L07 86 Arch

Charudatta Korde

Contents

Types of Malwares

- **Virus**: Attach themselves to clean files and infect other clean files.
- **Worms**: Spread over networks by exploiting vulnerabilities.
- **Trojans**: Disguise themselves as legitimate software.
- **Spyware**: Secretly monitors user activity and sends information to the attacker.
- **Ransomware**: Locks users out of their data and demands ransom.
- **Adware**: Automatically delivers advertisements.
- **Rootkits**: Provide privileged access to attackers.
- **Keyloggers**: Record keystrokes to capture sensitive information.

Types of Malware Analysis

- **Static Analysis**: Examining the malware without running it.
- **Dynamic Analysis**: Analyzing malware behavior by executing it.
- **Code Analysis**: Reviewing the code of the malware.
- **Memory Analysis**: Investigating malware's impact on system memory.

Forensic Importance

- **Evidence Collection**: Gathering evidence from infected systems.
- **Legal Proceedings**: Providing data for legal cases.
- **Incident Response**: Aiding in the quick response to incidents.
- **Mitigation Strategies**: Developing techniques to mitigate malware threats.
- **Attribution**: Identifying the source and motive behind the malware attack.

Malware Behaviour

- **Persistence Mechanisms**: How malware stays on the infected system.
- **Propagation Techniques**: Methods used by malware to spread.
- **Exfiltration Methods**: Techniques used to send data to the attacker.
- **Evasion Tactics**: Strategies to avoid detection by security software.
-

Payload Activities: Actions performed by malware on the infected system.

marp: true theme: default paginate: true class: invert author: charudatta created: 22-10-20024 title: "Malware L02 Hash" header: 'Malware Analysis Class' footer: 'Unit I: Malware Analysis - By Charudatta Korde' tags: nfsu, notes, ppt, malware —

Hashing, Finding Strings, and Decoding Obfuscated Strings Using FLOSS

Hashing

- **Definition**: Transforming data into a fixed-size string of characters.
- **Common Hash Functions**:
  - MD5
  - SHA-1
  - SHA-256

- **Uses**:
  - Data Integrity
  - Password Storage
  - Digital Signatures

- **Properties**:
  - Deterministic
  - Fast Computation
  - Pre-image Resistance
  - Small Changes in Input Produce Large Changes in Output

Finding Strings

- **Definition**: Extracting human-readable sequences from binary data.
- **Tools**:
  - `strings`: Command-line utility to find and print text strings in binary files.
  - **Use Case**: Identify embedded text, URLs, or function names in executables.
- **Example Command**:

```
strings malware_sample.exe
```

Decoding Obfuscated Strings Using FLOSS

- **FLOSS (FireEye Labs Obfuscated String Solver)**: A tool to automatically identify and decode obfuscated strings in malware.
- **Steps**:
  - **Installation**:
    `pip install floss`
  - **Running FLOSS**:
    `floss malware_sample.exe`

  - **Output**: Decoded strings displayed for analysis.
- **Benefits**:
  - Reveals hidden functionality.
  - Aids in understanding malicious behavior.

Questions & Discussion

- **Q1**: Why is hashing important for digital signatures?
- **Q2**: How can finding strings help in malware analysis?
-

Q3: What are the benefits of using FLOSS for decoding obfuscated strings?

marp: true theme: default class: lead author: charudatta created: 22-10-20024 title: "Malware L03 PE files" header: 'Malware Analysis Class' footer: 'Unit I: Malware Analysis - By Charudatta Korde' tags: nfsu, notes, ppt, malware —

PE Files: Headers and Sections, PE View, Linked Libraries and Functions

---

What is a PE File?

- **PE**: Portable Executable
- File format for executables, object code, DLLs, etc.
- Used in 32-bit and 64-bit versions of Windows operating systems
- Based on the Microsoft Common Object File Format (COFF)

PE File Headers

- **DOS Header**: Compatibility and executable info
- **PE Header**: General info about the file
- **Optional Header**: Detailed info for the loader
- **Section Table**: Info about the sections in the file

PE File Sections

- **.text**: Contains the executable code
- **.data**: Contains initialized data
- **.rdata**: Contains read-only data
- **.bss**: Contains uninitialized data
- **.edata**: Export table
- **.idata**: Import table

PE View

- Tool to analyze PE files
- Visual representation of headers and sections
- Provides detailed information about the file structure

Linked Libraries and Functions

- **Import Address Table (IAT)**: List of imported functions
- **Export Address Table (EAT)**: List of exported functions
- **Dynamic Link Libraries (DLLs)**: Libraries linked to the executable

---

Conclusion

- PE file format is essential for Windows executables
- Understanding headers and sections helps in malware analysis and reverse engineering
- Tools like PE View aid in analyzing and understanding PE files

---

Questions?

Feel free to ask any questions or provide feedback on the presentation!

marp: true theme: default class: author: charudatta created: 22-10-20024 title: “Malware L04 Tools - 1” header: ‘Malware Analysis Class’ footer: ‘Unit I: Malware Analysis - By Charudatta Korde’ tags: nfsu, notes, ppt, malware —

Tools for Windows Executables: Dependency Walker, CFF Explorer, Resource Hacker

---

Dependency Walker

- **Purpose**: Analyze dependencies of Windows applications
- **Features**:
  - Lists all dependent modules (DLLs)
  - Detects missing dependencies
  - Displays function calls and imported/exported functions
- **Usage**: Useful for troubleshooting and understanding application dependencies

---

CFF Explorer

- **Purpose**: Analyze and edit Portable Executable (PE) files
- **Features**:
  - PE Header Viewer
  - Hex Editor
  - Resource Editor
  - Dependency Scanner
- **Usage**: Ideal for malware analysis, reverse engineering, and PE file modification

---

Resource Hacker

- **Purpose**: View, modify, rename, add, delete resources in executables
- **Features**:
  - Resource viewing and editing (bitmaps, icons, menus, dialogs, etc.)
  - Script compiler and decompiler
  - Extract and replace resources
- **Usage**: Customizing application interfaces, modifying application resources

---

Comparison

Feature	Dependency Walker	CFF Explorer	Resource Hacker
Dependency Analysis	Yes	Yes	No
PE Header Viewing	No	Yes	Yes
Resource Editing	No	Yes	Yes
Hex Editing	No	Yes	No

Feature	Dependency Walker	CFF Explorer	Resource Hacker
Script Compilation	No	No	Yes

Conclusion

- These tools are essential for analyzing and modifying Windows executables
- Dependency Walker is perfect for dependency analysis
- CFF Explorer is a comprehensive tool for PE file analysis and modification
- Resource Hacker excels in resource viewing and editing

Questions?

Feel free to ask any questions or provide feedback on the presentation!

title: “06_PE_files.md” time: “2025-01-22 :: 18:01:18” tags: nfsu, malware, notes — ### 1. DOS Header

The DOS header is the first part of the PE file. It ensures backward compatibility with older DOS applications. If a Windows executable is run in DOS, it will display a message indicating that the program cannot be run in DOS mode.

Key Fields:

- e_magic: Signature word ‘MZ’ (0x4D 0x5A).
- e_lfanew: Offset to the PE header.

2. PE Header (Signature)

This header follows the DOS header and starts with the “PE00” signature, indicating the start of the PE file format.

Key Fields:

- Signature: “PE00” (0x50 0x45 0x00 0x00).

3. COFF File Header (PE Header)

The COFF (Common Object File Format) header provides important metadata about the PE file.

Key Fields:

- Machine: The target CPU architecture (e.g., 0x14C for x86).
- NumberOfSections: Number of sections in the PE file.
- TimeDateStamp: Timestamp indicating when the file was created.
- PointerToSymbolTable: Offset to the symbol table (usually not used in executables).
- NumberOfSymbols: Number of entries in the symbol table.
- SizeOfOptionalHeader: Size of the optional header.
- Characteristics: Flags indicating attributes of the file (e.g., executable, DLL).

4. Optional Header

Despite its name, this header is essential for executables and DLLs. It contains information required by the loader to manage the executable in memory.

Key Fields:

- Magic: Indicates PE32 (0x10B) or PE32+ (0x20B) format.
- AddressOfEntryPoint: The starting address of the program when executed.
- ImageBase: Preferred base address for loading the executable.
- SectionAlignment: Alignment of sections in memory.
- FileAlignment: Alignment of sections in the file.
- SizeOfImage: Total size of the image in memory.
- SizeOfHeaders: Size of all headers combined.
- Subsystem: Subsystem required to run the executable (e.g., Windows GUI, Console).
- DllCharacteristics: Attributes for DLLs (e.g., ASLR, DEP).
- SizeOfStackReserve: Reserved stack size.
- SizeOfHeapReserve: Reserved heap size.
- DataDirectories: Array of data directories (export table, import table, resource table, etc.).

5. Data Directories

A set of directories pointing to tables used during execution.

Key Tables:

- Export Table: Functions and symbols exported by the executable or DLL.

- Import Table: Functions and symbols imported from other DLLs.
- Resource Table: Various resources such as icons, images, menus.
- Exception Table: Information about exception handling.
- Certificate Table: Digital signatures for the file.
- Base Relocation Table: Addresses to be fixed if the executable is not loaded at its preferred base address.
- Debug Directory: Information for debugging tools.
- Architecture-specific Data: Reserved for specific CPU architecture.
- Global Pointer: Used for global pointer relative addressing.
- TLS Table: Thread-local storage data.
- Load Config Table: Configuration information for the loader.
- Bound Import: Information for bound imports.
- Import Address Table: Runtime address of imported functions.
- Delay Import Descriptor: Information for delay-loaded DLLs.
- COM Descriptor: Information for COM+ runtime support.

6. Section Headers

Each section header defines a specific section of the executable, detailing its characteristics and location in the file and memory.

Common Sections:

- .text: Contains the executable code.
- .data: Contains initialized data.
- .rdata: Contains read-only data.
- .bss: Contains uninitialized data.
- .rsrc: Contains resources like icons, bitmaps, and strings.
- .reloc: Contains base relocations.

Key Fields in Section Headers:

- Name: Name of the section (e.g., .text, .data).
- VirtualSize: Size of the section when loaded into memory.
- VirtualAddress: Address of the section when loaded into memory.
- SizeOfRawData: Size of the section in the file.
- PointerToRawData: Offset to the section in the file.

- Characteristics: Attributes of the section (e.g., executable, writable).

Example: "Hello World" PE File

Here's a simplified example of some key fields and their values:

DOS Header

```
4D 5A ; MZ (Magic number)

90 00 ; No of bytes in last block of file

03 00 ; No of blocks in file

40 00 ; Header size in paragraphs

...
```

PE Header

```
50 45 00 00 ; PE00 (Signature)
```

COFF File Header

```
4C 01 ; Machine (Intel 386)

03 00 ; Number of sections

60 89 4C 61 ; Timestamp

...

```shell

#### Optional Header:
```

0B 01 ; Magic (PE32)

10 00 ; Linker version

00 02 00 00 ; Size of code

00 01 00 00 ; Size of initialized data

...
```

.text Section:

Contains the actual executable code. A simple "Hello World" program might have:

```shell

48 65 6C 6C 6F 20 57 6F 72 6C 64 21 00 ; "Hello World!"
```

Instruction Set Architecture (ISA)

An Instruction Set Architecture (ISA)** is a part of the computer architecture related to programming, including the instruction set, word size, memory address modes, and processor registers43dcd9a7-70db-4a1f-b0ae-981daa162054. It serves as the interface between software and hardware, defining how software controls the CPU43dcd9a7-70db-4a1f-b0ae-981daa162054.**

RISC (Reduced Instruction Set Computing)

RISC is a type of ISA that uses a small, highly optimized set of instructions43dcd9a7-70db-4a1f-b0ae-981daa162054. The idea behind RISC is to simplify the processor design and improve performance by executing simple instructions very quickly43dcd9a7-70db-4a1f-b0ae-981daa162054. RISC architectures typically have fewer instructions but more registers, which can lead to more efficient use of the CPU43dcd9a7-70db-4a1f-b0ae-981daa162054.**

CISC (Complex Instruction Set Computing)

CISC is a type of ISA that includes a large number of complex instructions43dcd9a7-70db-4a1f-b0ae-981daa162054. These instructions can perform multiple low-level operations (such as memory access, arithmetic operations, and data movement) within a single instruction43dcd9a7-70db-4a1f-b0ae-981daa162054. CISC architectures aim to reduce the number of instructions per program, potentially simplifying the compiler's job43dcd9a7-70db-4a1f-b0ae-981daa162054.**

x86-64

x86-64 (also known as x64 or AMD64) is an extension of the x86 architecture, which adds 64-bit processing capabilities to the existing 32-bit x86 architecture43dcd9a7-70db-4a1f-b0ae-981daa162054. It combines elements of both RISC and CISC, offering a rich set of instructions and addressing modes43dcd9a7-70db-4a1f-b0ae-981daa162054. x86-64 is widely used in personal computers and servers43dcd9a7-70db-4a1f-b0ae-981daa162054.**

Would you like more details on any of these concepts?X86 is a computer

architecture for central processing units (CPUs) that uses a complex instruction set computer (CISC) design. It's the most common architecture for personal computers and servers. [1]

How it works [1]

- X86 CPUs can perform multiple tasks in a single operation. [1]
- X86 CPUs can execute multiple instructions in a single cycle. [1, 2]
- X86 CPUs use a modest number of special-purpose registers instead of large quantities of general-purpose registers. [3]

History [4]

- Intel developed the x86 architecture in 1978. [4]
- The name "x86" comes from the 8086, an early processor released by Intel. [1, 2]
- The x86 architecture was originally a 16-bit instruction set, but later grew to 32-bit instruction sets. [4]

Compatibility [2]

- X86 architecture is highly compatible and capable of running a vast array of software applications. [1, 2]
- X86 architecture is backward compatible with the 8-bit Intel 8080 processor. [3]

Limitations [4]

- The 32-bit version of x86 can handle a maximum of 4096 MB of RAM.

Generative AI is experimental.

[1] https://www.lenovo.com/in/en/glossary/x86/

[2] https://www.lenovo.com/gb/en/glossary/x86/

[3] https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/x86-architecture

[4] https://phoenixnap.com/kb/x64-vs-x86

Not all images can be exported from Search.

x64 vs. x86: Key Differences

July 20, 2022

KB » **Bare Metal Servers** » x64 vs. x86: Key Differences

Introduction

The x86 and x64 architectures refer to the two most widely-used types of instruction set architectures (**ISA**) created by Intel and AMD. An ISA specifies the behavior of machine code and defines how the software controls the **CPU**.

ISA is the hardware and software interface, defining what the CPU can do and how.

In this article, you will learn the difference between the x64 and x86 architectures.

What is x86 Architecture?

x86 is a type of ISA for computer processors originally developed by Intel in 1978. The x86 architecture is based on Intel's 80**86** (hence the name) **microprocessor** and its 8088 variant. At first, it was a 16-bit instruction set for 16-bit processors, and later it grew to 32-bit instruction sets.

The number of bits signifies how much information the CPU can process per cycle. For example, a 32-bit CPU transfers up to 32 bits of data per clock cycle.

Due to its capability of running on almost any computer, from laptops, home PCs, and servers, x86 architecture has become popular among numerous microprocessor manufacturers.

The x86 architecture's most significant limitation is that it can handle a maximum of 4096 MB of RAM. Since the total number of supported combinations is 2^32 (4,294,967,295), the 32-bit processor has 4.29 billion memory locations. Each location stores **one byte of data**, equating to approximately 4GB of accessible memory.

Today, the term x86 denotes any 32-bit processor capable of running the x86 instruction set.

Note: After choosing the right infrastructure, learn the **difference between a single and dual-core processor server** and meet your organization's demands.

What is x64 Architecture?

x64 (short for x86-64) is an instruction set architecture based on x86, extended to enable 64-bit code. It was first released in 2000, introducing two modes of operation - the 64-bit mode and the compatibility mode, which allows users to run 16-bit and 32-bit applications as well.

Since the entire x86 instruction set remains implemented in the x64 one, the older executables run with practically no performance penalties.

The x64 architecture supports much greater amounts of virtual and **physical memory** than the x86 architecture, allowing applications to store large data

The key differences between an x86 and x64 instruction set architecture.

Figure 1: The key differences between an x86 and x64 instruction set architecture.

amounts in memory. Additionally, x64 expands the number of general-purpose registers to 16, providing further enhancements and functionality.

The x64 architecture can utilize a total of 2^64 bytes, equaling to 16 billion gigabytes (16 **exabytes**) of memory. The much greater resource utilization makes it suitable for powering **supercomputers** and machines that need access to vast resources.

The x64 architecture allows the CPU to process 64-bits of data per clock cycle, much more than the x86 one.

x86 vs. x64

While both architecture types are based on the 32-bit set, some key differences make them suitable for different uses. The main difference between them is the amount of data they can handle with each clock cycle and the processor's register width.

The processor stores frequently used data in a register for quick access. A 32-bit processor on x86 architecture has 32-bit registers, while 64-bit processors have 64-bit registers. Thus, x64 allows the CPU to store more data and access it faster. The register width also determines the amount of memory a computer can utilize.

The following table shows an overview of the key differences between the x86 and x64 architecture sets:

ISA	x86	x64
Initial release	Introduced in 1978.	Introduced in 2000.
Creator	Intel	AMD
Origin	Based on the Intel 8086 processor.	Created as an extension of the x86 architecture.
Bit amount	32-bit architecture.	64-bit architecture.
Addressable space	4 GB.	16 EB.
RAM limit	4 GB (actual usable RAM 3.2 GB).	16 billion GB.
Speed	Slower and less powerful compared to x64.	Allows high-speed processing of large sets of integers; inherently faster than x86.

ISA	x86	x64
Data transmission	Supports parallel transmits of only 32-bits via a 32-bit bus in a single go.	Supports parallel transmits of larger chunks of data via the 64-bit data bus.
Storage	Utilizes more registers to split and store data.	Stores large data amounts with fewer registers.
Application support	No support for 64-bit apps and programs.	Supports both 64-bit and 32-bit apps and programs.
OS support	Windows XP, Vista, 7, 8, Linux.	Windows XP Professional, Windows Vista, Windows 7, Windows 8, Windows 10, Linux, Mac OS.

Note: See **how a CPU compares to a GPU**.

Features

Each architecture set has features that define it and give it an edge in specific use cases. The following lists showcase the features of x64 and x86:

x86

- It uses complex instruction set computing architecture (CISC).
- Complex instructions require multiple cycles to execute.
- x86 has more registers available but less memory.
- Designed with fewer pipelines, but it can handle complex addresses.
- System performance is optimized using the hardware approach - x86 relies on physical components to compensate for low memory.
- Uses software-based DEP (Data Execution Prevention).

x64

- Has 64-bit integer capability with backward compatibility for 32-bit applications.
- The (theoretical) virtual address space amounts to 2^64 bytes (16 exabytes). However, only a small portion of the theoretical 16-exabyte range is currently used in real life - about 128 TB.
- x64 processes large files by mapping the entire file into the process's address space.
- Faster than x86 due to its faster parallel processing, 64-bit memory and data bus, and larger registers.

- Supports simultaneous operation of large files on multiple address spaces. Additionally, x64 emulates two x86 tasks simultaneously and provides a faster experience than x86.
- Loads instructions more effectively and efficiently.
- Uses hardware-backed DEP (Data Execution Prevention).

Applications

Due to their different features and differences in resource access, speed, and processing power, each architecture set is used for different purposes:

x86 Contd I

- Many of the world's PCs are still based on x86 operating systems and CPUs.
- Used for gaming consoles.
- **Cloud computing** segments still use the x86 architecture.
- Older applications and programs usually run on 32-bit architecture.
- It is better for **emulation**.
- 32-bit is still preferred in audio production due to its compatibility with older audio equipment.

x64 Contd I

- An increasing number of PCs use 64-bit CPUs and operating systems based on the x64 architecture.
- All modern mobile processors use the x64 architecture.
- It is used to power supercomputers.
- Used in video-game consoles.
- **Virtualization** technologies are based on the x64 architecture.
- It is better suitable for newer game engines as it is faster and provides better performance.

Limitations

While both ISAs have limitations, x64 is a newer, more perfected type of architecture. Below is a list of limitations of both types of architectures:

x86 Contd II

- It has a limited pool of addressable memory.
- The processing speeds are lower compared to x64.
- Vendors no longer develop applications for 32-bit operating systems.
- Modern CPUs require a 64-bit OS.
- All devices on the system (video cards, **BIOS**, etc.) share the available RAM, leaving even less memory for the OS and applications.

x64 Contd II

- It doesn't natively run on older, legacy devices.
- Its high performance and speed usually consume more power.
- 64-bit drivers are unlikely to become available for older systems and hardware.
- Some 32-bit software isn't fully compatible with 64-bit architecture.

How to Check if Your Computer is x64 or x86?

If you own a PC purchased in the last 10-15 years, it likely runs on x64 architecture. Follow the steps below to check if your PC is 32-bit or 64-bit:

Step 1: Open Settings

On Windows 10, press the **Windows key** and click the **Settings**icon.

Step 2: Open System Options

In the settings menu, select the **System** option.

Step 3: Find Device Specifications

Select the **About** option in the left pane, and under the *Device specifications* section, find the system type. In the example above, the system is a 64-bit operating system with an x64-based processor.

Is x86 or x64 Better?

Although both x86 and x64 have advantages, the future doesn't tolerate limits, which means that x86 will eventually be scarcely used or completely dropped. Additionally, x64 is much faster, can allocate more RAM memory, and has parallel processing capabilities via the 64-bit data bus, making it a better choice between the two architecture types.

When choosing which OS type to install, it is always better to install a 64-bit OS because it can run both 32-bit and 64-bit software. On the other hand, an x86-based OS runs only 32-bit software.

Overall, x64 is much more capable than x86, utilizing all installed RAM, providing more hard drive space, faster bus speeds, and overall better performance.

Conclusion

This article shown the differences between an x86 and x64 instruction set architecture, along with their features, use cases, and limitations. Take into account all the features of each ISA and choose the one best suited for your configuration. — title: "L07 tools" created: 22-10-20024 header: 'Malware Analysis Class' footer: 'Unit I: Malware Analysis - By Charudatta Korde' tags: nfsu, notes, malware author: Charudatta Korde —

Dependency Walker

Dependency Walker is a free utility that scans any 32-bit or 64-bit Windows module (exe, dll, ocx, sys, etc.) and builds a hierarchical tree diagram of all dependent modules43dcd9a7-70db-4a1f-b0ae-981daa162054. It lists all the functions exported by each module and which of those functions are actually being called by other modules43dcd9a7-70db-4a1f-b0ae-981daa162054. It's useful for troubleshooting system errors related to loading and executing modules43dcd9a7-70db-4a1f-b0ae-981daa162054.

CFF Explorer

CFF Explorer is a freeware suite of tools for editing and analyzing portable executables, including .NET files43dcd9a7-70db-4a1f-b0ae-981daa162054. It offers a PE editor with full support for PE32/64, utilities, a hex editor, and support for .NET structures43dcd9a7-70db-4a1f-b0ae-981daa162054. It also includes a resource editor, disassembler, dependency walker, and more43dcd9a7-70db-4a1f-b0ae-981daa162054.

Resource Hacker

Resource Hacker is a resource compiler and decompiler for Windows applications43dcd9a7-70db-4a1f-b0ae-981daa162054. It allows you to view and edit resources in executables (*.exe,* .dll, *.scr, etc.) and compiled resource libraries (*.res, *.mui)43dcd9a7-70db-4a1f-b0ae-981daa162054. It supports various resource types, command-line options, scripts, and languages43dcd9a7-70db-4a1f-b0ae-981daa162054.

Would you like me to help you create a specific slide layout or content for your presentation?

Sure! Let's break down the basic algorithms and processes that each of these tools uses:

CFF Explorer I

CFF Explorer is designed for examining and editing Portable Executable (PE) files. Here's a simplified breakdown of its algorithm:

1. File Loading: Reads the PE file into memory.
2. Parsing Headers: Analyzes the DOS header, PE header, and optional headers to understand the file structure.
3. Section Analysis: Reads the section headers to identify different sections (text, data, resources, etc.).
4. Data Interpretation: Decodes and displays information such as imports, exports, and resource tables.
5. Editing: Allows users to modify various elements like headers, sections, and resources.
6. Validation and Saving: Validates changes to ensure consistency and saves the modified file.

Dependency Walker I

Dependency Walker analyzes the dependencies of a PE file. Here's a simplified breakdown of its algorithm:

1. File Loading: Loads the PE file into memory.
2. Parsing Headers: Extracts information from the PE headers.
3. Import Table Analysis: Reads the import table to identify all imported modules and functions.
4. Recursive Analysis: For each imported module, recursively performs the same analysis to build a complete dependency tree.
5. Function Mapping: Maps each function call to its respective module and function address.
6. Reporting: Displays the hierarchical dependency tree and details about each module and function.

Resource Hacker I

Resource Hacker is used for viewing and editing resources in PE files. Here's a simplified breakdown of its algorithm:

1. File Loading: Loads the PE file into memory.

2. Resource Table Parsing: Parses the resource table to identify all resources (icons, dialogs, strings, etc.).
3. Resource Extraction: Extracts resource data and decodes it for viewing and editing.
4. Editing: Allows users to modify existing resources or add new ones.
5. Validation and Saving: Ensures that changes are valid and writes the modified resource data back to the file.

Each of these tools follows a structured approach to analyze and manipulate PE files, making them valuable for software development, reverse engineering, and troubleshooting. — title: "08_wireshark" created: 22-10-20024 header: 'Malware Analysis Class' footer: 'Unit I: Malware Analysis - By Charudatta Korde' tags: nfsu, notes, malware author: Charudatta Korde marp: true paginate: true —

Wireshark: Network Protocol Analyzer

Introduction

- What is Wireshark?
- Importance of network protocol analysis

Key Features

- Real-time data capture
- Standard three-pane packet browser
- Powerful display filters
- Rich VoIP analysis
- Decoding of hundreds of protocols

Installation

- Download Wireshark from the official website
- Install on Windows, macOS, or Linux
- Basic setup and configuration

Capturing Packets

- Starting a capture
- Setting capture filters
- Saving captured packets

Analyzing Packets

- Using display filters
- Packet details pane
- Follow TCP/UDP streams
- Analyzing specific protocols (e.g., HTTP, DNS, TCP)

Advanced Features

- Coloring rules
- Customizing Wireshark
- Using Wireshark with other tools (e.g., tcpdump, tshark)

Best Practices

- Ethical considerations
- Legal implications
- Protecting sensitive data

Conclusion

- Recap of key points
- Q&A session

References

- Wireshark Official Documentation
- Online Tutorials and Forums
- Books and Articles on Network Analysis

Thank You

- Any questions?
-

Contact information

title: “09_software.md” time: “2025-01-15 :: 11:47:19” author: charudatta created: 22-10-20024 header: ‘Malware Analysis Class’ footer: ‘Unit I: Malware Analysis - By Charudatta Korde’ tags: nfsu, notes, ppt, malware —

1. **FLOSS (FireEye Labs Obfuscated Strings Solver)**
   - **Purpose**: Extracts obfuscated strings from malware binaries.
   - **Use Case**: Helps malware analysts uncover hidden strings, which might reveal valuable information about the malware’s functionality.

2. **CFF Explorer**
   - **Purpose**: A tool for PE (Portable Executable) file analysis and editing.
   - **Use Case**: Allows analysts to inspect and modify the structure of executable files, which is useful for reverse engineering and malware analysis.

3. **Resource Hacker**
   - **Purpose**: A resource editing tool for Windows.
   - **Use Case**: Enables users to view, modify, add, and delete resources in executable files, such as icons, dialogs, and menus.

4. **Dependency Walker**
   - **Purpose**: Analyzes the dependencies of Windows modules.
   - **Use Case**: Helps identify missing dependencies and troubleshoot issues related to DLL (Dynamic Link Library) files.

5. ClamAV (Clam AntiVirus)

- **Purpose**: An open-source antivirus engine.
- **Use Case**: Detects malware, viruses, trojans, and other malicious threats.

6. YARA

- **Purpose**: A tool for pattern matching and malware research.
- **Use Case**: Helps malware researchers identify and classify malware by defining rules based on patterns.

7. Process Monitor

- **Purpose**: Monitors real-time file system, registry, and process/thread activity.
- **Use Case**: Useful for debugging and identifying the behavior of software, including malware.

8. Regshot

- **Purpose**: Takes snapshots of the Windows registry and compares them.
- **Use Case**: Detects changes made by software installations, configurations, or malware.

9. Wireshark

- **Purpose**: A network protocol analyzer.
- **Use Case**: Captures and analyzes network traffic, which is essential for network troubleshooting, security analysis, and protocol development.

10. IDA Pro (Interactive DisAssembler)

- **Purpose**: A disassembler for software reverse engineering.
- **Use Case**: Converts machine code back into assembly code, helping analysts understand the inner workings of software.

11. OllyDbg

- **Purpose**: A debugger for Windows applications.
- **Use Case**: Primarily used for debugging binary code and analyzing malware behavior.

12. WinDbg

- **Purpose**: A multipurpose debugger for Windows.
- **Use Case**: Used for debugging kernel-mode and user-mode code, analyzing crash dumps, and debugging applications.

13. Cuckoo Sandbox

- **Purpose**: An automated malware analysis system.
- **Use Case**: Executes suspicious files in a virtual environment and monitors their behavior to detect malicious activity.

14. Volatility

- **Purpose**: A memory forensics framework.
-

Use Case: Analyzes memory dumps to extract digital artifacts and investigate system state, commonly used in incident response and malware analysis.

title: "10_cpuarch" time: "2025-01-16 :: 17:57:11" header: 'Malware Analysis Class' footer: 'Unit I: Malware Analysis - By Charudatta Korde' tags: nfsu, notes, malware author: Charudatta Korde —

x86 CPU Architecture Overview

The x86 CPU architecture is a widely used instruction set architecture (ISA) for computer processors. It was initially developed by Intel and has evolved over time to include advanced features and capabilities. Here's a breakdown of the key components:

Registers

Registers are small, fast storage locations within the CPU that hold data and addresses temporarily. In the x86 architecture, there are several types of registers:

- **General Purpose Registers (GPRs):**
  - `EAX`, `EBX`, `ECX`, `EDX` (used for arithmetic, logic, and data movement)
  - `ESP` (Stack Pointer, points to the top of the stack)
  - `EBP` (Base Pointer, used for stack frame referencing)
  - `ESI` (Source Index for string operations)
  - `EDI` (Destination Index for string operations)
- **Segment Registers:**
  - `CS` (Code Segment, points to the segment containing the current program)
  - `DS` (Data Segment, points to the segment containing data)
  - `ES`, `FS`, `GS` (Additional segments for various purposes)
  - `SS` (Stack Segment, points to the segment containing the stack)
- **Instruction Pointer:**
  - `EIP` (Extended Instruction Pointer, points to the next instruction to be executed)
- **Flags Register:**

  - `EFLAGS` (Contains status flags like Zero Flag (ZF), Carry Flag (CF), Sign Flag (SF), etc.)

Memory

Memory in the x86 architecture is organized into segments and pages to manage and access data efficiently:

- **Segmentation:**
  - Memory is divided into segments (code, data, stack, etc.)
  - Each segment is addressed by a segment register and an offset
- **Paging:**
  - Memory is divided into pages, typically 4KB in size
  - Used to implement virtual memory
  - Managed by the Memory Management Unit (MMU)

Instruction Set

The x86 instruction set includes a variety of instructions for arithmetic, data movement, control flow, and more. Some common instructions are:

- `MOV` (Move data)
- `ADD`, `SUB` (Arithmetic operations)
- `CMP` (Compare)
- `JMP` (Jump to another instruction)
- `CALL`, `RET` (Function call and return)
- `PUSH`, `POP` (Stack operations)

Summary

The x86 CPU architecture is a complex and versatile design that has been the foundation of many computer systems. Its registers and memory organization play a crucial role in its performance and capabilities, making it a powerful tool for both general computing and specialized applications.

RISC vs. CISC

Feature	RISC (Reduced Instruction Set Computing)	CISC (Complex Instruction Set Computing)
Instruction Set	Small and simple	Large and complex
Instruction Length	Fixed	Variable
Execution Time	Single-cycle execution	Multi-cycle execution
Pipelining	Highly pipelined	Less pipelined

Feature	RISC (Reduced Instruction Set Computing)	CISC (Complex Instruction Set Computing)
Memory Access	Load/store architecture	Memory-to-memory instructions
Design Philosophy	Simplify instructions to increase performance	Combine multiple operations in a single instruction
Examples	ARM, MIPS	x86, VAX

SIMD vs. Other Architectures

Feature	SIMD (Single Instruction, Multiple Data)	Other Architectures
Data Parallelism	High	Varies (generally lower)
Instruction Parallelism	Limited	Varies (depends on the architecture)
Usage	Multimedia, scientific computing	General-purpose computing
Performance	High for specific tasks	Varies
Examples	Intel AVX, ARM NEON	General CPU architectures

Processor Types Comparison

Feature	Superscalar Processors	Vector Processors	Scalar Processors
Parallelism	Multiple instructions per cycle	Multiple data elements per instruction	Single instruction per cycle
Pipelining	Deeply pipelined	Specialized pipelines for vector operations	Simple or moderately pipelined
Usage	General-purpose high-performance CPUs	Scientific and engineering computations	General-purpose low to moderate performance
Instruction Execution	Out-of-order execution	SIMD execution	In-order execution
Examples	Intel Core i9, AMD Ryzen	Cray-1, NEC SX-ACE	Early Intel and AMD CPUs, ARM Cortex-A series

General Terms in x86 Architecture

1. **ISA (Instruction Set Architecture)**: Defines the set of instructions that the CPU can execute. x86 is a type of ISA.
2. **Opcode**: The portion of a machine language instruction that specifies the operation to be performed.
3. **Microarchitecture**: The implementation of the ISA within a processor, including the design of its control and data paths, register set, and memory architecture.
4. **Clock Cycle**: The period of a CPU's clock signal. Instructions are executed in sync with these cycles.
5. **Pipeline**: A technique used to overlap the execution of multiple instructions by dividing them into several stages.
6. **Cache**: A small, fast memory located close to the CPU core that stores frequently accessed data to speed up processing.
7. **MMU (Memory Management Unit)**: Hardware responsible for handling virtual memory and memory protection.
8. **Bus**: A communication system that transfers data between different components of the computer, such as the CPU, memory, and I/O devices.

Fetch-Decode-Execute Cycle

The Fetch-Decode-Execute cycle, also known as the instruction cycle, is the basic operational process of a computer's CPU. Here's how it works in the context of the x86 architecture:

1. **Fetch**: The CPU fetches the instruction from memory. The address of the next instruction is held in the Instruction Pointer (IP) register. The fetched instruction is then placed into the Instruction Register (IR).

2. **Decode**: The CPU decodes the fetched instruction to determine what operation needs to be performed. The decoding process interprets the opcode and any operands that are part of the instruction.

3. **Execute**: The CPU executes the decoded instruction. This may involve performing an arithmetic or logic operation, accessing memory, or updating the registers. The result of the execution is then stored back in a register or memory.

Example Breakdown

To illustrate, let's consider a simple x86 instruction: `ADD EAX, EBX`

1. **Fetch**: The `ADD` instruction is fetched from memory and placed in the Instruction Register.
2. **Decode**: The CPU decodes the instruction and identifies that it needs to add the contents of register `EBX` to register `EAX`.
3. **Execute**: The CPU performs the addition and stores the result in `EAX`.

This cycle repeats continuously while the computer is running, allowing the CPU to process instructions efficiently.

Sure thing! Let's dive deep into opcodes, addressing modes, and different types of instructions in the x86 architecture.

Opcodes

An **opcode** (operation code) is a portion of a machine language instruction that specifies the operation to be performed. Opcodes are a crucial part of the instruction set and determine what action the CPU should take. Here are some examples of common x86 opcodes:

- `ADD`: Adds two values.
- `MOV`: Moves data from one location to another.
- `CMP`: Compares two values.
- `JMP`: Jumps to a specified address.
- `CALL`: Calls a procedure.
- `RET`: Returns from a procedure.

Addressing Modes

Addressing modes define how the operand of an instruction is chosen. The x86 architecture supports several addressing modes, each allowing flexible and efficient ways to access data. Here are the main addressing modes:

1. **Immediate Addressing Mode**:
   - The operand is a constant value.
   - Example: `MOV EAX, 10` (Moves the constant value 10 into the EAX register)
2. **Register Addressing Mode**:
   - The operand is a register.
   - Example: `MOV EAX, EBX` (Moves the value in the EBX register into the EAX register)
3. **Direct Addressing Mode**:
   - The operand is a memory address.
   - Example: `MOV EAX, [1234H]` (Moves the value at memory address 1234H into the EAX register)
4. **Indirect Addressing Mode**:
   - The operand is located in memory, and the address of the operand is held in a register.
   - Example: `MOV EAX, [EBX]` (Moves the value at the memory address pointed to by EBX into the EAX register)
5. **Base-plus-Index Addressing Mode**:
   - Combines a base register and an index register to calculate the memory address.
   - Example: `MOV EAX, [EBX + ECX]` (Moves the value at the address calculated by adding EBX and ECX into the EAX register)

6. **Scaled Index with Displacement**:
    - Uses an index register multiplied by a scaling factor, plus a displacement.
    - Example: `MOV EAX, [EBX + ECX * 4 + 8]` (Moves the value at the address calculated by adding EBX, ECX multiplied by 4, and 8 into the EAX register)

Types of Instructions

The x86 instruction set can be categorized into various types based on their functions. Here are some key categories:

1. **Data Transfer Instructions**:
    - Used to move data between registers, memory, and I/O ports.
    - Examples: `MOV`, `PUSH`, `POP`
2. **Arithmetic Instructions**:
    - Perform arithmetic operations like addition, subtraction, multiplication, and division.
    - Examples: `ADD`, `SUB`, `MUL`, `DIV`
3. **Logic Instructions**:
    - Perform logical operations like AND, OR, XOR, and NOT.
    - Examples: `AND`, `OR`, `XOR`, `NOT`
4. **Control Transfer Instructions**:
    - Change the flow of execution in a program.
    - Examples: `JMP`, `CALL`, `RET`, `JZ` (Jump if Zero), `JNZ` (Jump if Not Zero)
5. **String Instructions**:
    - Perform operations on strings of data.
    - Examples: `MOVS` (Move String), `LODS` (Load String), `STOS` (Store String)
6. **Bit Manipulation Instructions**:
    - Perform operations on individual bits.
    - Examples: `SHL` (Shift Left), `SHR` (Shift Right), `ROL` (Rotate Left), `ROR` (Rotate Right)
7. **System Instructions**:
    - Used for system-level operations and control.
    - Examples: `HLT` (Halt), `NOP` (No Operation), `CLI` (Clear Interrupt Flag), `STI` (Set Interrupt Flag)

Instruction Format

The format of an x86 instruction typically includes several components, each of which serves a specific purpose. Here are the main components of an x86 instruction:

1. **Opcode**: Specifies the operation to be performed (e.g., ADD, MOV).
2. **Operands**: Specifies the data to be operated on. Operands can be registers, immediate values, or memory addresses.

3. **Addressing Mode**: Determines how the operand is accessed (e.g., immediate, register, direct, indirect).
4. **Displacement**: An optional component used in some addressing modes to calculate the effective address.
5. **Prefix**: Optional bytes that modify the behavior of the instruction (e.g., segment override, repeat, and lock prefixes).

Here's an example of an x86 instruction with its components:

```
MOV EAX, [EBX+ECX*4]
```

- **Opcode**: MOV
- **Operands**: EAX, [EBX+ECX*4]
- **Addressing Mode**: Base-plus-index with scaling
- **Displacement**: None in this case

Detailed Explanation of Instructions

Let's delve into some of the key types of instructions in the x86 architecture:

Data Transfer Instructions

- **MOV**: Moves data from one location to another.
  - Example: `MOV EAX, EBX` (Moves the value in EBX to EAX)
- **PUSH**: Pushes data onto the stack.
  - Example: `PUSH EAX` (Pushes the value in EAX onto the stack)
- **POP**: Pops data from the stack.
  - Example: `POP EAX` (Pops the top value from the stack into EAX)

Arithmetic Instructions

- **ADD**: Adds two values.
  - Example: `ADD EAX, 5` (Adds 5 to the value in EAX)
- **SUB**: Subtracts one value from another.
  - Example: `SUB EAX, EBX` (Subtracts the value in EBX from EAX)
- **MUL**: Multiplies two values.
  - Example: `MUL EBX` (Multiplies the value in EBX with EAX)
- **DIV**: Divides one value by another.
  - Example: `DIV EBX` (Divides the value in EAX by the value in EBX)

Logic Instructions

- **AND**: Performs a bitwise AND operation.
  - Example: `AND EAX, EBX` (Performs a bitwise AND between EAX and EBX)
- **OR**: Performs a bitwise OR operation.
  - Example: `OR EAX, EBX` (Performs a bitwise OR between EAX and EBX)

- **XOR**: Performs a bitwise XOR operation.
  - Example: `XOR EAX, EBX` (Performs a bitwise XOR between EAX and EBX)
- **NOT**: Performs a bitwise NOT operation.
  - Example: `NOT EAX` (Inverts all bits in EAX)

Control Transfer Instructions

- **JMP**: Jumps to a specified address.
  - Example: `JMP 0x1234` (Jumps to memory address 0x1234)
- **CALL**: Calls a procedure.
  - Example: `CALL 0x1234` (Calls the procedure at memory address 0x1234)
- **RET**: Returns from a procedure.
  - Example: `RET` (Returns from the current procedure)
- **JZ**: Jumps if zero flag is set.
  - Example: `JZ 0x1234` (Jumps to memory address 0x1234 if the zero flag is set)
- **JNZ**: Jumps if zero flag is not set.
  - Example: `JNZ 0x1234` (Jumps to memory address 0x1234 if the zero flag is not set)

String Instructions

- **MOVS**: Moves string data from source to destination.
  - Example: `MOVS` (Moves data from the source string to the destination string)
- **LODS**: Loads string data into the accumulator.
  - Example: `LODS` (Loads data from the source string into the accumulator)
- **STOS**: Stores string data from the accumulator.
  - Example: `STOS` (Stores data from the accumulator into the destination string)

Bit Manipulation Instructions

- **SHL**: Shifts bits to the left.
  - Example: `SHL EAX, 1` (Shifts the bits in EAX one position to the left)
- **SHR**: Shifts bits to the right.
  - Example: `SHR EAX, 1` (Shifts the bits in EAX one position to the right)
- **ROL**: Rotates bits to the left.
  - Example: `ROL EAX, 1` (Rotates the bits in EAX one position to the left)
- **ROR**: Rotates bits to the right.

- Example: `ROR EAX, 1` (Rotates the bits in EAX one position to the right)

Let's dive into the world of assemblers and assembly language!

Assembly Language

Assembly language is a low-level programming language that is closely related to machine code instructions for a specific CPU architecture. It uses mnemonic codes and labels to represent machine-level instructions, making it easier for humans to read and write.

Assembly language provides direct control over the hardware, allowing programmers to write highly optimized code. However, it's more complex and less portable than high-level programming languages.

Assembler

An assembler is a program that translates assembly language code into machine code (binary) that the CPU can execute. The assembler reads the assembly language instructions and converts them into the corresponding machine language instructions.

Example of Assembly Language Code

Here's a simple example of an assembly language program for the x86 architecture. This program adds two numbers and stores the result in a register:

```
section .data
    num1 db 10     ; Declare a byte variable num1 with value 10
    num2 db 20     ; Declare a byte variable num2 with value 20

section .text
    global _start

_start:
    mov al, [num1] ; Move the value of num1 into register AL
    add al, [num2] ; Add the value of num2 to the value in AL
    mov [result], al ; Move the result into the memory location result

    ; Exit the program
    mov eax, 1     ; System call number (sys_exit)
    xor ebx, ebx   ; Exit code 0
    int 0x80       ; Call kernel
```

Explanation of the Code

1. **Data Section (`section .data`)**: This section defines and initializes data variables.
   - `num1 db 10`: Declares a byte variable named `num1` with a value of 10.
   - `num2 db 20`: Declares a byte variable named `num2` with a value of 20.
2. **Text Section (`section .text`)**: This section contains the code (instructions) to be executed.
   - `global _start`: Defines the entry point of the program.
   - `_start`: Label indicating the start of the program.
3. **Instructions**:
   - `mov al, [num1]`: Moves the value of `num1` into the `AL` register.
   - `add al, [num2]`: Adds the value of `num2` to the value in `AL`.
   - `mov [result], al`: Moves the result (sum) into the memory location `result`.
4. **Exit the Program**:
   - `mov eax, 1`: Sets up the system call number for `sys_exit`.
   - `xor ebx, ebx`: Sets the exit code to 0.
   - `int 0x80`: Calls the kernel to exit the program.

Using an Assembler

To convert the above assembly code into machine code, you can use an assembler like NASM (Netwide Assembler). The following command assembles the code and generates an executable binary:

```
nasm -f elf32 program.asm -o program.o
ld -m elf_i386 program.o -o program
./program
```

This process translates the human-readable assembly language code into the binary code that the CPU can execute.

- **Assembly Language**: A low-level programming language with mnemonic codes representing machine-level instructions.
- **Assembler**: A program that translates assembly language code into machine code.
-

Example Code: Demonstrates adding two numbers and storing the result using x86 assembly language.

title: “11_cpuarch” time: “2025-01-16 :: 22:15:23” created: 22-10-20024 header: ‘Malware Analysis Class’ footer: ‘Unit I: Malware Analysis - By Charudatta Korde’ tags: nfsu, notes, ppt, malware author: Charudatta Korde —

Memory Hierarchy

The memory hierarchy in computer architecture is designed to provide a balance between speed and size, optimizing cost and performance. It consists of different types of memory, each with distinct characteristics.

Types of Memory

1. **Cache Memory**:
   - **Speed**: Extremely fast
   - **Size**: Small (kilobytes to megabytes)
   - **Purpose**: Stores frequently accessed data to speed up processing.
2. **Main Memory (RAM)**:
   - **Speed**: Fast
   - **Size**: Moderate (gigabytes)
   - **Purpose**: Holds data and programs currently in use.
3. **Secondary Storage**:
   - **Speed**: Slower
   - **Size**: Large (gigabytes to terabytes)
   - **Purpose**: Stores data and programs not currently in use (e.g., HDD, SSD).

Speed and Size Comparison

Memory Type	Speed	Size
Cache Memory	Extremely Fast	Small (KB to MB)
Main Memory	Fast	Moderate (GB)
Secondary Storage	Slower	Large (GB to TB)

Cache Memory

Cache memory plays a crucial role in improving the performance of the CPU by storing frequently accessed data and instructions.

Importance and Function

- **Importance**: Reduces the time it takes to access data from the main memory.
- **Function**: Acts as a buffer between the CPU and main memory, storing copies of frequently accessed data to reduce latency.

Levels of Cache (L1, L2, L3)

1. **L1 Cache**:
   - Closest to the CPU core
   - Smallest size, but fastest

- Separate instruction and data caches (Harvard architecture)

2. **L2 Cache**:
   - Larger than L1
   - Slower than L1, but still fast
   - Shared between cores in some CPUs
3. **L3 Cache**:
   - Largest cache level
   - Slowest of the three, but still faster than main memory
   - Typically shared across all cores in a multi-core processor

Virtual Memory

Virtual memory allows a computer to compensate for physical memory shortages by temporarily transferring data from RAM to disk storage.

Concept and Benefits

- **Concept**: Creates an illusion of a large, continuous memory space by using both physical memory and disk space.
- **Benefits**: Increases the apparent amount of RAM, isolates processes, and provides memory protection.

Implementation (Paging, Segmentation)

1. **Paging**: Divides memory into fixed-size pages, managing them with a page table.
2. **Segmentation**: Divides memory into variable-sized segments, each representing a logical unit such as a function or data structure.

Pipelining and Its Stages

Pipelining is a technique used in CPU design to increase instruction throughput by overlapping the execution of multiple instructions.

Stages of Pipelining

1. **Fetch**: Retrieve the instruction from memory.
2. **Decode**: Interpret the instruction and prepare the necessary data.
3. **Execute**: Perform the operation specified by the instruction.
4. **Memory Access**: Read/write data from/to memory if needed.
5. **Write Back**: Store the result back into a register.

Instruction Level Parallelism (ILP)

ILP is the parallel execution of multiple instructions to improve CPU performance.

Concepts of ILP

- **Parallelism**: Executing multiple instructions simultaneously to increase performance.
- **Dependencies**: Handling data and control dependencies between instructions.

Techniques to Achieve ILP

1. **Pipelining**: Overlapping instruction execution stages.
2. **Superscalar Execution**: Issuing multiple instructions per cycle.
3. **Out-of-Order Execution**: Reordering instructions to avoid stalls and improve efficiency.

Advanced CPU Architectures

Superscalar Architecture

- **Description**: Executes more than one instruction per clock cycle by employing multiple execution units.
- **Benefits**: Increases instruction throughput and overall performance.

VLIW (Very Long Instruction Word) Architecture

- **Description**: Uses long instruction words that encode multiple operations to be executed in parallel.
- **Benefits**: Simplifies hardware by relying on the compiler to identify parallelism, but requires sophisticated compiler support.

Performance Metrics

Measuring CPU Performance

1. **CPI (Cycles Per Instruction)**: Average number of cycles needed to execute an instruction.
2. **MIPS (Million Instructions Per Second)**: Number of millions of instructions executed per second.
3. **FLOPS (Floating Point Operations Per Second)**: Number of floating-point operations executed per second.

Factors Affecting Performance

- **Clock Speed**: Higher clock speeds generally mean faster execution.
- **Instruction Set Efficiency**: Optimized instructions can execute more efficiently.
- **Parallelism**: Increased parallelism through techniques like ILP and superscalar execution improves performance.
- **Cache Performance**: Efficient caching reduces memory access time and improves speed.

Euclidean Algorithm for GCD

Assembly Language (x86) Here's a simple assembly language program to find the GCD of two numbers using the Euclidean algorithm:

```
section .data
    num1 dw 56      ; First number
    num2 dw 98      ; Second number

section .bss
    result resw 1   ; Result

section .text
    global _start

_start:
    mov ax, [num1]  ; Load num1 into AX
    mov bx, [num2]  ; Load num2 into BX

gcd_loop:
    cmp bx, 0       ; Compare BX with 0
    je done         ; If BX is 0, jump to done
    xor dx, dx      ; Clear DX for division
    div bx          ; Divide AX by BX, quotient in AX, remainder in DX
    mov ax, bx      ; Move BX into AX
    mov bx, dx      ; Move remainder into BX
    jmp gcd_loop    ; Repeat the loop

done:
    mov [result], ax ; Store result in memory

    ; Exit the program
    mov eax, 1      ; System call number (sys_exit)
    xor ebx, ebx    ; Exit code 0
    int 0x80        ; Call kernel
```

Explanation of Assembly Code

1. **Data Section**: Initializes two numbers (`num1` and `num2`).
2. **BSS Section**: Declares a variable (`result`) to store the result.
3. **Text Section**: Contains the code to execute the Euclidean algorithm.
    - **Loading Values**: `mov ax, [num1]` and `mov bx, [num2]` load the numbers into registers `AX` and `BX`.
    - **GCD Loop**: Repeats the division and remainder calculation until `BX` becomes 0.
    - **Store Result**: Stores the result in memory.

C Language

Here's the same algorithm implemented in C:

```c
#include <stdio.h>

int gcd(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

int main() {
    int num1 = 56;
    int num2 = 98;
    int result = gcd(num1, num2);

    printf("GCD of %d and %d is %d\n", num1, num2, result);
    return 0;
}
```

Explanation of C Code

1. **gcd Function**: Implements the Euclidean algorithm.
   - **While Loop**: Repeats the calculation until `b` becomes 0.
   - **Modulus Operation**: `a % b` calculates the remainder.
   - **Return Statement**: Returns the greatest common divisor.
2. **Main Function**:
   - **Initialize Numbers**: `num1` and `num2` are initialized.
   - **Call gcd Function**: Calls the `gcd` function and stores the result.
   - **Print Result**: Prints the result to the console.

Comparison

- **Low-Level vs. High-Level**: Assembly language provides low-level control over the hardware, allowing precise manipulation of registers and memory. In contrast, C is a high-level language that abstracts these details, making it easier to read and write.
- **Instruction Set**: Assembly language requires explicit instructions for each operation, while C uses high-level constructs like functions and loops.
- **Portability**: Assembly language is specific to a particular CPU architecture (x86 in this case), while C is more portable across different platforms with minimal changes.

- **Readability**: C code is generally more readable and maintainable, making it easier for developers to understand and modify.

Example C Code

Here's a simple C function that adds two integers:

```c
int add(int a, int b) {
    return a + b;
}

int main() {
    int x = 5;
    int y = 10;
    int result = add(x, y);
    return 0;
}
```

Corresponding Assembly Code

Let's see how this C code might be translated into x86 assembly language. The exact assembly code can vary depending on the compiler and optimization settings, but here's a typical translation:

```
section .data
    x dd 5          ; Declare variable x with value 5
    y dd 10         ; Declare variable y with value 10
    result dd 0     ; Declare variable result

section .text
    global _start

_start:
    ; Call add(x, y)
    mov eax, [x]    ; Move value of x into EAX register
    mov ebx, [y]    ; Move value of y into EBX register
    call add        ; Call add function
    mov [result], eax ; Move return value into result

    ; Exit program
    mov eax, 1      ; System call number (sys_exit)
    xor ebx, ebx    ; Exit code 0
    int 0x80        ; Call kernel

add:
    push ebp        ; Save base pointer
    mov ebp, esp    ; Set base pointer to current stack pointer
```

```
mov eax, [ebp+8] ; Move first argument (a) into EAX
mov ebx, [ebp+12] ; Move second argument (b) into EBX
add eax, ebx    ; Add EBX to EAX (a + b)
pop ebp         ; Restore base pointer
ret             ; Return from function
```

Translation Process

1. **Data Section**:
   - The data section declares variables `x`, `y`, and `result`.
2. **Text Section**:
   - The text section contains the executable code.
   - The `global _start` directive marks the entry point of the program.
3. **_start Label**:
   - The main function is represented by the `_start` label.
   - The values of `x` and `y` are loaded into registers `EAX` and `EBX`, respectively.
   - The `call add` instruction calls the `add` function.
4. **add Function**:
   - The `add` function is defined in assembly.
   - The `push ebp` and `mov ebp, esp` instructions set up the stack frame.
   - The arguments `a` and `b` are accessed from the stack using the base pointer (`[ebp+8]` and `[ebp+12]`).
   - The `add eax, ebx` instruction adds the values.
   - The `pop ebp` and `ret` instructions clean up the stack and return to the caller.

Key Points

- **Registers**: General-purpose registers (`EAX`, `EBX`) are used to hold values and perform operations.
- **Stack Frame**: The stack is used to pass arguments to functions and manage local variables.
- **Function Calls**: The `call` and `ret` instructions manage function calls and returns.
- **Instruction Mapping**: Each C statement typically maps to one or more assembly instructions.
- **C Code**: High-level, human-readable code that defines the logic of the program.
- **Assembly Code**: Low-level, machine-readable code that directly controls the CPU.
- **Translation Process**: The compiler translates C code into assembly code, optimizing it for the target architecture.

GCC Compiler

GNU Compiler Collection (GCC) is a versatile compiler that supports multiple programming languages, including C, C++, Objective-C, Fortran, Ada, Go, and Rust. It is widely used for compiling code for various hardware architectures and operating systems.

Key Features - I

- **Language Support**: Supports a wide range of programming languages.
- **Cross-Compilation**: Can compile code for different target platforms.
- **Optimization**: Includes various optimization options to improve performance.
- **Open Source**: Distributed under the GNU General Public License (GPL).
- **Extensibility**: Can be extended with plugins and additional front ends.

NASM Assembler

Netwide Assembler (NASM) is an assembler for the Intel x86 architecture, capable of producing 16-bit, 32-bit, and 64-bit programs. It is known for its simplicity and ease of use.

Key Features - II

- **Architecture Support**: Supports x86 and x86-64 architectures.
- **Output Formats**: Can generate various binary formats, including ELF, COFF, Mach-O, and binary files.
- **Syntax**: Uses a variant of Intel assembly syntax, which is simpler than AT&T syntax.
- **Open Source**: Distributed under the simplified BSD license.
- **Portability**: Can run on Unix-like systems, Windows, and DOS.

Comparison - II

Feature	GCC Compiler	NASM Assembler
Language Support	Multiple languages (C, C++, etc.)	Assembly language
Architecture	Multiple architectures	x86 and x86-64
Output Formats	Various formats (ELF, COFF, etc.)	Various formats (ELF, COFF, etc.)
License	GPL	BSD
Optimization	Yes	No
Cross-Compilation	Yes	No
Syntax	Depends on language	Intel assembly syntax

- **GCC** is a comprehensive compiler for high-level languages, offering extensive optimization and cross-compilation capabilities.
-

NASM is a specialized assembler for x86 architectures, known for its simplicity and ease of use.

title: "12_tools.md" time: "2025-02-05 :: 17:00:16" tags: nfsu, notes, malware —

1. **IDA Pro**: Used for disassembling and debugging complex malware to understand its structure and logic.
   - CLI Usage: Not applicable (GUI tool).
2. **Ghidra**: A free tool for reverse engineering that helps analyze obfuscated code and develop countermeasures.
   - CLI Usage: `ghidraRun`
3. **Wireshark**: Captures and analyzes network traffic to detect malicious communication patterns.
   - CLI Usage: `wireshark -r capture_file.pcap`
4. **OllyDbg**: Assists in dynamic analysis by debugging and tracing malware execution in real-time.
   - CLI Usage: Not applicable (GUI tool).
5. **Cuckoo Sandbox**: Executes and monitors malware in an isolated environment to observe its behavior.
   - CLI Usage: `cuckoo submit file.exe`
6. **CFF Explorer**: Used to inspect, edit, and rebuild PE files, providing detailed information about executable file structures.
   - CLI Usage: Not applicable (GUI tool).
7. **Dependency Walker**: Analyzes Windows module dependencies to detect missing or invalid files, helping troubleshoot application errors.
   - CLI Usage: `depends.exe file.exe`
8. **Regshot**: Compares registry snapshots before and after system changes to identify modifications, useful for monitoring software installations.
   - CLI Usage: `regshot -shot1 file1 -shot2 file2 -output differences.txt`
9. **FLOSS (FireEye Labs Obfuscated String Solver)**: Automatically extracts and deobfuscates strings from malware binaries. Useful for static analysis of unknown binaries.
   - CLI Usage: `floss malware.exe`
10. **Resource Hacker**: A resource extraction utility and resource compiler for Windows. It retrieves detailed information about the file in "Version Info".
    - CLI Usage: Not applicable (GUI tool).
11. **Wireshark**: A network protocol analyzer used for capturing and analyzing network traffic. It helps in identifying suspicious or malicious network activity.

- CLI Usage: `wireshark -r capture_file.pcap`

12. **YARA**: A tool for creating and using malware identification and classification rules. It's like a "Swiss knife" for malware researchers.
    - CLI Usage: `yara -r rule_file.yar target_file`
13. **Volatility**: An open-source memory forensics framework used to analyze RAM captures for digital forensics and incident response.
    - CLI Usage: `volatility -f memory_dump.raw --profile=Win7SP1x64 pslist`
14. **Any.RUN**: An interactive online malware sandbox that allows you to run and analyze suspicious files in a safe environment.
    - CLI Usage: Not applicable (web-based tool).
15. **Calmav**: A tool for analyzing and detecting malware based on behavioral patterns and heuristics.
    - CLI Usage: `clamscan -r /path/to/directory`