Chair of IT Security
Brandenburg University of Technology

Asya Mitseva, M.Sc.
Fabian Mildner, B.Sc.
Prof. Dr.-Ing. Andriy Panchenko

Summer Term 2025

Part III: System Security

Deadline: 4th June 2025

Cyber Security Lab (Ethical Hacking)
Advanced Buffer Overflows

Introduction

Buffer overflow attacks are one of the most popular attacks used to break into computer systems and spread malware such as viruses, worms, and Trojans. According to the news from the past decade, there is no software without a potential risk of buffer overflow. Even firewalls can be affected. Due to their severe risks, several security mechanisms have been developed over time to detect and prevent (less sophisticated) buffer overflows.

The learning objective of this laboratory exercise is to get familiar with advanced variants of buffer overflow attacks and how to bypass existing protection mechanisms currently implemented in major Linux operating systems. A common way to exploit a buffer overflow vulnerability is to overflow the buffer with a malicious shell code and, then, cause the vulnerable program to jump to the shell code that is stored on the stack or heap. This kind of attack was already studied in the course *Introduction to Cyber Security*. Nowadays, most general purpose operating systems enforce a rule that makes everything but the code-segment of a program non-executable, unless explicitly configured otherwise. Therefore, a jump to shell code located on the stack will cause the program to fail with a segmentation fault.

However, the above protection scheme does not guarantee perfect security. There are buffer overflow attacks which do not need an executable stack and do not even use shell code. Instead, they cause the vulnerable program to jump to an existing code, located either in the executable part of the stack or in an external library such as `libc`, which is already loaded in the memory. Such attacks are known as *return-to-libc* attacks. Another approach to protect against buffer overflows is by using *StackGuard* — a special pattern placed directly after the return address that serves as a barrier to detect any overwriting from the scope of the functions. However, as you will see, even this method can be bypassed in some cases...

Preparation of the Experimental Setup

Installing Necessary Software

For this laboratory exercise, it is highly recommended that you run any code and experiments on a suitable Linux distribution, be it bare metal or within a virtual machine. In the following, we will list what tools you need, and how to install them under Ubuntu Linux as an example. Please note that package names may differ if you elect to use a different Linux distribution.

To solve the sheet, you will need the following packages:

- `gcc-multilib` / `g++-multilib` for 32-bit C/C++ support:

  ```
  $ sudo apt install -y gcc-multilib
  $ sudo apt install -y g++-multilib
  ```

- `gdb` — The GNU debugger for debugging C/C++ and Assembly code:

  ```
  $ sudo apt install -y gdb
  ```

- Optionally: A gdb plugin *gdbpeda*, providing many helpful functions for buffer overflow exploit development:

  ```
  $ git clone https://github.com/longld/peda.git /tmp/gdbpeda
  $ sudo cp -r /tmp/gdbpeda  /opt
  $ echo "source /opt/gdbpeda/peda.py" >> ~/.gdbinit
  $ rm -rf /tmp/gdbpeda
  ```

  Note: On some linux distributions, you may also find gdb-peda within the official repositories of your package manager (e.g. `peda` on Arch Linux).

- A hex editor to view binary files, such as:

  ```
  $ sudo apt -y install hexedit
  $ sudo apt -y install bless
  ```

After installing the required software packages, you can install the vulnerable application that is provided to you in Moodle (see `application.zip`). We recommend you to decompress the code in a dedicated folder `buffer-overflow-lab`, which, in turn, is located in the home folder.

While working on the exercise sheet, we recommend you to do regular backups of your current solution progress, especially including the pieces of code that you prepare. To keep track of different versions and avoid several backup copies of the same files, it may be helpful to utilize a version control tool such as *git*. If you run within a virtual machine, you may also benefit from *Snapshots*, as provided e.g. by *Oracle VirtualBox*. A *Snapshot* allows you to restore a given VM stage, including all configurations, files, and installed applications. However, please keep in mind that the created *Snapshot(s)* are only as safe as the storage on your host system, i.e., in case that your *VirtualBox VM* is corrupted or deleted, the snapshots will be gone as well.

Disabling Existing Security Mechanisms

Common Linux distributions have several security mechanisms implemented to make buffer overflow attacks difficult. To simplify our task, we need to disable some of them first. During the development of some of the exploits, we will enable them again one by one. To this end, we briefly introduce the necessary compiler and/or linker flags here. Please keep in mind that you might need to play around with these flags in the provided `Makefile` at some point, i.e., you will have to modify the `CXXFLAGS` variable accordingly.

Address Space Layout Randomization (ASLR)

Many Linux-based systems use address space randomization to randomize the starting address of the heap and the stack. This makes it difficult to determine the exact memory addresses of code in each running instance of the program; determination of addresses is an important step of buffer overflow attacks. There are two main ways to disable ASLR – one for the system as a whole, and one that applies only for a specific execution of a program.

If you are running in a virtual machine, you can safely disable ASLR globally using the following commands:

```
$ sudo -s
Password: (enter root password)
$ sysctl -w kernel.randomize_va_space=0
$ exit
```

If you are working on the tasks directly on your personal computer, it may not be a good idea to disable ASLR for all of the processes running on your computer. In this case, there are two ways you can avoid ASLR: For one, ASLR is automatically disabled when running a program using GDB. In this case, you do not need to worry about ASLR unless you want to explicitly enable it. On the other hand, if for some reason you want to disable ASLR for a process that you are running without GDB, you can do so with the following command:

```
$ setarch $(uname -m) --addr-no-randomize <program> <arguments>
```

StackGuard Protection Scheme (CANARY)

The compiler `g++` implements a security mechanism called *StackGuard* that is effective in preventing simple buffer overflows. You can disable it by compiling your program using the option `-fno-stack-protector`. For example, to compile the program `example.c` with StackGuard disabled, you can use the following command:

```
$ g++ -fno-stack-protector example.c
```

Non-Executable Stack (NX)

Linux systems used to allow executable stacks, but this has long changed. Nowadays, the binary images of programs (and shared libraries) must explicitly declare if they require an executable stack, i.e., they need to mark a field in the program header. The kernel or the dynamic linker uses this mark to decide whether to make the stack of the running program executable or non-executable. This is done automatically by `g++`, setting the stack to be non-executable by default. To change that, you should use the following option when compiling your programs:

```
For executable stack:
$ g++ -z execstack -o example example.c

For non-executable stack:
$ g++ -z noexecstack -o example example.c
```

Fortify Source

Typically, modern C/C++ compilers optimize the code written by the developer during the compilation step. This allows the compiler to replace common code snippets corresponding to bad programming practice and to optimize the overall performance of the program. For instance, if a developer uses the vulnerable function `strcpy()`, the compiler will automatically detect this call and substitute it with a more secure version that additionally applies bound checking. However, in this laboratory exercise, we explicitly study these vulnerable functions. Hence, you should disable these compiler optimizations. This can be done using the flag *fortify source*:

```
$ g++ -U_FORTIFY_SOURCE -o example -example.c
```

To explicitly activate the feature, you can use `-D_FORTIFY_SOURCE=1`, or `2` for an even stronger optimization in favor of security.

Position Independent Code (PIE / PIC)

Nowadays, pieces of codes are compiled in such a way that their binaries can be executed without any dependence on the position they are stored in memory at. This is a very important feature considering dynamic libraries loaded only at run-time. However, independence of position can lead to surprising side effects when dealing with buffer overflows and can make the software analysis more difficult. Therefore, we deactivate this feature using the following option:

```
$ g++ -no-pie -o example -example.c
```

The Vulnerable Program

We consider a C++ program which is used to manage a database of students at a university. The students are represented by a `struct`, called `Student`.

```
1  struct Student
2  {
3          /**
4           * Password used to log in at University. The students
5           * passwords are allowed to be up to MAX_PASSWORD_LENGTH
6           * characters.
7           */
8          char password[MAX_PASSWORD_LENGTH];
9
10         /**
11          * Students identification number.
12          */
13         long id;
14
15         /**
16          * Last Name of the Student.
17          */
18         char* last_name;
19
20         /**
21          * Name of the Student.
22          */
23         char* name;
24 };
```

Listing 1: A structure used to maintain a single student record.

The students belong to a university which is represented by a `class`, called `University`. Each university offers operations as shown in the following UML diagram:

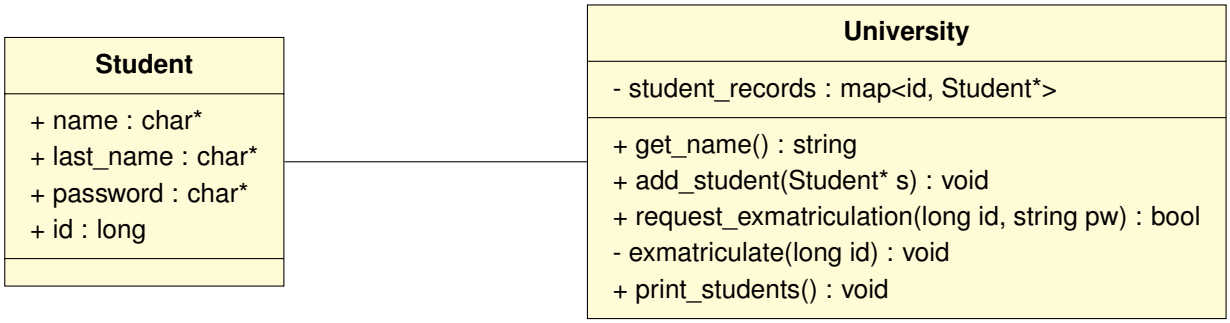

Figure 1: UML class diagram of the vulnerable program.

The complete source code of the program has been provided to you through Moodle.

To make student entries persistent, the `University` instances store all enrolled students in a file `<University-Name>.db`. This file contains the records separated line by line. Each line has the following format, in which all data fields are encoded using plain ASCII code:

```
Name LastName StudentID Password
```

The data fields are separated by whitespace; therefore, the fields themselves must not contain any whitespace. To make use of the classes described above, a simple command line interface is provided. The related source code is located in `b-tu/src/BTU.cpp`. This interface offers the following options:

Option	Optional Parameter	Description
add	Name LastName StudentID Password	Adds a new student to the database
remove	StudentID Password	Requests withdrawal of the student
print	—	Prints a list all enrolled students

To build the executable, we provide a `Makefile` that supports the options *release*, *debug*, and *clean*. Please note that for some of the tasks this file needs to be modified to activate or deactivate specific security features. To compile the (initial) release version from scratch, you can run the following commands:

```
$ make clean
$ make release
```

In addition, the application depends on a library called `libLog`, which provides logging functionality to the program. The compiled library is already given to you within the `lib/Log` folder and should work out of the box with the provided `Makefile`.

By default, the built executable `BTU` is located in `./build/bin` and can now be executed. Make sure to be in the same working directory as the database file `B-TU.db` when executing the program, because it requires access to this file to function correctly. To achieve this, we recommend running the command from within the root directory of the unpacked application as follows:

```
$ ./build/bin/btu <options>
```

Finally, during the process of exploiting you may need to adapt the `Makefile` accordingly to enable or disable the stack protection schemes mentioned above. Hence, it is mandatory to understand the contents of the `Makefile`. It is also recommended to get familiar with each of the provided files before starting the tasks.

The overall goal of the following tasks is to withdraw one of the BTU students without knowing his/her associated password. Therefore, we assume that you do not have read or write access to the database file `B-TU.db`. However, you have full control over the input to the `BTU` command line interface. The basic idea is to exploit a buffer overflow vulnerability to circumvene the password restrictions of the program. The source code of the vulnerable command line interface is given. For your exploit development, it is sufficient that all the implemented attacks run inside a program

instance that has been launched through `gdb`. Outside of `gdb`, the memory addresses needed for the exploit will differ slightly from those observed in the `gdb` session. Although there are ways to work around this slight issue, it adds additional complexity and time requirements to the task, which we want to avoid here.

Tasks

Task 1: Understanding the Program and Analyzing the Vulnerability

To be able to escape the restrictions laid upon you by the student management system, you will need to find some kind of vulnerability that you can exploit. Let's say you have the suspicion that there might be one or even multiple buffer overflow vulnerabilities within the student management system of BTU. Your first goal is to verify this fact, find the vulnerabilities and think about what you could do with them if you were to exploit them.

1. Firstly, let us assume that you do not have access to the source code of the BTU student management system. In such a case, it can be helpful to search for vulnerabilities using a trial-and-error approach, first, because statically analyzing compiled binaries can be very cumbersome. Try and look for *two buffer overflow vulnerabilities* in different functions by playing around with the binary – What do you discover? Describe your findings.
2. Given your findings, you should have a first idea where to look for the vulnerabilities in the source code of the student management system. Find the exact lines responsible for the vulnerabilities and describe what causes the discovered issues.

   To aid your explanations, you should be able to illustrate the memory layout of the program and its stack during execution, understand the sizes of buffers and memory locations utilized during the runtime of the program and describe what exactly is the issue with the vulnerable function calls used by the program. You can gain this understanding both through analysis of the C/C++ source files and by analyzing the runtime of the program using GDB. Please note that during the analytical step it is better to disable any protection mechanisms that you have control of. Be sure to explain any and all phenomena observed during testing and analysis of the application.

Task 2: Basic Buffer Overflow Attacks

In this task, we will look back on basic buffer overflow attacks such as the one studied in detail in the course *Introduction to Cyber Security*. To make this possible, you should *disable all buffer-overflow protections* described in the preparation section (as should already be the case within the provided `Makefile`). Your task is to inject a simple shell code `exit()` that terminates the program with a return value of 5. The shell code is given by the following assembler program:

```
1 exiter:     file format elf32-i386
2
3 Disassembly of section .text:
4 08048060 <_start>:
5  8048060:       31 c0            xor    %eax,%eax
6  8048062:       b0 01            mov    $0x1,%al
7  8048064:       31 db            xor    %ebx,%ebx
8  8048066:       b3 05       mov    $0x5,%bl
9  8048068:       cd 80            int    $0x80
```

It can be represented in the hexadecimal form `x31 xc0 xb0 x01 x31 xdb xb3 x05 xcd x80`.

Your goals for this task are the following:

1. Firstly, you should successfully inject the shell code into the *BTU* program and execute it. To this end, it is sufficient if you can exploit a process launched through GDB. How would it be possible to verify that the attack works if you were running it against the release version of the software?
2. Next, you should examine what happens if you enable certain protections against buffer overflows again. Research what the following protection mechanisms do and how exactly they may or may not affect the viability of this basic buffer overflow attack. What happens if you enable only:
   a) *Non-Executable Stack*?
   b) *StackGuard*?
   c) *Address Space Layout Randomization (ASLR)*?

Hints:

1. Just as in Introduction to Cyber Security, examining the running program through GDB will help you in finding out the necessary memory addresses to craft your payload.
2. It is easy to pass more complicated arguments (such as your payload) to the program or to GDB using e.g. Python. For this, you may be interested in the `python -c` option, which can be used to execute short Python code straight from the command line. If you're considering this option, you may need to think about the differences between the function

`print()` in Python and the C/C++ method `stdout.buffer.write()`.

3. The plugin `gdb-peda` can help you in your process by providing additional information and commands. When you encounter unexplainable issues, make sure to check the status of the overflow protection mechanisms by using the `gdb-peda` commands `checksec` and `aslr`.
4. It is also possible to exploit the release version of the *BTU* program. The main issue we face here is that the exact memory addresses in the release version will differ slightly (by a few bytes) from those encountered in GDB. We encourage you to think about how this issue can be dealt with, even if it's not a required part of the task.

Task 3: Attacking Non-Executable Stack

In the previous task, you have been able to observe that non-executable stack is a powerful countermeasure to prevent basic buffer overflows that rely on the execution of an attacker-provided shell code. However, in many situations, the code necessary to achieve the attacker's goal is already included in the vulnerable program. In this task, we will use this fact to withdraw the student *Klaus Komisch* (student ID 1782914303) from the database and even spawn a general purpose shell, just like we would have been able with an injected shell-code. For this task, you should *enable non-executable stack (NX)*, but still keep *ASLR* and *StackGuard* disabled. Your task consists of three main steps:

1. To get familiar with the capabilities of this attack, you shall first look for a way to change the flow of the program to call the withdrawing function on *Klaus Komisch* without having to provide the necessary password. Once again, you will abuse the buffer overflow in the *remove* action of the BTU program to manipulate the stack. You want to overwrite the memory in such a way that the flow of the program changes to the function `exmatriculate(long id)`, withdrawing *Klaus Komisch*. Be ready to describe and demonstrate your developed attack as part of your submission.

2. Once you are able to successfully exmatriculate students with this technique, it is time to consider what else an attacker can do with it. We will now utilize the fact that, next to the functionality directly provided by the vulnerable program, there is also code included from external libraries during the compilation and/or linking stage.

   Check what libraries are loaded by `BTU` – Based on them, you should create an exploit to initiate a shell (e.g. `/bin/bash`, `/bin/sh` or a similar available shell) using the function `system()` from the external library `libc`. This is commonly known as a *ret2libc Attack*. Having the possibility to launch a shell, an attacker can do most of the things that would have also been possible with malicious shellcode, thus rendering non-executable stack alone ineffective at protecting vulnerable systems.

   For our purposes, it is enough if you can successfully start a shell process, i.e. end up dropping into a shell within GDB or receive the following feedback from the debugger:

   ```
   [New inferior]
   process 22333 is executing new program: /usr/bin/bash
   ...
   ```

   Successfully connecting to and using such a shell as an attacker in the release binary requires some additional tricks (since in this example, we do not guarantee a vulnerable netcat version to be present). We will not focus on this during this exercise, but you are encouraged to think about solutions if you find the time.

   Finally, you should extend *both of your exploits* to exit the `BTU` program safely after performing the exmatriculation and after leaving the shell. The function `exit()` provided by `libc` will be of use for this purpose. Thus, the attacker's goal is achieved without leaving any logs about segmentation faults, which would ease detection and forensic analysis of our attack.

It is also possible to control the exit return code (exit status) of the program to avoid any suspicions.

3. After both of your attacks are executed and perfected, think again about how the attacks are affected when *ASLR* and *StackGuard* are enabled. Can you withdraw *Klaus Komisch* and/or get the shell? How do *ASLR* and *StackGuard* make your attacks not possible?

Hints:

1. Keep in mind that the function `exmatriculate(long id)` is a member function of the class `University`, i.e. it relates to a specific instantiated University object. How does the function know which university it belongs to when it is called?
2. While selecting the type of shell to spawn within your exploit, you should carefully consider the impacts on your attack. E.g., some shells such as `/bin/bash` are automatically included in the environment of the program due to the `libc` library, while others have to be explicitly exported (see the practical hints provided in the appendix). In addition, some shells have special protection mechanisms.
3. When using addresses of functions such as `exit()` or `system()`, you may have to jump to an address like, for example, `0xF7C10420`. Converting this address to little-endian as required for the attack, you will obtain `0x2004C1F7`. Depending on alignment and your concrete implementation, you might face a problem in which the `00` causes not all of your attack payload to be copied into the vulnerable buffer. In this case, you should take a look at the disassembly of the function you want to jump to: You will find that it is possible to jump to an address four bytes before or after it, without affecting the execution flow / semantic.

Task 4: Sneaking Past the StackGuard

The introduction of *StackGuard* defeated several types of buffer overflow exploits which relied on overwriting local buffers stored in the stack to modify the running functions return address. The core idea of *StackGuard* is to place a well defined, unpredictable four byte pattern between the return address and the saved base pointer. Once the attacker overwrites a buffer in such a way that the return address is affected, the pattern is overwritten as well. During the function's epilogue, this modification is detected and handled appropriately.

On first glance, it may seem like this approach solves all of our buffer overflow issues. However, taking into account its implementation, it becomes obvious that *StackGuard* can only protect against exploits that try to overwrite memory above the stack guard directly by overflowing some vulnerable buffer on the local stack frame. Are there ways for an attack to succeed if the buffer only overflows *within* the confines of its current stack frame?

In this task, we study an unlucky vulnerability within our program that allows an attacker to write arbitrary data into arbitrary memory regions. This means you will be able to precisely overwrite, e.g., just 4 bytes of the return address without affecting any surrounding memory regions. The pattern of *StackGuard* will remain valid and the attack will remain undetected!

We will appproach this task in the following steps:

1. **Scenario Setup.**

   For this task, you have to enable both *non-executable stack* and the *StackGuard* as described above. All other security measures, such as *ASLR*, *PIC* and *RelRo* should be kept disabled.

2. **Examining the Vulnerability.**

   In the previous tasks, we have studied in detail the vulnerability associated with the *remove* action of the *BTU* program. However, there is a second buffer overflow vulnerability within the *BTU* program - and due to the instructions surrounding it, we will find that it happens to be more powerful than the previous one. As you hopefully discovered by this point, the vulnerability can be found when *adding* a new student to the university database.

   If you did not do so already in Task 1, now is the time to take a close look at this vulnerability again. What makes it different from the vulnerability we discovered in the *remove* action? In particular, how can this specific vulnerability and the instructions that follow after it allow us to write data to arbitrary memory locations? In order to fully understand what happens, you will have to both debug the code dynamically using GDB, and perform static analysis of the code to understand the memory layout of the objects that are being worked on within the vulnerable function. To this end, we recommend you to create a depiction of the process memory and how it is affected by the overflow so that you can add it to your writeup.

3. **Verifying the Exploitability.**

   Not all vulnerabilities yield in a threat which an attacker can exploit for malicious purposes. Often, programs just crash or result in unexpected behavior. Therefore, you should verify your suspicions of this vulnerability allowing you to perform arbitrary writes to arbitrary memory locations. To this end, you should write a basic exploit which allows you to write the number `0xDEADBEEF` to memory address `0xFFFFABCD`. How can you verify whether your exploit worked out correctly with `GDB`? How can you observe whether you are able to overwrite a specific address or not?

4. **Sneaking past the StackGuard.**

   Once you have successfully verfied the usefulness of the vulnerability, it has come time to weaponize it. Your goal in this task is to exmatriculate *Klaus Komisch* (ID 1782914303) once again, this time with both *Non Executable Stack* and *StackGuard* enabled. We suggest that you use the capabilities of this vulnerability to redirect the execution flow to the exmatriculate function, similar to what you did in Task 3. To help you in crafting your payload, we suggest you draw a side-by-side view between how the stack looks before your exploit and how it should look after your exploit has written the desired data. Be sure to take special care about how the data you do overwrite may or may not impact the execution flow of the vulnerable function.

   Note: It might not be possible to ensure a graceful exit of the program this time. That's alright as long as you manage to achieve your main objective.

Task 5: Avoiding Buffer-Overflow Vulnerabilities

Throughout the course of this task sheet, you should have gained a fundamental understanding of how various different general protection schemes make it difficult for an attacker to actually exploit buffer overflow vulnerabilities. A system with all of these security measures enabled could be thought of as reasonably secured against buffer overflows. And yet, these tasks should have also given you an idea of how very specific constellations can still make it possible to defeat these countermeasures and allow an attacker to break into supposedly secured systems – a situation that we hear of time and time again when following cyber-security related news. Thus, it should be clear that the best way to truly secure a program is to avoid these types of vulnerabilities in the first place.

To conclude our journey into advanced buffer overflows, your final task is to fix the vulnerabilities that we have exploited within the *BTU* program. Make sure that the fixes you propose do not impact the capabilities or go against the programmer's intent of the affected functions. Apart from that, feel free to be creative in how you decide to fix the issues you discovered – it can be said that both of the vulnerabilities can be fixed without introducing a lot of new code.

Preparation of Your Submission

Your submission for this lab should contain the final versions of any and all source code you have written and files you have generated, along with a detailed writeup about the tasks you solved. Your writeup should be given in an appropriate format, containing detailed descriptions of how you solved each task, what theoretical insights you gained, what observations you have made as well as images / screenshots / listings to aid the understanding of your approach. Your writeup should be submitted in PDF format.

In this lab, the documentation of the steps you took to solve the tasks is of particular importance. Make sure to include relevant gdb commands you executed. If you mention specific memory addresses, make sure to comment on their meaning and how you obtained them, for example: "The entry point of the function `University::exmatriculate()` lies at address `0x80c4fcff`, which I found using the gdb command `disas University::exmatriculate()`." Last but not least, the final commands and payloads you used to run your exploits should be submitted within your writeup and/or as dedicated and well-documented scripts. The steps that you took to solve the tasks *must* be given in a reproducible manner in order to yield points! Moreover, you have to submit a code snippet implementing and highlighting the fixes to the vulnerabilities.

Please submit your solutions in a packed zip file named after your full name, with the following structure and minimum required contents:

Preparation for the Q&A Session

Prepare yourself for a Q&A session of up to 40 minutes. During the Q&A session, you should be prepared to *explain and demonstrate* how you solved the tasks. Take special care to explain how your solution works and why you took the approach you took. Note that simply presenting a piece of code without any explanation will not yield any points! Last but not least, you should be ready to answer spontaneous (theoretical) questions asked by the reviewer in relation to the topics of this lab.

Appendix

A. Practical Hints for Working with the GNU Debugger

After a program is loaded by `gdb <program name>`, the debugger offers many features which can be used for both dynamic and statical analysis of binary code. Table 1 is just a short outline of some useful GNU debugger commands.

function	command (hot key)
print source code	`l [function name, line number]`
insert break point	`b <line number>`
set condition for break point	`condition <number> <expression>`
print value of variable, address or expression	`p <variable>`
print function in assembler syntax[1]	`disas <function>`
run program[2]	`r <parameter list>`
show register contents	`i r`
inspect memory at address	`x/<format> <address>`
print help to specific command	`h <command>`
switch into a tui layout (c code)	`layout src`
switch into a tui layout (assembly code)	`layout asm`
switch into a tui layout (with registers)	`layout reg`
disable tui	`tui disable`

Table 1: Useful gdb commands

The `p` and `x` commands may become a valuable tool for your exploit development. When working with memory addresses, they also provide the powerful options of formatting the output they provide. For instance, `p/x` will print the value in hexadecimal, while `p/d` prints it in decimal. You can also evaluate common C-expressions using the `p` command - this includes simple calculations, e.g. adding 13 bytes to an address `0xffffabcd` could be done using `p/x 0xffffabcd + 13`. It also allows inspecting where variables and even functions lie in memory - for example, you can find out where the libc "system" function lies by issuing the command `p &system`.

Regarding the current memory layout, the `x` command is helpful. To display four bytes in hex at a certain memory address, you can use `x/4bx <address>`. You can also use this command to see what lies at a certain address when interpreted as a null-terminated string: `x/s <address>`. And finally, if you're curious about what the data at an address (for example, the value of the current program counter `$pc`) means when interpreted as 20 assembly instructions, you can run `x/20i $pc`.

Note: By default the assembler code is printed in AT&T syntax. To change to Intel notation (as is used by `nasm`), one can change the format within gdb's interactive shell using `(gdb) set disassembly-flavor intel`.

Since you will be working with complicated payloads that you likely have to adjust several times to reach a working exploit, it is possible that you will have to restart GDB several times across your development process. Surely, you will find it very cumbersome at some point to have to set the same breakpoints and issue the same series of commands over and over again to test your exploit and investigate the current memory layout. Fortunately, there are ways to automate this process. Namely, GDB allows specifying commands to execute immediately from the command line. With this, you can configure your environment with just one command. You may benefit from putting your commands into a little script such as the following:

```
1 #!/bin/sh
2 gdb build/bin/btu \
3         -ex "set args print '$(cat payloadfile)'" \
4         -ex "set disassembly-flavor intel" \
5         -ex "set debuginfod enabled off" \
6         -ex "b University::request_exmatriculation" \
7         -ex "run"
```

This script makes GDB launch the program with a call akin to `build/bin/btu print $(cat payloadfile)`, configures some preferences and also sets a breakpoint in the program before immediately running it.