Binary Analysis and Secure Coding

Federico Conti

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).

x86 Architecture

Registers vs Memory Cells

- **Speed**: Registers are significantly faster than memory cells as they are located within the CPU and do not require external communication.
- **Size**: Registers are smaller in number but can store more data per unit compared to individual memory addresses.
- **Quantity**: CPUs typically have a limited number of registers (e.g., 10-30), while modern computers have billions of memory addresses (gigabytes of memory).
- **Addressing**:
  - Memory cells are addressed numerically (e.g., address 0, 1, 2, etc.).
  - Registers are addressed by names (e.g., A, B, C), which vary depending on the CPU architecture.

Registers are fast, limited in number, and named, while memory cells are slower, abundant, and numerically addressed.

- **MMU Role**: The Memory Management Unit acts as an intermediary between physical memory and processes. It translates virtual addresses used by processes into physical addresses.
- **Virtual Memory**: Processes operate in a virtual memory space, giving the illusion of having the entire memory available to themselves.
- **Abstraction**: The MMU, along with paging, enables this abstraction, ensuring processes remain isolated and unaware of the physical memory layout.

CPU Execution Cycle

The CPU operates in a repetitive cycle to execute instructions. This cycle consists of three main stages:

1. **Fetch**: The CPU reads the instruction pointed to by the instruction pointer. For example, if the instruction pointer holds the value 10, the CPU retrieves the instruction from memory address 10. The complexity arises from varying instruction lengths (e.g., 1 to 15 bytes in x86 architecture).
2. **Decode**: The CPU interprets the fetched instruction. For instance, the binary pattern `55` might represent the `push rbp` instruction, depending on the context.
3. **Execute**: The CPU performs the operation specified by the instruction. For example, if the instruction is an addition, it sums the specified operands.

This cycle repeats continuously throughout the CPU's operation.

In kernel mode, the CPU may handle interrupts. When an interrupt occurs, the CPU pauses its current task, executes the corresponding interrupt handler, and then resumes its normal operation. This ensures efficient handling of system events.

8086 Specifics

The term **word** refers to the natural data size a processor operates on:

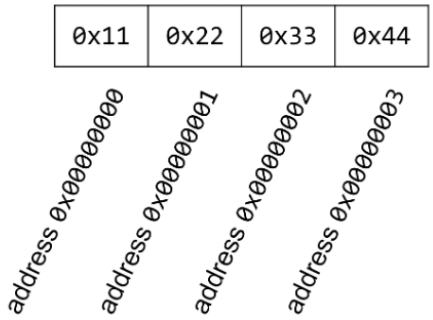
- For a 16-bit processor, a word is 16 bits.
- For a 32-bit processor, a word is 32 bits.
- For a 64-bit processor, a word is 64 bits.

Since the 8086 is a 16-bit processor, the convention of referring to 16 bits as a **word** has persisted. Today, we use the following terminology:

- **Byte**: 8 bits
- **Word**: 16 bits
- **Double Word**: 32 bits
- **Quad Word**: 64 bits

x86 instructions have a **variable length** for the instruction, ranging from 1 byte to 15 bytes. This contrasts with architectures like ARM, where instruction lengths are typically fixed (e.g., 4 bytes). Variable-length instructions enable advanced techniques, such as code obfuscation.

Intel x86 is **little-endian** by default, meaning that in memory, the bytes appear in reverse order compared to their numerical value. x86 is a little-endian system. This means that when storing a word in memory, the least significant byte is stored at the lowest address, and the most significant byte is stored at the highest address. For example, here we are storing the word `0x44332211` in memory:



In this course, we will primarily focus on the **user mode** of x86. The main operational modes are:

1. **64-bit mode**: Used by modern processes on Windows and Linux.
2. **Compatibility mode**: Used for 32-bit processes, functioning similarly to the true 32-bit mode of older processors.

   Note: There is also a third compatibility mode for legacy systems, which we will not cover.

When using an address in user mode, the first transformation is **segmentation**. Technically, each address passes first through the segment translation unit and then through paging.

In modern operating systems, however, (except for thread local storage), all selectors have a 0 offset and a '+infinity' limit, so segmentation does not alter the address. In practice: the hardware for segmentation exists, but is not used. The real virtualisation of memory occurs through **paging**.

Intel vs. AT&T Syntax

x86 assembly has two main syntaxes:

1. **Intel Syntax**: More readable and used in this course.
2. **AT&T Syntax**: Used in some tools and articles.

Key Differences in AT&T Syntax:

- Operand Order: Source on the left, destination on the right.
- Register Prefix: Registers are prefixed with `%` (e.g., `%eax`).
- Immediate Value Prefix: Immediate values are prefixed with `$` (e.g., `$1`).
- Size Suffix: Instruction size is indicated by a suffix (`l` for 32-bit, `q` for 64-bit).
- Memory Addressing : The format differs between the two syntaxes.

Intel Code	**AT&T Code** (destination on the right)
`mov eax, 1`	`movl $1, %eax`
`mov ebx, 0FFh`	`movl $0xff, %ebx`
`int 80h`	`int $0x80`
`mov ebx, eax`	`movl %eax, %ebx`
`mov eax, [ecx]`	`movl (%ecx), %eax`
`mov eax, [ebx+3]`	`movl 3(%ebx), %eax`

Intel Code	**AT&T Code** (destination on the right)
`mov eax, [ebx+20h]`	`movl 0x20(%ebx), %eax`
`add eax, [ebx+ecx*2h]`	`addl (%ebx, %ecx, 0x2), %eax`
`lea eax, [ebx+ecx]`	`leal (%ebx, %ecx), %eax`
`sub eax, [ebx+ecx*4h-20h]`	`subl -0x20(%ebx, %ecx, 0x4), %eax`

Types of Registers and Structure

Note: Since the values in these three registers are usually addresses, sometimes we will say that a register points somewhere in memory.

- **AX, BX, CX, DX**: These are general-purpose registers. Historically, the accumulator (AX) was used to store arithmetic operation results.
- **Extended Registers**: The "X" in AX, BX, etc., stands for "extended," indicating compatibility with earlier architectures.
- **AX (16-bit)**: Composed of two 8-bit registers:
  - **AL (Low)**: Represents the lower 8 bits.
  - **AH (High)**: Represents the upper 8 bits.

For example:

```
MOV AX, 0x1234     ; AX = 0001001000110100 (binario)
                   ; AH = 00010010 (0x12)
                   ; AL = 00110100 (0x34)

MOV AL, 0xFF       ; AL change only
                   ; Now AX = 0x12FF
                   ; AH remains 0x12, AL becomes 0xFF

MOV AH, 0xAB       ; AH change only
                   ; Now AX = 0xABFF
                   ; AH becomes 0xAB, AL remains 0xFF
```

Assembly instructions follow this general syntax:

```
INSTRUCTION DESTINATION, SOURCE
```

- The **destination** is the target operand.
- The **source** is the value or location being used.

For example: - `MOV AH, 2`: Assigns the value 2 to the AH register.

1. **Registers**: e.g., AX, BX, CX, DX.
2. **Immediate Values**: Constants directly encoded in the instruction.
3. **Memory Addresses**: Accessed using square brackets in Intel syntax.

Examples:

- `MOV AX, 2`: Assigns the immediate value 2 to AX.
- `MOV AX, [2]`: Assigns to AX the value stored at memory address 2.

In the latter case, the number 2 represents an address, not a value.

The **SI** (Source Index) and **DI** (Destination Index) registers are 16-bit registers that cannot be divided like AX (into AH and AL). These registers are used as pointers, and some instructions work exclusively with them.

To read or write memory efficiently:

- **SI** is used as a source pointer.
- **DI** is used as a destination pointer.

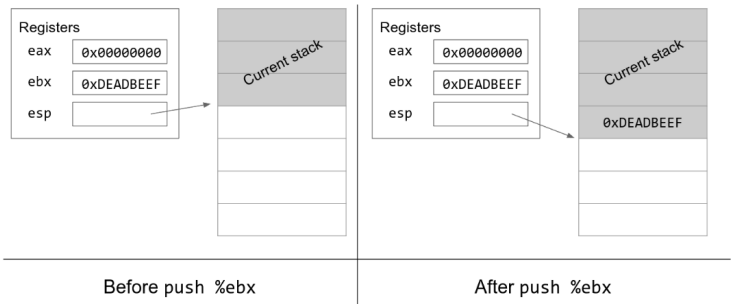
Today, however, almost any register can be used to access memory.

Another important register is

- **IP** (Instruction Pointer). This register points to the current instruction being executed.
- **SP** (Stack Pointer). The stack is a special area of memory used to "remember" information, such as where to resume execution after a function call. It stores the address of the top of the stack.
- **BP** (Base Pointer). This register is used to reference function parameters and local variables within the stack.
  - It stores the address of the top of the current stack frame.

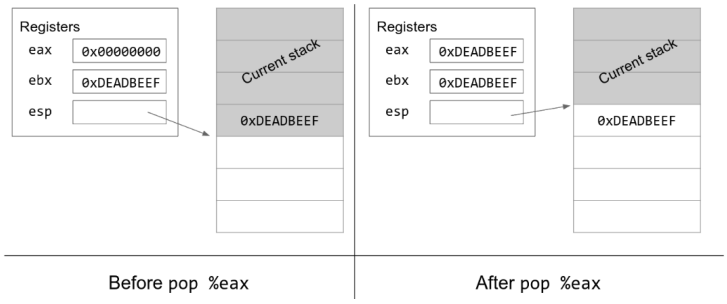
Pushing and popping

Sometimes we want to remember a value by saving it on the stack. There are two steps to storing a value on the stack. First, we have to allocate additional space on the stack by decrementing esp. Then, we store the value in the newly allocated space. The x86 push instruction does both of these steps to store a value to the stack.



We may also want to remove values from the stack. The x86 pop instruction increments esp to remove the next value on the stack. It also takes the value that was just popped and copies the value into a register AX.

Note that when we pop a value off the stack, the value is erased from memory. However, we increment esp so that the popped value is now below esp. The esp register points to the bottom of the stack, so the popped value below esp is now in undefined memory.



Memory Layout

What is needed to execute a program?

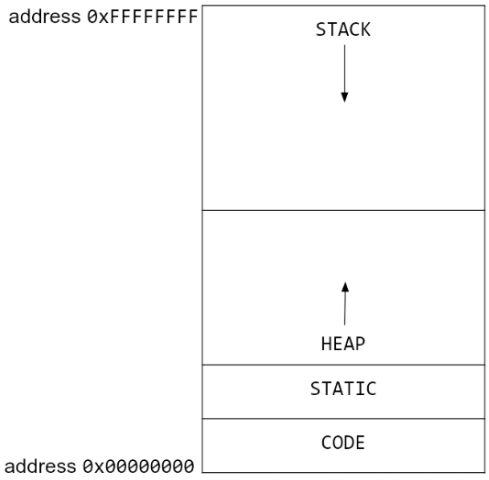
- **Code**: The instructions of the program.
- **Data**: Global and static variables.

- **Stack**: Used to keep track of function calls and local variables (LIFO).
- **Heap**: Used for dynamic memory allocation (e.g., `malloc` in C or `new` in Java/C++).

Stack and Heap Growth Directions

- The stack grows towards lower memory addresses (decrements).
- The heap grows towards higher memory addresses (increments).
- These two areas must never overlap.

With 64-bit addressing, it is theoretically possible to have the stack and heap grow in the same direction, separated by a vast address space. However, historically, the stack was designed to grow downwards, and this behavior persists today.



At runtime, the operating system gives the program an address space to store any state necessary for program execution. You can think of the address space as a large, contiguous chunk of memory. **Each byte of memory has a unique address.**

The **size of the address** space depends on your operating system and CPU architecture. In a 32-bit system, memory addresses are 32 bits long, which means the address space has 2^32 bytes of memory. In a 64-bit system, memory addresses are 64 bits long.

Calling Convention

A classic method (commonly used in 32-bit systems) involves:

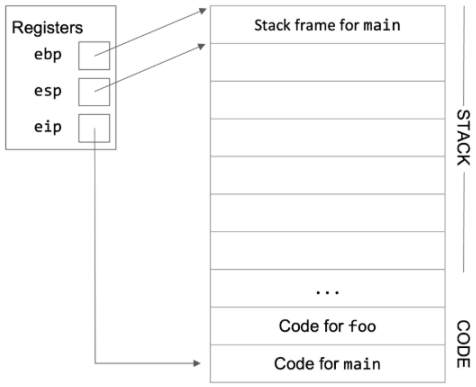
- Pushing arguments onto the stack (`push`).
- Storing the return value in the accumulator (now EAX or RAX instead of AX).

The last argument is pushed first, and the first argument is pushed last.

- This ensures that when the function starts, the first argument is always at a fixed distance from the return address on the stack.
- This is crucial for functions with a variable number of arguments (e.g., `printf("...", a, b, c, ...)`).
- If the arguments were placed in the opposite order, the function would not know where the first parameter begins without additional information.

Example (32-bit)

Consider the function caller `main` and the corresponding function callee `foo`:




```c
int main(void) {
    foo(1, 2);
}

void foo(int a, int b) {
    int bar[4];
}
```

```
main:

    # Step 1. Push arguments on the stack in reverse order
    push $2
    push $1

    # Steps 2-3. Save old eip (rip) on the stack and change eip
    call foo

    # Execution changes to foo now. After returning from foo:

    # Step 11: Remove arguments from stack
    add $8, %esp

foo:

    # Step 4. Push old ebp (sfp) on the stack
    push %ebp

    # Step 5. Move ebp down to esp
    mov %esp, %ebp

    # Step 6. Move esp down
    sub $16, %esp

    # Step 7. Execute the function (omitted here)

    # Step 8. Move esp
    mov %ebp, %esp

    # Step 9. Restore old ebp (sfp)
```

```
pop %ebp

# Step 10. Restore old eip (rip)
pop %eip
```

The number 16 is determined by the compiler depending on the function being called. In this case, the compiler decides 16 bytes are required to fit the local variable and any other data needed for the function to execute.

NOTE:

- Steps 8-9 are often combined into a single instruction: `leave`, which performs both operations.
- The instruction in step 10 is sometimes abbreviated as the `ret` instruction.
- Steps 4-6 are sometimes called the function **prologue**, since they must appear at the start of the assembly code of any C function.
- Similarly, steps 8-10 are sometimes called the function **epilogue**.
- Starting with EBP, with positive offsets we can find the arguments, and with negative offsets we can find the local variables.

see: Memory Safety

Conventions (32-bit)

1. Optional Use of EBP as Frame Pointer
   The EBP (Base Pointer) register is used to point to the start of the current stack frame, making it easier to access function parameters and local variables. However, in many modern compilers, the use of EBP as a frame pointer is optional. When not used, the compiler accesses local variables and parameters directly via the ESP (Stack Pointer), freeing up the EBP register for other purposes.
2. Caller-Saved Registers
   Certain registers must be preserved by the caller during a function call. These include:
   - EBP: Used as the frame pointer.
   - EBX: Typically used for persistent data.
   - EDI and ESI: Commonly used for data copying or manipulation.
   - ESP: Must always point to the top of the stack.
     If a function modifies any of these registers, it must save their values at the start and restore them before returning control to the caller.
3. Function Return Values
   - Integral values (e.g., `int`, `char`) or pointers (e.g., `int*`) are returned in the EAX register.
   - For 64-bit return values (e.g., `long long`), the EDX register is also used to store the most significant bits.

Convention (64-bit)

1. The first six arguments are passed through registers in the following order:

- `RDI`
- `RSI`
- `RDX`
- `RCX`
- `R8`

- `R9`

If there are more than six arguments, the stack is used for the remaining ones.

2. The return value of a function is stored in:

- `RAX` (for most types, such as `int`, pointers, etc.)
- If the return value is larger (e.g., a `struct`), `RDX` may also be used as a secondary register to return data.

3. Certain registers must be preserved either by the caller or the callee to ensure that important values are not accidentally overwritten.

- `RBP` (base pointer)
- `RBX`
- `R12`, `R13`, `R14`, `R15`
- `RSP` (stack pointer)
- **Notes:**
  - In 32-bit systems, `EDI` and `ESI` also had to be preserved.
  - In 64-bit systems, `RDI` and `RSI` can be freely modified.

Program Entry

When a compiled C program starts, the entry point is not `main`. Instead, it begins with library code responsible for initialization, which then calls `main`.

```
int main(int argc, char** argv, char** envp)
```

1. Number of Command-Line Arguments (`argc`)
2. Pointers to Command-Line Arguments (`argv`)
3. A Null Pointer
4. Pointer to the Environment (`envp`)
5. Another Null Pointer
6. ELF (Executable and Linkable Format) Data
7. Environment Strings, Program Name, etc.

   Note: This stack layout does not adhere to the calling convention typically used for functions.

System Calls

System calls are invoked by a process to request the operating system to perform specific tasks.

When programming in C or C++, system calls appear as regular library function calls. However, these library functions are wrappers that contain special instructions (which vary depending on the processor) to invoke the actual system call.

There is no portable way in C to directly make a system call.

During a system call, a context switch occurs from user mode to kernel mode. This switch is significantly slower than a regular function call.

A wrapper function performs the following steps:

1. Uses assembly code to:
   - Place the arguments into CPU registers.
   - Place the syscall number into a register. Syscall Reference
   - Trigger a “trap” into the kernel.

2. The kernel executes the system call handler, which:
    - Validates the syscall number and arguments.
    - Calls the actual routine corresponding to the system call.
    - Places the result in a register.
    - Switches back to user mode using a special instruction.
3. The wrapper function:
    - Checks the result.
    - On error, sets `errno` and returns an error code (typically `-1`).
    - Otherwise, returns the result.

32-bit

In Linux 32-bit assembly programming, system calls are handled as follows:

1. System Call Number: Place the system call number in the `EAX` register.
2. Parameters: The six parameters are passed in the following registers, in order:
    - `EBX`
    - `ECX`
    - `EDX`
    - `ESI`
    - `EDI`
    - `EBP`
3. Additional Arguments: If more than six arguments are needed, use structures or pointers.
4. Return Value: The return value of the system call is stored in the `EAX` register.
5. Register Preservation: All registers are preserved during the system call.

Example

The following example demonstrates how to use system calls to write "Hello World" to the standard output and then exit:

```
section .text
global _start

_start:
     mov eax, 4          ; System call number for write
     mov ebx, 1          ; File descriptor (stdout)
     mov ecx, msg        ; Address of the message string
     mov edx, msglen     ; Length of the message
     int 0x80            ; Make the system call

     mov eax, 1          ; System call number for exit
     mov ebx, 0          ; Exit status code
     int 0x80            ; Make the system call

section .data
msg: db "Hello World", 10  ; Message string with newline
msglen equ $-msg           ; Calculate the length of the message
```

> Note: If the `exit` system call is not invoked, the processor will continue executing the subsequent code, which may lead to a segmentation fault.

64-bit

In Linux 64-bit assembly programming, system calls are handled as follows:

1. System Call Number: Place the system call number in the `RAX` register.
2. Parameters: The six parameters are passed in the following registers, in order:
    - `RDI`
    - `RSI`
    - `RDX`
    - `R10`
    - `R8`
    - `R9`
3. Additional Arguments: If more than six arguments are needed, use structures or pointers.
4. Return Value: The return value of the system call is stored in the `RAX` register.
5. Register Preservation: All registers are preserved during the system call, except for `RAX` and `R11`.
    - `RCX` and `R11` are implicitly used by the syscall instruction for saving `RIP` and `RFLAGS`.

Example

```
section .text
global main
main:
    mov rax, 1          ; use the write syscall
    mov rdi, 1          ; write to stdout
    mov rsi, msg        ; use string "Hello World"
    mov rdx, msglen     ; write 12 characters
    syscall             ; make syscall
    mov rax, 60         ; use the exit syscall
    mov rdi, 0          ; error code 0
    syscall             ; make syscall

section .data
msg: db "Hello World", 10
msglen equ $-msg
```

Debugging

Each process operates under the illusion that it has exclusive access to the CPU and memory. Virtual address spaces create this perception, making it seem as though all memory belongs to a single process. While the memory is shared among threads within a process, each thread has its own virtual set of CPU registers.

When a process is paused, its registers can be inspected. For single-threaded processes, there is only one set of registers. For multi-threaded processes, each thread has its own set of CPU registers, while the memory remains shared. Debuggers allow you to modify registers or memory, enabling you to test different execution paths, such as altering conditions in an `if` statement.

Despite its name, a debugger does not remove bugs. Instead, it slows down execution, allowing you to observe and analyze the program's behavior step by step. This helps in identifying and understanding errors.

Types of Debuggers

1. **Source-Level vs Assembly-Level Debuggers**
   - Source-Level Debuggers: Operate on the source code, allowing step-by-step execution, variable inspection, and more. Example: Visual Studio.
   - Assembly-Level Debuggers: Work directly with machine instructions, useful for binary analysis or reverse engineering. Example: GDB, which supports both levels if debug information is available.
2. **Local vs Remote Debuggers**
   - Local Debuggers: Run on the same machine as the target program.
   - Remote Debuggers: Analyze code on a separate device, often used for embedded systems or firmware debugging.
3. **User-Mode vs Kernel-Mode Debuggers**
   - User-Mode Debuggers: Analyze user-space processes.
   - Kernel-Mode Debuggers: Analyze kernel code, typically requiring remote debugging to avoid halting the entire system.

Breakpoints

Debuggers typically allow you to set breakpoints; two kinds:

Software breakpoints are the most common type of breakpoints. They allow the execution to stop when a specific instruction is reached.

- Unlimited number of breakpoints can be set.
- They only work during execution (not on memory access).

Hardware breakpoints are more powerful but limited in number (e.g., a maximum of four on x86 architectures). They can be used not only for execution but also to:

- Halt when a specific memory cell is read or written.
- Monitor access to specific variables or buffers.

Implementing this functionality with software breakpoints would be too costly in terms of performance, so it is directly managed by the processor hardware.

When a software breakpoint is set, the debugger modifies a byte of the program at the desired location. This involves replacing the byte with the machine code `CC`, which represents the instruction:

```
INT 3
```

The `INT 3` instruction generates software interrupt number 3, specifically used to signal the operating system and debugger that a breakpoint has been reached.

Internal Process

1. The debugger saves the original byte at the breakpoint address.
2. It replaces the byte with `CC`.
3. When execution reaches this position, the processor generates an interrupt (interrupt 3).
4. The debugger gains control, restores the original byte, and adjusts the Instruction Pointer (RIP/EIP) back by 1 byte so that the next instruction executed is the original one.

If the breakpoint needs to remain active, the debugger: - Executes a single instruction (single step). - Re-inserts `CC` at the same location.

This ensures normal execution while keeping the breakpoint valid.

Difference from Other Interrupts

In general, the instruction to generate an interrupt is `CD` followed by the interrupt number, e.g.:

`CD 05 → INT 5`

However, `INT 3` is a single-byte instruction (`CC`), making it more efficient for debuggers to insert and detect.

Memory Sharing

Software breakpoints have a significant side effect: to insert them, the debugger modifies the process memory.

Typically, the code pages of a program (e.g., `notepad.exe`) are shared among all processes running the same binary to save memory.

However, when a debugger modifies the code of a process (by inserting a `CC` instruction), that page can no longer be shared. The operating system creates a private copy of the page for that process using the copy-on-write mechanism.

Example:

- Three instances of `notepad.exe` share the same code pages.
- If one instance is debugged and a breakpoint is inserted, only that process sees the modification.
- The other instances continue executing the original code without interruption.

Anti-Debug Techniques

Since the debugger modifies the programme code, a software can detect this by checking its memory. For example, a program might:

- Calculate a checksum or hash of its code.
- Check for unexpected bytes (e.g., `0xCC`) in specific locations.
- Restore the original code autonomously.

These anti-debug techniques are often used in protected software or malware to hinder dynamic analysis. They allow the program to detect the presence of a debugger or even remove the breakpoints.

While uncommon in regular software, such mechanisms are prevalent in copy-protected programs and malware samples.