple Data-Only Attack (3 Points)** The following code snippet contains a vulnerable password authentication that allows for a successful authentication even with an incorrect password.

```c
int check_authentication(char *password) {
    int auth_flag = 0;
    char password_buffer[16];
    strcpy(password_buffer, password);
    if(strcmp(password_buffer, "5W0Rdf15H") == 0)
        auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[]) {
    if(argc < 2) {
        printf("Usage: %s <password>\n", argv[0]);
        exit(0);
    }
   char *password = argv[1];
   if (check_authentication(password)) printf("Correct!\n");
   else printf("You shall not pass!\n");
}
```

Find and submit a valid input not equal to `5W0Rdf15H`, such that the program outputs *Correct!*. Assume the compiler allocates local variables in the same order as defined by the source code. Why does your attack work? Sketch the stack frame layout of function `check_authentication`. What would happen if the compiler reorders local variables? Explain concisely. Please note that this part is purely theoretic.

b) **Hardening (4 Points)** Assume that the problem in the code was identified by the developer and the vulnerable function `check_authentication` was hardened accordingly—the outcome is the `doa_hardened` program. Analyze the program `doa_hardened` with this updated authentication routine. What has changed and how can this fix be bypassed? **Briefly** describe the new check(s) and, again, find and submit a password not equal to `5W0Rdf15H`, such that the program outputs the success message. Sketch the stack frame layout of function `check_authentication`. Please note that the source code of the hardened program is not available—you need to reverse the (interesting parts of) the program!

Taskname for remote: `doa_hardened`

Task 3: System Calls and Shellcode (6 points)

On modern operating systems, system calls or *syscalls* are used as a well-defined interface for applications to communicate with the kernel of an operating system (e.g., for reading and writing files). For example, the Linux kernel defines 384 system calls for the x86 architecture. This interaction is usually abstracted away from the programmer through the means of high-level library functions (e.g., using the standard library `libc`). However, when exploiting a binary we often want to skip this extra level of abstraction.

In this task, we want to take a closer look at system calls and how we can use them for binary exploitation.

a) **Hello Kernel (2 Points)**

Your first task is simple: Create the file `/home/user/greetings` and write `himum!` into it. However, other than in previous assembly-programming tasks, you are no longer allowed to use the C standard library `libc`. Specifically, you should now link the object files directly with the GNU linker `ld` (rather than `gcc`) as follows

```
nasm -felf32 code.asm -o code.o && ld -m elf_i386 code.o -o solution
```

This means there is neither a `main` function nor any library functions to use. Instead, you will have to resort to system calls. Submit your *commented* source code. Make sure that your program does not terminate with a `segmentation fault`.

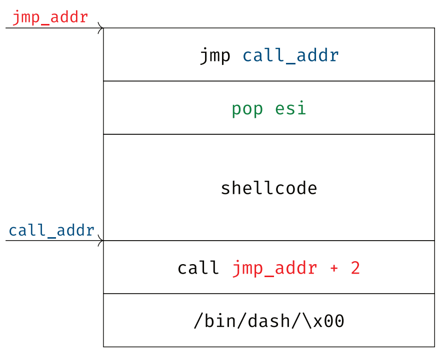
Taskname for remote: `syscalling`

- The `hexdump` command might become handy if you want to check a files content.
- Linux Syscalls: `https://bit.ly/2TKlGD1`

b) **Shellcode (4 Points)**

One key technique for binary exploitation is the injection of attacker-controlled code into the target program. This code often spawns a "shell" and, therefore, is usually referred to as *shellcode*. In this task you will write and assemble your own shellcode. To this end, use the system call `execve` to spawn the shell `/bin/dash`. Take care to avoid null-bytes and keep your shellcode independent of specific addresses.

In order to obtain position-independent code, you might want to use the jmp/call trick:



Idea:

The `call` instruction pushes the `eip` (i.e., the return address) on the stack. If we know the offset to the `call` target, we can calculate the address of the shellcode.

Approach:

1. jump forward (relative offset in bytes)
2. `call` back (negative relative offset in bytes)
3. `esi` holds address of string `/bin/dash`

Submit your *commented* source code as well as the raw bytes of your shellcode as a comment in the code (i.e., `SHELLCODE: "\x90\x90\x90..."`).

Taskname for remote: `shellcode`

- Shellcoding 101: `https://bit.ly/31ILgtb`
- Assemble and link your shellcode with:
  `nasm -felf32 shellcode.asm -o x.o && ld -m elf_i386 x.o -o shellcode`
- You can check whether you spawned `/bin/dash` with `ps -p "$$"`
- To extract the raw bytes, use `objdump -D shellcode` to disassemble your shellcode. You can test your code blob with `test_shellcode.c`

Task 4: Smashing the Stack for Fun and Profit (8 points)

Consider the following vulnerable application:

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

/// DEBUG NOTE: admin_pw @ 0xABAD1DEA
char *admin_pw = "sup3r_s3cr3t_passw0rd!!1elf";

/// DEBUG NOTE: start of function @ 0xC001D00D
int authenticate(char *password){
        // first pad the password to hopefully avoid timing attacks
        char padded_password[28];
        strcpy(padded_password, password);
        // check the password
        if (strcmp(padded_password, admin_pw) == 0) return 1;
        return 0;
}

/// DEBUG NOTE: start of function @ 0xBADEAFFE
void accepted(){
        printf("Welcome to the admin console, trusted user!\n");
}

void rejected(){
        printf("Go away! You smell :(\n");
        exit(1);
}

/// DEBUG NOTE: argv[1] = 0xCAFEBABE
int main(int argc, char *argv[]){
        int authed = authenticate(argv[1]);
        if (authed) accepted();
        else rejected();
        return 0;
}
```

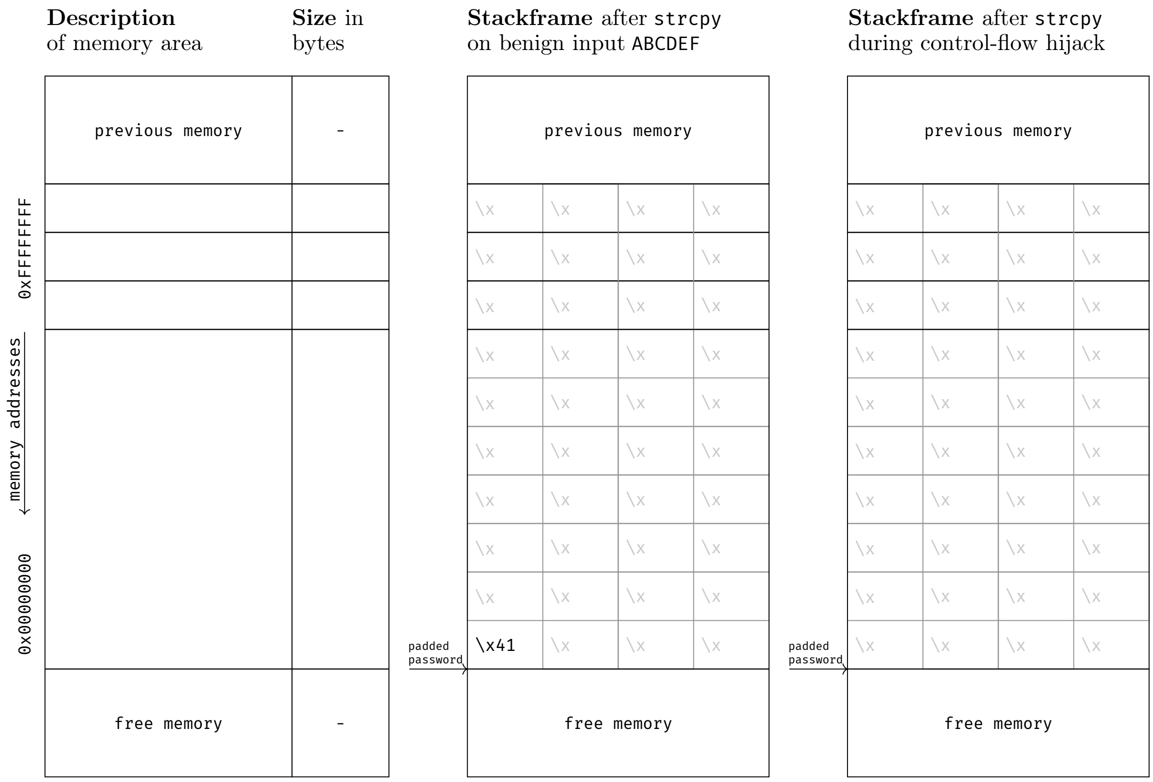
a) **Control-flow Hijacking (5 points)**

Assume that you were able to extract a couple of function and memory addresses from the program execution by debugging the application (denoted as `DEBUG NOTE` comments in the source code). Your task is to use this information to craft an exploit that overwrites the return address of the authentication function in such a way that the execution proceeds directly to the `accepted()` function (without having entered the correct password).

To this end, consider the stack frame after execution of `strcpy` in `authenticate(·)` for a "normal" execution. Assume the program is called with parameter `ABCDEF` and fill out the template's left part from (description + stack frame). Remember, strings in C are terminated by a null-byte and `strcpy` copies up to and including `0x00`. You can mark all undefined or unknown bytes/values with a '?'.

Then construct a password, i. e., a command line parameter for the application, such that `accepted(·)` is executed (without using the real password). If you need padding bytes, use the character `A`. How does the stack frame look after the execution of `strcpy`? Fill out the right hand side of the template. Submit the resulting command line parameter together with the completed template.

Reminder: The byte order on x86 processors is little-endian.



b) **Code Injection (2 points)**

The buffer overflow not only allows us to change the program's control flow but also to inject our own code. Assume the buffer `padded_password` is located at address `0xDEADBEEF` during execution. How could a command line parameter with a possible exploit look like? Assume you want to inject the following shellcode with 21 bytes into the buffer and execute it.

`<SHELLCODE> = "\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80"`

Briefly explain why it would be problematic if `padded_password` is not located at address `0xDEADBEEF` but address `0xDEAD0070`.

c) **Countermeasure (1 point)**

How could you modify the source code to effectively prevent the vulnerability? Explain concisely.

Task 5: Stack-based Buffer Overflow (7 points)

We now want to actually implement stack-based buffer overflows, the programs `basic_overflow` and `slide_rider` are susceptible to this kind of vulnerability. Your task is to exploit this fact by writing suitable shellcode. 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 the shell, you should be able to retrieve the `flag` located within the same directory.

a) **Basic Buffer Overflow (3 points)**

First analyze the program `basic_overflow`. The buffer overflow is located in line 12 as the input length is not checked. Before writing your exploit, think about how many bytes are needed to overwrite the return address and how you can inject your shellcode into the process. The target program helps you by printing the address of the buffer during execution.

Taskname for remote: `basic_overflow`

b) **Nop Sliding (4 points)**

In practice, memory addresses may change, for example due to different environment variables. This can render exploits useless that depend on a specific memory address. One technique commonly used to increase an exploit's reliability are so-called NOP sleds / NOP slides. This term refers to a large number of consecutive No-Operations (NOPs) prefixing your actual shellcode and serving as a *landing area* for the jump into the shellcode.

Your task is again to inject your shellcode and read the secret flag—but this time use a NOP sled such that your exploit is stable even if addresses change slightly. Remember, you can overflow more than the current stack frame.

Note: The source code is not available for this task.

Taskname for remote: `slide_rider`

Description

What does the stack memory contain during execution?

Snapshot 1

After executing the instruction at 0x5A2.

Snapshot 2

After executing the prologue of function f.

Snapshot 3

After entering function g, but before executing 0x52d.

Snapshot 4

Before executing the epilogue of function f.