# Architecture Security

## Contents

# 1 Binary Exploitation

## 1.1 AMD64

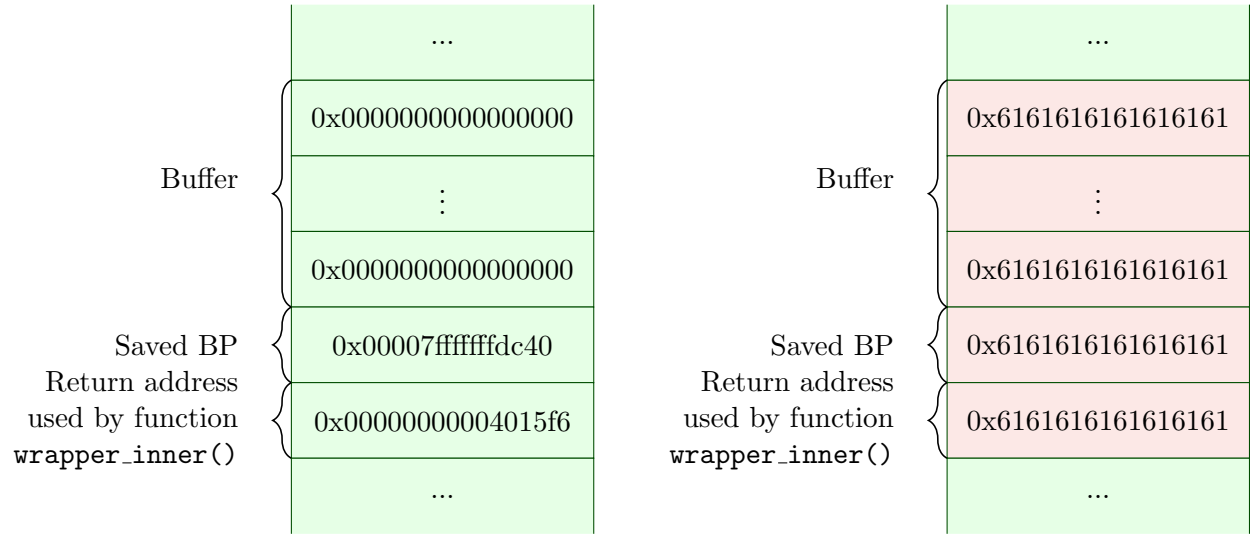
### 1.1.1 a)

The program takes one argument (a word) from the CLI. It then reverses the word and prints it back. The buffer has a length of 128 characters, and the instruction `strcpy(buf, argv[1])` is flawed and poses a security risk, i.e. it can be exploited to perform a buffer overflow.

### 1.1.2 b)

The following command can make the program crash and produce a *Segmentation Fault*: `flip $(python3 -c "print('a' * 144)")`. This happens because the return address on the stack gets overwritten with the characters `a`, and thus the program tries to return to a restricted area in memory.

The following visualization shows part of the stack before (left) and after (right) executing the `strcpy(buf, argv[1])` command. The address `0x4015f6` is used by the function `wrapper_inner()` in order to return to the function `wrapper_outer()`.




**Note:** BP stands for Base Pointer.

### 1.1.3 c)

i. The address of the function `secret()` is `0x4012b6`.

ii. Yes, the address of the function `secret()` remains the same with each execution of the `flip` program. This happens because the executable `flip` itself has no ASLR/PIE protection enabled (`hardening-check /blatt5/a1/flip/amd64/flip`) [1].

iii. The command `python3 -c "print('a' * 135 + '\xb6\x12\x40')"` will print the payload needed to call the `secret()` function

iv. By executing the command `flip $(python3 -c "print('a' * 135 + '\xb6\x12\x40')")`, we can inject the output of the Python command into flip as its first argument. Therefore,

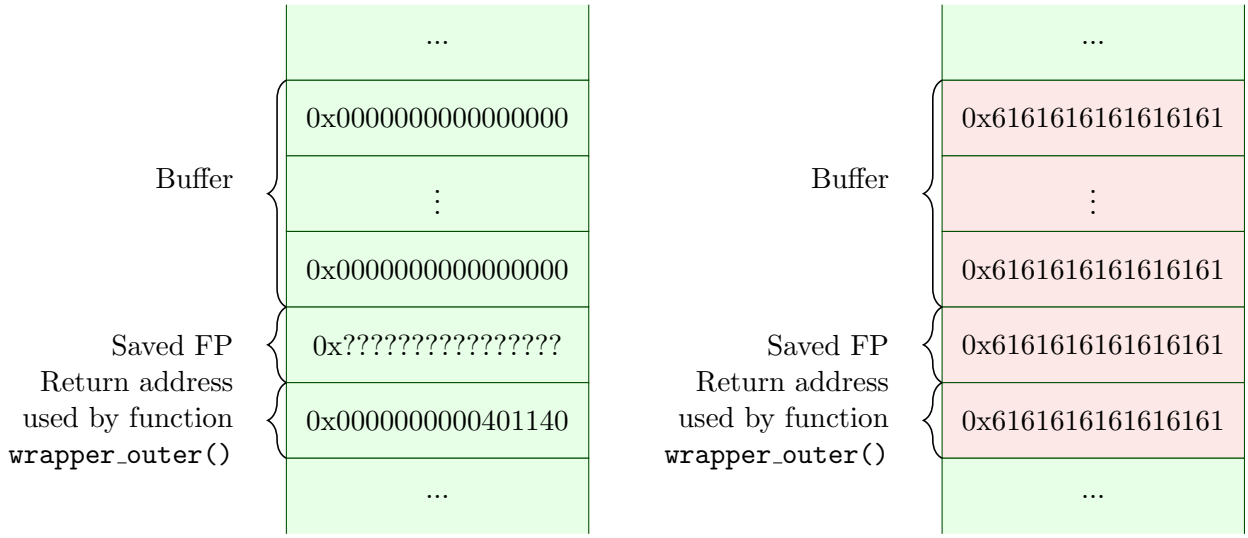
instead of returning from `wrapper_inner()` to `wrapper_outer()`, the program will jump to the function `secret()` and print the flag `I1binarYF1@GrETri3VED`.

## 1.2 ARM64

### 1.2.1 a)

The following command can make the program crash and produce a *Segmentation Fault*: `flip $(python3 -c "print('a' * 144)")`. This happens because the return address on the stack gets overwritten with the characters `a`, and thus the program tries to return to a restricted area in memory.

The following visualization shows part of the stack before (left) and after (right) executing the `strcpy(buf, argv[1])` command. The address `0x401140` is used by the function `wrapper_outer()` in order to return to the function `main()`.




**Note:** FP stands for Frame Pointer.

### 1.2.2 b)

Yes, both outputs of the crashed programs are sligthly different, i.e. the first program does not output the text `Done flipping word!`. The AMD64 program produces a *Segmentation Fault* after returning from the function `wrapper_inner()`, while the ARM64 program produces a *Segmentation Fault* after returning from the function `wrapper_outer()`.

This happens, because of the different ways both architectures write to the stack. As we can see from the drawings in chapters *1.1.2* and *1.2.1*, different function return addresses are present after the buffer. On AMD64, this address is used to return from `wrapper_inner()` to `wrapper_outer()`, while on ARM64, this address is used to return from `wrapper_outer()` to `main()`.

### 1.2.3 c)

The command `python3 -c "print('a' * 135 + '\xb0\x0d\x40')"` will print the payload needed to call the `secret()` function. By executing the command

`flip $(python3 -c "print('a' * 135 + '\xb0\x0d\x40')")`, we can inject the output of the Python command into flip as its first argument.

# 2 Binary Cracking

## 2.1 crackme1

### 2.1.1 a)

The correct password is `PA5sW0rD_4X0107L`. It can be found by analyzing the `crackme1` program with *Ghidra*.

Starting inside the `main()` function, we can see that the function `check_password()` is being called. While analyzing this function, we can see the correct password being passed to the `strcmp()` function.

### 2.1.2 b)

The flag `11crACkME1NiC31YdoN3` is returned by the function `retrieve_flag()`, which retrieves the flag from the URL https://pastebin.com/raw/QNbQbGMq. The URL can be retrieved by analyzing the function `retrieve_flag()` inside the *Listing View* in *Ghidra*.

## 2.2 crackme2

### 2.2.1 a)

The program uses a *Caesar Cipher (ROT13)* in order to hide the original password (`redpelicanbluetiger` $\xrightarrow{\text{rot+13}}$ `erqcryvpnaoyhrgvtre`).

### 2.2.2 b)

Yes, we can write such a program. The source code for this program can be found inside blatt5/a2/caesar.py.

### 2.2.3 c)

The flag `yoUSO1V3d11Cr4CKme2hUrRaY` is returned by the function `retrieve_flag()`. The function retrieves the flag from the URL https://pastebin.com/raw/5ttHCxNc. The URL can be retrieved by analyzing the function `retrieve_flag()` inside the *Listing View* in *Ghidra*. But this time, the *5ttHCxNc* part of the URL is actually encrypted by using a *ROT* cipher with a rotation of 18 (5ttHCxNc $\xrightarrow{\text{rot+18}}$ 5bbPKfV).

## 2.3 crackme3

### 2.3.1 a)

The program uses a *SHA-256* hash in order to hide the original password (`elephant.` $\xrightarrow{\text{sha-256}}$ a0a585828a2644361236d2ca69345d3bc15eb940a401d6e50f59f7ce3080c06f).

#### 2.3.2 b)

For the general case, no, we cannot write such a program that executes sufficiently fast, because we cannot brute-force all possible combinations in a timely manner. But for hint given at the end of exercise 2.3, we can. The source code for this program can be found inside blatt5/a2/sha256.py.

#### 2.3.3 c)

The correct password is `elephant.`.

#### 2.3.4 d)

The flag `I1_CraCKme3_S0M3_HA5H` is returned by the function `OOO()`. The function retrieves the flag from the URL https://pastebin.com/raw/1aqRHu2V. The URL can be retrieved by analyzing the function `III()` inside the *Listing View*. But this time, the whole URL is actually encrypted by XORing each character with the value $0xf1$.

In order to get all bytes of the URL, we can double click the variable *url* inside the `III()` function, which will show us all 33 bytes in the *Listing View*. A simple Python program to mimic the decryption process would look like this:

```
1 url = [0x99, 0x85, 0x85, 0x81, 0x82, 0xcb, 0xde, 0xde, 0x81, 0x90, 0x82, 0x85,
  ↪  0x94, 0x93, 0x98, 0x9f, 0xdf, 0x92, 0x9e, 0x9c, 0xde, 0x83, 0x90, 0x86, 0xde,
  ↪  0xc0, 0x90, 0x80, 0xa3, 0xb9, 0x84, 0xc3, 0xa7]
2
3 print(''.join(chr(c ^ 0xf1) for c in url))
```

## 3 Kernel Exploitation

### 3.1 Subtask 1

The new system call expects two addresses *arg1* and *arg2*. It then uses those addresses and creates two char arrays *in* and *out* which point to them. At the end, it copies all characters from *in* into *out*.

The source code for the example program can be found inside blatt5/a3/subtask1.c and it can be compiled by using `aarch64-linux-gnu-gcc -fno-stack-protector -no-pie subtask1.c -o subtask1`.

### 3.2 Subtask 2

Because the system call does not check if the char array *in* is bigger than the char array *out*, it can overwrite bytes in memory located exactly after *out*. This can be misused in order to perform buffer overflows and overwrite variables, return addresses, etc.

Our example program overwrites the value of the variable *proof* to $0x13371337$. The source code for it can be found inside blatt5/a3/subtask2.c and it can be compiled by using `aarch64-linux-gnu-gcc -fno-stack-protector -no-pie subtask2.c -o subtask2`.

### 3.3 Subtask 3

Our program uses the vulnerable syscall two times: one time in order to overwrite the content of the variable *impl_pointer*, so that it points to our own `hook_sys_capital_impl()` instead of the original `sys_capital_impl()` function, and the other time to call our `hook_sys_capital_impl()` function.

Our `hook_sys_capital_impl()` function executes `commit_creds(prepare_kernel_cred(0))` in order to create a new privileged *cred* struct and assign it to the currently running process [2]. We can get the addresses for `impl_pointer`, `prepare_kernel_cred()` and `commit_creds()` by executing `cat /proc/kallsyms` and piping its output to `grep` in order to only display the relevant results.

At the end, our program checks if the UID is 0 [3], and if that is true, it will spawn a root shell by executing `/bin/sh`. We can than execute `cat /flag` in order to get the flag `7h15keRNELwa$PWND8y11StUD`.

The source code for the exploit can be found inside blatt5/a3/subtask3.c and it can be compiled by using `aarch64-linux-gnu-gcc -fno-stack-protector -no-pie subtask3.c -o subtask3`.

## 4 Requirements

```
sudo apt install -y gcc-aarch64-linux-gnu
# if hardedning-check is not installed
sudo apt install -y devscripts
```

## References


1. https://www.romanh.de/article/binary-exploitation
2. https://ctf-wiki.mahaloz.re/pwn/linux/kernel/ret2usr/
3. https://gist.github.com/n4sm/e032f84bf3ffd7f790cbbc1eccbfb898