Binary Analysis and Secure Coding

Federico Conti - 4991779

2025/26

Contents

Binary Analysis

Introduction

Binary analysis involves understanding the structure and content of files, which are essentially sequences of bytes. File extensions are not critical for the operating system; instead, the content determines how files are interpreted. Different parsers may interpret the same sequence of bytes in various ways:

- **ZIP/JAR Parsers**: Look for the "End of Central Directory" at the end of the file.
- **BMP/PE/ELF Parsers**: Expect a header at the beginning of the file.
- **PDF Parsers**: Prefer the header at the beginning, but most viewers accept it within the first 1024 bytes.

Files that conform to multiple file format specifications are known as **polyglots**. For example:

- **Janus.com**: A 512-byte file that functions as an x86 bootloader, COM executable, ELF, ZIP, RAR, GNU Multiboot2 Image, and Commodore 64 PRG executable.
  Read more here.
- **Ange Albertini's Work**: Explore fascinating insights into file formats and polyglots in Funky File Formats.
- **Magika**: A tool for file format analysis. GitHub Repository.
- **Polyfile**: A utility for analyzing and manipulating polyglot files. GitHub Repository.

Tools for Executable Analysis

These tools and resources provide a solid foundation for understanding and analyzing binary files.

ELF-Specific Tools - **readelf** and **objdump**: Tools for analyzing ELF files. `objdump` can also disassemble and parse PE files. - **XELFViewer**: A tool for viewing ELF files. GitHub Repository.

PE-Specific Tools - **dumpbin**: Similar to `objdump`, but for PE files. - **PE Bear**: A lightweight tool for PE analysis. GitHub Repository. - **PE Studio**: A comprehensive tool for analyzing PE files. Official Website.

General Tools

- **hte**: A hex editor that supports ELF and PE files. Official Website | GitHub Repository.

we trust the compiler?

Source code written in C is essentially a plain text file. However, CPUs do not understand C; they execute only machine code. Therefore, to run a program, the source code must be compiled into an executable file.

A critical question arises: can we trust the compiler to preserve the semantics of the source code in the compiled program?

- In theory: Yes, the compiler should maintain the semantics.
- In practice: Compilers can introduce surprises.

For example, consider a source file with a `printf` call. After compilation, the executable may not contain a `printf` call at all. Instead, it might use a `puts` call.

The compiler makes this substitution for two main reasons:

1. **Semantic Preservation**: If the string being printed is simple (e.g., `Hello World\n`) and does not include format specifiers like `%d` or `%s`, the logic of `printf` is unnecessary.
2. **Optimization**: The `puts` function is more efficient because it directly prints the string followed by a newline, without checking for format specifiers. This optimization results in lighter, faster code while maintaining the program's behavior.

If you want the compiler to retain the exact `printf` call, you can use specific compiler flags, such as:

```
gcc file.c -fno-builtin
```

Static vs Dynamic Analysis

When analyzing binaries, two primary approaches are used: static analysis and dynamic analysis. Each has its strengths and limitations.

Dynamic Analysis Dynamic analysis involves executing the binary and observing its behavior during runtime. Key characteristics include:

- Advantages:
  - Simpler to perform as it observes runtime states directly.
  - Provides insights into the binary's behavior during execution.
- Disadvantages:
  - Potentially harmful, especially when analyzing malware.
  - Limited to the specific execution path taken during the run, which may miss interesting or critical parts of the code. For example:
    ```
    if (random() == 0xcafebabe) {
            /* interesting stuff */
    }
    ```
    If the condition is not met during execution, the "interesting stuff" remains unobserved.

Static Analysis Static analysis involves examining the binary without executing it. Key characteristics include:

Advantages:

- Enables analysis of the entire binary in one go.
- Does not require a compatible CPU or system to run the binary.
- Disadvantages:
- Provides little to no knowledge of runtime states.
- Can be challenging to identify the most relevant or interesting parts of the binary.

Both approaches are complementary and often used together to gain a comprehensive understanding of a binary's behavior and structure.

Compilation and Linking

Compilation Stages

1. **Preprocessing**: Handles macros and `#include` directives.
2. **Compiler**: Translates preprocessed code into assembly (.s).
3. **Assembly**: Converts assembly into object files (.o), which are relocatable and contain unresolved references.
4. **Linking**: Resolves references and produces the final executable.

By default, **dynamic linking** is used:

- Libraries are not copied into the executable.
- The executable contains metadata declaring required libraries.
- At runtime, the OS or a user-space process loads and maps the libraries.

Examples:

- **Linux**: Dynamic linker runs in user space.
- **Windows**: Library resolution occurs in the kernel.

Dynamic linking defers library resolution to runtime, reducing executable size and enabling updates without recompilation.

Assembler Files

Assembler files are a step toward machine code and include:

- **Assembler instructions**: Translated into machine code (e.g., `push rbp` → `55`).
- **Pseudo-instructions**: Emit arbitrary data/bytes (e.g., `.string "Hello world!"` → `48 65 6c 6c 6f...`).
- **Directives**: Provide the assembler with information (e.g., `.text` for the `.text` section).

Assembler files use symbols (names) instead of addresses, which are resolved during linking.

- **Defined**: e.g., `main: push rbp` . . .
- **Undefined**: e.g., `call printf` gets translated into:
  1. `e8 ?? ?? ?? ??`; that is, machine code with "holes"
  2. and metadata: relocations that tell the linker how to "fill such holes"

Example

```
gcc -c hello.c -o hello.o
gcc -c hello.c -o hello.o
objdump -d -M intel hello.o
```

```
0000000000000000 <say_hello_world>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   rbp
   5:   48 89 e5                mov    rbp,rsp
   8:   48 8d 05 00 00 00 00    lea    rax,[rip+0x0] # f <say_hello_world+0xf>
   f:   48 89 c7                mov    rdi,rax
  12:   b8 00 00 00 00          mov    eax,0x0
  17:   e8 00 00 00 00          call   1c <say_hello_world+0x1c>
  1c:   90                      nop
  1d:   5d                      pop    rbp
  1e:   c3                      ret

000000000000001f <newline>:
  1f:   f3 0f 1e fa             endbr64
  23:   55                      push   rbp
  24:   48 89 e5                mov    rbp,rsp
  27:   bf 0a 00 00 00          mov    edi,0xa
  2c:   e8 00 00 00 00          call   31 <newline+0x12>
  31:   90                      nop
  32:   5d                      pop    rbp
  33:   c3                      ret

0000000000000034 <main>:
  34:   f3 0f 1e fa             endbr64
  38:   55                      push   rbp
  39:   48 89 e5                mov    rbp,rsp
  3c:   e8 bf ff ff ff          call   0 <say_hello_world>
  41:   b8 00 00 00 00          mov    eax,0x0
  46:   e8 00 00 00 00          call   4b <main+0x17>
```

```
4b:   b8 00 00 00 00          mov    eax,0x0
50:   5d                      pop    rbp
51:   c3                      ret
```

Note: `call` with different addresses: 3 of the 4 calls have 00 00 00, but one is different:

```
// Example of the relocations to be resolved by the linker:

17:   e8 00 00 00 00          call   1c <say_hello_world+0x1c>  // → printf
2c:   e8 00 00 00 00          call   31 <newline+0x12>          // → putchar
46:   e8 00 00 00 00          call   4b <main+0x17>             // → newline

// But this is already solved:
3c:   e8 bf ff ff ff          call   0 <say_hello_world>        // Local, already calculated
```

Even if `newline` is in the same file, the assembler leaves `00 00 00` because during linking it could be overwritten by a symbol with the same name in another object file. Only static symbols are guaranteed not to be overwritten.

File Sections in .o

An object file's contents are organized into sections:

- **.text**: Contains machine code (historical name, now less meaningful).
- **.data / .rdata or .rodata**: Stores initialized data or read-only data.
- **.bss**: Reserved space for uninitialized data.

The acronym `.bss` originates from "before start symbol," a historical term retained for compatibility with early linkers.

Other minor sections may exist, but the general structure of an executable or object file is divided into these main areas of code and data.

During the linking process:

1. Sections from various object files are concatenated.
2. Symbol tables are unified.
3. Undefined symbols are resolved.

Example

- **object1.o**: Defines `function1` (in `.text` at offset 0) and `data1` (in `.data` at offset 0). Calls `function2` (undefined).
- **object2.o**: Defines `function2` (in `.text` at offset 0).

If the linker processes `object1` first and then `object2`:

- `function1` remains at offset 0.
- `function2` is placed after, e.g., at offset 16.
- The symbol table is updated accordingly.

If the order is reversed (`object2` first):

- `function2` is at offset 0.
- `function1` is placed at the next offset.

When compiling, for example:

```
gcc file1.c file2.c
```

The compiler automatically includes the standard C library (`-lc`). Additional libraries can be specified using options like `-l lib1 -l lib2`.

1. **file1.o** and **file2.o** contain defined and undefined symbols.
2. After merging, if undefined symbols remain (e.g., `printf`), the linker searches libraries (collections of `.o` files like `printf.o`, `abs.o`, etc.).
3. If a missing definition is found (e.g., `printf.o`), it is included. This process continues until no undefined symbols remain.

Static vs Dynamic linking

Static Linking

The library code is included within the executable.

The final program is self-contained: all necessary code is embedded.

Advantages:

- The executable can run without external libraries.
- Independence from the runtime environment.

Disadvantages:

- Larger executable file size.
- Higher RAM usage if multiple processes use the same library.

Dynamic Linking (Default)

The executable contains only:

- The program code (compiled source),
- Data,
- Metadata declaring the required libraries (e.g., libc, lib1, lib2).

When the programme is executed, the system:

- The system checks if the libraries are available.
- Maps the libraries into the process's address space.
- If a library is unavailable $\rightarrow$ runtime error.

Dynamic linking can be implemented using two primary strategies:

Early Binding (Eager Resolution)

- Library functions (e.g., `printf`) are resolved immediately at program startup.
- If any required function is missing, the program fails to start.

Lazy Binding (Deferred Resolution)

- Library functions are resolved only when they are first called.
- If a function is never called, the program runs without issues.
- If a missing function is called, a runtime error occurs.

Computing platforms

When a program is executed, the operating system creates a process identified by a **PID** (Process ID).

- Each process has its own **address space**, meaning the addresses seen by a process are **virtual**.
- Two different processes can have the same variable at the same virtual address but with different contents, as the physical memory is separate.
- Only the **kernel** can directly work with physical addresses (supervisor/kernel-mode); programs in **user space** (user-mode) only see virtual addresses.

The operating system provides its services through **system calls**, typically encapsulated in **APIs** (Application Programming Interfaces). - **Linux/POSIX**: System calls like `open`. - **Windows**: System calls like `CreateFile` (misleadingly named, as it can also open an existing file).

For C/C++ Programmers

- System calls are not invoked directly but through **wrapper functions** in libraries:
  - `libc` on Linux.
  - `ntdll.dll` (and others) on Windows.

Many programming languages provide additional abstractions (e.g., `fopen` in standard C), which internally invoke the underlying APIs.

Application Binary Interface (ABI)

A compiled program must comply with an **ABI (Application Binary Interface)**.

The ABI defines:

- Executable and object file formats
- Memory representation of fundamental types (e.g., how many bytes an `int` occupies, byte order: big-endian vs little-endian)
- Calling conventions: how arguments are passed to functions and how return values are handled
- Ensures that code compiled with different compilers can work together.
- Different architectures have different ABIs

Multiple ABIs can coexist on a single system (e.g., 32-bit and 64-bit)

- Machine instructions to invoke a system call:
  - **x86 32-bit systems:**
    - `int` (interrupt)
    - `sysenter` (on modern CPUs)
  - **x86 64-bit systems:**
    - `syscall` (dedicated instruction)
- System calls are identified by numbers that vary across platforms:
  - **32-bit**: `open` is syscall #5 (from Linux)
  - **64-bit**: `open` is syscall #2 (from Linux)
  - Numbering differs but remains stable within the same kernel version
  - Numbers change between versions (Windows 8, 8.1, 10, even service packs)

For normal developers → irrelevant, just use wrapper functions (open, CreateFile, etc.). For advanced analysis → also important to recognise direct system calls, i.e. invoked without library wrappers.

Malware

Often uses direct calls because they can:

- Maintain number/version correspondence tables
- Extract numbers by disassembling `ntdll.dll` code (which knows correct values)

Detection Risk: If malware directly invokes syscalls, antivirus software can detect it.

Evasion Techniques: Advanced malware still uses `ntdll` but alters call numbers to mask their activities.

User space

When a program runs in user space, it operates in a restricted environment known as **user mode**. In this mode, the program cannot directly access hardware or critical system resources. Instead, it relies on the operating system's kernel to perform privileged operations. Non-privileged instructions are executed normally by the CPU.

When a program in user space invokes a system call:

- Control is transferred to the kernel, in an area configured at startup.
- The kernel executes the necessary code.
- Control is returned to the program in user mode.

From the user's perspective:

- A system call appears as a "magical" macro-instruction,
- Without visibility into how it is implemented.

Essentially, the hardware + the kernel + the APIs provide the programmer with an abstraction of a virtual machine, which is simpler to use compared to the real hardware.

Emulators

An **Instruction Set Architecture (ISA)** is typically implemented in hardware, but emulators are (hardware or) software enabling one computer system to behave like another one.

Some notable open-source emulator projects:

- **QEMU**: A generic machine emulator and virtualizer. It supports a wide range of architectures and is widely used for virtualization and system emulation.
  Reference: [Bellard, 2005]
- **MAME**: A multi-purpose emulation framework designed to preserve decades of software history. It focuses on accurately emulating arcade systems and other platforms.
- **DOSBox**: A DOS emulator that allows users to relive the "good old days" by running classic DOS-based games and applications on modern systems.

Translator Layers

An ABI (Application Binary Interface) can be implemented on top of another. Examples include:

WINE allows many Windows applications to run on Linux.

- A Windows program calls `CreateFile` → `user32.dll` → `ntdll.dll` → Windows kernel.
- In WINE, `user32.dll` is a fake library that intercepts the call and translates it.
  - Example: `CreateFile` → `open` in `libc.so` → Linux syscall → Linux kernel.
- Non-privileged instructions run natively on the CPU.
- System call wrappers are translated at runtime.

WSL1: Performs the reverse of WINE.

- A Linux executable thinks it uses `libc.so`, but a layer translates it to `user32.dll` → `ntdll.dll` → Windows kernel.
- Linux processes appear as special Windows processes (called Pico processes).

WSL2: Introduces a real Linux virtual machine.

- Ensures full compatibility by running an actual Linux kernel.
- However, the direct syscall translation approach of WSL1 was more intriguing, albeit less stable.

WOW: On Windows, when running a 32-bit program on a 64-bit system, WOW64 (Windows on Windows) comes into play.

- All processes are technically 64-bit, even on Linux.
- A 32-bit process operates in 32-bit mode, but:
  - Each syscall enters a translation "stub."
  - Parameters are adapted from 32-bit to 64-bit.
  - Results are converted back to 32-bit.

It's even possible to dynamically switch between 32-bit and 64-bit code within the same process (a technique sometimes used to bypass certain controls).