Information Security Lab
Module 4: Software Security

Teaching Assistants: Dr. Giovanni Camurati, Claudio Anliker
Lab Environment: Carl Friess

v1.1
November 8, 2022

Responsible: Prof. Dr. Srdjan Capkun

Contents

History:

- 1.0 / 04.11.22: Initial release
- 1.1 / 07.11.22: Added grader timeouts and cheatsheets, minor corrections

1 Organisation

Welcome to Module 4 of the Information Security Lab! The next two weeks are about software vulnerabilities. You will learn how software security mechanisms work and how one can exploit vulnerable programs to obtain root access on a target system.

1.1 Lecture

The lecture will cover the theory and concepts behind software security mechanisms and attacks. Within the lecture, we will introduce the lab environment and describe your assignment on a conceptual level.

1.2 Exercise Sessions

In the exercise session, we will focus on your assignment. We will show you how to work with our new lab environment and how to use the tools you need to develop your exploits.

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

2.1 Goal of this Module

The goal of this lab is to understand memory corruption vulnerabilities. These vulnerabilities allow an attacker to change and overwrite the memory of an affected process. They can, for example, modify data (e.g., variables), or divert the program's control flow (e.g., through return address manipulation). If the attacker is able to inject their own code and direct the program's execution to it, they can take over the process. If the process runs as root, it can be used for a *privilege escalation* attack.

Imagine a scanario in which an attacker has already established low-privileged access to a system and, in a next step, wants to obtain root access. One way to do this is by exploiting binaries that are run as root (i.e. owned by root and with `suid` bit set) and affected by a vulnerability. To that end, the attacker can set up a machine that resembles the target regarding software versions and configurations, and run the vulnerable binary there. This facilitates exploit development tremendously, since the attacker may debug and reverse-engineer the binary in this setup. The resulting attack code can then be copied to and run against the target system. If the exploit manages to spawn a shell on the target, then this shell will be owned by root, giving the attacker full control over the machine.

In this module, you will find yourselves in the shoes of such an attacker, essentially following the workflow explained above. However, you don't have to spawn a shell to proof that you elevated your privileges. Instead, we ask you to print the content of a *flag* file, which is only readable by root.

2.2 Lab Environment

This section should give you an overview of the lab environment and the exploit development workflow. More detailed explanations for the setup can be found in the next section.

Every exercise is based on a different, vulnerable binary that you have to exploit. We give you these binaries and your own, individual machine (or *container*, more on that shortly), which is under your full control. You can use your container to develop your exploit and, once it works, run it against a target system. However, in contrast to the attacker described above, you will not get shell access to the target for practical reasons. Instead, you will have to submit your exploit to a *grader* machine. The grader will run your submission for you, with restricted privileges and against the same vulnerable, root-owned binary. If executing your exploit on the grader prints the flag, you have successfully elevated your privileges and get the point for that exercise. This allows you to obtain immediate feedback about the result of your code.

Container

The purpose of your container is to provide you with an environment that is as close to the target machine (i.e. the grader) as possible. This has the advantage that the exploit that you develop within your container also fits the target machine, without you having to run a virtual machine or adapt your own system.

Important

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

1. You have only shell-based access to the container (SSH).
2. 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.
3. We **strongly recommend** that you develop your exploit on your own machine, not in the container.

The last point might sound a bit inconvenient, but we will show you how you can interact with your container very efficiently in section 3.2. If set up correctly, you will hardly notice that you are working remotely. Furthermore, this allows you to use whatever IDE or text editor you prefer on a system you are comfortable with.

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

Important

Please make sure not to use '/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. You can leave the exercise files in the handout, since the exploits find them by absolute path and you do not have to change them anyway.

Grader

The grader allows you to submit your exploits and get immediate feedback. Since the grader is shared by all students, submitted exploits 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 carry out plagarism checks on your submissions.

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 your system for the lab.

3.1 Local Machine

Please install node (https://nodejs.org/en/download/package-manager/) on your own system. Make sure you have at least version 16.15.0 installed. Now you can run the isl-tool. By using @latest you always get the latest version of the tool, which makes it easy for us to deploy updates.

```
> 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
```

3.2 Access your Container

First, you need to connect to the ETHZ VPN. Please refer to the official instructions (https://unlimited.ethz.ch/display/itkb/VPN).

You can now login using 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 two weeks, so you will need to login only once (unless you change machine or delete the token). Next, you can initialize your remote environment:

```
> npx isl-tool@latest reset
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 only be able to change it by resetting your environment. Even if you setup ssh keys, you will need this password for sudo.

You can now SSH into your remote environment using the information above. Make sure to specify the right port.

Key-based SSH access

We recommend that you use key-based SSH access to facilitate working with the container. For Linux, you can use the following script, which is also available on Moodle:

```
export USER=student
export HOST=<your hostname, e.g., isl-desktop7.inf.ethz.ch>
export 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 $USER
  HostName $HOST
  Port $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 ssh-keygen command will ask you to set a passphrase for your key. Take note of this passphrase, you will have to remember it.

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)

From now on you can run, for example:

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

Mounting a remote file system over SSH

Although not strictly necessary, we strongly recommend that you use sshfs to mount your container's file system on your local 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 synched to your container and ready to be run.

In short, for Linux (Ubuntu):

```
export USER=student
export HOST=<your hostname, e.g., isl-desktop7.inf.ethz.ch>
export PORT=<your port, e.g., 2001>
sudo apt install sshfs
sudo mkdir /mnt/islremotefs
sudo sshfs -o allow_other,IdentityFile=~/.ssh/isl_id_ed25519 -p $PORT
    $USER@$HOST:/home/student/ /mnt/islremotefs/
```

For other systems, consider the guide on https://phoenixnap.com/kb/sshfs.

Now, the `/home/student/` of your remote environment is mounted on your local machine as `/mnt/islremotefs/`. You can now run one of the following commands:

```
ls /mnt/islremotefs # List files in your remote environment
echo "print('Hello from remote env')" >> /mnt/islremotefs/test.py #
    create a python script
ssh isl-env python3 /home/student/test.py # run the python script
    remotely
```

tmux

Tmux is a terminal multiplexer. When working within your container via SSH, it is very convenient to use tmux because of two reasons: (i) if your connection is unstable and you are being disconnected from your container, your tmux session will stay alive and (ii) you can create one SSH session and then use tmux to manage multiple terminals. This is particularly important when you want to start your debugger within your exploit script, which spawns the debugger in a new terminal.

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

Minimal tutorial:

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.

Important

To copy/paste within tmux, press `Ctrl+Shift+C` *before* you mark text with your mouse (note that it might be a different key combination on Mac and Windows). Pasting then works with `Ctrl+Shift+P`. This is necessary because the tmux mouse integration is enabled to allow scrolling with your mouse wheel. Feel free to adapt your tmux configuration (`/home/student/.tmux.conf`) if you see fit.

Container Setup

We give you root access to your container and you can generally install Ubuntu packages, python packages, and other tools. However, everything you need should already be there. Furthermore, with every change that you apply, your system deviates more from the grader, which we generally try to avoid. Also keep in mind that your container only has very limited resources.

Important

Install as little software as possible in the container. Use your own system and SSHFS to write the exploits.

3.3 Workflow and Exploit Development

Here is our suggested workflow:

- Connect to the VPN and to your container
- Create a copy of the handout on which you will work (copy the exploits in the home directory of your environment). Keep in mind that the handout folder will be wiped on container reset.
- Write the exploits on your local machine. You can then either copy them to your environment with scp, or, better, mount a remote file system with ssh as explained in this guide.
- Open a shell to your container with tmux. There, you can run binaries, debug them with gdb, and run your exploits.

To submit your solution, the following commands are available:

Submit your exploit:

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

Show the solution relevant for grading:

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

Show your score:

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

3.4 Exploit Development

All the binaries you are going to exploit can be run and interacted with over the command line. Exploiting them requires you to provide them with carefully crafted input, generally containing unprintable characters. To facilitate this interaction, your exploit will be 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 script that can be run on the command line. The exploit script will start the vulnerable program by itself and provide you with the means to interact with it.

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 addressed 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 13 exercises grouped in six parts. For all exercises, the command run on the grader will be

```
python3 exploit<nr>.py
```

Important

All sub-exercises until 5b) are worth 1 point, exercises 6a) and 6b) are worth 2 points each. This results in a total of 14 points. You only need 12 points for full marks (corresponding to 100 "module points").

Exercise 1a) is *not* graded, as it will be used as an introductory example in the exercise session.

Edit

07.11.22: 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

The first two executable files you have to work with are meant to show you how to leverage an overflow to write data on the stack in a controlled manner.

Exercise 1a - Changing a variable (ungraded!)

This simple exercise shows you how to change values of other variables on the stack thanks to a buffer overflow.

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 second simple exercise has a slight change compared to 1a: 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 the previous location in the main.

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

4.2 Part 2 - Simple Overflows

As you saw in the lecture, until 2003 memory pages on the stack used to be executable. The immediate danger posed by this design is that attackers with an overflow of a vulnerable buffer could write their own code in the stack, and change the return address to make the program return to that code and execute it. Very often, 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

We saw in the lecture that one of the first mitigations against buffer overflows was the use of stack canaries, i.e., random values that protect the return address of functions from being overwritten by an overflow. Unfortunately, a canary can be defeated by vulnerabilities that leak the portion of the stack containing its value.

Goal: Your exploit, similarly to Exercise 1b, needs to call the `uncallable` function. 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. Observe carefully the overflow in this exercise: it is different from the overflow in Part 1 and Part 2. This new overflow allows you to overwrite arbitrary memory and does not automatically terminate the strings – recall that functions that print strings, such as printf, assume that a string is terminated by 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 they OS was not able to activate ASLR for the executable itself. 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: disabled, 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: disabled, W^X: enabled

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 as a starting point.*

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.

5 Appendix

5.1 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`

tmux

- Vertically split pane: `Ctrl+b, %`
- Horizontally split pane: Ctrl+b, "
- Switch focus between panes: `Ctrl+b, <arrow keys>`
- Enter scrolling mode (then scroll with arrow keys): `Ctrl+b, [`

5.2 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.