Praktikum: Binary Exploitation

Introduction

Julian Kirsch    Sebastian Vogl    Lorenz Panny

Technische Universität München

14.04.2015
Contents

x86_64 (/Linux)
  Architecture overview
  Virtual memory
  ELF & dynamic linking
  Call stack
  Calling conventions
  Exploit mitigation

Helpful tools
  GDB
  objdump
  readelf
  Python

First assignment
  SSH Tunnel
Complex instruction set computer (→ variable-length instructions)
Architecture overview

- **Complex instruction set computer** (⇔ variable-length instructions)
- Pitfall: little-endian architecture!, i.e. least significant bytes first
Architecture overview

- **Complex instruction set computer** (⟷ variable-length instructions)
- Pitfall: little-endian architecture!, i.e. least significant bytes first
- 16 registers of 64 bits each
  - instruction pointer `%rip` (pitfall: points to the next instruction)
  - stack pointer `%rsp` (pitfall: stack grows to lower addresses!)
- 14 general-purpose registers
Architecture overview

- Complex instruction set computer (→ variable-length instructions)
- Pitfall: little-endian architecture!, i.e. least significant bytes first
- 16 registers of 64 bits each
  - instruction pointer %rip (pitfall: points to the next instruction)
  - stack pointer %rsp (pitfall: stack grows to lower addresses!)
  - 14 general-purpose registers
- Linux: system calls via syscall or int $0x80 (legacy) instruction
Virtual memory

- Programs have separate, disjoint address spaces
- **Virtual addresses** map to **physical memory pages** via page table
Virtual memory

- Programs have separate, disjoint address spaces
- Virtual addresses map to physical memory pages via page table
- x86_64/Linux (userspace): pages of \(0x1000 = 4096\) bytes
Virtual memory

- Programs have separate, disjoint address spaces
- Virtual addresses map to physical memory pages via page table
- x86_64/Linux (userspace): pages of $0x1000 = 4096$ bytes
- Memory access errors (unmapped addresses, protection errors, ...): handled by the kernel, typically kills the process (segmentation fault)
Main goal: Enable dynamic linking of shared libraries, i.e. standard libraries need not be compiled into every program.
Executable & Linking Format

- Main goal: Enable dynamic linking of shared libraries, i.e. standard libraries need not be compiled into every program
- On program startup, a loader uses the information in the executable file to supply the required imports
Executable & Linking Format

- An ELF file contains...
  - Information about the target environment, architecture, ...
  - The program image, i.e. code, static data, ...
  - A description of how everything shall be put together
  - Optional: Debug information like variable names, line numbers, ...
Executable & Linking Format

- An ELF file contains...
  - Information about the target environment, architecture, ...
  - The program image, i.e. code, static data, ...
  - A description of how everything shall be put together
  - Optional: Debug information like variable names, line numbers, ...

- Segments tell the loader which parts of the file go where in memory
An ELF file contains...

- Information about the target environment, architecture, ...
- The program image, i.e. code, static data, ...
- A description of how everything shall be put together
- Optional: Debug information like variable names, line numbers, ...

Segments tell the loader which parts of the file go where in memory

Symbols specify where a function or global variable starts
Executable & Linking Format

- An ELF file contains...
  - Information about the target environment, architecture, ...
  - The program image, i.e. code, static data, ...
  - A description of how everything shall be put together
  - Optional: Debug information like variable names, line numbers, ...

- Segments tell the loader which parts of the file go where in memory
- Symbols specify where a function or global variable starts
- Relocations tell the loader which symbols from imported libraries the program requires and where it expects their address
An ELF file contains...
- Information about the target environment, architecture, ...
- The program image, i.e. code, static data, ...
- A description of how everything shall be put together
- Optional: Debug information like variable names, line numbers, ...

- **Segments** tell the loader which parts of the file go where in memory
- **Symbols** specify where a function or global variable starts
- **Relocations** tell the loader which symbols from imported libraries the program requires and where it expects their address
- **Sections** are relevant to linking only
The **Procedure Linkage Table**

- **Problem:** large programs use library functions in many places
  \[\Rightarrow\] huge relocation table

- **Solution:** PLT (gcc, clang)
The Procedure Linkage Table

- Problem: large programs use library functions in many places
  ⇒ huge relocation table
- Solution: PLT (gcc, clang)
- Each library function gets a "proxy"; address resolved only once
The **Procedure Linkage Table**

- Problem: large programs use library functions in many places
  ⇒ huge relocation table
- Solution: **PLT** (gcc, clang)
- Each library function gets a "proxy"; address resolved only once

```c
/* .got.plt */
.printf_addr:
    /* loader fills this in: */
    .qword printf

/* .plt */
printf@plt:
    jmp *.printf_addr

// call printf becomes call printf@plt
```
The Procedure Linkage Table

- Problem: large programs use library functions in many places ⇒ huge relocation table

- Solution: PLT (gcc, clang)

- Each library function gets a "proxy"; address resolved only once

```c
/* .got.plt */
.printf_addr:
    /* loader fills this in: */
    .qword printf

/* .plt */
.printf@plt:
    jmp *.printf_addr
```

- call printf becomes call printf@plt

- Typically, the .got.plt entry is filled in lazily, that is: when the function is first called
Call stack

► Rationale:
  ► Where to continue after current function is finished?
  ► Space for local variables
  ► Single global memory region for each function is unsuitable:
    recursion, memory requirement, (cache efficiency)
Call stack

- **Rationale:**
  - Where to continue after current function is finished?
  - Space for local variables
  - Single global memory region for each function is unsuitable: recursion, memory requirement, (cache efficiency)

- **Solution:** Stack data structure
  - Each function stores its information in a stack frame
  - Frames are pushed on top of the call stack on procedure entry
  - ...and popped on procedure exit
Call stack: example

```c
char *greeting(char *dest, char *name)
{
    sprintf(dest, "Hello, %s!", name);
    return dest;
}

void greet(char *whom)
{
    char buf[0x40];
    puts(greeting(buf, whom));
}

int main() { greet("world"); }
```
Call stack: example

```c
// char *greeting(char *dest, char *name)
{
    sprintf(dest, "Hello, %s!", name);
    return dest;
}

// void greet(char *whom)
{
    char buf[0x40];
    puts(greeting(buf, whom));
}

// int main()
{
    greet("world");
}
```
Calling conventions (caller)

- Put arguments into
  1. %rdi
  2. %rsi
  3. %rdx
  4. %rcx (pitfall: system calls use %r10 instead!)
  5. %r8
  6. %r9
  7. (stack)
Calling conventions (caller)

- Put arguments into
  1. %rdi
  2. %rsi
  3. %rdx
  4. %rcx (pitfall: system calls use %r10 instead!)
  5. %r8
  6. %r9
  7. (stack)

- Use call instruction to push return address and transfer control
Calling conventions (caller)

- Put arguments into
  1. `%rdi`
  2. `%rsi`
  3. `%rdx`
  4. `%rcx` (pitfall: system calls use `%r10` instead!)
  5. `%r8`
  6. `%r9`
  7. (stack)

- Use call instruction to push return address and transfer control

- Example:

```assembly
main:
    mov $.greeting, %rdi
    mov $.name, %rsi
    call printf
    xor %rdi, %rdi
    call exit

.greeting: .asciz "Hello, %s!\n"
.name: .asciz "world"
```
Calling conventions (callee: prologue)

- Set up stack frame (space for local variables)
Calling conventions (callee: prologue)

- Set up stack frame (space for local variables)
- Example:
  - without frame-pointer:
    ```
    fun :
    sub $0x100 , %rsp
    /* ... */
    mov %rdi , 0x42(%rsp)
    ```
Calling conventions (callee: prologue)

- Set up stack frame (space for local variables)
- Example:
  - without frame-pointer:
    ```assembly
    fun :
        sub $0x100, %rsp
        /* ... */
        mov %rdi, 0x42(%rsp)
    
    with frame-pointer:
    
    fun :
        push %rbp
        mov %rsp, %rbp
        sub $0x100, %rsp
        /* ... */
        mov %rdi, −0xbe(%rbp)
    ```
Calling conventions (callee: epilogue)

- Clean up stack frame (restore state before call)
- Put the return value in the %rax register
- Use the ret instruction to continue after the call
Calling conventions (callee: epilogue)

- Clean up stack frame (restore state before call)
- Put the return value in the %rax register
- Use the ret instruction to continue after the call
- Example:
  - without frame-pointer:
    ```
    mov 0xccc(%rsp), %rax
    add $0x100, %rsp
    ret
    ```
  - with frame-pointer:
    ```
    mov -0xccc(%rbp), %rax
    mov %rbp, %rsp
    pop %rbp
    ret
    ```
    (no 0x100 ⇒ frame-pointer enables dynamic allocation on the stack!)
Calling conventions (callee: epilogue)

- Clean up stack frame (restore state before call)
- Put the return value in the `%rax` register
- Use the `ret` instruction to continue after the call

Example:

- Without frame-pointer:
  
  ```
  mov 0xcc(%rsp) , %rax
  add $0x100 , %rsp
  ret
  ```

- With frame-pointer:
  
  ```
  mov −0xcc(%rbp) , %rax
  mov %rbp , %rsp
  pop %rbp
  ret
  ```

(no $0x100 ⇒ frame-pointer enables dynamic allocation on the stack!)
Non-executable mapping / \( w^x \)

- Address ranges that are **executable** must not be **writable**
- ...and vice-versa
- Enforced by CPU and kernel
Address space layout randomization

- Stack and dynamically linked libraries are mapped at (pseudo)random addresses
- Idea: attacker can not (ab)use these objects in exploits
- Randomization may also include the executable image itself; a.k.a. position-independent executable
The GNU Debugger

Run

```
gdb /path/to/program
```

or

```
gdb --args /path/to/program arguments
```
The GNU Debugger

- Get help
  - General: help
  - About a command: help $cmd
The GNU Debugger

Controlled execution

- Start the program: `run`
- Set a breakpoint at a function: `break main`
- Set a breakpoint at an address: `break *0x400100`
- Single-step (execute one instruction): `stepi`
- Continue execution: `continue`
- Step over call: `next`
- Stop on forks and execves: `catch fork / catch exec`
The GNU Debugger

- Dump registers
  - `info registers`
  - `print/x $rax`
The GNU Debugger

- Dump registers
  - `info registers`
  - `print/x $rax`

- Dump (examine) memory
  - `x/8i $rip` (disassemble the next 8 instructions)
  - `x/40wx $rsp` (show the 40 topmost 32-bit words on the stack in hex)
  - `x/s 0x401000` (show the null-terminated string starting at 0x400000)
The GNU Debugger

- Dump registers
  - `info registers`
  - `print/x $rax`
- Dump (examine) memory
  - `x/8i $rip` (disassemble the next 8 instructions)
  - `x/40wx $rsp` (show the 40 topmost 32-bit words on the stack in hex)
  - `x/s 0x401000` (show the null-terminated string starting at 0x400000)
- Generally:
  - `x/`
    - count
    - type (instruction, string, byte, 32-bit word, g: 64-bit word)
    - format (hexadecimal, signed decimal, unsigned decimal, character)
    - address expression (may involve arithmetic, e.g. `$rsp-0x20`)
The GNU Debugger

- Examine call stack (unreliable)
  - Stack frames: `info stack`
  - Current frame: `info frame`

- Get process info
  - Process id: `info proc`
  - Mapped memory ranges: `info proc mappings`
  - Memory protections: `cat /proc/$pid/maps` in a shell
  - A whole bunch of stuff: `info proc all`
The GNU Debugger

- File `.gdbinit` can be used to execute commands on every startup
- ...including hooks on common events, like program interruptions
The GNU Debugger

- File `.gdbinit` can be used to execute commands on every startup
- ...including hooks on common events, like program interruptions
- Check out https://home.in.tum.de/~panny/?l=gdbinit
Dumping binary object files

- Dump data and disassemble
  - AT&T syntax: `objdump -sd /path/to/program`
  - Intel syntax: `objdump -sdM intel /path/to/program`
Reading ELF files

- Show ELF header: `readelf -h`
- Show program headers (segments!): `readelf -l`
- Show symbols: `readelf -s`
- Show relocations: `readelf -r`
- Show everything: `readelf -a`
Python

...is found to be a very useful tool.

- **socket**: `socket()`, `connect()`, `recv()`, `send()`
- **struct**: `pack()`, `unpack()`
- **binascii**: `hexlify()`, `unhexlify()`
- **Helper functions**: `str.encode()`, `str.decode()`, `str.format()`, `hex()`, `int()`
- **Tons of others.**
First assignment

- Tasks in /opt/tasks/$foo on praksrv
- Scoreboard at https://pwning.sec.in.tum.de.
- Submission to hp-abgabe@sec.in.tum.de until the deadline (multiple files: .tgz or .zip)
- Mailing list: hacker-praktikums@sec.in.tum.de
SSH Tunnel

- On your machine: `ssh -L 1234:hacky1:13701
team9@praksrv.sec.in.tum.de`

- ...then `localhost:1234` is forwarded to `hacky1:13701`