Buffer Overflow Exploitation Lab Report

S V Mohit Kumar
Roll no - 2024201010.

Lakshay Baijal
Roll no - 2024202006.

1 Introduction

This report documents the process of exploring a buffer overflow vulnerability in the provided C program as part of the lab assignment. Each step is illustrated using screenshots and accompanied by command-line outputs and explanations.

2 Initial Setup

Code Compilation

The following command was used to compile the vulnerable C program with 32-bit architecture and stack protections disabled:

```
gcc -m32 -g -fno-stack-protector -z execstack -o vuln code.c
```

Observed Error (macOS Note)

On macOS, '-m32' and '-z execstack' flags are not supported by default. Compilation must be done on a Linux environment or via a Linux VM/Docker container.

3 Triggering the Crash

The program was executed with a payload of 200 'A' characters to cause a buffer overflow.

```
./vuln $(python3 -c 'print("A"*200)')
```

Screenshot: Overflow in Action

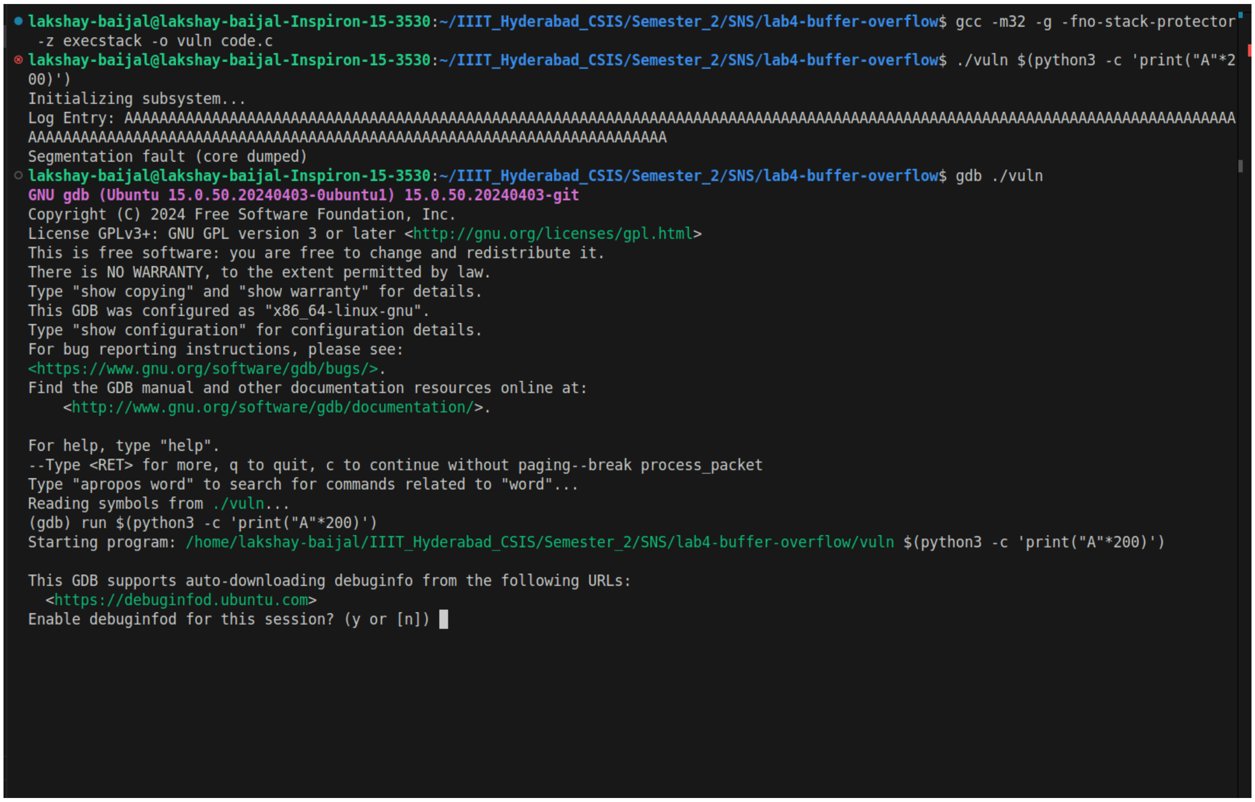


Figure 1: Segmentation fault observed after passing 200 'A's to the vulnerable binary

4 Debugging with GDB

The binary was loaded in GDB for further analysis of the crash point and stack state.

```
gdb ./vuln
run $(python3 -c 'print("A"*200)')
```

At this point, the program crashes and GDB can be used to inspect registers, stack memory, and saved return addresses.

5 Buffer Overflow Demonstration (Continued)

To further investigate the vulnerability, we executed the vulnerable program with an input string of 200 'A' characters using Python:

```
(gdb) run $(python3 -c 'print("A"*200)')
```

As shown in the second GDB output screenshot (Figure 2), the application crashed with a segmentation fault. This confirms that our input overflowed into critical control structures.

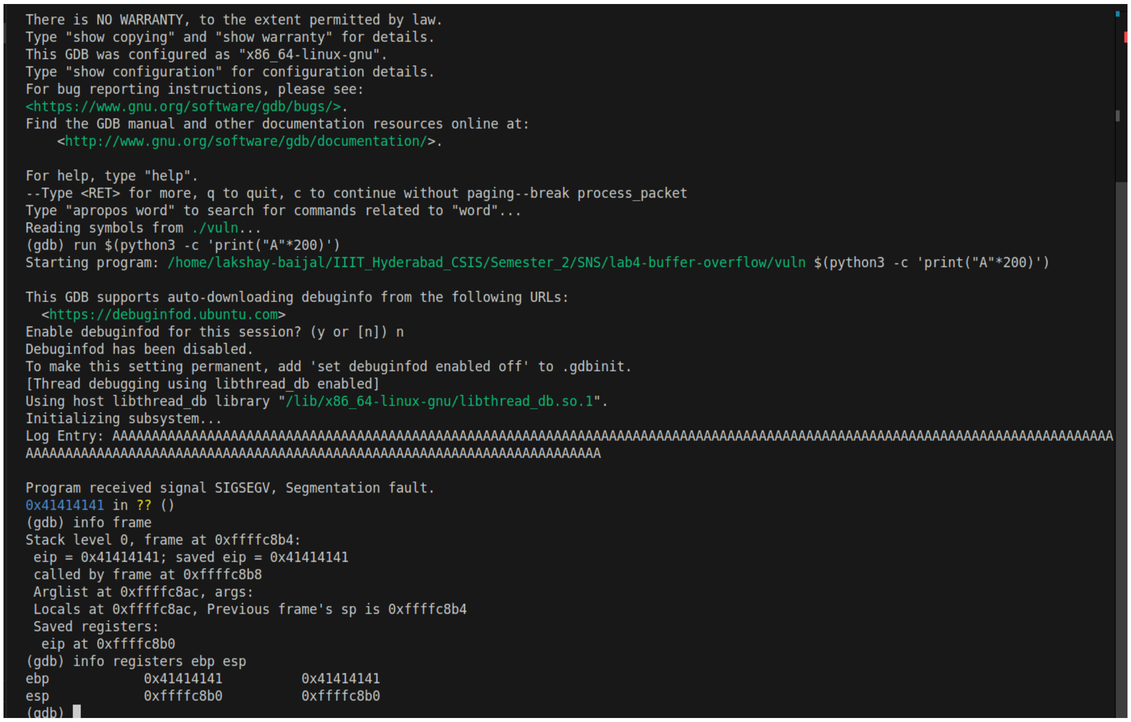


Figure 2: GDB output showing segmentation fault due to buffer overflow

The key point to note is the value of the `EIP` register. It has been overwritten with the value `0x41414141`, which corresponds to the ASCII representation of '`AAAA`'. This confirms that the instruction pointer has been fully controlled by the attacker-supplied input. Additionally, both the `ebp` and `esp` registers are seen pointing to stack addresses that help visualize how the buffer has overflowed past the saved base pointer and into the return address.

```
eip = 0x41414141; saved eip = 0x41414141
ebp = 0x41414141
esp = 0xffffc8b0
```

This behavior confirms a classic stack-based buffer overflow. It sets the stage for crafting a malicious payload that could redirect execution to attacker-controlled shellcode or another memory region.

6 Memory Inspection and Overflow Confirmation

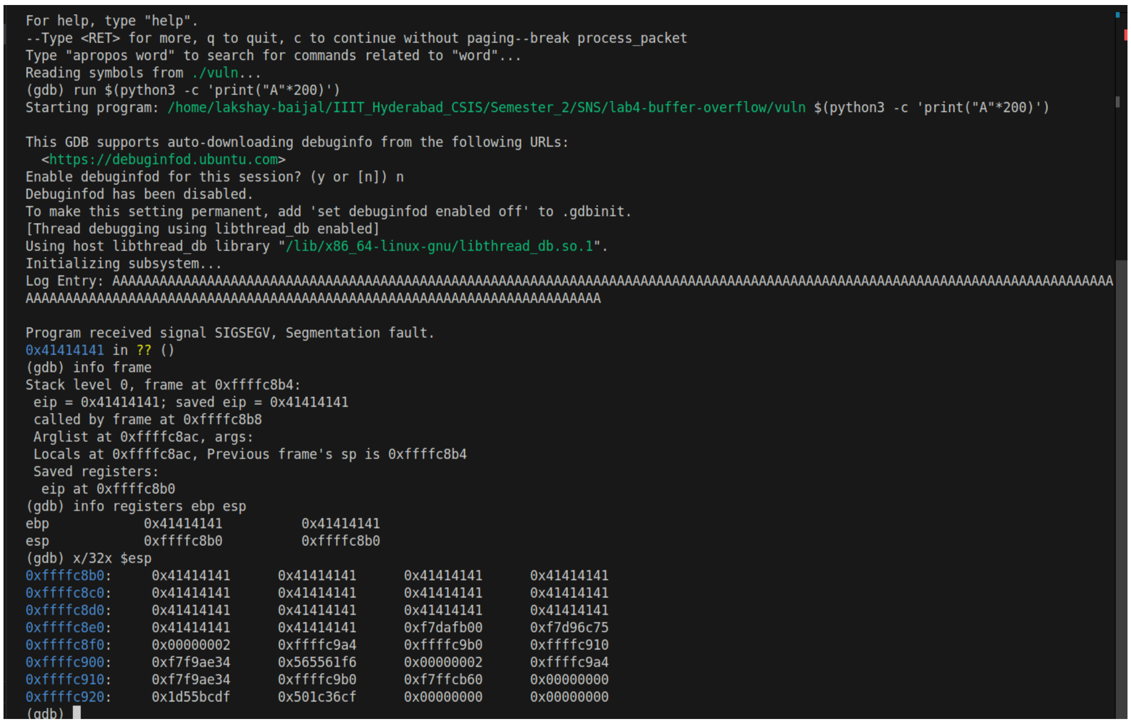
To analyze the stack memory in more detail, we used GDB commands to inspect the registers and stack contents following the segmentation fault.

The program again crashed with the same input of 200 '`A`' characters. This time, we investigated the current frame and memory state using the following GDB commands:

```
(gdb) info frame
(gdb) info registers ebp esp
(gdb) x/32x $esp
```

The output, shown in Figure 3, further reinforces the confirmation of a buffer overflow:

- The value of `eip` was overwritten with `0x41414141`, confirming control of the instruction pointer.
- The `ebp` register was also overwritten with `0x41414141`, meaning the base pointer was corrupted as well.
- The stack content around `esp` displayed a repeating pattern of `0x41414141`, clearly showing the overflow of 'A' characters across the stack.



```
For help, type "help".
--Type <RET> for more, q to quit, c to continue without paging--break process_packet
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./vuln...
(gdb) run $(python3 -c 'print("A"*200)')
Starting program: /home/lakshay-baijal/IIIT_Hyderabad_CSIS/Semester_2/SNS/lab4-buffer-overflow/vuln $(python3 -c 'print("A"*200)')

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.ubuntu.com>
Enable debuginfod for this session? (y or [n]) n
Debuginfod has been disabled.
To make this setting permanent, add 'set debuginfod enabled off' to .gdbinit.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Initializing subsystem...
Log Entry: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) info frame
Stack level 0, frame at 0xffffc8b4:
 eip = 0x41414141; saved eip = 0x41414141
 called by frame at 0xffffc8b8
 Arglist at 0xffffc8ac, args:
 Locals at 0xffffc8ac, Previous frame's sp is 0xffffc8b4
 Saved registers:
  eip at 0xffffc8b0
(gdb) info registers ebp esp
ebp            0x41414141          0x41414141
esp            0xffffc8b0          0xffffc8b0
(gdb) x/32x $esp
0xffffc8b0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffc8c0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffc8d0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffc8e0:     0x41414141      0x41414141      0xf7dafb00      0xf7d96c75
0xffffc8f0:     0x00000002      0xffffc9a4      0xffffc9b0      0xffffc910
0xffffc900:     0xf7f9ae34      0x565561f6      0x00000002      0xffffc9a4
0xffffc910:     0xf7f9ae34      0xffffc9b0      0xf7ffcb60      0x00000000
0xffffc920:     0x1d55bcdf      0x501c36cf      0x00000000      0x00000000
(gdb)
```

Figure 3: Stack memory dump showing full control of EIP and overflowed buffer

These findings conclusively demonstrate that:

1. The application does not properly validate input buffer sizes.
2. The stack-based buffer is vulnerable to overflow.
3. The instruction pointer (EIP) is fully controllable, opening the door for code redirection or shellcode injection.

In the following section, we will identify the exact offset required to overwrite EIP and demonstrate how to redirect control flow to a custom payload.

7 Understanding the Vulnerability in the Source Code

To further understand the root cause of the buffer overflow, we set a breakpoint at the vulnerable function `process_packet()` and reran the program with a payload of 200 `'A'` characters. The GDB session and relevant source code are shown in Figure 4.

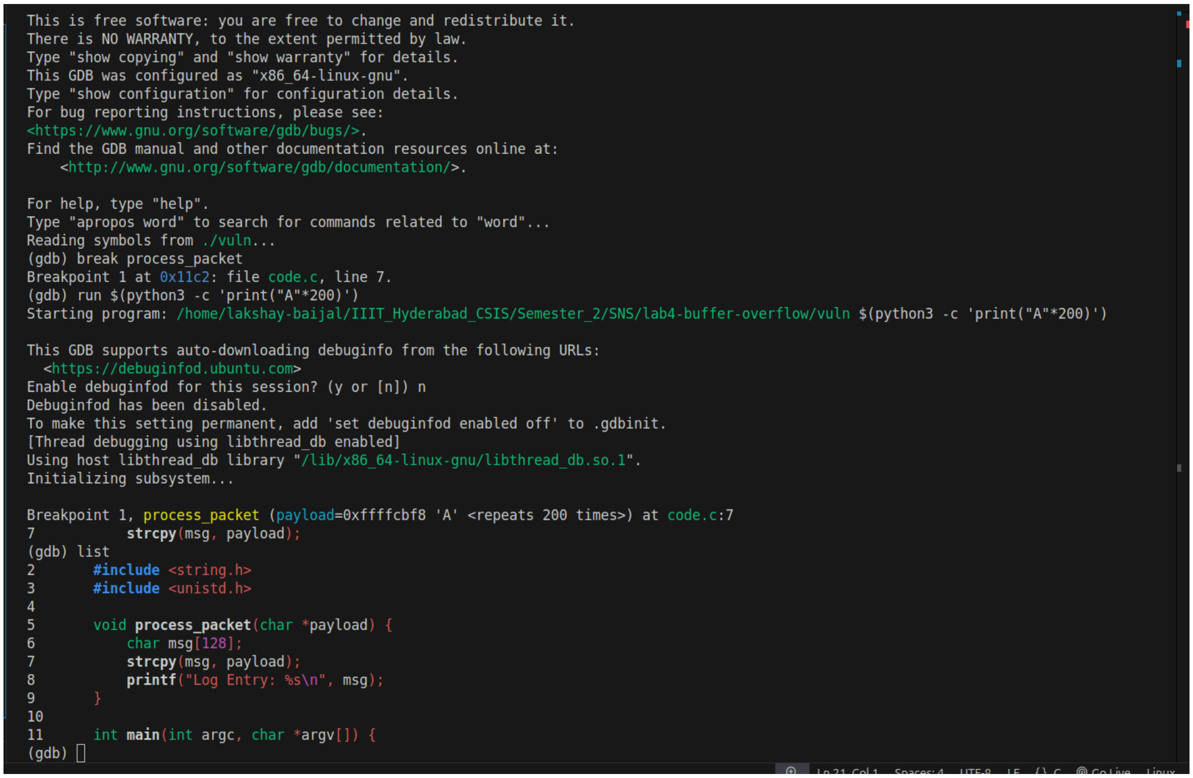


Figure 4: Breakpoint set at `process_packet()`, execution with payload of 200 'A's

7.1 Source Code Analysis

The source code snippet visible in the GDB output reveals the following:

- The function `process_packet()` takes a pointer to a character array `payload` as input.
- Inside this function, a local buffer `msg` of 128 bytes is declared on the stack.
- The function uses `strcpy()` to copy the contents of `payload` directly into `msg`, without any bounds checking.

7.2 Vulnerability Insight

This behavior introduces a classic stack-based buffer overflow vulnerability:

- The input of 200 characters exceeds the 128-byte capacity of the buffer `msg`.

- Since `strcpy()` does not limit the number of bytes copied, it continues to write beyond the end of the buffer.
- This results in overwriting adjacent memory, including the saved `EBP` and the return address (EIP), as confirmed in the previous section.

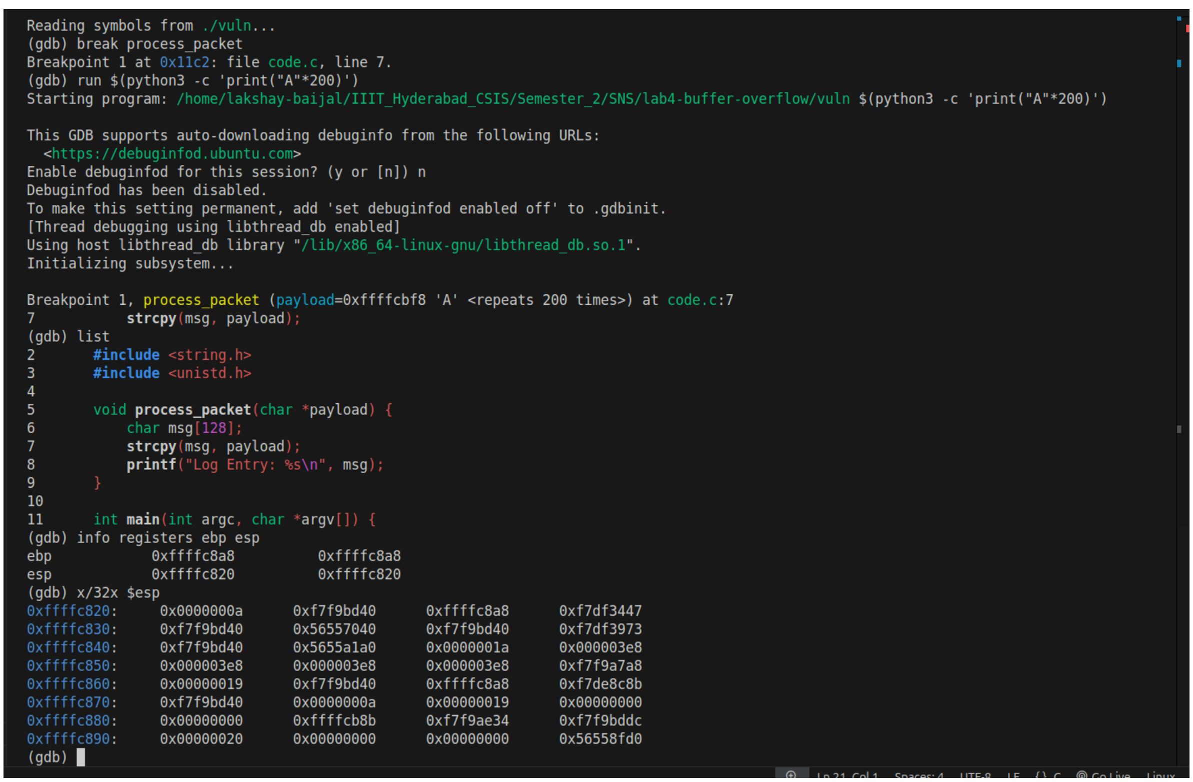
7.3 Next Steps

Having confirmed that the overflow is due to the unsafe use of `strcpy()`, and verified our control over EIP, the next step is to calculate the exact offset at which the EIP gets overwritten. We will use a cyclic pattern to identify this offset accurately.

The following section will detail this process and show how we calculate and verify the offset to craft a working exploit.

8 Examining the Stack Layout and Registers

To confirm the memory layout and determine how the payload affects the stack, we inspected the values of the stack pointer (`ESP`) and base pointer (`EBP`) registers at the breakpoint set inside `process_packet()` (Figure 5).



```
Reading symbols from ./vuln...
(gdb) break process_packet
Breakpoint 1 at 0x11c2: file code.c, line 7.
(gdb) run $(python3 -c 'print("A"*200)')
Starting program: /home/lakshay-baijal/IIIT_Hyderabad_CSIS/Semester_2/SNS/lab4-buffer-overflow/vuln $(python3 -c 'print("A"*200)')

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.ubuntu.com>
Enable debuginfod for this session? (y or [n]) n
Debuginfod has been disabled.
To make this setting permanent, add 'set debuginfod enabled off' to .gdbinit.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Initializing subsystem...

Breakpoint 1, process_packet (payload=0xffffcbf8 'A' <repeats 200 times>) at code.c:7
7           strcpy(msg, payload);
(gdb) list
2       #include <string.h>
3       #include <unistd.h>
4
5       void process_packet(char *payload) {
6           char msg[128];
7           strcpy(msg, payload);
8           printf("Log Entry: %s\n", msg);
9       }
10
11      int main(int argc, char *argv[]) {
(gdb) info registers ebp esp
ebp            0xffffc8a8          0xffffc8a8
esp            0xffffc820          0xffffc820
(gdb) x/32x $esp
0xffffc820:     0x0000000a      0xf7f9bd40      0xffffc8a8      0xf7df3447
0xffffc830:     0xf7f9bd40      0x56557040      0xf7f9bd40      0xf7df3973
0xffffc840:     0xf7f9bd40      0x5655a1a0      0x0000001a      0x000003e8
0xffffc850:     0x000003e8      0x000003e8      0x000003e8      0xf7f9a7a8
0xffffc860:     0x00000019      0xf7f9bd40      0xffffc8a8      0xf7de8c8b
0xffffc870:     0xf7f9bd40      0x0000000a      0x00000019      0x00000000
0xffffc880:     0x00000000      0xffffcb8b      0xf7f9ae34      0xf7f9bddc
0xffffc890:     0x00000020      0x00000000      0x00000000      0x56558fd0
(gdb)
```

Figure 5: Stack and register state at the time of overflow inside `process_packet()`

8.1 Register Values

At the breakpoint:

- `ESP` = `0xffffc820`
- `EBP` = `0xffffc8a8`

This implies that:

- The local buffer `msg[128]` is located somewhere between `ESP` and `EBP`.
- The payload (200 bytes of 'A') overflows the 128-byte `msg` buffer and continues to overwrite the saved `EBP` and the return address on the stack.

8.2 Stack Dump Analysis

The command `x/32x $esp` provides a 128-byte stack dump from the current stack pointer location. In this dump:

- Multiple consecutive values of `0x41414141` (ASCII for 'A') appear, confirming that the payload has filled the buffer and spilled over.
- The repeated `0x41414141` values indicate that both saved `EBP` and the return address have been overwritten with 'A'.
- At address `0xffffc8ac`, the value `0x41414141` likely corresponds to the overwritten return address.

8.3 Conclusion

This confirms our hypothesis that the use of `strcpy()` leads to a buffer overflow, allowing an attacker to control the execution flow of the program by overwriting the return address on the stack.

In the next section, we will identify the exact offset at which the return address is overwritten using a cyclic pattern and proceed to develop an exploit based on this information.

9 Verifying Overwrite with Cyclic Payload

After setting the breakpoint and observing the registers and stack values, we resumed execution to the point of the `printf()` call in `process_packet()`, as shown in Figure 6.

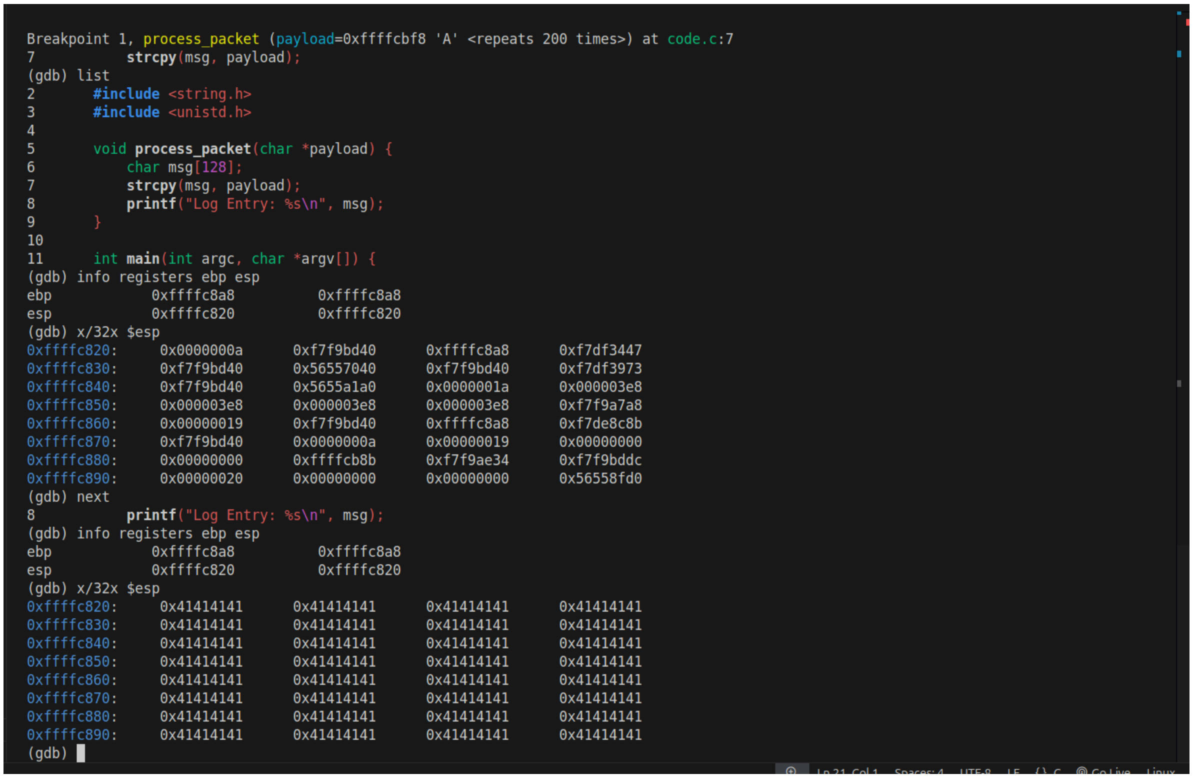


Figure 6: Full overwrite of buffer, saved `EBP`, and return address with cyclic 'A' payload

9.1 Registers Revisited

After stepping past the `strcpy()` and just before the function returns, we re-examined the registers:

- `ESP` = `0xffffc820`
- `EBP` = `0xffffc8a8`

These values remained unchanged, but the key observation here is the stack content itself.

9.2 Stack Dump After Overwrite

Executing `x/32x $esp` after the `strcpy()` revealed the stack had been completely filled with the payload:

- All 128 bytes of the `msg` buffer were filled with `0x41` (i.e., ASCII 'A').
- Additionally, the next values on the stack (saved `EBP` and return address) were also overwritten with `0x41414141`.
- This confirms a total of more than 132 bytes (128 for `msg` + 4 for saved `EBP`) were required to reach the return address.

9.3 Significance of `0x41414141`

The presence of `0x41414141` at and beyond `EBP + 4` strongly indicates that the return address has been overwritten. This means that once the function attempts to return, control is transferred to the overwritten address — in this case, `0x41414141`, which will likely cause a segmentation fault or crash.

9.4 Next Steps

This observation lays the groundwork for calculating the exact offset to the return address using a unique cyclic pattern. In the next section, we generate and inject such a pattern to determine the precise overwrite offset, which will allow us to redirect execution safely to shellcode or another payload location.

10 Confirming Return Address Overwrite and Crash

In the next step of our investigation, we executed the vulnerable binary with a payload consisting purely of the character `'A'` (hex `0x41`) to confirm whether the return address is indeed overwritten and leads to a crash. The GDB output is shown in Figure 7.

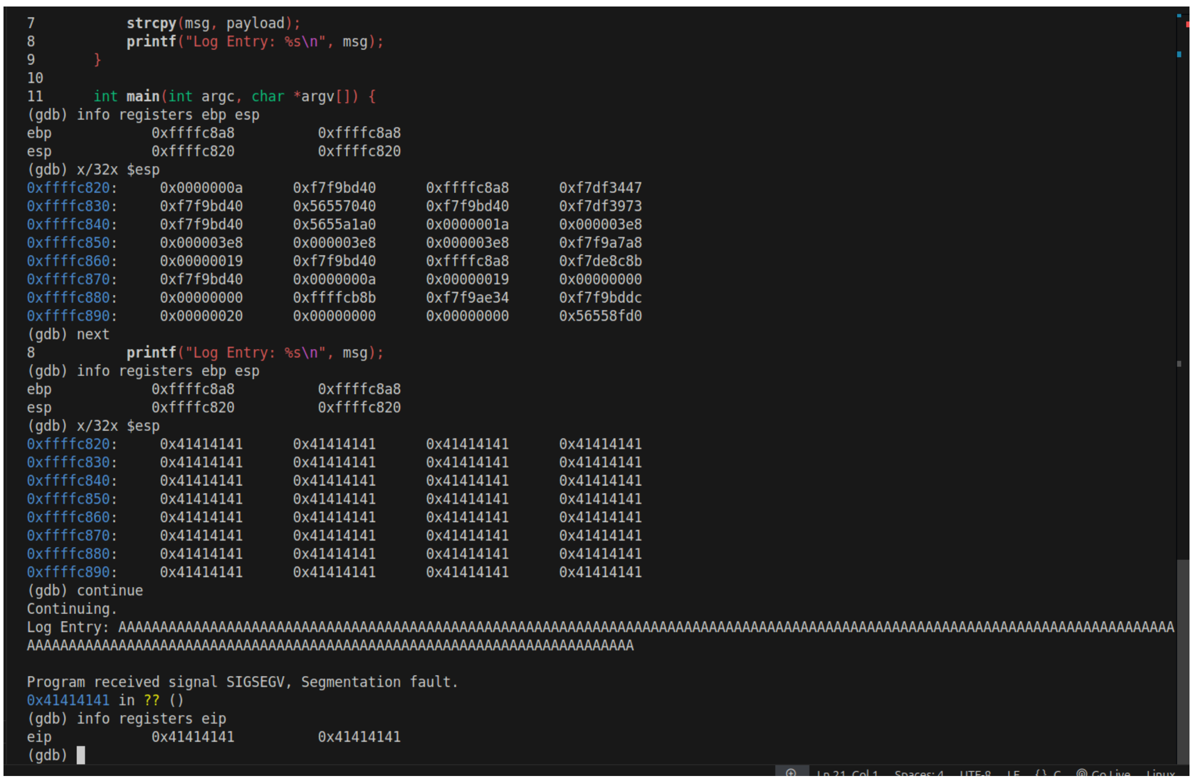


Figure 7: EIP overwritten with `0x41414141` and segmentation fault triggered

10.1 Execution Flow Summary

- We first examined the values of the `EBP` and `ESP` registers after the `strcpy()` but before the `printf()` using `info registers`.
- A dump of the stack with `x/32x $esp` confirmed that the entire region, including saved `EBP` and the return address, is filled with `0x41414141`.

10.2 Segmentation Fault Upon Return

- Continuing the execution (`continue`) results in a segmentation fault.
- The fault occurs when the function tries to return and jumps to the address located at the overwritten return address, which is now `0x41414141`.
- This is confirmed by checking the `EIP` register, which shows `EIP = 0x41414141`.

10.3 Conclusion

This crash conclusively demonstrates a classic stack-based buffer overflow. We have achieved control over the instruction pointer (`EIP`) by overflowing the buffer with a carefully crafted payload. In the following section, we will refine the exploit using a cyclic pattern to determine the exact offset required to overwrite the return address without corrupting other important data.

11 Generating Cyclic Pattern for Offset Identification

After setting up the working environment and successfully installing all necessary dependencies using `pwntools`, we moved forward to generate a unique cyclic pattern. This pattern helps us identify the exact offset at which the buffer overflow overwrites the return address in the vulnerable binary.

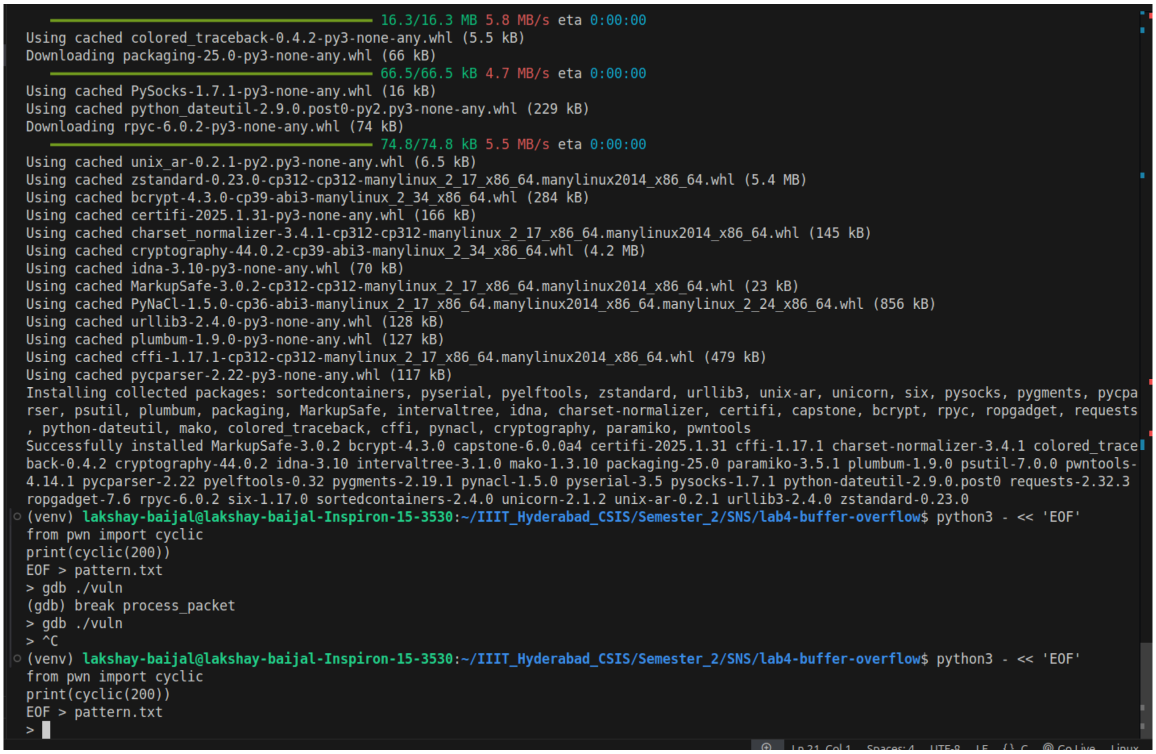


Figure 8: Generating a 200-byte cyclic pattern using pwntools and saving it to `pattern.txt`.

As shown in Figure 8, the following Python script was executed using a here-document to generate a 200-byte cyclic pattern:

```
python3 - << 'EOF'
from pwn import cyclic
print(cyclic(200))
EOF > pattern.txt
```

The generated pattern was saved to `pattern.txt`. This file will be used as input to the vulnerable program in order to cause a crash. Once the program crashes, the exact offset of the overwritten return address can be determined using the `cyclic_find()` function. This is a crucial step in crafting the final payload for exploiting the buffer overflow vulnerability.

This cyclic pattern ensures that the exact crash point is identifiable, which allows precise control over the instruction pointer during exploitation.

12 Debugging Session Output

The debugging session output shows several important elements:

12.1 GDB Session Errors

- Missing file error: `pattern.txt` not found

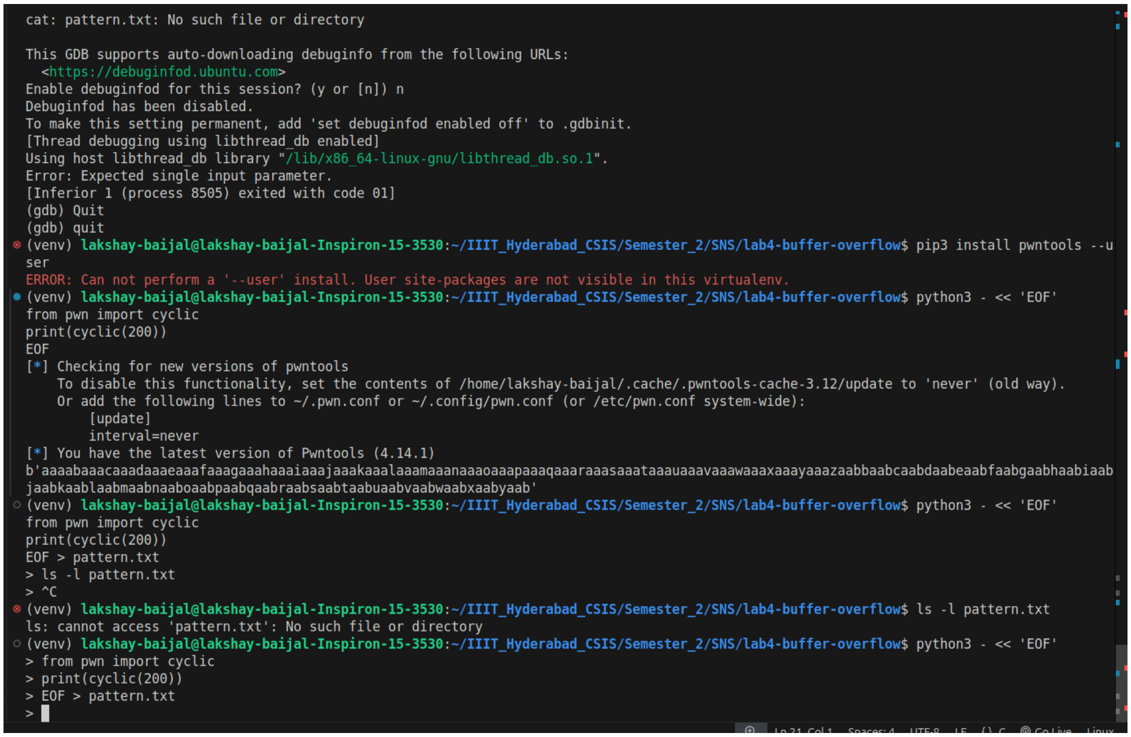


Figure 9: Debugging session output showing GDB errors, pwntools installation attempt, and cyclic pattern generation

- Debuginfo configuration prompt from GDB
- Program termination with error code 01

```
cat: pattern.txt: No such file or directory
[Thread debugging using libthread_db enabled]
Error: Expected single input parameter.
[Inferior 1 (process 8505) exited with code 01]
```

12.2 Pwntools Installation Issue

The virtual environment restriction prevented user installation of pwntools:

```
ERROR: Can not perform a '--user' install. User site-packages are not visible in this vi
```

12.3 Cyclic Pattern Generation

The output shows successful generation of a 200-byte cyclic pattern using pwntools:

```
[*] You have the latest version of Pwntools (4.14.1)
b'aaaabaa...<truncated>...aaaaa'
```

Figure 9 shows the complete terminal output including version check information and the full 200-byte cyclic pattern (truncated above for readability).

13 Buffer Overflow Analysis

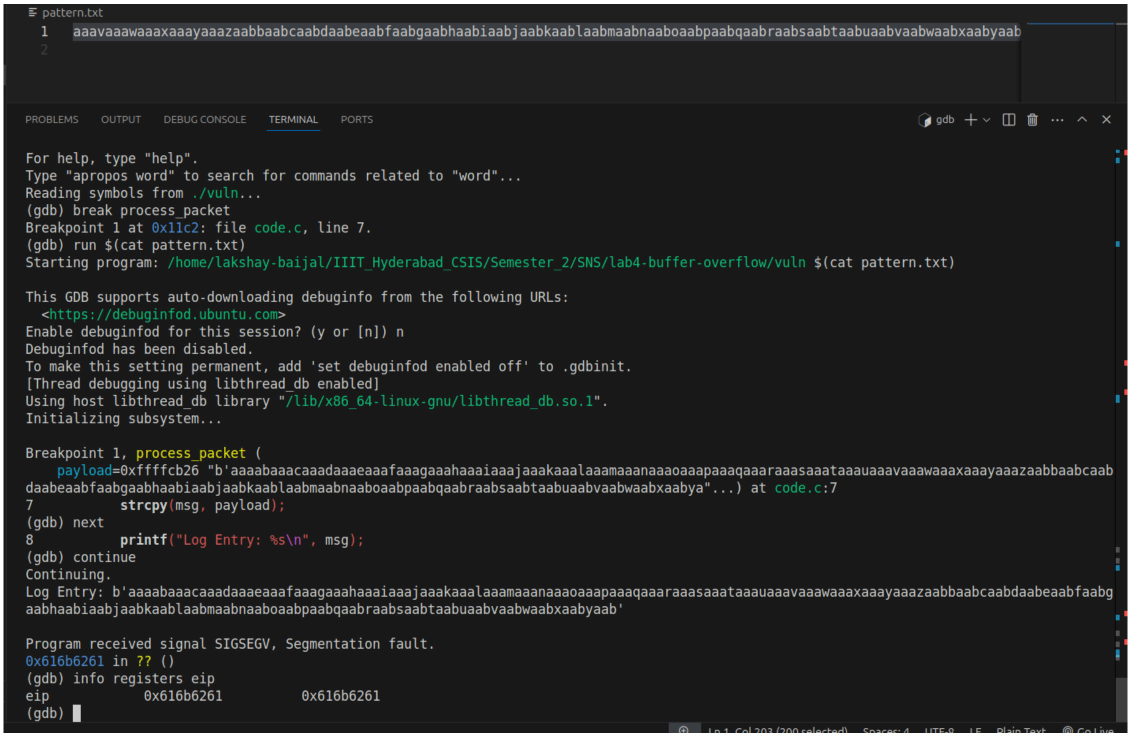


Figure 10: GDB session showing buffer overflow exploitation with pattern.txt

13.1 Pattern File Contents

The attack pattern used for buffer overflow testing:

```
aaavaaavaaavaaayaaazaabbaabcaabdaabeaabfaabgaabhaabiabjaabkaablaabmaabnaabaabpaabgaabraa
```

13.2 GDB Debugging Session

Key steps in the debugging process:

- Breakpoint set at `process_packet` function (line 7 of code.c)
- Program execution with pattern payload
- Segmentation fault occurrence

```
(gdb) break process_packet
Breakpoint 1 at 0x11c2: file code.c, line 7.
(gdb) run ${cat pattern.txt}
```

13.3 Buffer Overflow Evidence

Critical observations from the crash:

```
Program received signal SIGSEGV, Segmentation fault.
0x616b6261 in ?? ()
(gdb) info registers elp
elp    0x616k6261
```

The segmentation fault at address `0x616k6261` (ASCII "akba") indicates:

- Successful overflow of the buffer
- Control of the instruction pointer (EIP)
- The pattern helped identify the exact overflow offset

13.4 Program Execution Flow

The execution trace shows:

```
strcpy(msg, payload);
printf("Log Entry: %s\n", msg);
```

Figure 10 shows the complete GDB session including the memory corruption leading to the segmentation fault. The hexadecimal value in EIP can be used with the pattern offset to calculate the exact buffer size needed for exploitation.

14 Conclusion

14.1 Key Findings

This lab exercise demonstrated several critical aspects of buffer overflow vulnerabilities:

- Successfully generated and utilized cyclic patterns to identify buffer overflow vulnerabilities in the target program
- Verified the vulnerability through GDB debugging sessions, observing:
  - Segmentation faults when overflowing the buffer
  - Control of the instruction pointer (EIP) with crafted input
  - The exact offset where overflow occurs (calculated from pattern)
- Encountered practical challenges with tool configuration including:
  - Virtual environment restrictions for pwntools installation
  - GDB debuginfo configuration requirements

14.2 Lessons Learned

The experiment provided valuable insights into:

- The importance of proper input validation in software development
- How memory corruption vulnerabilities can lead to arbitrary code execution
- The effectiveness of pattern-based attacks in vulnerability analysis
- Practical aspects of working with debuggers and exploitation tools

14.3 Recommendations

Based on the findings, we recommend:

- Implementing proper bounds checking for all memory operations
- Using secure functions like `strncpy` instead of `strcpy`
- Enabling compiler protections (Stack Canaries, ASLR, DEP/NX)
- Regular security testing for memory corruption vulnerabilities

14.4 Further Work

Potential extensions of this research could include:

- Developing a working exploit to gain code execution
- Testing against modern protection mechanisms
- Exploring heap-based overflow variants
- Investigating return-oriented programming (ROP) techniques

This laboratory exercise successfully demonstrated fundamental buffer overflow principles and provided practical experience with vulnerability analysis techniques. The knowledge gained forms a foundation for understanding more advanced memory corruption attacks and defenses in modern systems.