Information Security Lab
Module 3: Software Security and Side-Channel Attacks

Teaching Assistants: Dr. Giovanni Camurati, Claudio Anliker,
and Benedict Schlüter

v2023.1
November 2, 2023

Responsible: Prof. Dr. Srdjan Čapkun (SYSSEC),
Prof. Dr. Shweta Shinde (SECTRS)

Contents

History:

- v2023.1 / 02.11.23: Initial release

1 Organisation

Welcome to Module 3 of the Information Security Lab! This module is a collaboration between the System Security Group and the Secure & Trustworthy Systems Group. The first part of the module is about memory corruption bugs in vulnerable software, and the second about side-channel attacks.

1.1 Lecture

06.11.23: This lecture covers memory corruption vulnerabilities and related security mechanisms relevant for the first part of the lab (exercises 1-6). Furthermore, we are introducing the lab environment and your assignment on a high level.

13.11.23: This lecture is about Trusted Execution Environments (TEEs) and their weaknesses. It is relevant for exercise 7.

20.11.23: *TBD*

1.2 Exercise Sessions

We use the exercise sessions only to prepare you for your lab assignments. We will not introduce additional tasks or exercises in these sessions.

07./08.11.23: We will show you how to work within our lab environment and how to use the tools you need to develop your exploits for exercises 1-6.

14./15.11.23: We will introduce additional tools required for exercise 7 (side-channel attacks).

21./22.11.23: *TBD*

1.3 Lab Sessions

The lab sessions are intended to discuss any problems that you might have with the assignment. You are also welcome to post your questions and join ongoing discussions on the Moodle forum. Please make sure that you do not share solutions when asking or answering questions. Lab session attendance is not mandatory.

2 Introduction

This section introduces the methodology behind developing exploits. If you are experienced in binary exploitation, it may be enough to skim this part.

2.1 Goal of this Module

The main goal of this lab is to understand memory corruption vulnerabilities and, in particular, how they can be exploited to conduct a *privilege escalation* attack.

Imagine a scenario in which an attacker wants to completely compromise a target system by spawning a root shell. Furthermore, assume that they were already partially successful and able to get low-privileged shell access. In reality, this might happen due to a vulnerability in an internet-facing service (e.g. a web server) or an incautious user opening a malicious email appendix. The objective is now to elevate or escalate these privileges to establish full control over the target.

Memory Corruption

As the name indicates, memory corruption allows an attacker to change the memory of an affected process. Depending on the vulnerability, it may be possible to modify data (e.g., variables) or even divert the program's control flow (e.g., through return address manipulation). If the attacker manages to inject their own code, they may completely control the process' behaviour. If the affected process is additionally running with higher privileges (usually root), privilege escalation can be achieved.

On Linux, a process typically runs with the privileges of the user who *starts* it, not of the one who *owns* it. The exception are binaries that have the `suid` bit set, since the very purpose of this bit is to execute the binary with the owner's privileges. This is necessary for programs that are run by low-privileged users but perform privileged actions. One example is `passwd`, which is used to change a user's password and has to write to protected files (i.e., `/etc/passwd` and `/etc/shadow`).

The `suid` bit is indicated with an "s" instead of an "x" in the owner's privilege mask, as show below:

```
user@linux:~ ls -l /usr/bin/passwd
68 -rwsr-xr-x 1 root root 68208 Nov 29  2022 /usr/bin/passwd
```

Naturally, binaries that are owned by root, have the `suid` bit set, and contain exploitable vulnerability can be used for privilege escalation. Throughout exercises 1 to 6, you will attack several such binaries.

Developing an exploit

The standard approach to writing exploits is to set up an environment resembling the target regarding software versions and configurations as closely as possible, and develop/test the exploit there. This simplifies the process tremendously, since the attacker may debug the running process on a system they control. The finished exploit can then be copied to and run on the target system. By and large, we will follow this methodology in this module.

2.2 Lab Environment

This section should give you an overview of the lab environment. You will essentially be following the workflow explained above, with the following exceptions:

- **Container:** You provide you with your own, personal *container* (more on this later) that resembles the target system, but in which you have root privileges. You can use this environment to test your exploit without having to worry about setup or configuration.
- **No access to the target:** In contrast to a real attacker, you do not have low-privileged shell access to the target system, but you will upload your (hopefully working) exploit to a *grader*, which will run it automatically and check if the privilege escalation works. In a sense, you will perform a controlled attack on a machine that we provide.
- **Reading a root-owned flag:** Your goal is to read a file on the grader that is only readable by root. This task is largely equivalent to spawning a root shell, since the reading operation will fail unless you have successfully escalated your privileges. However, it has the advantage that our grader can verify the privilege escalation by simply checking whether the standard output contains the flag or not.

Container

Your container has a read-only directory "/home/student/handout" in which you find your individual binaries and the exploit templates.

Important

The container has very limited resources, so keep the following points in mind:

- You have only shell-based access to the container (SSH).
- You can only run limited software within the container, such as your exploit (python), your debugger (GDB), or a text editor. Do not install tools such as Ghidra in your container.
- As a result, we recommend that you write your exploits either using a text-based editor in your container or, if you prefer a visual editor, on your own, local machine.).

We will show you in Section 3 how you can interact with your container using SSH. If you like a more seamless SSH configuration with less typing and transparent file synchronization, check out the additional instructions in Section 5.2. Note that all our instructions are tailored towards Linux. Therefore:

Important

Please use a Linux system to run the tools (Ghidra, SSH, etc.) we are about to introduce. Any distribution should do, and using a VM is perfectly fine. If you decide to develop your exploit on another OS, be aware that configuring SSH may be troublesome (especially `sshfs`), and that debugging potential issues is your own responsibility.

Grader

The grader allows you to submit your exploits and get immediate feedback. Since the grader has to evaluate the solutions of all students, they are executed in order of submission. Although you may submit as many attempts as you want, we encourage you to use the grader with caution, since delays induced by a high workload affect all students. Develop your exploits on your own machine, run them in the container, and submit them to the grader once you are fairly confident that they work. The grader considers your last working submission for your final mark, meaning you will not risk losing your points with additional submissions.

Important

Be aware that we will perform plagarism checks on your submissions. Apart from that, passing an exercise on the grader means that you get the corresponding point for your final grade.

ISL Tool

The ISL tool is a npm package that you can run on the command line of your host system to perform the following tasks:

1. Authenticate with the ISL lab environment using your ETH credentials.
2. Start and reset your container.
3. Submit exploits to the grader.
4. Show your current score.

3 Setup and Workflow

In this section, you will learn how to setup and use your environment.

3.1 ISL Tool

First, install Node.js (https://nodejs.org/en/download/package-manager/) on your Linux system. Make sure you have at least version 16.15.0 installed.

> **Important**
>
> The version provided by packet managers (like apt/snap) is usually too old, so we recommend installing Node.js from an alternative repository as shown here https://github.com/nodesource/distributions#debian-and-ubuntu-based-distributions. Using version "NODE_MAJOR=21" has proven to work fine.
>
> If the installation results in an error, check this post https://unix.stackexchange.com/questions/627635/upgrading-nodejs-on-ubuntu-how-to-fix-broken-pipe-error.

Now you can run the `isl-tool`. By using `@latest` you always get the latest version of the tool:

```
> npx isl-tool@latest help
Usage: isl-tool [options] [command]
Information Security Lab Tool (ETH Zurich)

Options:
  -h, --help                display help for command

Commands:
  login                     authenticate for access to student
    environment
  status                    check the status of your environment
  start                     start your environment
  stop                      stop your environment
  reset                     initialize or reset your environment
  submit <exercise> <file>  submit solution for grading
  results                   list the current grading results for your
    submissions
  submission <exercise>     show the submission file considered for the
    grading

                            result of an exercise
  help [command]            display help for command
```

How to use the tool:

Submit your exploit:

```
npx isl-tool@latest submit ex1a exploit1a.py
```

Show the solution currrently considered for your grade:

```
npx isl-tool@latest submission ex1a
```

Show your score:

```
npx isl-tool@latest results
```

3.2 Your Container

First, you need to connect to the ETHZ network. If you are working outside of ETH, use the VPN. Please follow the official instructions for the VPN installation (https://unlimited.ethz.ch/display/itkb/VPN).

Next, you have to authenticate yourself to our ISL infrastructure. Use your nethz credentials:

```
> npx isl-tool@latest login
? Username (nethz): username
? Password (nethz): [hidden]

Login successful!
Authentication token stored at /home/giovanni/.isl-auth-token
```

The token will be valid for three weeks, so you will need to authenticate yourself only once. Next, you can initialize your container using the `isl-tool`:

```
> npx isl-tool@latest start
Container does not exist yet, starting...
Successfully reset container!

Username:   student
Password:   blablablabla
Take note of this password! You will need to reset your container to
    reset it!

Your container is available at isl-desktop7.inf.ethz.ch on port 2001.
You can connect to it using: ssh student@isl-desktop7.inf.ethz.ch -p
    2001
```

Take note of your username, password, hostname and port.

Important

Save your password. You will need it to access your environment and to execute sudo. If you forget it, you will have to reset your container. You can change your password in the container if you like.

If you set the variables `$ISL_HOST` and `$ISL_PORT` to your values (or if you define them as environment variables), you can interact with your container like this:

- Access your container:

  ```
  ssh student@$ISL_HOST -p $ISL_PORT
  ```

- Download the handout directory ("-r") to your local machine:

  ```
  scp -r -p -P $ISL_PORT student@$ISL_HOST:/home/student/handout .
  ```

- Upload the file "exploit1a.py" to your container's home directory:

  ```
  scp -p -P $ISL_PORT exploit1a.py student@$ISL_HOST:/home/student
  ```

This concludes the minimal setup. If you do not mind to synchronize files with `scp` (or to develop your exploit in a plain-text editor directly in the container), you are ready to get started with the exercises.

However, if you prefer to work in a visual editor or or have seamless, transparent file synchronization between your local machine and your container, check out the advanced setup in Section 5.2.

3.3 Exploit Development Tools

All the binaries you are going to exploit can be run and interacted with over the command line. Exploiting them requires you to provide carefully crafted inputs, generally containing unprintable characters. To facilitate this interaction, you will not run the binary on the command line, but from within a Python script, in which you can define what you want to send to the binary and when. Every exercise comes with an exploit template.

We will show how this can be done and introduce the tools explained below in the first exercise session.

pwntools

pwntools is a state-of-the-art python framework for exploit development that contains a plethora of useful functions. It allows you to execute a vulnerable binary and send inputs to it as if you were providing them on the command line. You can receive outputs, compute and convert addresses and offsets, and do much more. As a rule of thumb, we recommend that you check the frameworks's documentation (`https://docs.pwntools.com/en/stable/`) whenever you think that a task is difficult or cumbersome. In most cases, you will find that pwntools already offers you a ready-to-use function that does just what you need.

gdb

The GNU debugger, or *GDB*, will be crucial for exploit development. It allows you to run a program, stop its execution, or observe and change its data at runtime. Use GDB to figure out what your exploit code has to look like, and step through the binary's execution when you run your exploit to find mistakes and fine-tune it. You can use GDB in the following ways:

1. Start the vulnerable binary directly with GDB: `gdb -q exercise1a`
2. Attach GDB to a running process: `gdb --pid <pid>`
3. Start GDB within your exploit script (same as one, but while running your exploit): `python3 exploit1a.py GDB`

GDB is installed in your container together with the *GDB Enhanced Features* (gef, `https://hugsy.github.io/gef/`), which provides, among others, a useful view showing you information relevant to the programs execution.

Ghidra

Ghidra `https://ghidra-sre.org/`) is an open-source reverse-engineering framework. It is powerful and comprises a lot of features, most of which you will not have to use. However, its decompiler might come in handy to reconstruct the C code of the binaries you have to exploit. Although this reconstruction is generally not perfect, it is much easier to read than plain assembly and make your work easier.

A note on ASLR

Address Space Layout Randomization (ASLR) is enabled on both your container and the grader for all binaries and is also active within GDB. Therefore, you cannot rely on memory regions (such as the stack, or linked libraries) being consistently mapped to pre-defined addresses. However, if the binary itself (e.g. exercise1a) has not been compiled as a Position-Independent Executable (PIE), then the OS cannot change the static addresses defined *in* the binary. In that case, the text section will always stay at the same memory address, even with ASLR enabled. You can check the memory layout of the process with the `vmmap` command within GDB.

4 Exercises

This module comprises a total of 14 exercises grouped in seven parts. We will announce the point distribution on Moodle in due time.

Before you proceed to the actual exercises, please keep the following points in mind:

Important

- **Please do not use "/home/student/handout" as your working directory** because its content will be overwritten when you reset your container. Instead, copy the *exploit* templates into your home directory.
- **Please leave the *exercise* files in "/home/student/handout"**: The exploits find them by absolute path (also on the grader!), and copying changes the privileges (i.e., removes root ownership).
- **Please check your files' privileges:** Remember, your exploits should be owned by `student` and your exercise binaries by `root`. Furthermore, the exercise binaries should have the `SUID` bit set. Finally, your flag should only owned by `root` and only readable by `root`.
- **Timeouts:** The grader has a default timeout of 10 seconds for all exploits. Exceptions are exercises 4a, 4b, 4c, and 6a, for which the timeout is 90 seconds.

4.1 Part 1 - Simple Overflows

In this exercise, you learn how to leverage an overflow to write data on the stack in a controlled manner.

Exercise 1a - Changing a variable (ungraded example)

This example shows you how to use a buffer overflow to change a variable on the stack.

Goal: Your exploit needs to trick the executable into calling the `uncallable` function. Do this by changing the value of the variable that `check_authorization` returns. Where can you overflow a buffer in the `check_authorization` function? Are the buffer and the variable that is returned by `check_authorization` neighboring in the stack's memory?

Info: 64-bit executable, PIE: disabled, canaries: disabled, W^X: enabled

Exercise 1b - Changing a return address

This exercise is slightly different to the previous one: now the `uncallable` function is never called, so changing the return value would not help.

Goal: Your exploit needs to make the executable call the `uncallable` function. Do so by changing the return address of the `check_authorization` function so that it returns to `uncallable` instead of main.

Info: 64-bit executable, PIE: disabled, canaries: disabled, W^X: enabled

4.2 Part 2 - Simple Overflows

As you saw in the lecture, the memory page segment containing the stack was mapped as executable until 2003. This is dangerous because it may allow attackers to write their own code on the stack, jump/return to it, and execute it. Generally, this code spawns a shell, and is therefore called a *shellcode*.

Exercise 2a - Simple Shellcode

This first exercise offers you a big buffer where there's plenty of space for your shellcode.

Goal: Your exploit needs to inject and jump (i.e., return to) some code that ultimately allows you to read the flag file. You can inject a complete shellcode and use the spawned shell to read the flag file (e.g., with *cat flag*), or a piece of code that only reads the flag file. To generate some shellcode, you can find plenty of sources and tools (e.g. `http://shell-storm.org/shellcode/`), or you can refer to the dedicated pwntools module (`http://docs.pwntools.com/en/stable/shellcraft/amd64.html`, that can help you generate one.

Info: 64-bit executable, PIE: disabled, canaries: disabled, **W^X: disabled**.

Exercise 2b - Small Buffer Shellcode

This exercise is similar to the previous one, but now the buffer, and thus the space from the start of the buffer to the return address of the `check_authorization` function is very small.

Goal: Your exploit needs again to inject and return to your (shell)code, and print out the content of the flag file. Depending on your approach in 2a), you may need to find a solution that works despite the small buffer space.

Info: 64-bit executable, PIE: disabled, canaries: disabled, **W^X: disabled**

4.3 Part 3 - Elementary Countermeasures

In this part, we work with vulnerable programs that have W^X enabled, meaning the program stack is not executable anymore. You will need to reuse code that is already in the binary and its linked libraries; however, first we will understand how to leverage leaks of memory from the stack, and use these leaks to prepare our exploits and defeat common mitigations.

Exercise 3a - Stack Canaries

One of the first mitigations against buffer overflows was the use of stack canaries, i.e., random values placed before the return addresses on the stack. Since overwriting the return address also changes the canary, the attack can be detected. Unfortunately, a canary can be defeated by vulnerabilities that leak the canary itself.

In this and following exercises, the binaries are protected by our own stack canary implementation (the default canaries of gcc are not active).

Goal: Your exploit needs to call the `uncallable` function, similarly to Exercise 1b. To do so, you need to find a way to leak the canary that protects the return address of `check_authorization` and rewrite it when you overwrite the return address. Pay attention to the overflow in this exercise: it is different from the overflow in Part 1 and 2. This new overflow allows you to overwrite arbitrary memory and does not automatically terminate with a NULL byte, i.e., '\x00'.

Info: 64-bit executable, PIE: disabled, **canaries: enabled**, W^X: disabled

Exercise 3b - Position-Independent Executables

So far, the text section of all executables were always at a fixed memory location, despite ASLR being enabled. The reason is that the binaries were not compiled to be PIEs, meaning the OS was not able to activate ASLR for the memory segments of the executable (only for libraries). In this exercise, your binary is a PIE, and addresses within in the text section change with every execution. Therefore, you will have to find a way to break this randomization.

Goal: Your exploit needs to call the `uncallable` function. To do so, you need to compute its runtime address: if you can leak the runtime address of any function of the program, you can compute the runtime address of `uncallable` by computing the offset between the two functions, and knowing that this offset dos not change even under the effect of ASLR.

Info: 64-bit executable, **PIE: enabled**, **canaries: enabled**, W^X: enabled

4.4 Part 4 - Code Reuse

As hinted, W^X defeats code injection - but does not prevent code reuse. If you can (i) find interesting functions that are loaded by the program and (ii) redirect its control flow by, e.g., overwriting a function pointer or return address, you can reuse (parts of) such functions for your evil purposes.

You will now see two simple forms of code reuse: return-to-system on 32-bit and 64-bit systems, where you need to redirect control to the `system` function in libc - you "only" need to make sure to pass parameters to `system` in the right way. For this, you will need to understand the calling conventions of x86 and x86-64, for which you can find plenty of explanations online (e.g. https://en.wikipedia.org/wiki/X86_calling_conventions#cdecl).

Exercise 4a - Return-to-libc on 32-bit

The calling convention of x86 requires parameters to be passed on the stack.

Goal: Your goal is to execute "`system(“cat flag”)`" by changing the return address of `check_authorization`. You need to prepare the stack correctly for the call to `system`, in particular passing the correct address of the argument (i.e., the address of the string "`cat flag`", that you have to write somewhere in the stack) in the expected stack position.

Info: **32-bit executable**, PIE: disabled, **canaries: enabled**, W^X: enabled

Exercise 4b - Return-to-libc on 64-bit

The calling convention of x86-64 has a twist: the first few function parameters are now passed via registers - so you need to be able to load the content you would like as parameters in registers, by using existing code in the binary.

Goal: Your goal is to execute `system(“cat flag”)` by changing the return address of `check_authorization`. You need to prepare the environment correctly for the call to system, in particular loading the address of the argument (i.e., the address of the string "`cat flag`", that you need to write somewhere in the stack) into the correct register.

Info: 64-bit executable, PIE: disabled, **canaries: enabled**, W^X: enabled

Important

If you think that your exploit is correct but you still get segmentation faults, check out the hint in Section 5.4.

Exercise 4c - Custom ROP-Chain

In this exercise, your binary does not contain the string "cat flag", meaning you cannot utilize its address to prepare the stack for a call to `system`. This reflects a much more realistic scenario, where you only have a buffer overflow vulnerability to start with and have to prepare the parameters for your call to system yourself. This can be done with return-oriented programming (ROP), which is a more generic approach of what you have seen in the last two exercises. Instead of a single call to system, you might need to prepare the stack for several calls to different gadgets. A gadget is a set of one of several useful instructions in executable memory that are followed by a `ret` instruction. Providing addresses of such gadgets in the stack can enable you to return from one gadget directly to the next, and thus put them together to produce the code you want to execute. ROP might seem a bit difficult on first glance, but you can find plenty of information on it online.

Goal: Your goal is again to execute `system(“cat flag”)`, this time by changing the return address of `get_message()`. Coming up with a ROP chain by yourself can be tedious, we recommend that you look into ways to find a chain automatically, e.g. using ropper or pwntools.

Info: 64-bit executable, **PIE: enabled**, canaries: disabled, W^X: enabled

4.5 Part 5 - Format Strings

Format string vulnerabilities are another way for an attacker to read from or write to memory. A detailed explanation of format strings can be found on `https://crypto.stanford.edu/cs155old/cs155-spring08/papers/formatstring-1.2.pdf` and `http://phrack.org/issues/59/7.html`.

Our exercises work with 64-bit binaries, which brings additional challenges. However, once you understand the principle of how format strings work on 32-bit architectures, writing the exploits should become easier (*Hint: do not try to provide the 64-bit address you want to read/write*).

Exercise 5a - Format String (Read)

In the previous exercises, you saw how to defeat a stack canary protection using a leak. In this exercise you will do the same by leveraging a format string vulnerability.

Goal: As usual, your goal is to execute the `uncallable` function that prints the flag. After having identified a format string on which you have control, try to use it to leak the value of the canary. Where is the value of the canary located? Is it passed as parameter to a function? Is it close to another known value? Once you have found the canary, you will be able to continue with a buffer overflow exploit. Since the goal is to focus on the format string vulnerability, the exploit is easier than previous exercises and it does not require to modify the return address.

Info: 64-bit executable, PIE: disabled, **canaries: enabled**, W^X: enabled

Exercise 5b - Format String (Write)

In this exercise we will see that format strings can also be exploited to write memory. In this exercise you will use a format string vulnerability to modify a variable.

Goal: As usual, your goal is to execute the `uncallable` function that prints the flag. Look at the binary; is there a variable that you can modify to call the `uncallable` function? What is the address of this variable? Where is it stored in memory? Can you write at that address and modify it?

Info: 64-bit executable, PIE: disabled, canaries: disabled, W^X: enabled

4.6 Part 6 - Advanced Challenges

Exercise 6a - Snake

Each of the previous exercises focused on a specific vulnerability inserted in a toy program, and you knew in advance which exploit technique you had to use. However, in the real world, vulnerabilities are hidden in large code bases and you do not know in advance what they look like. In this part, we provide you with a more realistic example. Despite being still fairly simple, it is a complete game written in around 250 lines of C code. It contains a vulnerability that you have to exploit. The goal of the snake game is feed the snake by leading it from fruit to fruit. After every meal, the snake grows longer, making the game more difficult. You win when the snake reaches a certain size. Have fun!

Goal: As usual, you need to print the flag by running the `uncallable function`. This time, it is up to you to find how to do that. Try to understand how the program works, what vulnerability it contains, how to exploit it to reach your goal.

Hint: this program uses a lot of function pointers. Even a small change to a function pointer could be dangerous. Hint: do not waste time writing a program that finds the right move to go to the next fruit, you can just use the command 'A'.

Info: 64-bit executable, **PIE: enabled**, canaries: disabled, W^X: enabled

Exercise 6b - Shellcode with Limited Character Set

In this part, we are building on what you have learned about shellcodes in Part 2. The following exercise contains a small program to manage notes. It is very simplistic, only allowing you to take new notes or display all notes that have been taken so far. The notes themselves are stored in notes.txt in a proprietary file format that starts with a predefined file header, i.e. a 32-byte sequence. Consequently, your input must not contain these byte patterns. If it does, it is rejected by the program. This limitation does not harm the program's intended functionality of managing notes, since most printable ASCII characters are allowed. However, it poses additional challenges when writing shellcode. Your job is to come up with a shellcode that works under these additional constraints. To solve this exercise, we recommend that you do follow these steps:

- Inspect the buffer overflow vulnerability and find the offset to overwrite the return address.
- Find a way to redirect your execution into the buffer.
- Analyze the binary to identify the forbidden characters
- Build an exploit that does not rely on the forbidden characters. We recommend that you use the shellcode from Exercise 2 as a starting point. *Hint: Think of how you could utilize the executable stack to build your shellcode or, more specifically, generate the forbidden characters that you cannot enter directly. You might want to look at the* `add` *and* `sub` *instructions.*

Info: 64-bit executable, PIE: disabled, canaries: disabled,**W^X: disabled**

Goal: You need to exploit the buffer overflow vulnerability by injecting a shellcode that reads the flag.

4.7 Part 7 - Side-Channels

Exercise 7 - Private Key Recovery

In the last part of this module, you will steal a Diffie Hellman key from an SGX enclave. Intel SGX is known for containing many side-channels. This task focuses on page faults in particular. SGX's memory is protected against direct access from applications outside the enclave, but the OS is still responsible for paging the enclave's memory. Concretely, this implies that the OS can change certain bits in the page table. The modification can induce page faults when the enclave tries to access/write/read/execute the page. The page fault is delivered to the OS, which can infer information about the enclave's execution state. Due to the hardware requirements for SGX and the complex setup for emulating enclaves, we provide traces for you.

Goal: Your task is to write a **generic** python script that is capable of extracting a secret key from **one** execution trace (series of page faults) of the binary. For simplicity, we removed all external library calls from the trace. Thus the trace only contains page faults in the binary's `text` segment itself

- The traces were generated with ASLR **enabled**. This implies that the function addresses change across different executions. However, the **relative** position within the `text` segment is fixed.
- The filename of the trace is equal to the secret key generating it.
- The page size is 4 KiB.
- $k \in [0, \texttt{G} - 1]$. Where k is the private key and `|G|` the order of the Elliptic Curve.

5 Appendix

5.1 Persistent SSH sessions with tmux

Tmux is a terminal multiplexer. We recommend using tmux because it allows to to manage multiple terminals over a single SSH connection. This is particularly useful when you want to start your debugger *within* your exploit script, as this spawns the debugger in a new terminal. This would not work over a plain SSH connection.

Additionally, if your connection is unstable and you are being disconnected from your container, your tmux session will persist, and you can simply reconnect over SSH and continue where you left off.

You can find a cheatsheet for tmux here (`https://tmuxcheatsheet.com/`).

An excerpt:

1. `tmux` (in your container) to start a new session.
2. `Ctrl-b, c` to create a new terminal.
3. `Ctrl-b, n` to move to next terminal.
4. `Ctrl-b, p` to move to previous terminal.
5. `Ctrl-b, d` to detach from the session.
6. `tmux a` (in your remote env) to reattach.

5.2 SSH Configuration Tips

Key-based SSH access

If you do not want to enter your password, port, and host every `ssh`/`scp` command, change your SSH configuration with the script below. Since copying from PDFs might cause troubles, we recommend that you use the shell script from the Moodle forum.

```
export ISL_HOST=<your hostname, e.g., isl-desktop7.inf.ethz.ch>
export ISL_PORT=<your port, e.g., 2001>

ssh-keygen -t ed25519 -f ~/.ssh/isl_id_ed25519

cat << EOF >> ~/.ssh/config
AddKeysToAgent yes
ServerAliveInterval 5
Host isl-env
    User student
    HostName $ISL_HOST
    Port $ISL_PORT
    IdentityFile ~/.ssh/isl_id_ed25519
EOF

eval $(ssh-agent)
ssh-add ~/.ssh/isl_id_ed25519

cat ~/.ssh/isl_id_ed25519.pub \
| ssh isl-env "mkdir -p ~/.ssh && cat >> ~/.ssh/authorized_keys"
```

The script does the following:

- Set host and port as local variables
- Generate a private/public key pair.
- ***Append*** an entry to the SSH config file.
- Add the newly generated private key to the `ssh-agent`.
- Copy the public key to the container.

Execute the script with `bash`. From now on you can run, for example:

```
ssh isl-env # connect to your container and open a shell
ssh isl-env <command> # run a command in your container
scp <localpath> isl-env:<remotepath> # Copy a file to your remote env
scp isl-env:<remotepath> <localpath> # Copy a file from your remote env
```

For further information or other systems, consider these guides:

- https://www.digitalocean.com/community/tutorials/ssh-essentials-working-with-ssh-servers-clients-and-keys#ssh-overview
- https://wiki.archlinux.org/title/SSH_keys#Generating_an_SSH_key_pair,
- https://learn.microsoft.com/en-us/windows-server/administration/openssh/openssh_keymanagement#user-key-generation(Windows)

Automatic File Synchronization with sshfs

Although not strictly necessary, we recommend that you use sshfs to mount your container's file system on your local Linux machine. The main advantage of this setup is that you can use your favorite editor/IDE on your own system to work on your exploits. Whenever you save them, they are immediately synchronized to your container and ready to be run.

For Linux (Ubuntu) you can use the following command, replacing your `$HOST` and `$PORT`:

```
sudo apt install sshfs
sudo mkdir /mnt/islremotefs
sudo sshfs -o allow_other,IdentityFile=~/.ssh/isl_id_ed25519 \
-p $ISL_PORT student@$ISL_HOST:/home/student/ /mnt/islremotefs/
```

Now, "/home/student/" of your remote environment is mounted on your local machine as "/mnt/islremotefs/". You can now work with the files as if they were local. However, the mapping is not persistent. If your connection fails, simply rerun the `ssfhs` command above.

5.3 Cheatsheets

pwntools

- Pack an integer into a 32/64 `bytes` object:

```
>>> p32(0xdeadbeef)
b'\xef\xbe\xad\xde'
>>> p64(0xdeadbeef)
b'\xef\xbe\xad\xde\x00\x00\x00\x00'
```

- Unpack a 32/64-bit bytes object into a an integer:

```
>>> u32(b'\xef\xbe\xad\xde')
3735928559
>>> u64(b'\xef\xbe\xad\xde\x00\x00\x00\x00')
3735928559
```

- Get the ELF (binary) of a loaded program / library:

```
>>> elf = ELF("/home/student/handout/exercise1/exercise1a")
[*] '/home/student/handout/exercise1/exercise1a'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

- Get the static address of a symbol of a loaded program / library:

```
>>> elf.symbols['check_authorization']
4198861
```

- Wait a total of three seconds for reception:

```
>>> r.recvall(3)
```

- Load ROP gadgets from an ELF

```
>>> rop = ROP(elf)
[*] Loading gadgets for '/home/student/handout/exercise1/exercise1a'
```

GDB

- Run process: `r`
- Continue process: `c`
- Show content of register eax: `i r eax / i r $eax`
- Show content at address in eax: `x/xw $eax`
- Show ten 64-bit words starting from address 0xdeadbeef: `x/10xg 0xdeadbeef`
- Set breakpoint at address `0xdeadbeef`: `break *0xdeadbeef`
- Set breakpoint at `main+55`: `break *main+55`
- Compute difference between `0xdead` and `0xbeef`: `print(0xdead-0xbeef)`
- Set variable a to 0x1000: `set $a=0x1000`

5.4 Further tips

- **Pay attention to the lifespan of your processes:** This is important when you attach GDB to a running process (i.e. `gdb --pid`) or start the vulnerable program with GDB within your exploit (`python3 exploit1a.py GDB`). For example, assume your exploit waits one second for a response from the vulnerable process, (e.g. with `recvall(1)`). If you attach a debugger, your exploit will time out, terminate, and kill the process of the binary your debugger is attached too.
- **Pay attention to your send() and recv() methods:** There exists a huge number of primitives to exchange bytes between the exploit and the running program. Some of them send a newline, some don't, some time out automatically, and

others listen forever. If you have any issues such as exploits that do not terminate or end-of-file errors, check the documentation of your send and receive functions.

- **Resolving runtime addresses with `r.libc`:** This is one of the functions that would only work if exploit and exercise binary had the same privileges: an exploit run as `student` cannot inspect the memory of an exercise run as `root`. However, you can still use `r.libc` to resolve runtime addresses if you figure out and set the base address yourself. Alternatively, the `ELF("path to library")` could be helpful.
- **(Exercise 4) Segmentation faults in `system`:** Your code may cause a segmentation fault when it tries to execute the `movaps` instruction in `system`. In that case, check if your stack pointer (i.e., `RSP`) is aligned to a 16-byte boundary, as this is required by the instruction. You can correct a misalignment by executing an additional `ret` instruction before you return into `system`. If things remain unclear, you might want to look at exercise 4 c) first and familiarize yourself with return-oriented programming.
- **(Exercise 4) Issues running "python3 exploit4a.py GDB":** You can try to start the exploit and attach a GDB as root from another terminal window, i.e.:

  1. Put a pause() in you exploit4.py somewhere after r.start().
  2. Run python3 exploit4a.py (without GDB) and learn the PID of the binary.
  3. In another terminal, run "sudo gdb -p <PID>"
  4. In gdb you can, e.g., put a breakpoint (`b check_authorization`) and then continue (c).
  5. Go back to the terminal with the `exploit.py` and press enter so that `pause()` exits and the python script continues
  6. Go back to GDB and it should have stopped `in check_authorization`.

  Important note: When your exploit terminates, so does the process of the exercise. Therefore, make sure that your exploit is kept alive as long as you need it.
- **GDB does not disable ASLR in the container:** A side-effect of your environment being run by docker is that you cannot disable ASLR (not even as root, and not even in a debugger). Thus, the NOASLR flag (which you are not supposed to use anyway) of pwntools will not work. Whenever we think that you should not bother with ASLR, we provide your with an obvious address leak on the command line.