# 1. Introduction

This book is based on the lectures notes of the course of *System and Network Hacking* of the University of Pisa. The course is about learning to think like an attacker in order to become a better engineer, researcher, or defender.

The book covers only the "System" part of the course. Check the following url for updated versions of the book and other information about the course:

https://lettieri.iet.unipi.it/hacking

The common theme of the course are *bugs*, how they can be exploited, how they can be prevented, and how their exploitation can be mitigated.

> A common story claims that U.S. Navy Rear Admiral Grace Hopper coined the term "bug" to refer to computer errors after finding a moth lodged in a relay of the Harvard Mark II in 1947. She removed the moth—the first instance of debugging!—and taped it into the logbook with the annotation, "First actual case of bug being found" [65].
> Many "origin stories" are myths, and this is one of them. The episode itself is historical, once we correct the protagonist. Hopper was the one who popularized the story and donated the 1947 logbook, moth still attached, to the Smithsonian Institution [67]. Yet, she never claimed to have found the insect or written the logbook entry herself, always attributing it to her colleagues on the Mark II team. The true myth is the connection of this event to the origin of the term "bug" for a technical defect. In reality, the term had described glitches in telegraphy since at least the 1870s, and Hopper herself acknowledged its common use among radar technicians by the 1940s [18]. Moreover, as the Jargon File acutely states, the way the logbook annotation is worded "establishes that the term was already in use at the time in its current specific sense" [103]. What does this mean? Let's read the annotation again: "First *actual* case of bug being found." (Emphasis added). It was the first time they found a literal insect instead of—we must presume—a hardware or software defect. It's a pun, a programmer's joke, not a pivotal event.
> However, the desire to give this story historical significance remains strong. The current trend is to claim that the episode popularized the term in the larger computing community thanks to Hopper's retellings [13]. Personally, I find this more cautious claim to be weak as well. Hopper certainly told the anecdote at many conferences to get a laugh. But the pun only works if the audience's primary understanding of "bug" is already the technical one. The fact that people laughed proves they were already in on the joke.

R The one above was a *sidenote*. Sidenotes contain mostly historical annotations and sometimes curiosities that you can safely skip. The one you are reading now, instead, is a *remark*. Skipping these is somewhat less safe.

The book is divided into three parts that descend from higher to lower software layers relative to user programs.

- Part I deals with bugs occurring above the user programs, at the user-facing interface.
- Part II deals with bugs inside the user programs.
- Part III addresses bugs that occur below the user programs, in the support system.

Part II, and most of Part III, deal with the important topic of *memory corruption bugs*. These bugs have a long history. Hackers realized that seemingly innocent bugs could be exploited to compromise an entire system. Then, countermeasures for these exploits were developed, only to find that these countermeasures could be bypassed. For didactic reasons, the book will roughly follow this historical development. Due to the arms race, things have become progressively more complex over time. Therefore, it makes sense to start with the simpler, older techniques and progress to more complex, newer ones.

> Here, we use the word "hackers" in the historical, neutral sense [8]. Throughout the book, we use the term "attackers" to refer to those who intend to harm a system.

Many of the techniques you'll practice were "solved problems" years ago in production systems. Stack smashing, format string attacks, and simple return-to-libc payloads are rarely effective on a modern, fully patched distribution. So why study them? Because they teach fundamental ideas: how memory is laid out, how the toolchain transforms source into machine code, and how defenses such as stack canaries, ASLR, or pointer mangling work. Without these foundations, modern exploit development and defense strategies will look like black magic.

## 1.1 Setting up your environment

I'm firmly convinced that you cannot really understand something until you actually do it, so the focus of the course is on practical, hands-on activities: you will compile code, crash programs, analyze memory, and sometimes even break into the kernel. Along the way you will see where systems fail, and more importantly, why they fail.

The price to pay is that we can examine only one system. In particular, you need an environment with Linux for x86_64. Any recent distribution will work, including Ubuntu or Kali.

If you have an x86_64 PC/laptop you have several options:

- Run Linux natively, either exclusively or in dual-boot, or with a Live USB.
- Install Linux in a VM (e.g., using VirtualBox or VMware).
- Rent a cheap x86_64 virtual server from a cloud provider.

The Windows Subsystem for Linux (WSL) should work too, for user-space exercises. Ensure you are using WSL 2 as it utilizes a real Linux kernel and will work for most exercises. WSL 1 may have limitations for low-level topics. For kernel module development or low-level kernel exploitation labs (Part III) prefer a full VM or cloud VM.

If you only have an Apple Silicon (ARM) Mac, you cannot run x86_64 Linux natively or in a fast VM (no, you can't). Your best options are:

- Rent a cloud-based x86_64 VM.
- Use a software emulator like UTM [125], accepting that it will be slower.

To make UTM run faster, consider installing Linux without a graphical user interface (e.g., Ubuntu

server) and use a shared folder for file exchange. Once your environment is running, I highly recommend connecting to it via SSH from your host machine. This makes cut-and-paste seamless and allows you to open multiple terminals easily. Note that x86_64 is needed for part II and most of part III. For part I and the network/web part of the course, Linux for ARM is sufficient. For these parts, you can install a Linux/ARM distribution in a fast VM such as VMware.

## 1.2 Prerequisites

The languages used in the course are C, x86_64 assembly using the Intel syntax, bash and Python (and a tiny bit of C++). It is assumed that the student already knows C. Basic familiarity with bash is also assumed. Previous exposure to Python 3 is useful, but probably not as important, as it can be acquired along the way. Appendix A can be used as a recap of the x86 architecture, Intel assembly and the ELF file format, which are needed starting from Part II.

We will make extensive use of the debugger, so it is a good idea to familiarize with it. In particular, we will use GDB with the *pwndbg* extension [121].

## 1.3 Capture The Flag

A solution to each exercise is provided in Appendix D. Some of the exercises are actually Capture The Flag (CTF) challenges, in which you must exploit a bug to steal a secret from a remote system. You can identify the CTF exercises because they have names. All secrets are strings of the form "`SNH{...}`". There is CTFd site reachable from this url:

https://lettieri.iet.unipi.it/snh-ctfd-25.

You need an account to access it. Ask the instructor if you don't have one. The site contains the instructions and the supplementary files for each CTF challenge. The challenges are enabled during the course, and some of them are shown only after you solve a previous one. Each challenge has a score. When you select a challenge, a pop-up will open showing the score, instructions, and the command you should use to start interacting with the vulnerable binaries. There are two kinds of challenges:

If the instructions contain a command like the following:

```
ssh -p port challenge@lettieri.iet.unipi.it
```

then you need to connect via SSH using this command. When prompted, enter the provided password. Some of these challenges will greet you with a message like this:

```
Welcome to challenge. Session: SESSION-CODE
```

If you see this message, then you have a private home directory on the remote system where you can create your own files. Write down the session code; you can use it to reconnect with a command like this:

```
ssh -p port challenge@lettieri.iet.unipi.it -t SESSION-CODE
```

This is useful for logging out from the challenge and reconnecting later, to continue working on it. It can also be useful for connecting multiple times concurrently to have several terminals active at the same time.

- If you see something like

```
nc lettieri.iet.unipi.it port
```

you need to establish a TCP/IP connection to the specified host and port. You can do this from the shell using the netcat (`nc`) utility suggested in the command or by writing a program in your preferred programming or scripting language (Python is highly recommended). These challenges use a vulnerable server that `fork()`s a new process for each connection.

After capturing the flag, enter it in the pop-up window of the challenge. The site will verify that you have actually captured the correct flag and will keep track of your score. There is nothing to win or lose, so don't cheat.