Saarland University
Systems Security
Prof. Dr. Thorsten Holz

Systems Security

WiSe 2023/2024

Assignment 2 / November 21th, 2022
Due December 5th, 2023, 23:59
Carefully read “Assignments” in CISPA CMS before working on the tasks.

x86 Assembly

For the following tasks, you are expected to read, write and understand x86 assembly, a necessary prerequisite to inspect C programs on the binary level. Besides the lecture notes, consider the following resources as needed: instruction reference, short guide, example-driven introduction, and NASM tutorial (note: x86-64 bit, but still mostly relevant).

Task 1: Assembly Instructions (5 Points)

Consider the following three x86 assembly snippets:

```
; Data Movement
mov esi, 4
push 8
push esi
mov eax, [esp]
lea eax, [esp + eax * 2 + 4]
sub eax, 8
mov eax, [eax]
pop ebx
add esp, 4
add eax, ebx
```

```
; Arithmetic and Logic
xor eax, eax
add eax, 8765h
ror eax, 16
or eax, 42h
inc eax
shl ax, 8
mov al, 21h
```

```
; Control Flow
mov eax, 42h
neg eax
mov ebx, 0xFFFFFFF8
cmp eax, ebx
jl false
true:
    mov eax, 1
    jmp done
false:
    mov eax, 0
done:
```

a) For each snippet, analyze which value `eax` contains after execution. Submit this value in hexadecimal notation.

b) How does the result change when changing the jump instruction `jl` to `jb`? Explain concisely. *Reminder:* Negative numbers are represented as 2’s-complement in x86.[1]

Task 2: x86 Assembly Programming (15 Points)

This task will introduce you to x86 assembly programming. Submissions are required to use the Intel syntax. To assemble and link programs, use

```
nasm PROGRAM.asm -felf32 -o PROGRAM.o && gcc ./PROGRAM.o -m32 -o PROGRAM
```

with `PROGRAM` being the name of your program. The flags `-felf32` and `-m32` are necessary to link 32-bit programs on 64-bit operating systems. In this case, we use GCC to link the program in order to gain access to functions from the C standard library (such as `printf`). In case your program malfunctions (e. g., crashes with a segmentation fault), you can use GDB to trace the execution and locate the bug.

[1] 2’s-complement: http://igoro.com/archive/why-computers-represent-signed-integers-using-twos-complement/

a) **A small how-to (4 Points)**

Write and submit a small program using x86 assembly which calculates the following expression:

$$\texttt{eax} = (\texttt{ecx * ecx + ebx - eax * 1337}) \oplus (\texttt{ecx - 0x42 })$$

A template providing a fundamental structure is provided in the remote environment. The result of your calculation should be placed in `eax` such that the template prints it. The template sets initial register values that should be used for the registers found on the right-hand side of the expression. Do not modify these initial values.

Taskname for remote: `hello_x86`

- Use the provided template to solve this task
- Over- and underflows may occur; You can ignore them for this task

b) **Sorting (11 Points)**

In this task, you are expected to complete the template by providing missing functionality, mostly parsing integers and sorting them (at designated points in the template). Lookout for `TODO` and do not modify parts marked as `DO NOT MODIFY`. The following functionality is required:

1. The user provides a variable number of integer arguments on the command line. You assume that less than 20 values are provided and their values are in the range $[-1000, 1000]$.
2. The program should print `string_error` and properly terminate the program if *no* arguments are provided.
3. The program should parse these numbers and return them as an integer array: To do so, in function `parse`, first use `malloc` to reserve memory for an array in which the numbers will be placed. Consider the case where `malloc` returns NULL: Use the `exit` system call to exit the program with returncode 1[2]. If the memory was allocated successfully, convert each input to a number using `strtol` and store it in the array. Return the array from the function `parse`.
4. The array is then passed to the function `sort`, which should sort the array *in-place*. Use whatever (reasonable efficient) sorting algorithm you prefer.

Both `parse` and `sort` should adhere to the calling convention `cdecl` and System V ABI[3]. Consider especially how to pass parameters to the functions, which registers you need to backup before using them in the function body, and how to return any computed result.

Taskname for remote: `sorting`

- Use GDB to debug segmentation faults: You can walk your instructions step-by-step
- Double-check if you clobbered registers
- Double-check if you forgot to properly clean the stack
- Double-check if you aligned the stack

Task 3: Size Directives (5 Points)

Usually the assembler can infer the type or *size* of an operand automatically from the context. However, in some situation the size cannot uniquely determined and need to be defined explicitly.

Determine for the following snippets whether the *size directive* is either *necessary*, *optional*, or *wrong*. Briefly explain your decision.

a) Reading a `word` from address in `esp`

```
mov ax, word ptr [esp]
```

b) Reading a `byte` from address in `ebx`

```
movzx eax, byte ptr [ebx]
```

c) Writing `FFh` as a `word` at address `eax`

```
mov word ptr [eax], 00FFh
```

d) Writing `BAADF00Dh` as a `dword` on top of the stack

```
mov dword ptr [esp], BAADF00Dh
```

e) Writing the Least Significant Byte (LSB) from `ecx` to `eax`

```
movzx byte ptr eax, cl
```

[2] https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86-32_bit
[3] https://wiki.osdev.org/System_V_ABI (i386)

Task 4: Crackme (10 Points)

a) **Analysis of an Unknown Program (5 Points)**

Analyze the program `crackme` of which the source code is not available. The program expects the user to input a *secret key*, consisting of 6 capital letters. Your task is to find a correct key by analyzing the program with GDB. Briefly explain how the program derives the password from the secret key and submit the input yielding the success message "*Key is valid :)*".

Hints:

- The user input is checked in function `verify_key`. The function contains a `call` <*address*> instruction which calls the C function `strcmp`. There is no need to analyze this function.
- Within GDB, `x/s` <*address*> may help you to print strings.

b) **Reconstruction of C Code (5 Points)**

Reverse engineer the program `crackme`. Understand its functionality and submit C source code providing the same functionality, including the password derivation.

Taskname for remote: `winter_is_coming`