Lecture 10-11: Return to libc

Security in libc

Xin Liu

Florida State University
xl24j@fsu.edu

CIS 5370 Computer Security
https://xinliulab.github.io/cis5370.html

Outline

We have already learned that an ”executable file” is a data structure that describes the initial state of a process. Through the Funny Little Executable, we explored the compilation, linking, and loading processes involved in generating an executable file.

Today’s Key Question:

- As the software ecosystem evolved, the need for **”decomposing”** software and dynamic linking emerged!

Main Topics for Today:

- Dynamic Linking and Loading: Principles and Implementation
- Security in **libc**

“Disassembling” an Application

Software Ecosystem Requirements

How Many Executable Files Exist in Our OS?

Have you ever wondered how many executable files are in your system?

- We can count the number of files in `/usr/bin` with:

```
ls -l /usr/bin | wc -l
```

- Most of these executables rely on `libc`. We can verify this with:

```
ldd /usr/bin/bash | grep libc
```

Why Dynamic Linking Matters?

What if every executable included its own copy of `libc`?

- Assume `libc` is 1MB in size.
- There are 1,500 executables in `/usr/bin`.
- Total storage required:

Without Dynamic Linking

$$1MB \times 1500 = 1.5GB$$

With Dynamic Linking:

- The system only needs **one copy** of `libc.so`.
- All executables share the same library at runtime.
- Saves **disk space** and **memory usage**.

”Disassembling” Application Requirements (1)

Achieving Separation of Runtime Libraries and Application Code

- **Library Sharing Between Applications**
  - Every program requires `glibc`.
  - But the system only needs a single copy.
  - Yes, we can check this with the `ldd` command.
- **Decomposing Large Projects**
  - Modifying code does not require relinking massive 2GB files.
  - Example: `lib5370.so`, etc.

Library Dependencies: A Security Risk

The shocking xz-utils (liblzma) backdoor incident (CVE-2024-3094)

- In March 2024, a serious security backdoor was discovered in ‘xz-utils‘, which provides the ‘liblzma‘ compression library.
- The backdoor allowed an attacker to remotely gain control over affected Linux systems.
- The attack was stealthy, bypassing security checks and remaining undetected for months.

How Did This Happen?

- The attacker, known as ‘JiaT75‘, contributed code to ‘xz-utils‘, slowly introducing malicious modifications.
- The malicious code was cleverly hidden within performance improvements and obfuscated commits.
- Even advanced security tools, like Google’s oss-fuzz, did not detect the attack at first.

Why Was This So Dangerous?

- Many Linux distributions (e.g., Debian, Fedora) rely on ‘xz-utils‘ for compression.
- ‘liblzma‘ is a core dependency in multiple system components, including OpenSSH.
- A compromised ‘liblzma‘ meant that attackers could intercept SSH traffic, effectively gaining remote access to Linux machines.

What Was the Response?

- Security researchers discovered and reported the issue before it was fully exploited.
- Major Linux vendors immediately released patches, removing the compromised versions.
- The incident raised concerns about supply chain security in open-source software.

Key Takeaways:

- Open-source projects can be targeted by long-term attacks.
- Even trusted libraries like ‘liblzma‘ can become attack vectors.
- Automated security tools like ‘oss-fuzz‘ are helpful, but not foolproof.
- Regular auditing and manual code reviews are crucial for security.

What If This Happened to Other Critical Libraries?

- Imagine if ‘libc.so‘ or ‘libssl.so‘ were compromised in a similar way.
- How would this affect millions of Linux systems worldwide?

What Happened?

- In 2021, researchers from the University of Minnesota (UMN) intentionally submitted malicious patches to the Linux kernel as part of a security study.
- Their goal was to demonstrate that vulnerabilities could be introduced through seemingly legitimate contributions.
- This research was conducted without prior disclosure to the Linux maintainers.

Community Response

- Greg Kroah-Hartman, a senior Linux maintainer, reacted strongly and reverted all commits from UMN.
- The entire UMN domain (‘umn.edu‘) was temporarily banned from contributing to the Linux kernel.
- The incident raised ethical concerns about conducting security research without consent.

References: UMN Incident Report, Reversion of UMN Commits, S&P’21 Statement on Ethics

”Decomposing Applications” Requirements (2)

Library Dependencies are Also a Code Weakness

- The shocking xz-utils (liblzma) backdoor incident
  - JiaT75 even bypassed oss-fuzz detection
  - Linux incident:
    Greg Kroah-Hartman reverted all commits from umn.edu;
    S&P’21 Statement

What if the Linux Application World was Statically Linked...

- libc releases an urgent security patch $\rightarrow$ all applications need to be relinked
- Semantic Versioning
  - ”Compatible” has a subtle definition
  - ”Dependency hell”

Does It Really Not Exist?

If this is a weapon of mass destruction, does it truly not exist?

- Consider the real world—certain nations possess nuclear weapons.
- They shape global stability.
- Could a similar balance exist in the digital world?

The Computer World Runs on a Fragile Equilibrium

- Zero-day vulnerabilities are discovered, but not always disclosed.
- Some entities have the capability to exploit them but choose restraint.
- Security and control often depend on an unspoken balance between offense and defense.

Verifying ”Only One Copy”

Approach 1: `libc.o`

- Relocation is completed during loading.
    - Loading method: static linking
    - Saves disk space but consumes more memory.
    - Key drawback: **Time** (Linking requires resolving many undefined symbols).

Approach 2: `libc.so` (Shared Object)

- Compiled as **position-independent code**.
    - Loading method: `mmap`
    - However, function calls require an extra lookup step.
- Advantage: Multiple processes share the same `libc.so`, requiring only a single copy in memory.

Verifying ”Only One Copy”

How to Achieve This?

- Create a very large `libbloat.so`
  - Our example: 100M of `nop` (0x90)
- Launch 1,000 processes dynamically linked to `libbloat.so`
- Observe the system’s memory usage:
  - 100MB or 100GB?
- If it’s the latter, the system will immediately crash.
  - However, the **out-of-memory killer** will terminate the process with the highest `oom_score`.
  - We can also use `pmap` to observe the addess of `libbloat.so`.
    - Do all of the addresses point to the same shared library?

How Shared Libraries Shape Process Address Space

Shared Libraries and Virtual Memory

- When a process loads `libc.so`, the operating system maps it into the process's virtual address space.
- The same physical memory holding `libc.so` can be shared across multiple processes.
- This is achieved via `mmap/munmap/mprotect`, which maps shared objects to the address space without duplication.

Address Translation: From Virtual to Physical

- The CPU translates virtual addresses using **paging**.
- In x86 systems, the **CR3 register** holds the base address of the **page table**.
- When a process accesses a function in `libc.so`, the CPU:
  - Reads the virtual address from the instruction.
  - Uses CR3 to locate the correct page table.
  - Translates the virtual address into a physical address.

Implementing Dynamic Loading

All problems in computer science can be solved by another level of indirection. (Butler Lampson).

Dynamic Linking: A Layer of Indirection

At Compilation: Function Calls Use an Indirect Lookup

```
call *TABLE[printf@symtab]
```

At Linking: Symbols Are Collected and Mapped

- The linker gathers all symbol references.
- It generates symbol information and the necessary code.

Symbol Table and Resolution

```
#define foo@symtab 1
#define printf@symtab 2
...

void *TABLE[N_SYMBOLS];

void load(struct loader *ld) {
   TABLE[foo@symtab] = ld->resolve("foo");
   TABLE[printf@symtab] = ld->resolve("printf");
   ...
}
```

Compilation and Linking

- Borrowing from the GNU toolchain works well
  - `ld` is borrowed from `objcopy` (referred)
  - `as` is borrowed from `GNU as` (also referred)

Parsing and Loading

- The rest needs to be done manually
  - `readelf` (`readelf`)
  - `objdump`
  - Similarly, we can ”borrow” `addr2line`, `nm`, `objcopy`, ...
- The loader is simply the ”INTERP” field in ELF

What Have We Implemented?

We "Discovered" the GOT (Global Offset Table)!

- Each dynamically resolved symbol has an entry in the GOT.
- ELF: Relocation section `.rela.dyn`.

Offset	Info		Type
000000000003fe0	00030006	R_X86_64_GLOB_DAT	printf@GL

Examining Offset 0x3fe0 in the GOT using objdump:

- `printf("%p", printf);` reveals that this is not the actual `printf`.
- `*(void **)(base + 0x3fe0)` gives the real address.
- We can set a "read watchpoint" to see who accesses it.

Implementing Dynamic Linking and Loading of Code

- `main (.o)` calls `printf (.so)`
- `main (.o)` calls `foo (.o)`

Challenge: How to Decide Whether to Use a Lookup Table?

```
int printf(const char *, ...);
void foo();
```

- Should it be determined within the same binary (resolved at link time)?
- Or should it be handled within the library (loaded at runtime)?

A Historical Legacy Issue: Compile First, Link Later

Compiler Option 1: Fully Table-Based Indirect Jump

```
ff 25 00 00 00 00 call *FOO_OFFSET(%rip)
```

- Each call to `foo` requires an additional table lookup, leading to performance inefficiency

Compiler Option 2: Fully Direct Jump

```
e8 00 00 00 00 call <reloc>
```

- `%rip`: `0000555982b7000`
- `libc.so`: `00007fdcfd800000`
  - The difference is `2a8356549000`
- A 4-byte immediate cannot store such a large offset, making the jump impossible

What Can We Do?

For Performance, "Fully Direct Jump" is the Only Choice

```
e8 00 00 00 00 call <reloc>
```

- If a symbol is resolved at link time (e.g., `printf` from dynamic loading), then a small piece of code is "synthesized" in `a.out`:

```
printf@plt:
    jmp *PRINTF_OFFSET(%rip)
```

- This leads to the invention of the **PLT (Procedure Linkage Table)**!

Do We Really Need the PLT?

- If compilation and linking were done together, we would already know the target of every `call` instruction.

```
puts@PLT:
    endbr64
    bnd jmpq *GOT[n] // *offset(%rip)
```

- Why does the PLT use `endbr64` and `bind jmpq` for jump resolution?
- In reality, there are many ”other” possible solutions.

ELF Dynamic Linking and Loading

Dynamic Loading and Linking of Data

- `main (.o)` accesses `stderr` (`libc.so`)
- `libjvm (.so)` accesses `stderr` (`libc.so`)
- `libjvm (.so)` accesses `heap` (`libjvm.so`)
- Just like code, the compiler does not know where the data is located.

Same Challenge as Code: What Exactly is a Symbol?

```
extern int x;
```

- Is it in the same binary (resolved at link time)? Or is it in another library?

PLT: The Unresolved Data Access Issue

For Data, We Cannot Use "Indirect Jump"!

- `x = 1`, within the same `.so` (or executable)

```
mov $1, offset_of_x(%rip)
```

- `x = 1`, in a different `.so`

```
mov GOT[x], %rdi
mov $1, (%rdi)
```

An Inelegant Solution

- `-fPIC` by default adds an extra layer of indirection for all `extern` data accesses.

```
__attribute__((visibility("hidden")))
```

Takeaways: Finding the Core Structure in Complexity

Understanding Dynamic Linking Through Implementation

- By attempting to implement dynamic linking and loading ourselves, we gain deep insights into the process.
- In doing so, we ”invent” key ELF concepts, such as:
  - The **Global Offset Table (GOT)** for resolving addresses dynamically.
  - The **Procedure Linkage Table (PLT)** for indirect function calls.
- This hands-on approach reveals the underlying principles behind complex systems.