Lecture 10-11: 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

”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: `libyjm.so`, `libart.so`, etc.
  - **NEMU:** ”Insert the CPU into the motherboard.”

C Standard Library (libc)

Program Dependencies

Why Do We Need `libc`?

- "Bare-metal" Programming: Works (and technically enough), but not user-friendly
- Essential definitions that all programs use:
  - `stddef.h` – Provides types like `size_t`
  - `stdint.h` – Defines standard integer types like `int32_t`, `uint64_t`

```c
#include <stdio.h>
#include <assert.h>

int main() {
      int a;

   printf("Size␣of␣int␣=␣%ld\n", sizeof(int));
   printf("Size␣of␣long␣=␣%ld\n", sizeof(long));

      assert(sizeof(a) == 4);
}
```

Can you guarantee that this code will pass on all machines?

- **Preferred way**: Use fixed-width types, e.g., `int32_t a;` for consistency across platforms.
- **Best practice**: Refer to the official C++ reference for detailed information on types and usage.

Why Do We Need `libc`?

- "Bare-metal" Programming: Works (and technically enough), but not user-friendly
- Essential definitions that all programs use:
  - `stddef.h` – Provides types like `size_t`
  - `stdint.h` – Defines standard integer types like `int32_t`, `uint64_t`
  - `inttypes.h` – Defines formats for printing integer types

```
#include <stdio.h>
#include <stddef.h>
#include <stdint.h>

int main() {
   int64_t x = 1;
   printf("%ld\n", x); // %ld : this is a long = 4 bytes
        for 32-bit machine!
}
```

Why Do We Need `libc`?

- "Bare-metal" Programming: Works (and technically enough), but not user-friendly
- Essential definitions that all programs use:
  - `stddef.h` – Provides types like `size_t`
  - `stdint.h` – Defines standard integer types like `int32_t`, `uint64_t`
  - `inttypes.h` – Defines formats for printing integer types
  - `stdbool.h`
  - `float.h`
  - `limits.h`
  - `stdarg.h`
    - Used for handling variable arguments (essential in `syscall`, though custom `syscall0`, `syscall1`, etc., can be more efficient)

Making System Calls Easier to Use!

System Calls: The Minimal Interface of the OS

- System calls provide the OS's smallest, most compact interface.
- Not all system calls are as straightforward as `fork`; some require additional setup.

Low-Level API:

```
extern char **environ;
char *argv[] = { "echo", "hello", "world", NULL, };
if (execve(argv[0], argv, environ) < 0) {
  perror("exec");
}
```

High-Level API:

```
execlp("echo", "echo", "hello", "world", NULL);
system("echo␣hello␣world");
```

Comparison:

- Low-level APIs like `execve` offer more control but require more setup.
- High-level APIs like `execlp` and `system` simplify common tasks, making code easier to read and write.

Endless Abstractions: Layers of Encapsulation

- **Encapsulation (1): Pure Computations**
- **Encapsulation (2): File Descriptors**
- **Encapsulation (3): More Process / OS Functions**
- **Encapsulation (4): Address Space**

Specification Overview (similar to Lab1):

- Manage a set of non-overlapping intervals $M = \{[\ell_0, r_0), [\ell_1, r_1), \ldots, [\ell_n, r_n)\}$ within a large interval $[L, R)$.

Operations

- `malloc(s)`:
  - Returns a segment of memory of size s.
  - May request additional memory from the OS if needed (observe this with `strace`).
  - Can "deny" requests if memory is insufficient.
- `free(`ℓ`, `r`)`:
  - Given a starting address ℓ, removes the interval $[\ell, r) \in M$.

Considerations

- Inspired by concepts from *Introduction to Algorithms*.
- **Thread Safety**: Ensuring `malloc` and `free` work correctly in a multithreaded environment.
- **Scalability**: Handling multiple allocation and deallocation requests efficiently becomes a significant challenge.

Premature optimization is the root of all evil.
— D. E. Knuth

Optimizing without workload analysis is risky

- **Workload Analysis:** Analyze common memory usage patterns to guide optimization.
- Key Principle: Allocating objects of size $O(n)$ should typically involve at least $\Omega(n)$ read/write operations. Otherwise, it's a performance bug.

Further Reading for Workload Analysis:

- Mimalloc: free list sharding in action (APLAS'19)

Types of Memory Allocations:

- **Small Objects (high frequency):**
  - Strings, temporary objects (tens to hundreds of bytes), with varying lifespans.
- **Medium-Sized Objects (moderate frequency):**
  - Arrays, complex objects with longer lifespans.
- **Large Objects (low frequency):**
  - Huge containers, allocators, with very long lifespans.

Key Challenges:

- **Parallelism:** Allocations occur across all processors; parallel strategies are essential.
- **Data Structures:** Using linked lists or interval trees (e.g., first fit) is not ideal for high concurrency.

Designing Two Systems for Memory Allocation:

- **Fast Path**
  - Optimized for high performance and parallelism.
  - Covers most allocation cases with minimal latency.
  - Has a small probability of failure, in which case it falls back to the slow path.
- **Slow Path**
  - Not focused on speed, but handles complex cases reliably.
  - Manages difficult allocations that cannot be handled by the fast path.

Common Pattern in Computer Systems:

- This fast-and-slow path design is common in systems, such as cache mechanisms or futex (as we discussed earlier).
- The fast path handles frequent, simple requests, while the slow path ensures robustness for exceptional cases.

`malloc`: Fast Path Design

Goal: Enable all CPUs to allocate memory in parallel

- **Thread-Local Allocation Buffer (TLAB):**
  - Each thread has its own “territory” or buffer for allocations.
  - By default, memory is allocated from its own buffer, minimizing contention.
- **Efficient Locking:**
  - Locks are rarely contested because threads typically allocate from their own buffer.
  - Only in rare cases, such as when memory is freed by another CPU, might a lock be required.
- **Global Pool Backup:**
  - When a thread’s buffer runs low, it borrows memory from a global pool.
  - This approach allows for minor memory waste but improves allocation speed.
- **Alignment to 2^k Bytes:**
  - Aligning allocations to 2^k bytes helps maintain efficient memory access and reduces fragmentation.

Small Memory Allocation: Segregated List (Slab)

Allocation Strategy: Segregated List (Slab)

- Each **slab** contains objects of the same size.
- Each thread has its own slab for each object size.
- **Fast Path:** Allocation is completed immediately from the thread-local slab.
- **Slow Path:** Calls `pgalloc()` to allocate additional memory.

Two Implementation Approaches:

- **Global List:** A single global list of slabs.
- **List Sharding:** A small list per page, reducing contention.

Reclaiming Memory:

- Freed objects are returned directly to their respective slab.
- If the slab belongs to another thread, a **per-slab lock** is needed to prevent data races.

Moving Beyond C: Building on `libc`

- **C++ Compiler:** Expands on `libc` to support the C++ Standard Library.
- **OpenJDK (HotSpot):** Java runtime built on layers extending from C.
- **V8 (JavaScript):** A JavaScript engine that relies on foundational libraries for execution.
- **CPython:** The C-based Python interpreter extends further from `libc`.
- **Go:** Initially compiled with C, now capable of self-compilation. ("Goodbye, C!")

The Endless Cycle of Encapsulation:

- Each language and runtime builds upon lower-level abstractions, creating a layered stack.
- Over time, these layers form a "patched-up" world of interconnected technologies, making our computing environment both powerful and complex.

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 � all applications need to be relinked
- Semantic Versioning
  - "Compatible" has a subtle definition
  - "Dependency hell"

Decomposing Applications

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`.

Prototypes are easy. Production is hard. (Elon Musk)

The Address Space Appears as ”Thousands of Contiguous Memory Segments”

- Maintained via `mmap`, `munmap`, and `mprotect`
- In reality, it is a ”mirage” maintained by the paging mechanism

Dynamic Loading

Let's create our own dynamic loading.

dlbox: Reimplementing binutils Once Again

Compilation and Linking

- Stealing the GNU toolchain works fine
  - `ld` = `objcopy` (stolen)
  - `as` = `GNU as` (also stolen)

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

Implementing the Dynamic Loader (2)

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")))
```

Understanding Executable Files and Buffer Overflow

- What is an Executable File?
  - An executable file is a data structure (a sequence of bytes) that describes the initial state of a state machine.
  - The loader transfers this ”initial state” into the operating system.
  - It is difficult to read because it was never designed for human readability.
- It helps us understanding the buffer overflow:
  - Why can we use `gdb` to compute stack offsets that helps analyze function call stack structures?
  - Observing local variables, return addresses, and how an overflow can overwrite the return address.
  - Redirecting execution to malicious code (e.g., shellcode) reveals how control flow is hijacked.
  - This process provides insight into program execution, stack management, and security vulnerabilities.