Analysis of Function Call Mechanisms: PLT/GOT vs Direct Syscall Implementation

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

***Abstract*—This report presents a comprehensive analysis of the Procedure Linkage Table (PLT) and Global Offset Table (GOT) mechanisms in dynamically linked C programs compared to direct system call implementations. Through systematic examination of static versus dynamic symbol resolution, this study demonstrates the lazy binding process, captures GOT update mechanics during runtime, and contrasts these with direct syscall approaches that bypass PLT/GOT entirely. The analysis provides insights into modern linking mechanisms, their security implications, and performance characteristics essential for cybersecurity professionals and system developers.**

I. Introduction

The PLT and GOT are essential components in modern dynamically linked programs that enable efficient symbol resolution and code reuse. Understanding these mechanisms is crucial for cybersecurity analysis, reverse engineering, and system optimization. This analysis aims to understand:

- Fundamental differences between static and external symbol handling
- The complete lazy binding resolution process
- Runtime GOT update mechanics and timing
- Comparative analysis with direct syscall implementations
- Security implications of different linking approaches

The study employs a controlled experimental approach using custom C programs and assembly code to demonstrate each mechanism in isolation, providing clear visibility into the underlying processes.

II. Methodology

A. Environment Setup

Component	**Specification**
Operating System	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
Analysis Tools	readelf, objdump, nm

Table I
Development Environment Configuration

B. Disable Address Space Layout Randomization (ASLR)

To ensure consistent memory addresses across multiple runs for analysis purposes:

Listing 1. ASLR Disabling and Verification

```
sudo sysctl -w kernel.randomize_va_space
    =0
# Verify consistent library loading
ldd ./plt_got_test
ldd ./plt_got_test
ldd ./plt_got_test
```

C. Experimental Design

The experiment systematically demonstrates four distinct symbol resolution scenarios through carefully designed test cases:

1) **Static data access** - Direct memory addressing without indirection
2) **Static function calls** - Direct function invocation within the same compilation unit
3) **External data access** - GOT-mediated data access from external modules
4) **External function calls** - PLT/GOT lazy binding for external functions

D. Program Structure

1) Primary Program Design (Appendix A): The main program file was structured to systematically test different memory access patterns and function call mechanisms with clear separation between each test case.

Component	Implementation Purpose
Static Data	Test direct RIP-relative memory addressing
Static Function	Analyze direct function calls without PLT
External References	Setup PLT/GOT resolution mechanisms
Main Function	Execute controlled test sequence

Table II
PRIMARY PROGRAM COMPONENTS AND THEIR TESTING PURPOSE

The test sequence in the main function follows a deliberate order to isolate each mechanism:

- Access static data using direct addressing
- Access external data through GOT indirection
- Execute static function call with direct addressing
- Trigger PLT/GOT resolution with external function call

2) Support Module Design (Appendix B): The secondary file provides external symbols necessary for testing PLT/GOT mechanisms, implemented as separate compilation unit to force dynamic resolution.

Component	Testing Purpose
Global Variable	Enable GOT-mediated data access testing
Global Function	Trigger PLT/GOT lazy binding process

Table III
SUPPORT MODULE COMPONENTS FOR DYNAMIC TESTING

E. Compilation Process

The compilation strategy was designed to enable detailed PLT/GOT analysis while maintaining predictable memory layout:

Stage	Purpose
Separate Compilation	Generate independent object files for dynamic linking
Non-PIE Linking	Create executable with fixed addresses for analysis
Debug Information	Enable runtime debugging and step-through analysis

Table IV
COMPILATION STRATEGY FOR ANALYSIS

Listing 2. Compilation Commands

```
# Compile with debugging symbols and disable PIE for fixed addresses
gcc -g -fno-pie -no-pie -c -o file1.o file1.c
gcc -g -fno-pie -no-pie -c -o file2.o file2.c
gcc -g -fno-pie -no-pie -o plt_got_test file1.o file2.o
```

F. Analysis Methodology

The analysis follows a systematic approach combining static analysis and dynamic debugging:

1) Static Analysis Phase: First, examine the binary structure and symbol information:

Listing 3. Static Analysis Commands

```
# Examine section headers and memory layout
readelf -S plt_got_test

# View symbol table and binding information
nm plt_got_test

# Analyze relocation entries
readelf -r plt_got_test

# Disassemble relevant sections
objdump -d plt_got_test
```

2) Dynamic Analysis Phase:

1) **Case 1: Static Symbol Analysis**
   Analyze direct memory access and function call patterns:

Analysis Type	Target Information
Static Data	RIP-relative addressing calculation
Static Function	Direct call instruction without PLT

Table V
STATIC SYMBOL ANALYSIS TARGETS

Listing 4. Static Analysis Commands

```
gdb ./plt_got_test
pwndbg> b main
pwndbg> run
pwndbg> disass main
# Examine RIP-relative addressing for static data
# Analyze direct call instructions for static functions
```

2) **Case 2: PLT/GOT Resolution Analysis**
   Monitor the complete resolution process for external symbols:
   External Data Access Analysis:

Observation Point	Data Collected
Initial GOT State	Pre-resolution GOT entries
PLT Execution	Resolution chain execution
Memory Updates	GOT entry modifications
Final State	Post-resolution addresses

Table VI
PLT/GOT ANALYSIS COLLECTION POINTS

Listing 5. External Data Analysis

```
# Check data relocations
readelf -r plt_got_test | grep
    shared_value
# Examine GOT entry for data
pwndbg> x/gx &shared_value
# Analyze access pattern in main
pwndbg> disass main
```

External Function Call Analysis:

Listing 6. Function Resolution Tracking

```
# Set strategic breakpoints
pwndbg> b main
pwndbg> b *0x401060   #
    shared_function@plt entry
pwndbg> run

## At main - examine initial state
# Check PLT entry structure
pwndbg> x/3i 0x401060
# Examine initial GOT entry (should
    point to resolver)
pwndbg> x/gx 0x404008

## Continue to PLT entry - observe
    resolution
pwndbg> c
pwndbg> x/3i $rip  # Current PLT
    instruction
pwndbg> si  # Step through resolution
     process

## After resolution - verify GOT
    update
pwndbg> x/gx 0x404008  # Should now
    contain actual function address
```

3) **Case 3: Direct Syscall Implementation (Appendix D)**
Analyze direct system call mechanism bypassing PLT/GOT:

Listing 7. Direct Syscall Analysis

```
gcc -m32 -nostdlib -o syscall_test
    execveShell.S
gdb ./syscall_test
pwndbg> disass _start
# Examine register setup for syscall
# Analyze stack layout for arguments
```

III. Results

A. Binary Structure Analysis

The compiled binary exhibits the expected section layout for PLT/GOT functionality:

Section	Type	Address	Flags
.plt	PROGBITS	0x401020	AX (Allocated, Executable)
.got	PROGBITS	0x403fd8	WA (Writable, Allocated)
.got.plt	PROGBITS	0x403fe8	WA (Writable, Allocated)
.text	PROGBITS	0x401040	AX (Allocated, Executable)

Table VII
CRITICAL BINARY SECTIONS FOR PLT/GOT ANALYSIS

Section permissions are crucial for security and functionality:

- **PLT (AX)**: Executable for resolution code, not writable for security
- **GOT (WA)**: Writable to enable runtime address updates
- **Text (AX)**: Contains main program code and static functions

B. Symbol Resolution Comparison

Symbol Type	Access Method	Resolution Time	Indirection
Static Data	RIP-relative	Link time	None
Static Function	Direct call	Link time	None
External Data	GOT-mediated	Load time	Single
External Function	PLT/GOT	First call (lazy)	Double

Table VIII
COMPREHENSIVE SYMBOL RESOLUTION ANALYSIS

C. Memory Access Patterns

Analysis of the disassembled main function reveals distinct access patterns:

Access Type	Instruction Pattern	Address Calculation
Static Data	mov 0x2e9a(%rip),%eax	RIP + offset
External Data	mov 0x2f8e(%rip),%rax	RIP + GOT offset
	mov (%rax),%eax	Dereference GOT entry
Static Function	call 0x401156	Direct address
External Function	call 0x401060 <shared_function@plt>	PLT entry

Table IX
MEMORY ACCESS PATTERN ANALYSIS

D. PLT/GOT Resolution Process

The complete resolution process demonstrates the lazy binding mechanism:

Stack Position	Content	Purpose
ESP+16	0x0	String terminator
ESP+12	0x68732f2f	"//sh" string
ESP+8	0x6e69622f	"/bin" string
ESP+4	0x0	argv terminator
ESP	ptr to command	argv[0] pointer

Table X
DIRECT SYSCALL STACK LAYOUT

E. Direct Syscall Implementation

The direct syscall approach completely bypasses PLT/GOT mechanisms:

Component	Implementation	Purpose
Stack Setup	String construction in-place	Command preparation
Register Loading	Direct syscall number	System call identification
Syscall Invocation	`int 0x80`	Direct kernel interface

Table XI
DIRECT SYSCALL IMPLEMENTATION COMPONENTS

Stack layout for syscall execution:

Stack Position	Content	Purpose
ESP+16	0x0	String terminator
ESP+12	0x68732f2f	"//sh" string
ESP+8	0x6e69622f	"/bin" string
ESP+4	0x0	argv terminator
ESP	ptr to command	argv[0] pointer

Table XII
DIRECT SYSCALL STACK LAYOUT

IV. DISCUSSION

A. Static vs Dynamic Resolution Mechanisms

The analysis reveals fundamental differences in symbol resolution approaches:

Static Symbol Resolution:

- **Efficiency**: Direct addressing with single instruction execution
- **Predictability**: Fixed addresses determined at link time
- **Security**: Addresses visible in static analysis
- **Example**: `mov 0x2e9a(%rip),%eax` for static data access

Dynamic Symbol Resolution via PLT/GOT:

- **Flexibility**: Runtime address resolution enables shared libraries
- **Lazy Loading**: Functions resolved only when first called
- **Memory Efficiency**: Shared library code reused across processes
- **Overhead**: Additional indirection and resolution complexity

B. GOT Update Mechanism Analysis

The GOT update process follows a precise sequence ensuring thread safety and consistency:

1) **Initial Setup**: GOT entry contains address of PLT resolver stub
2) **First Access**: PLT entry redirects to resolver via GOT
3) **Resolution**: Dynamic linker locates actual function address
4) **Update**: GOT entry atomically updated with resolved address
5) **Subsequent Calls**: Direct jump through updated GOT entry

This mechanism provides several advantages:

- **Performance**: First call overhead, subsequent calls efficient
- **Memory Conservation**: Unused functions never resolved
- **Security**: Write-protected after resolution in some implementations

C. Security Implications

Each approach presents distinct security characteristics:

PLT/GOT Security Considerations:

- **Vulnerability**: Writable GOT enables GOT overwrite attacks
- **Mitigation**: RELRO (Read-Only Relocations) protection
- **Analysis Complexity**: Dynamic resolution complicates static analysis
- **Runtime Flexibility**: Enables function interposition and hooking

Direct Syscall Security Profile:

- **Stealth**: Bypasses library-based monitoring and hooking
- **Simplicity**: Reduced attack surface through minimal dependencies
- **Detection Difficulty**: Harder to intercept for security tools

- **Functionality Limitation**: Fixed syscall numbers and interfaces

D. Performance Analysis

The timing characteristics differ significantly between approaches:

Call Type	First Call	Subsequent	Overhead
Static Function	1 instruction	1 instruction	None
PLT/GOT (cold)	50+ instructions	2 instructions	High initially
PLT/GOT (warm)	2 instructions	2 instructions	Minimal
Direct Syscall	3–5 instructions	3–5 instructions	Context switch

Table XIII
PERFORMANCE COMPARISON OF CALL MECHANISMS

E. Practical Applications

Understanding these mechanisms is crucial for:

Cybersecurity Applications:

- **Reverse Engineering**: Understanding program flow and dependencies
- **Malware Analysis**: Identifying evasion techniques and system interactions
- **Exploit Development**: Leveraging GOT overwrites and ROP chains
- **Defense**: Implementing and understanding protective mechanisms

System Development:

- **Performance Optimization**: Choosing appropriate linking strategies
- **Security Hardening**: Implementing proper memory protections
- **Debugging**: Understanding resolution failures and timing issues

V. CONCLUSION

This comprehensive analysis demonstrates the complete spectrum of function call and symbol resolution mechanisms in modern systems. The PLT/GOT system provides essential flexibility for dynamic linking while introducing complexity and potential security considerations. The lazy binding process optimizes memory usage and startup time but creates resolution dependencies that must be understood for effective system analysis.

Direct syscall implementations offer an alternative approach with different trade-offs, providing stealth and simplicity at the cost of flexibility and maintainability. Understanding these mechanisms is essential for cybersecurity professionals, system developers, and anyone working with low-level system interactions.

The ability to observe and analyze GOT updates in real-time provides crucial insights into runtime symbol resolution, enabling better understanding of system behavior, security implications, and performance characteristics. This knowledge forms the foundation for advanced topics in system security, reverse engineering, and low-level system optimization.

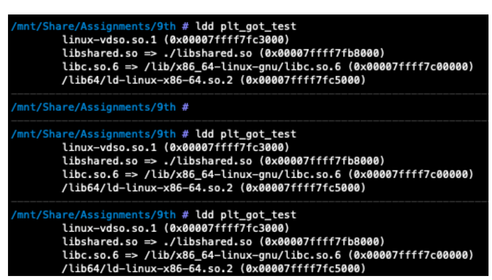

Figure 1. Fixed Memory Address Verification

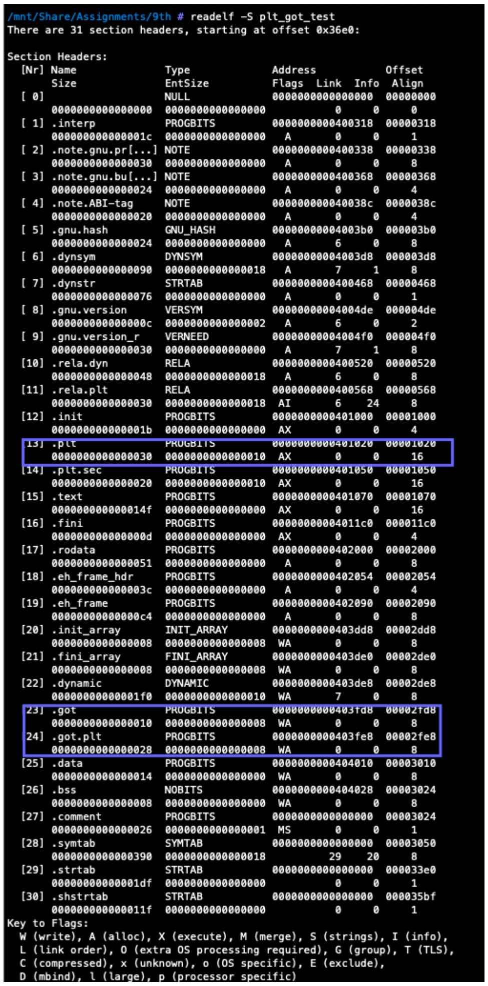

Figure 2. Binary Section Information

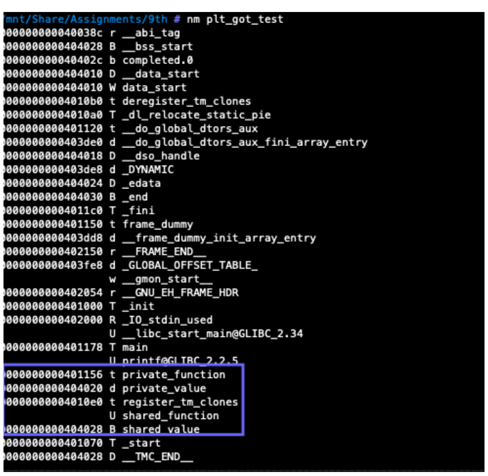

Figure 3. Symbol Table and Binding Information

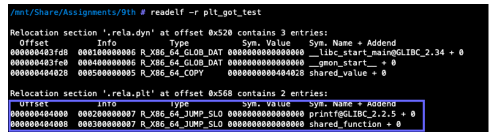

Figure 4. Relocation Entries Analysis

Figure 5. Main Function Disassembly (file1.c)

Figure 6. PLT/GOT Resolution: Before and After States

Figure 7. Direct Syscall Implementation Disassembly (execveShell.S)

VI. Appendices

A. Appendix A: Primary Test Program (file1.c)

```c
// file1.c
#include <stdio.h>

// External variable and function declarations
extern int shared_value;
extern void shared_function(void);

// Static variable - only visible in this file
static int private_value = 100;

// Static function - only visible in this file
static void private_function(void) {
    printf("Inside private function, value: %d\n", private_value);
}

int main() {
    printf("Private value: %d\n", private_value);
    printf("Shared value: %d\n", shared_value);
    private_function();
    shared_function();
    return 0;
}
```

B. Appendix B: External Symbol Provider (file2.c)

```c

// file2.c - This will be compiled into a shared library
#include <stdio.h>

// Shared variable and function definitions
int shared_value = 42;

void shared_function(void) {
    printf("Inside shared function\n");
}
```

C. Appendix C: Alternative Main Implementation (main.c)

```c
#include <stdio.h>

static int stt_data = 1;
int ext_data;

static int Stt_func(){
        printf("static function\n");
}
int Ext_func(){
        printf("extern function\n");
}

int main()
{
        Stt_func();
        Ext_func();
        printf("static variable stt_data: %d\n", stt_data);
        printf("extern variable ext_data: %d\n", ext_data);
}
```

D. Appendix D: Direct Syscall Implementation (execveShell.S)

```asm
// execveShellstorm.S
.global _start
_start:
    xor    %eax,%eax        # Zero EAX register
    push   %eax             # Push NULL terminator
    push   $0x68732f2f      # Push "//sh"
    push   $0x6e69622f      # Push "/bin"
    mov    %esp,%ebx        # First arg: filename pointer
    push   %eax             # Push NULL
    push   %ebx             # Push argv[0]
    mov    %esp,%ecx        # Second arg: argv pointer
    mov    $0xb,%al         # syscall number for execve
    int    $0x80            # Make syscall
```

References

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

[2] Ulrich Drepper, “How To Write Shared Libraries,” Red Hat Inc., 2011.

[3] “System V Application Binary Interface AMD64 Architecture Processor Supplement,” Available: https://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf

[4] Ian Lance Taylor, “ELF Symbol Versioning,” Available: https://www.akkadia.org/drepper/symbol-versioning