Information Security Lab
Module 2: Software Security

Teaching Assistant: Claudio Anliker

v2025.4
October 27, 2025

Lecturer: Prof. Dr. Srdjan Čapkun

Contents

1 Introduction

Welcome to module 2 of the Information Security Lab! In this module, we focus on software security, specifically on *binary exploitation.* The goal is to understand how unsanitized program inputs can corrupt the memory of a running process and ultimately compromise an entire computer system.

1.1 Administrativa

This module has only one lecture on October 20 and one exercise session on October 21/22 (both exercise slots cover the same content). **The exercise session will be recorded, but not the lecture.**

The lecture reviews fundamental concepts of computer programs, such as memory organization, the stack, and program control, and introduces binary exploitation.

The exercise session shows you how to setup the lab environment and how to approach the challenges. We will demonstrate the necessary tools and solve Exercise 0 together.

The Thursday lab sessions provide an opportunity to ask questions about the lab setup, tools, and exploitation techniques. However, TAs are not permitted to give hints about solving the exercises that go beyond this script.

1.2 This Document

First, Section 2 introduces the attacks covered in this module on a high level. Section 3 and Section 4 guide you through the lab setup and the exercise workflow, respectively. We will cover most of this content in the exercise session, but we strongly recommend reading these sections at least once before starting with the lab. The actual exercise descriptions can be found in Section 5.

Version History:

- v2025.1 / 17.10.25: Initial release
- v2025.2 / 17.10.25: Add SUID to binaries in container
- v2025.3 / 19.10.25: Typo/error in shell command
- v2025.4 / 27.10.25: Clarification of the grader file system (Section 5)

2 On Privilege Escalation Attacks

An exploit is piece of program code or a technique that takes advantage of a vulnerability in a program to run instructions chosen by an attacker. In most cases, the attacker's goal is to fully compromise a target machine by gaining the highest level of privileges. On Linux, this is equal to running code as `root`, on Windows as `NT AUTHORITY\SYSTEM`. In this lab, we focus exclusively on Linux.

Acquiring this level of privileges typically involves a multi-stage attack. An attacker may first obtain a foothold on the target, for example by exploiting an internet-facing service or, more likely, by using phishing to steal a user's credentials. If the compromised account is not already privileged, the attacker must then perform a *privilege escalation attack* to obtain root access. **Privilege escalation can be achieved, for example, by injecting code into a vulnerable process that runs as root.**

On Linux, every binary is owned by a user, which may represent an actual person or a technical role (e.g., "www-data" for an Apache webserver). When a user launches a *binary*, the Linux operating system creates one or several *processes*. To prevent privilege escalation attacks, Linux grants a process only the privileges of the user who launched it, regardless of the binary's owner. In other words, even a root-owned binary will not run with higher privileges when started by a non-privileged user.

Unfortunately, some binaries that require root privileges need to be available to non-privileged users. Common examples include `ping` (to test network reachability), `passwd` (to change a user's password), or `sudo` (to temporarily grant elevated privileges for authorized users). These binaries all either use OS features or files that non-privileged users cannot access. Linux solves this problem with the Set User ID (SUID) bit. When the SUID bit is set on a binary, any process created from it runs with the privileges of the binary's owner. The presence of the SUID bit is indicated by an "s" in the binary file's permission string, as in:

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

In this lab, you will perform privilege escalation attacks on a Linux host by exploiting vulnerable, root-owned SUID binaries. To demonstrate that your exploit is correct, you will print the content of a file that is only readable by root (the *flag*).

3 Lab Environment

The lab environment of this module consists of three components.

- The **grader** simulates the target of your attacks. You will submit your exploits as Python files and the grader will execute them in an emphemeral, isolated environment. If the exploit prints the flag on the grader, you get the point for the corresponding exercise.
- Your personal **remote container** mirrors the grader environment and serves as your development sandbox, in which you have root privileges. It runs on our infrastructure, and is accessible via SSH.
- The **ISL tool** is a command line utility to submit solutions to the grader and to manage the remote container. You should install it on your device.

3.1 Remote Container

If you are not familiar with Docker, you can think of the container as a light-weight virtual machine (VM). Unlike a traditional VM, a container shares the host's kernel, making it fast and efficient. You do not need to know Docker for this lab.

Since we have to host containers for many students, they have *very* limited resources. Thus, you only have shell access to your container (no GUI), and you can only use it to run command-line tools with a small resource footprint. Examples are `python3` (to execute your exploit), `gdb` (to debug it), or text editors like `nano` or `vim` (to write the exploit directly in the container). If you want to use a graphical editor or other, more resource-intensive tools, we kindly ask you to install them on your own device. We will discuss file synchronization with the container in Section 3.2.3.

3.1.1 Local Alternatives

We recommend using the remote container for this module because it is virtually identical to the grader, and because it saves you from having to set up your own environment.

However, **you may solve the exercises on your own (virtual) machine at your own risk**. If you use a fresh Ubuntu 24.04 (Noble Numbat) installation, you should be fine. If you use a different Linux distribution, you should at least ensure that it uses GLIBC 2.39, as other versions are bound to cause compatibility issues in later exercises.

Important

Note that solving the exercises locally requires x86-64 hardware, which excludes most Apple devices (`https://support.apple.com/en-us/116943`).

Besides, be aware that we cannot provide support for local setups and that discrepancies between your system and the grader may affect your solutions. If your exploit works locally but fails on the grader, we expect you to debug it in the remote container.

3.2 Setup

We will demonstrate the setup in the first exercise session.

On your local device, you need to be able to run the following tools:

- SSH: to connect to your remote container
- Node.js: to run the ISL tool
- (optional) Ghidra: to decompile and analyze the binaries

These tools are available on all major platforms. However, since this is a master's level computer science course, we only provide instructions for Linux (you may use any VM of any Linux distribution). If you prefer to work on macOS or Windows, you can find well-documented installation guides online. You may also use additional or different tools if you wish.

3.2.1 Installing the ISL Tool

1. Install Node.js on your system.

Use the packet manager of your Linux distribution or download Node.js it from its website. With APT:

```
sudo apt update
sudo apt install npm -y
```

2. Connect to the ETH network:
Your remote container is only accessible from within the ETH network. If you are not connected to an ETH Wi-Fi, please connect through VPN.

3. Login using your nethz credentials:
Authenticate with the ISL tool with the following command. By using `@latest` you always get the latest version of the tool. The tool will be automatically installed on first use:

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

Login successful!
Authentication token stored at /home/<yourname>/.isl-auth-token
```

3.2.2 Setting up the Remote Container

1. Create your container

Execute the *start* command of the ISL tool and save port, hostname, and password. If you don't see any output, try again with `reset`.

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

Username:   student
Password:   <random string>
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
```

2. Change your container password

Use the connection information from above to connect to the container via SSH, and change the password with `passwd`. You will likely have to enter this password once in a while, so choose one that is reasonably secure but still easy to type/remember.

3. IMPORTANT: Setup your files.

Your home directory is `/home/student` and contains two subdirectories:

- `handout` (read-only): contains your exercise files, namely mostly exploit skeletons and the vulnerable binaries.
- `workdir`: will contain your solutions and work in progress

Important

You *must* store your work in `/home/student/workdir`. It is the only part of your container's file system that persists after a reset - the rest will be wiped and reset to default. Thus, if you store your files elsewhere, they will be permanently lost when you reset your container.

We provide exploit skeletons for all exercises. This skeletons are the only files you need to modify in this module. Thus, copy now all of them to your working directory with
`cp /home/student/handout/exercise*/exploit*.py /home/student/workdir/`

If the skeletons are read-only, make them writeable with

```
chmod +w /home/student/workdir/exploit*.py
```

Note that you should keep the exercise binaries (i.e. `binaryX`) in the handout directory. The exploit skeletons expect them there, and grader uses the same directory layout. Moving the binaries may also alter their ownership, which can break the exercises.

Edit v2025.2: *Make sure the binaries in `/home/student/handout` have the SUID bit set. You can set them for all binaries with the command*

```
sudo chmod u+s /home/student/handout/exercise*/binary*
```

If the SUID is set, then the "x" flag is replaced with an "s" as in:

```
student@env:~/handout/exercise0$ ls -lsa binary0
16 -r-sr-xr-x 1 root root 16272 Nov  1  2024 binary0
```

4. Setup the flag

On the grader, the flag and your exploits will be in the same directory. To replicate this, and to ensure that the flag can only be read by root, run:

```
sudo cp /home/student/handout/flag /home/student/workdir/flag
sudo chown root:root /home/student/workdir/flag
sudo chmod 400 /home/student/workdir/flag
```

Confirm that you cannot read the file anymore *without* sudo:

```
cat /home/student/workdir/flag
cat: /home/student/workdir/flag: Permission denied
```

3.2.3 Setting up File Synchronization

Technically, you are ready to get started with the exercises now. However, if you want to use graphical editors or tools like Ghidra, you need to transfer files between the container and your local system.

With `scp`, you can either transfer single files or complete directories with the `"-r"` flag. To download the complete handout to your host system, execute the following command (replacing the variables).

```
scp -r -p -P $ISL_PORT student@$ISL_HOST:~/workdir workdir
```

When you save an exploit on your local system, you can upload it to the container by flipping the argument:

```
scp -p -P $ISL_PORT exploit0.py student@$ISL_HOST:~/workdir/exploit0.py
```

Now that you have this command in your history, you can re-use it with `Ctrl+r` to synchronize your file(s) whenever you save them locally.

Advanced file synchronization (optional)

If you do not want to keep typing your password, port, and host everytime you use ssh/scp, you can set up key-based authentication in a few simple steps (explained in Section 6.4.1). Alternatively, you can mount your container's file system on your local device using `sshfs`, which synchronizes files automatically (see Section 6.4.2).

3.2.4 Installing Ghidra (optional)

Now would be a good time to download and install Ghidra, a powerful open-source reverse-engineering framework. Follow the instructions in the readme in the Ghidra repository (`https://github.com/NationalSecurityAgency/ghidra/`). Installing Ghidra is as easy as extracting the ZIP archive, but you need to install a Java Runtime Environment (JRE) in order to run it.

3.2.5 Resetting Your Container

As mentioned above, you can reset your container with the ISL tool's `reset` command. A reset is usually a good idea when the container becomes unresponsive or if you accidentally corrupt important files. However, please keep in mind that a reset wipes everything besides `/home/student/workdir` (see the warning in Section 3.2.2, step 3).

4 How To Solve The Exercises

We will demonstrate the steps below in the first exercise session.

4.1 Identifing the Vulnerability

Most binaries in these exercises are so small that finding the vulnerability is quite easy. Start by executing the binary a few times on the command line with different inputs. If you manage to crash it, you have probably already found the vulnerability, as your input likely overflowed its buffer and corrupted neighboring memory.

To get a better understanding of the binary, you can decompile it with Ghidra. The resulting C code is not quite correct but it is much easier to read than assembly.

Finally, run the binary with GDB with `gdb binary0`. GDB is a command line debugger that allows you to step through execution and get a more detailed understanding of the binary. GDB may seem a bit daunting at first, but knowing a few useful commands already goes a long way, which is why we compiled a cheatsheet in Appendix 6.5.2.

4.2 Using the Exploit Skeleton

Exploiting a binary by entering inputs directy on the command line is cumbersome, especially when they contain non-printable characters. Instead, we use *pwntools*, a Python framework for exploit development. Each exercise includes a pwntools-generated exploit skeleton that automatically starts the corresponding binary and connects to it via a *pipe* (the object `io`). The pipe can be used for all input/output with the binary, and the exploit can simply be written in Python.

A good first step is to implement the command-line interaction with the binary in Python, using functions like `io.sendline()` and `io.recvline()`. To get a more detailed output, you can add the `DEBUG` flag as in:

```
python3 exploit0.py DEBUG
```

As a rule of thumb, we recommend that you check the pwntools documentation (`https://docs.pwntools.com/en/stable/`) whenever you feel that a (sub-)task is difficult or cumbersome. In many cases, you will find that pwntools already offers you a ready-to-use function.

4.3 Debugging

The actual exploit development involves *a lot* of command-line debugging. We compiled a list with useful GDB commands in Appendix 6.5.2, and you can find many sources online for more details.

It is often helpful to run the exploit *and* attach GDB to the process of the binary, as this allows you to analyze the effect of your exploit on the binary step by step. The

easiest way to do this is to open two terminal windows with two concurrent SSH sessions, one to run the exploit and one to interact with the debugger:

1. In the first window, add two pauses to your exploit, such as `pause()` or `input("<some string>")`, one after `io.start()` and the other at the end of the file.
2. Run the exploit. This starts the binary and prints its process ID (PID) to the terminal. Then, the exploit pauses, giving you time to attach GBD.
3. In the second window, attach GDB to the binary by running "`sudo gdb -p <PID>`". You need sudo because the binary runs as root (which is not the case if you run it directly with GDB using `gdb binary0`).
4. In GDB, add a breakpoint (for example, `gef> b check_authorization`). Then, let the process continue with `c`.
5. Go back to the first terminal and press Enter to end the `pause()` in the exploit. Now, you should hit your breakpoint in GDB, allowing you to step through the binary's process.

The purpose of the second pause in the end is to prevent your exploit from terminating automatically when it hits the end of the file. Since the exploit started the binary's process, it will also kill it when the python script terminates, thus, ending your debugging session.

Important

Note that we do not recommend pwntools' `GDB` parameter, as it tries to open GDB in a new terminal. This is not possible over SSH in your current configuration. If you want to use this feature, you can use tmux (see Section 6.3) or one of the local setups discussed in Section 3.1.1.

4.3.1 Submitting your solutions

You can submit your solutions with the ISL tool:

```
npx isl-tool@latest submit ex0 exploit0.py
```

We recommend using the ISL tool on your local system, especially if you already write your exploits there. The ISL tool can be used in the container too, but installing it will take a moment due to the container's limited resources.

To grade your submission, the grader simply runs

```
python3 exploitX.py
```

and checks if the output contains the flag. If that is the case, you will get the point for the corresponding exercise. Thus, your exploit does not have to work reliably but only once, and you cannot lose it anymore due later erroneous submissions.

We recommend testing your exploit in the remote container and only to submit to the grader once it works. You can submit as often as you like, but there may be rate-limiting in case of high load or if you still have a submission running (e.g., if it runs indefinitely and has not reached the timeout yet), as the grader must be available to all students equally.

Important

If your exploit does not work on the grader, please check the points listed in Section 6.1.

5 Exercises

This module comprises a total of 8 exercises. In all of them, your goal is to exploit the corresponding binary and use its root privileges to print the content of the file `/home/student/workdir/flag`, which is only readable by root. You can run your exploits with `python3 exploitX.py`.

Edit

v2025.4 - Clarification of grader file system: In your container, your exploits and the flag should be in "`/home/student/workdir`". On the grader, this directory does not exist, and the files are put into `/home/student` instead. Note that this difference does not affect your exploits, as long as you follow these two rules:

1. Run the exploit *in* the directory containing your flag:

```
student@env-XX-XXX-XXX:~/workdir   python3 exploitX.py
```

2. Access the flag directly by name and not by its absolute path.
Example: "`cat flag`" and not "~~`cat/home/student/workdir/flag`~~".

Doing so ensures that your exploit finds the flag both in your container and on the grader.

Important

Please do not ideas or parts of solutions to fellow students, and do not make them publicly available. All exercises must be solved independently, and submissions will be checked for plagiarism.

5.1 Changing a Program's Control Flow

In this exercise, you learn how to leverage buffer overflows to write data on the stack.

5.1.1 Exercise 0 - Changing a variable (ungraded example)

We solve this example together in the first exercise session.

This binary contains a function to print the flag, but it is never called. Your goal is to change this by making `check_authorization` return a non-zero value. Execute your exploit with `python3 exploit0.py`.

Info: 64-bit executable, no PIE, canaries disabled, stack not executable
Grader timeout: 10 seconds

5.1.2 Exercise 1 - Changing a return address (1 point)

This binary ignores the result of `check_authorization` completely. Thus, you will have to find another way of redirecting the control flow to `print_flag`.

Info: 64-bit executable, no PIE, canaries disabled, stack not executable
Grader timeout: 10 seconds

5.1.3 Exercise 2 - Address Space Layout Randomization (1 point)

Modern operating systems try to randomize runtime addresses. This technique is called *Address Space Layout Randomization* (ASLR) and is an important defense against binary exploitation.

The OS can always randomize the memory layout of linked libraries or dynamic regions, like the stack. However, it can only randomize memory sections defined in the binary (like the `.text` section containing the binary's machine code) if the binary was compiled to support this relocation. Such binaries are called *Position-independent Executables* (PIEs). With modern compilers, binaries are usually PIEs by default.

In contrast, the binaries of Exercises 0 and 1 were *not* PIEs: they relied on hard-coded addresses. Thus, functions like `print_flag` were always at the same location in memory.

In this exercise, the binary is a PIE. Although it still contains a `print_flag` function, you will now need to determine the function's address at runtime.

Info: 64-bit executable, PIE, canaries disabled, stack not executable
Grader timeout: 10 seconds

Tipp

- You can concatenate `bytes` in Python with `+`, e.g., `b'\x00\x01' + more_bytes`.
- You can convert an 8-bytes `bytes` object to a 64-bit `int` and back with `u64()` and `p64()`, respectively.

5.2 Shellcode

In this exercise, the binary does not contain a `print_flag` function, so we have to inject our own. In the early days, operating systems used to map stack memory as executable, which allowed hackers to write machine code onto the stack and redirect the control flow to that code. As this code often spawns a shell, it is referred to as *shellcode*.

5.2.1 Exercise 3 - Shellcode (1 point)

Use the buffer overflow vulnerability to inject shellcode onto the stack to print the flag, and redirect the binary's control flow to this code.

To learn more about shellcode, you can find plenty of resources online (e.g. `http://shell-storm.org/shellcode/`). In this exercise, you can either inject shell code that reads the flag directly, or you can spawn a shell and use it to read the flag (i.e., by sending *cat flag* to your shell over your exploit's pipe).

Info: 64-bit executable, PIE disabled, canaries disabled, stack executable
Grader timeout: 10 seconds

Tipp

Split the task into two parts. First, find out where your inputs are stored and how you could redirect execution to that location, using what you have learned so far. Execute the binary in GDB and keep your eyes on the stack.

Then, for shellcode generation, you may use tools like the pwntools module *shellcraft* (`http://docs.pwntools.com/en/stable/shellcraft/amd64.html`). However, we strongly encourage you to step through your shellcode in GBD to understand how it works.

5.2.2 Exercise 4 - Short buffer (1 point)

In this exercise, the buffer is too small to hold meaningful shellcode. However, all is not lost: you can still use the shellcode from the last exercise, but you have to figure out where to place and how to reach it.

If your solution for Exercise 3 also works for this exercise, you may resubmit it.

Info: 64-bit executable, no PIE, canaries disabled, stack executable
Grader timeout: 10 seconds

5.3 Format String Vulnerabilies

The exercises so far were designed to simplify the leakage and injection of data. In practice, this can be more difficult, since functions working with C strings like `scanf` and printf terminate strings with null bytes, ignoring following characters.

This is where format string vulnerabilities can come in handy; they also allow to read from or write to memory, but they are not affected by null bytes. Detailed explanation of format strings can be found online*.

In a nutshell, some functions in the C standard library support format specifiers to format in- and output. Examples are `"%s"` (for C strings) or `"%d"` (for decimal numbers). Format strings can be used like this:

```
printf ("var is %d in decimal and %x in hex", var, var);
```

* We recommend `https://crypto.stanford.edu/cs155old/cs155-spring08/papers/formatstring-1.2.pdf` and `http://phrack.org/issues/59/7.html`

Note that the data replacing the format placeholders is provided in additional function parameters. This is where it gets tricky: If an attacker can enter a string with format specifiers, the program assumes additional parameters that do not exist. Thus, the program uses whatever data is at the corresponding location (registers or stack). For example:

```
printf(some_var);       // Attacker sets some_var="show me %d" ...
printf("show me %d");   // ... causing a call like this...
printf("show me %d", some_leaked_data) // ... which results in a leak.
```

5.3.1 Exercise 5 - Leaking Credentials (1 point)

In this exercise, you have a binary that prints the flag if you provide the correct username and password. However, both are set randomly at runtime. You have to leak the credentials first, then enter enter them in order to pass authentication.

Info: 64-bit executable, no PIE, canaries disabled, stack not executable
Grader timeout: 10 seconds

Tipp

- Look up where parameters for function calls are stored in the System V AMD64 ABI calling convention (`https://wiki.osdev.org/System_V_ABI`) and verify this in GDB. This will help you to to draw conclusions about the format strings you need to use.
- The format specifiers "`%x`" and "`%d`" access data differently than "`%s`".

5.4 Return-Oriented Programming

Under normal execution, the stack only maintains a program's state and does not contain executable code. Consequently, marking it non-executable (by changing the permissions of the containing memory page) is a straightforward countermeasure against binary exploitation and is deployed in all modern operating systems.

However, placing executable code on the stack is not the only way to "inject" functionality into a running process; in *Return-Oriented Programming (ROP)*, the attacker redirects execution to existing code snippets of the program or loaded libraries. These snippets are called *gadgets* and usually contain only one or a few instructions and end with a control flow instruction like `ret`, which gives the technique its name. For example, the end of every function could serve as a gadget. If an exploit redirects the CPU to the first gadget, the gadget is executed until the program hits the `ret`, at which point the CPU reads the next address from on the stack. Thus, if you put the addresses of more gadgets on the stack, they can effectively be chained together into one bigger piece of code. This is a generalization of your exploit in Exercise 1, where `print_flag` was the only "gadget" necessary.

5.4.1 Exercise 6 - Simple ROP (1 point)

Your goal is to print the flag by redirecting execution in `check_authorization` and calling `system("cat flag")` or a similar function. This time however, the program does not contain a `print_flag` function.

Tipp

- Use GDB to analyze the assembly code of previous programs that called `system`. This should help you figure out what your ROP chain has to do. In particular, pay attention to the additional `ret` instruction (stack alignment).
- **In case of segmentation faults:** At some point, the `system` function executes the `movaps` instruction, which requires the RSP register to be 16-byte aligned (otherwise it causes a segfault). If this happens, you can add a simple `"ret"` instruction to your ROP chain before calling `system` - this will align RSP and prevent the segfault. However, if the segfault happens anywhere else, the cause is likely a bug in your exploit.

Info: 64-bit executable, no PIE, canaries disabled, stack not executable
Grader timeout: 90 seconds

5.4.2 Exercise 7 - A Complete ROP attack (2 points)

The binary in this exercise is compiled to use *stack canaries*. A stack canary is a random value on the stack, just before the return address. Overwriting the return address means overwriting the canary, and a corrupt canary results in program termination. Therefore, one has to find a way to learn the canary and inject it where necessary to make an exploit work.

In this exercise, we're putting all the pieces together: besides the stack canaries, your binary is a PIE, does not have an executable stack, and does not contain any "gadget gifts" to simplify your ROP chain. Combine the techniques you have learned to bypass all countermeasures and print the flag.

Info: 64-bit executable, PIE, canaries enabled, stack not executable
Grader timeout: 90 seconds

Tipp

Solve the exercise one step at a time: First, try to figure out how the stack canaries work and how you can bypass them. Then, think about the ROP chain: which addresses you have to leak and where can you find them?

6 Appendix

6.1 Exploit is not working on the grader

Please check that

- your exploit **works on the remote container**, not just locally.
- your exploit prints the flag without the `"DEBUG"` parameter.
- your exploit does not contain any `pause()` or similar statements that require user interaction.
- your exploit is in `/home/student/workdir`, runs as `student` (i.e., owned by `student` and you are not running it as root or with sudo).
- your flag is in `/home/student/workdir/flag` and only readable by root (see the Section 3.2.2, point 4).
- your binary is in `/home/student/handout`, owned by `root`, and with SUID set.
- your exploit does not time out (check the timeout listed in the corresponding exercise).

6.2 ASLR in the container / grader

Address Space Layout Randomization (ASLR) is explained in the lecture slides and in Exercise 2. **For security reasons related to Docker, ASLR is always enabled in the container and on the grader, even when you start a program in GDB.**

Outside of Docker (in a Linux VM, for example) GDB usually disables ASLR if you start the binary with GDB, as in `gdb binary0`. This is because ASLR adds unnecessary complexity if your objective is only to debug a process, and starting a program with GDB drops root privileges anyway. Notably, GDB cannot disable ASLR when it attaches to an already running process.

The first exercises are structured so that ASLR does not affect your work. In Exercise 2, you will learn techniques to bypass ASLR.

Tipp

The `vmmap` command in GDB shows the memory mapping of your process and how ASLR works.

6.3 Persistent SSH sessions with Tmux

Tmux is a terminal multiplexer, meaning it helps you manage multiple terminals over a single SSH connection. This is particularly useful when you want to start your debugger *within* your exploit script with pwntools' `GDB` parameter, as Tmux spawns the debugger in a new, split terminal.

Additionally, if your network connection to the container is unstable and your SSH session terminates, the tmux session on the container will persist, and you can simply reconnect over SSH and continue in your tmux session 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 container) to reattach.

6.4 SSH Configuration Tips

6.4.1 Key-based SSH access

Key-based authentication means you will not have to type your user password anymore. We provide a setup script on moodle (`ssh_setup.sh`). Additionally, the script adds an entry to your local SSH config that stores your key identifier, host, and port, allowing you to connect to your container with "`ssh isl-env`".

First, set the host/port variables at the top of the script to your values. Then, make sure the script is executable (`chmod +x ssh_setup.sh`).

The script creates a key pair, creates the config entry, and uploads the public key. Remember that if you protect your private key with a password (you can leave it empty), you will have to type that password whenever you use the key. Finally, the script creates a backup of your container's SSH directory in `/home/student/workdir/` to restore the uploaded public key after a potential container reset.

If the script crashes, make sure to remove the dangling entry from your `~/.ssh/config` and delete the key pair `~/.ssh/isl_id_ed25519(.pub)` before running it again.

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 to do the setup on 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)

6.4.2 Automatic File Synchronization with sshfs

You can use sshfs to mount your container's home directory on your local machine over SSH. The main advantage of this setup is that you can work on your local system and synchronize transparently without issuing "scp".

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/`, and you can access all your files locally. Note that this mount is not persistent and has to be renewed whenever your connection fails.

6.5 Cheatsheets

6.5.1 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/binary1")
[*] '/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/binary1'
```

6.5.2 GDB

- Run process: `r`
- Continue process: `c`
- Show content of register rax: `i r rax`

- Show content at address in eax: `x/xw $rax`
- Show ten 64-bit words starting from address 0xdeadbeef: `x/10xg 0xdeadbeef`
- Show ten instructions starting from address 0xdeadbeef: `x/10xi 0xdeadbeef`
- Show ten ASCII strings starting from address 0xdeadbeef: `x/10s 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`
- Show address of `main`: `info address main`
- Show memory mapping of process: `vmmap`
- Show runtime address of a function: `info function <name>`

6.6 Further tips

- **Beware of different send() and recv() methods:** There exists a vast number of different versions of send and receive functions. Some of them send a newline, some don't, some time out automatically, and others continue listening forever. If you have issues with exploits that do not terminate or raise end-of-file (EOF) errors, check the documentation of your send and receive functions.
- **Resolving runtime addresses with `io.libc`:** This object can be used to resolve runtime addresses of symbols in a process. This does not work in our context, since the binary runs with higher privileges (`root`) than the exploit (`student`). However, if you find the base address of libc through a leak, you can configure io.libc with this base address to easily compute runtime addresses.
- **Segmentation faults in `system`:** If you call `system` and the program runs into a segmention fault at a `movaps` instruction, it is likely because your stack pointer (i.e., `RSP`) is not aligned to a 16-byte boundary. This alignment is a strict requirement for `movaps`, and you can prevent the segmentation fault if you correct the misalignment by executing an additional `ret` instruction *before* you return into `system`.