Malware Analysis

Contents

1 Introduction **3**
- 1.1 Initial Analysis with Antivirus Programs . . . 3
- 1.2 Limitations of Antivirus Tools . . . 3
- 1.3 Benefits of Using Multiple Antivirus Programs . . . 3

2 Basic Static Techniques **4**
- 2.1 Antivirus Scanning . . . 4
- 2.2 Hashing . . . 4
- 2.3 Finding Strings . . . 4
- 2.4 Packed and Obfuscated Programs . . . 4
- 2.5 Portable Executable File Format . . . 4
- 2.6 Linked Libraries and Functions . . . 5

3 Malware Analysis in Virtual Machines **7**
- 3.1 Setting Up a Safe Environment for Malware Analysis . . . 7
- 3.2 Challenges with Physical Machines . . . 7
- 3.3 Preference for Virtual Machines . . . 7
- 3.4 The Structure of a Virtual Machine . . . 7
- 3.5 Creating Your Malware Analysis Machine . . . 8
- 3.6 Using Your Malware Analysis Machine . . . 8
  - 3.6.1 Simulating Network Services for Malware Analysis . . . 8
  - 3.6.2 Connecting Malware to the Internet . . . 8
  - 3.6.3 Managing Peripheral Devices . . . 9
  - 3.6.4 Taking Snapshots in VMware . . . 9
  - 3.6.5 Workflow with Snapshots . . . 9
  - 3.6.6 Transferring Files from a VM . . . 9
- 3.7 The Risks of Using VMware for Malware Analysis . . . 9
- 3.8 Record/Replay: Running Your Computer in Reverse . . . 9
- 3.9 Conclusion . . . 10

4 Basic Dynamic Analysis **11**
- 4.1 Sandboxes: The Quick-and-Dirty Approach . . . 11
- 4.2 Running Malware . . . 11
- 4.3 Monitoring with Process Monitor . . . 12
- 4.4 Viewing Processes with Process Explorer . . . 12
- 4.5 Comparing Registry Snapshots with Regshot . . . 13
- 4.6 Faking a Network . . . 13
  - 4.6.1 Using ApateDNS . . . 13
  - 4.6.2 Monitoring with Netcat . . . 14
- 4.7 Packet Sniffing with Wireshark . . . 14
- 4.8 Using INetSim . . . 14

4.9 Basic Dynamic Tools in Practice 14
4.10 Conclusion 15

5 Advanced Static Analysis. A Crash Course in x86 Disassembly 17
5.1 Levels of Abstraction 17
5.2 Reverse-Engineering 18
5.3 The x86 Architecture 18

6 Recognizing C Code Constructs in Assembly 20
6.1 Global vs. Local Variables 20
6.2 Disassembling Arithmetic Operations 20
6.3 Recognizing if Statements 21
6.4 Recognizing Loops 22
6.5 Understanding Function Call Conventions 22
6.6 Analyzing Switch Statements 23
6.7 Disassembling Arrays 23
6.8 Identifying Structs 24
6.9 Analyzing Linked List Traversal 25

7 Types of Malware 26

1 Introduction

1.1 Initial Analysis with Antivirus Programs

A good starting point for analyzing potential malware is to scan it with multiple antivirus programs, which might already recognize it through their databases.

1.2 Limitations of Antivirus Tools

- **Reliance on Databases**: Antivirus software mainly uses databases of known malware signatures and heuristic analysis for detection.
- **Evading Detection**: Malware authors can modify their code to change the malware's signature, making it harder for antivirus tools to detect.
- **Rare Malware**: Less common malware may go undetected because it is not included in the antivirus databases.
- **Heuristic Analysis**: While heuristic methods can identify unknown malicious code, they are not foolproof and can be bypassed by new, unique malware types.

1.3 Benefits of Using Multiple Antivirus Programs

Different antivirus programs use varied signatures and heuristic methods, making it beneficial to scan a suspected malware file with several programs to increase the chances of detection.

- **VirusTotal as a Resource**: VirusTotal is a website that allows files to be scanned by multiple antivirus engines simultaneously, providing a comprehensive report on whether any engines identified the file as malicious, the name of the malware, and additional information if available.

2 Basic Static Techniques

2.1 Antivirus Scanning

A primary step in malware analysis is utilizing antivirus programs to identify known malware through file signatures and heuristics. However, limitations include the ease with which malware authors can modify code signatures and the potential for new or rare malware to evade detection. Using multiple antivirus engines, such as those provided by VirusTotal, can offer broader detection capabilities.

2.2 Hashing

Employing hashing functions (e.g., MD5, SHA-1) to generate unique identifiers for malware samples allows analysts to label, share, and search for information on specific malware instances efficiently.

2.3 Finding Strings

Analyzing strings within a program helps hint at its functionality, such as network connections (URLs) or specific actions (error messages). Tools like the Strings program facilitate this analysis by extracting ASCII and Unicode strings from executable files.

2.4 Packed and Obfuscated Programs

Malware authors often obfuscate or pack their malware to hinder analysis. Packed malware compresses or encrypts the original code, making static analysis challenging. Tools like PEiD can identify packing algorithms, but unpacking is often required for further analysis.

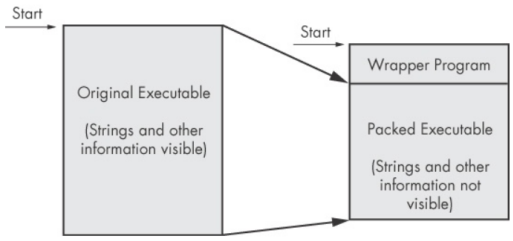

Figure 1: The file on the left is the original executable, with all strings, imports, and other information visible. On the right is a packed executable. All of the packed file's strings, imports, and other information are compressed and invisible to most static analysis tools.

2.5 Portable Executable File Format

The PE file format is standard for Windows executables and DLLs, containing headers with valuable metadata for analysis. Tools like PEview and Dependency Walker can explore this metadata, including imports, exports, and sections, to infer malware functionality.

DLL	Description
Kernel32.dll	This is a very common DLL that contains core functionality, such as access and manipulation of memory, files, and hardware.
Advapi32.dll	This DLL provides access to advanced core Windows components such as the Service Manager and Registry.
User32.dll	This DLL contains all the user-interface components, such as buttons, scroll bars, and components for controlling and responding to user actions.
Gdi32.dll	This DLL contains functions for displaying and manipulating graphics.
Ntdll.dll	This DLL is the interface to the Windows kernel. Executables generally do not import this file directly, although it is always imported indirectly by Kernel32.dll. If an executable imports this file, it means that the author intended to use functionality not normally available to Windows programs. Some tasks, such as hiding functionality or manipulating processes, will use this interface.
WSock32.dll and Ws2_32.dll	These are networking DLLs. A program that accesses either of these most likely connects to a network or performs network-related tasks.
Wininet.dll	This DLL contains higher-level networking functions that implement protocols such as FTP, HTTP, and NTP.

Table 1: Common DLLs and Their Descriptions

2.6 Linked Libraries and Functions

Analyzing the libraries and functions that an executable imports can reveal its capabilities and intentions. This includes evaluating dynamically linked functions, which are critical for understanding how malware interacts with the operating system and other software.

Executable	**Description**
.text	Contains the executable code.
.rdata	Holds read-only data that is globally accessible within the program.
.data	Stores global data accessed throughout the program.
.idata	Sometimes present and stores the import function information; if this section is not present, the import function information is stored in the .rdata section.
.edata	Sometimes present and stores the export function information; if this section is not present, the export function information is stored in the .rdata section.
.pdata	Present only in 64-bit executables and stores exception-handling information.
.rsrc	Stores resources needed by the executable.
.reloc	Contains information for relocation of library files.

Table 2: Sections of a PE File for a Windows Executable

3 Malware Analysis in Virtual Machines

3.1 Setting Up a Safe Environment for Malware Analysis

Before conducting dynamic analysis on malware, it's crucial to create a secure environment to prevent the malware from compromising your machine or spreading to others in the network.

- **Use of Dedicated Machines**: Malware analysis can be safely conducted using either dedicated physical machines or virtual machines. These setups prevent malware from affecting other networked devices.
- **Airgapped Networks**: Utilizing physical machines within airgapped networks (isolated from the Internet and other networks) helps in safely running malware by preventing its spread. However, this setup lacks Internet connectivity, which can be a limitation since many malware types require an online connection to activate their full functionalities.

3.2 Challenges with Physical Machines

1. **Dependency on the Internet**: Some malware needs an internet connection for activities like updates, command, and control, which is not possible on airgapped networks.
2. **Difficulty in Removal**: Malware on physical machines can be hard to remove. Tools like Norton Ghost are used for managing OS backup images for easy restoration after analysis.
3. **Detection by Malware**: Some malware can detect when they are run on virtual machines and may alter their behavior to avoid analysis, which is less of an issue with physical machines.

3.3 Preference for Virtual Machines

Despite the potential for malware to detect and react differently in virtual environments, virtual machines are preferred for dynamic malware analysis due to the ease of managing and resetting the environment, alongside the reduced risk of spreading malware.

3.4 The Structure of a Virtual Machine

Virtual Machines (VMs) function as a computer within a computer, where a guest operating system (OS) is installed within a host OS. This setup maintains isolation between the guest OS running on the VM and the host OS, ensuring that any malware active on the VM does not affect the host system.

- **Recovery and Safety Measures**: If malware compromises the VM, the solution is straightforward — either reinstall the guest OS or revert the VM to a previously saved, clean state. This ease of restoration makes VMs particularly suited for malware analysis.

3.5 Creating Your Malware Analysis Machine

Before utilizing a virtual machine (VM) for malware analysis, one must first create the VM. The recommended starting point for virtual hard drive size is 20GB, sufficient for the guest OS and malware analysis tools, with VMware dynamically adjusting the disk space based on actual usage.

1. **Installing OS and Applications:**
   - Windows is often the chosen OS for malware analysis due to its popularity and the prevalence of malware targeting it. Windows XP is highlighted as a common choice despite its age.
   - Essential applications for malware analysis should be installed on the VM.
2. **Configuring VMware for Malware Analysis:**
   (a) Networking Considerations: Given that many malware specimens exhibit network behavior, VMware's networking options are crucial for analysis. However, it's important to restrict malware from accessing the analyst's network to prevent spread.
   (b) Disconnecting the Network: While possible, disconnecting the VM's network is not generally advised as it limits the ability to analyze network-based malware activities.
   (c) Host-Only Networking: This configuration isolates the VM's network from the internet, creating a private LAN between the host and guest OS, which is ideal for containing malware while observing its network behavior.
      - Ensure the host machine is fully patched and has a restrictive firewall setup as a precaution against malware spreading.
      - VMware creates a virtual network adapter for host-only networking, connecting the host and VM without using the host's physical network adapter.
   (d) Using Multiple VMs for Advanced Analysis: An advanced setup involves linking multiple VMs within a LAN but disconnected from both the internet and the host machine. This allows for a controlled environment where one VM can analyze malware while another provides services, without risking external networks.
      - Virtual machine teams can be created for easier management of multiple VMs, allowing coordinated control over their power and network settings.

3.6 Using Your Malware Analysis Machine

3.6.1 Simulating Network Services for Malware Analysis

To effectively analyze malware, it's crucial to replicate the network services the malware relies on, such as DNS for IP address resolution and HTTP servers for downloading additional payloads. This setup allows for detailed observation of the malware's network behavior within a controlled environment.

3.6.2 Connecting Malware to the Internet

While risky, connecting malware to the internet can provide a more realistic analysis environment. This step should be taken with caution, assessing the risks of malicious

activities like spreading malware, participating in DDoS attacks, or alerting the malware's author. Bridged network adapters or Network Address Translation (NAT) mode in VMware can facilitate this connection, with NAT mode allowing the VM to share the host's IP connection.

3.6.3 Managing Peripheral Devices

Peripheral devices, such as USB drives, can pose security risks when connected to virtual machines, especially with malware that spreads via USB. VMware provides options to manage the connection of external devices, allowing users to prevent automatic connection of new USB devices to the VM.

3.6.4 Taking Snapshots in VMware

Snapshots are a powerful feature in VMware, enabling users to save the current state of a VM and revert back to that point later, effectively undoing any changes made, including malware infection. This feature simplifies the analysis process by allowing for easy resets to a clean state without needing to reinstall the OS.

3.6.5 Workflow with Snapshots

Analysts can use snapshots to manage multiple analysis sessions, branching out to investigate different malware samples without losing progress. This approach allows for efficient exploration of various scenarios by reverting to specific snapshots as needed.

3.6.6 Transferring Files from a VM

A limitation of using snapshots is the loss of data upon reverting to an earlier state. VMware offers solutions for saving work, such as drag-and-drop file transfers and shared folders between the host and guest OS, facilitating the preservation of analysis results and other important data.

3.7 The Risks of Using VMware for Malware Analysis

Like any software, VMware is susceptible to vulnerabilities that could potentially compromise the host operating system. These vulnerabilities could lead to system crashes or allow malicious code execution on the host OS. Past issues have been identified in features like shared folders and the drag-and-drop functionality, highlighting the importance of keeping VMware updated with the latest patches.

Inherent Risks in Malware Analysis: Conducting malware analysis, even within the presumed safety of a VM, always carries some level of risk. Despite taking extensive precautions, the possibility of unintended consequences remains. Analysts are advised against performing malware analysis on machines that are critical to personal or organizational operations or that contain sensitive information, to minimize potential damage.

3.8 Record/Replay: Running Your Computer in Reverse

VMware Workstation's record/replay feature lets you record everything that happens in a virtual machine and replay it with perfect accuracy. This means every action and instruction is replayed exactly as it happened, even rare events that are hard to recreate. Different from just recording video, this feature lets you actually run the recorded actions, pause, and interact during the replay. This is especially useful for going back to moments before errors were made when you can't simply undo them.

3.9 Conclusion

Procedure
Start with a clean snapshot of the virtual machine.
Transfer the malware specimen to the virtual machine.
Conduct detailed analysis within the virtual environment.
Document findings, take screenshots, and gather data, then transfer these to the physical machine.
Revert the virtual machine back to its clean snapshot to eliminate any trace of the malware.

Maintaining the Analysis Environment: As malware analysis tools evolve, it's necessary to update the virtual machine's base image with the latest tools and updates, followed by creating a new clean snapshot. This ensures that the analysis environment remains current and effective for analyzing new malware threats.

Safety and Cleanliness in Malware Analysis: Running malware poses risks of infecting the analyst's computer and network. VMware provides a secure environment to mitigate these risks, enabling safe execution of malware for analysis purposes. The book emphasizes the importance of conducting malware analysis within a virtual machine to maintain safety and control throughout the analysis process.

4 Basic Dynamic Analysis

Basic dynamic analysis is a crucial step in the malware analysis process that follows basic static analysis, especially when static methods hit a limit due to obfuscation or other complexities.

- Dynamic analysis involves observing malware in action or inspecting the system post-execution, offering insights into the malware's actual behavior that static analysis can't provide.
- This approach is particularly effective for identifying specific functionalities of malware, such as locating a keylogger's log files, understanding its record-keeping, and determining its data transmission methods.
- However, dynamic analysis is recommended only after completing static analysis due to potential risks to the network and system. Limitations exist, such as not all code paths executing during analysis, which might require more advanced techniques to thoroughly examine the malware's capabilities.

4.1 Sandboxes: The Quick-and-Dirty Approach

Sandboxes offer a streamlined approach for conducting basic dynamic malware analysis, utilizing virtual environments to safely execute and analyze untrusted programs without risking real systems. They mimic network services to allow normal functioning of the analyzed software, making them particularly useful for initial assessments.

However, using sandboxes involves certain drawbacks, such as limitations in executing malware that requires specific conditions or command-line arguments, and potential detection by the malware that it's running in a virtual environment. Despite these challenges, sandboxes remain valuable for quick analyses but cannot replace the depth of insight gained from more comprehensive analysis methods.

4.2 Running Malware

Basic dynamic analysis requires the ability to run malware, typically EXEs and DLLs, with EXEs being straightforward to execute, but DLLs presenting challenges since Windows doesn't automatically run them.

1. **Launching DLLs with rundll32.exe:**
   (a) Syntax: Use rundll32.exe DLLname, Export arguments to run a DLL, where Export is a function name or ordinal from the DLL's export table.
   (b) Tools: PEview or PE Explorer can be used to view the DLL's export table.
   (c) Example: To launch rip.dll which has an Install export function, use rundll32.exe rip.dll, Install.
   (d) Ordinal Exports: Functions exported by ordinal can be called with rundll32.exe using syntax like `rundll32.exe xyzzy.dll, #5`, where 5 is the ordinal number.
2. **Modifying the PE Header to Run DLLs as Executables:**
   (a) Method: Alter the PE header by removing the `IMAGE_FILE_DLL` flag from the Characteristics field, allowing Windows to load the DLL as an executable.

(b) Limitations: This may not run imported functions but will execute DLLMain. It might cause the malware to crash but can still provide valuable analysis data.

3. **Installing DLL Malware as a Service:**

   (a) Example: Use `rundll32 ipr32x.dll,InstallService ServiceName` and `net start ServiceName` to install and start a service, providing the necessary ServiceName argument.

   (b) Manual Installation: If no convenient export function like InstallService is available, manually install the service using the Windows sc command or by modifying the registry at `HKLM/SYSTEM/CurrentControlSet/Services`, then start it with net start.

4.3 Monitoring with Process Monitor

Process Monitor (Procmon) is an advanced tool for monitoring registry, file system, network, process, and thread activity on Windows. Does not capture device driver activity, certain GUI calls, or consistently log network activity.

- Monitors system calls as soon as it runs, capturing a vast amount of data.
- Can quickly consume available memory due to logging numerous events; advisable to run for short periods.
- To prevent memory overload, stop capturing by selecting File > Capture Events and clear irrelevant data via Edit > Clear Display before analysis.
- Analyzing Procmon data requires patience and practice due to the sheer volume of standard system activities recorded.

The **Procmon Display** shows detailed information about events, including sequence number, timestamp, process name, operation, path, and result. We can also use **filtering**, which is essential for managing large volumes of data and focusing on specific activities. Filters include Process Name, Operation, and Detail, among others.

4.4 Viewing Processes with Process Explorer

Process Explorer Overview is a powerful, Microsoft-provided task manager for dynamic analysis. It lists active processes, DLLs, process properties, and system information. It also allows killing processes, logging out users, and process validation.

- Process Explorer should be running during dynamic analysis for valuable insights.
- It can crash a VM due to memory consumption from logging many events; use for limited times.
- Filtering and examining specific processes or properties helps in focusing the analysis on relevant details.
- Regularly updating Process Explorer and using it alongside other analysis tools enhances malware detection and analysis capabilities.

4.5 Comparing Registry Snapshots with Regshot

Regshot Overview is an open-source tool for comparing registry snapshots before and after malware execution. It's useful for identifying registry changes made by malware.

```
Regshot
Comments:
Datetime: <date>
Computer: MALWAREANALYSIS
Username: username

----------------------------------
Keys added: 0
----------------------------------

----------------------------------
Values added:3
----------------------------------
(1) HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\ckr:C:\WINDOWS\system32\
  ckr.exe
  ...
  ...
----------------------------------
Values modified:2
----------------------------------
(2) HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed: 00 43 7C 25 9C 68 DE 59 C6 C8
  9D C3 1D E6 DC 87 1C 3A C4 E4 D9 0A B1 BA C1 FB 80 EB 83 25 74 C4 C5 E2 2F
  4E E8 AC C8 49 E8 E8 10 3F 13 F6 A1 72 92 28 8A 01 3A 16 52 86 36 12 3C C7
  5F 99 19 1D 80 8C 8E BD 58 3A DB 18 06 3D 14 8F 22 A4
  ...

----------------------------------
Total changes:5
----------------------------------
```

Registry snapshots were compared before and after running spyware `ckr.exe`. As you can see ckr.exe creates a value at:

`HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run`

as a persistence mechanism (1). A certain amount of noise (2) is typical in these results, because the random-number generator seed is constantly updated in the registry. As with procmon, your analysis of these results requires patient scanning to find nuggets of interest.

4.6 Faking a Network

Faking a Network for Malware Analysis is essential for identifying DNS names, IP addresses, and packet signatures without real internet connectivity. It requires preventing malware from detecting it's in a virtual environment for effectiveness.

4.6.1 Using ApateDNS

ApateDNS is a tool by Mandiant that spoofs DNS responses, redirecting malware DNS requests to a specified IP address.

- Listens on UDP port 53 and can show DNS requests made by malware, offering insights into its network behavior.
- To use, set a response IP address, start the server, and run the malware to observe DNS requests.
- Features include redirecting to localhost or an external IP, like a fake web server, and utilizing the NXDOMAIN option to reveal additional domains malware may query.

4.6.2 Monitoring with Netcat

- Versatile tool used for listening to inbound connections, port scanning, tunneling, and more.
- In listen mode, Netcat can act as a server to catch malware communications, especially useful for analyzing malware that communicates over common ports like 80 (HTTP) or 443 (HTTPS).
- Example shows listening on port 80 to intercept and analyze malware communications, revealing attempts to disguise traffic as legitimate HTTP POST requests.

4.7 Packet Sniffing with Wireshark

Wireshark is an open-source packet capture tool for intercepting and logging network traffic, offering detailed analysis capabilities. It's used for analyzing network issues, studying protocols, and can also be misused for unethical purposes like sniffing passwords.

- **Safety Warning**: Wireshark has known security vulnerabilities; it should be operated in a secure environment to avoid risks.
- Wireshark is invaluable for understanding malware's network behaviors by capturing its communication packets. For analysis, it can be paired with an internet connection or simulated network environment, such as using Netcat for internet simulation.

4.8 Using INetSim

INetSim is a Linux-based simulation tool designed for analyzing malware's network behavior by emulating various Internet services on a virtual network. It's an effective solution for creating a controlled environment to observe how malware interacts with network services without exposing real systems or networks to risk. INetSim supports a wide range of services including HTTP, HTTPS, FTP, IRC, DNS, and SMTP, making it versatile for different types of network analysis.

Its capability to mimic real servers, serve any requested file to keep malware operational, and log all inbound requests helps in understanding malware's network communication strategies. Additionally, its high configurability and unique features like the Dummy service for capturing all client-sent traffic to unbound ports make INetSim particularly valuable for comprehensive malware analysis tasks.

4.9 Basic Dynamic Tools in Practice

An **Integrated Malware Analysis Setup** utilizes a combination of tools for comprehensive dynamic analysis, including procmon, Process Explorer, Regshot, INetSim, ApateDNS, and Wireshark.

- Involves running procmon with specific filters, starting Process Explorer, taking registry snapshots with Regshot, and configuring a virtual network with INetSim and ApateDNS.

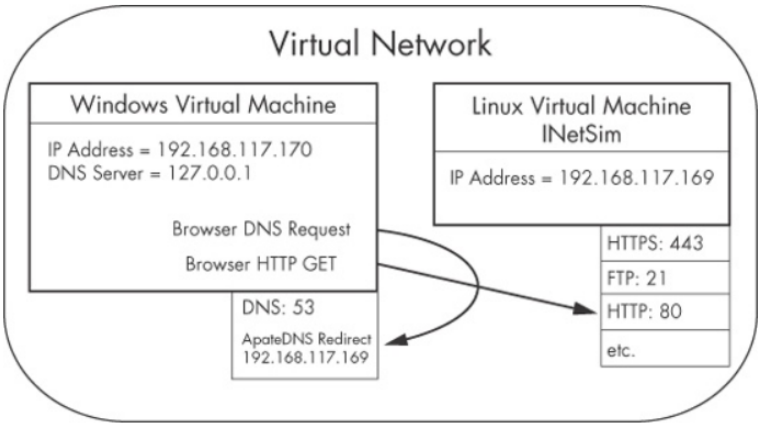

Figure 2: Example of a virtual network

4.10 Conclusion

Basic dynamic analysis of malware can assist and confirm your basic static analysis findings. Most of the tools described in this chapter are free and easy to use, and they provide considerable detail.However, basic dynamic analysis techniques have their deficiencies, so we won't stop here. For example, to understand the networking component in the `msts.exe` fully, you would need to reverse-engineer the protocol to determine how best to continue your analysis. The next step is to perform advanced static analysis techniques with disassembly and dissection at the binary level, which is discussed in the next chapter.

Tool/Technique	Description
Dynamic Analysis	Involves observing malware in action or inspecting the system post-execution to provide insights into the malware's actual behavior.
Sandboxes	Utilize virtual environments to safely execute and analyze untrusted programs without risking real systems, effectively for initial assessments.
Running Malware	Essential for basic dynamic analysis, with methods for executing EXEs directly and launching DLLs using rundll32.exe or by modifying the PE header.
Process Monitor (Procmon)	Monitors registry, file system, network, process, and thread activity, capturing vast amounts of data for analysis.
Process Explorer	Offers detailed insights into active processes, DLLs, and system information, crucial for dynamic analysis.
Regshot	Compares registry snapshots before and after malware execution to identify changes made by the malware.
Faking a Network	Includes using ApateDNS to spoof DNS responses and Netcat for monitoring network interactions, essential for analyzing network behavior.
Wireshark	An open-source packet capture tool used for in-depth network traffic analysis, identifying malware's communication patterns.
INetSim	Simulates common Internet services on a Linux-based virtual machine, useful for observing malware's network interaction in a controlled environment.

5 Advanced Static Analysis. A Crash Course in x86 Disassembly

Basic static and dynamic malware analysis methods are useful for initial assessments. These methods, however, do not offer comprehensive insights necessary for full malware analysis.

Limitations of Basic Analysis Techniques

1. Basic Static Analysis:
   (a) Comparable to an external examination in an autopsy.
   (b) Offers preliminary conclusions, such as identifying imported functions.
   (c) Does not reveal how or if these functions are utilized within the malware.
2. Basic Dynamic Analysis:
   (a) Can show malware's response to specific stimuli (e.g., receiving a specially crafted packet).
   (b) Falls short in providing detailed insights, such as the packet's format or deeper malware functionalities.

The Role of Disassembly in Malware Analysis. Disassembly is introduced as a crucial step for deeper analysis. It bridges the gap left by the basic analysis methods, offering a pathway to understanding the inner workings of malware.

5.1 Levels of Abstraction

- Abstraction hides implementation details.
- Malware authors often write their programs in high-level languages, which are then compiled into machine code.
- Malware analysts and reverse engineers work primarily with low-level languages, utilizing disassembly to understand and analyze the operations of a program.
- Disassembly is critical for translating machine code back into a form that is analyzable by humans, bridging the gap between the complex, binary nature of machine instructions and the need for comprehensible analysis.

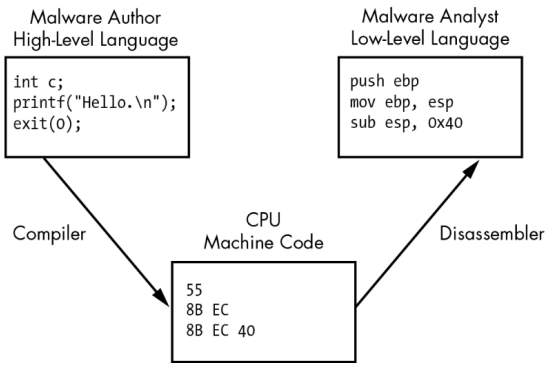

Figure 3: Code Level Examples

5.2 Reverse-Engineering

When malware is stored on a disk, it is typically in binary form at the machine code level. As discussed, machine code is the form of code that the computer can run quickly and efficiently. When we disassemble malware, we take the malware binary as input and generate assembly language code as output, usually with a disassembler.

- Assembly language is actually a class of languages. Each assembly dialect is typically used to program a single family of microprocessors, such as x86, x64, SPARC.
- x86 is by far the most popular architecture for PCs.
- Most 32-bit personal computers are x86, also known as Intel IA-32.

5.3 The x86 Architecture

The x86 Architecture follows the Von Neumann architecture with three main components:

1. **CPU** (Central Processing Unit): Executes code.
    - *Control Unit*: Retrieves instructions from RAM using the instruction pointer register, which holds the address of the next instruction.
    - *Registers*: Serve as the CPU's basic data storage units, reducing the need to access RAM.
    - *Arithmetic Logic Unit* (ALU): Executes instructions fetched from RAM, outputting results to registers or memory.
2. **Main Memory (RAM)**: Stores all data and code. Divided into four major sections:
    (a) *Data Section*: Holds static or global values.
    (b) *Code Section*: Contains executable instructions.
    (c) *Heap*: Used for dynamic memory allocation during program execution.
    (d) *Stack*: Manages local variables, function parameters, and program flow.
3. **I/O System**: Interfaces with external devices like hard drives and keyboards.

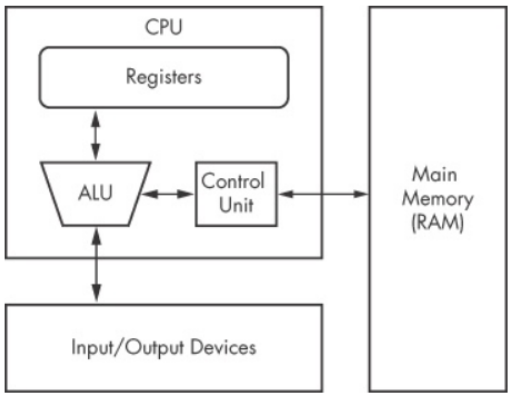

Figure 4: x86 Architecture

Component	Description
Instructions	Assembly Instructions are made from mnemonics and operands, translating into opcodes for CPU execution. Opcodes dictate CPU operations, with x86 using little-endian for data storage.
Registers	Categorized into general, segment, status flags, and instruction pointers, each serving specific execution purposes.
Flags	The EFLAGS register holds flags such as Zero Flag and Carry Flag to control operations or indicate results.
EIP	The Instruction Pointer, critical for program flow, indicates the next instruction to execute.
Simple Instructions	Includes commands like mov for data transfer and lea for loading effective addresses.
Arithmetic Instructions	Encompasses basic arithmetic, logical operations, and data shifting/rotation.
The Stack	A LIFO structure used for function calls, local variables, and control flow, with x86 built-in support.
Function Calls	Describes the process of function calling, stack frame setup, and return, emphasizing stack use for temporary storage and flow control.
Conditionals	Utilizes conditional instructions such as test and cmp for decision-making, and jump instructions for flow control.
Rep Instructions	Manipulates data buffers for operations like data copying, comparison, and initialization.
C Main Method	Details translation of C main method arguments (argc, argv) into assembly, showing parameter and offset handling.

Table 3: Summary of x86 Architecture Components

6 Recognizing C Code Constructs in Assembly

Successful reverse engineers focus on understanding code at a high level rather than evaluating each instruction individually due to the tedious nature and sheer volume of instructions in disassembled programs.

Note. Malware is often developed using high-level languages, predominantly C, which emphasizes the importance of recognizing code constructs such as loops, if statements, linked lists, and switch statements to understand the program's functionality.

6.1 Global vs. Local Variables

Variable Type	**Characteristics**	**Assembly Representation**
Global Variables	Accessible by any function within the program. Declared outside any function.	Referenced by specific memory addresses. Changes impact all functions that use the variable.
Local Variables	Accessible only within the function they are defined. Declared within a specific function.	Referenced by stack addresses, relative to `ebp`. Specific to the function, isolated impact.

Table 4: Comparison of Global and Local Variables in C

6.2 Disassembling Arithmetic Operations

Many different types of math operations can be performed in C programming. In this example, we have the C code on the left and their Assembly code on the right.

```
int a = 0;
int b = 1;
a = a + 11;
a = a - b;
a--;
b++;
b = a % 3;
```

```
00401006   mov     [ebp+var_4], 0
0040100D   mov     [ebp+var_8], 1
00401014   mov     eax, [ebp+var_4] 1
00401017   add     eax, 0Bh
0040101A   mov     [ebp+var_4], eax
0040101D   mov     ecx, [ebp+var_4]}
00401020   sub     ecx, [ebp+var_8] 2
00401023   mov     [ebp+var_4], ecx
00401026   mov     edx, [ebp+var_4]
00401029   sub     edx, 1 3
0040102C   mov     [ebp+var_4], edx
0040102F   mov     eax, [ebp+var_8]
00401032   add     eax, 1 4
00401035   mov     [ebp+var_8], eax
00401038   mov     eax, [ebp+var_4]
0040103B   cdq
0040103C   mov     ecx, 3
00401041   idiv    ecx
00401043   mov     [ebp+var_8], edx
```

Variables `a` and `b` are initialized to 0 and 1, respectively. IDA Pro labels these as `var_4` (for a) and `var_8` (for b). The variable `a` is moved into `eax` (1), and then `0x0b` is added to `eax`, thereby incrementing `a` by 11. The variable `b` is then subtracted from `a` (2). The compiler decided to use the sub and add instructions (3) and (4), instead of the inc and dec functions.

The final five assembly instructions implement the modulo. When performing the div or idiv instruction (5), you are dividing `edx:eax` by the operand and storing the result in `eax` and the remainder in `edx`. That is why `edx` is moved into `var_8` (5).

6.3 Recognizing if Statements

Programmers use if statements to alter program execution based on certain conditions. if statements are common in C code and disassembly.

```
int x = 1;
int y = 2;

if(x == y){
      printf("x is y");
} else {
      printf("x not y");
}
```

```
00401006     mov      [ebp+var_8], 1
0040100D     mov      [ebp+var_4], 2
00401014     mov      eax, [ebp+var_8]
00401017     cmp      eax, [ebp+var_4]  1
0040101A     jnz      short loc_40102B  2
0040101C     push     offset aXEqualsY_; "x␣is␣y."
00401021     call     printf
00401026     add      esp, 4
00401029     jmp      short loc_401038  3
0040102B loc_40102B:
0040102B     push     offset aXIsNotEqualToY; "x␣not␣y.
00401030     call     printf
```

Construct	Assembly Characteristics
Simple `if` Statement	Characterized by a conditional jump (e.g., `jnz`) based on a prior comparison (e.g., `cmp`). This jump decides whether to execute the `if` block or skip to the `else` or end of the statement.
Nested `if` Statements	Involves multiple conditional jumps corresponding to each level of nesting. Each `if` or `else if` condition is checked via a `cmp` followed by a conditional jump, leading to complex paths in the assembly.

Table 5: Recognizing `if` and Nested `if` Statements in Assembly

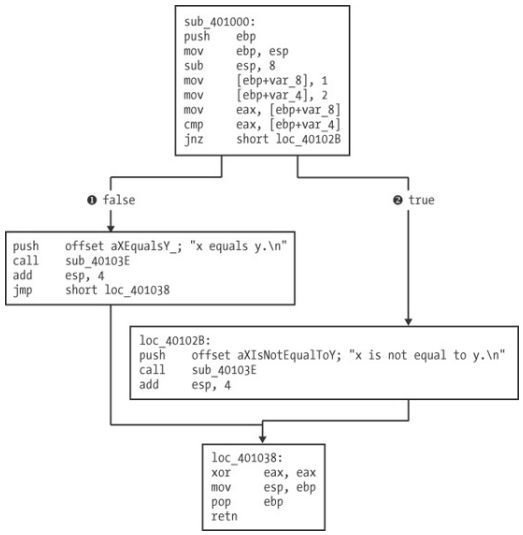

Figure 5: Disassembly graph for the if statement example

6.4 Recognizing Loops

Loop Type	Characteristics in C	Assembly Recognition
`for` Loop	Consists of initialization, comparison, execution, and increment/decrement.	Identified by steps for initialization, comparison, execution instructions, and increment/decrement. Look for a conditional jump after comparison and an unconditional jump that loops back.
`while` Loop	Executes as long as a condition remains true. Lacks an explicit increment/decrement section in its definition.	Similar to `for` loop but usually lacks an increment section. Identified by a conditional jump that exits the loop if the condition is not met and an unconditional jump that maintains the loop.

Table 6: Recognizing Loops in Assembly: `for` and `while` Loops

6.5 Understanding Function Call Conventions

The calling convention used depends on the compiler, among other factors. There are often subtle differences in how compilers implement these conventions, so it can be difficult to interface code that is compiled by different compilers.

Convention	Parameter Placement and Stack Cleanup	Return Value	Compiler Variations
cdecl	Parameters pushed right to left; caller cleans stack.	Return value in EAX.	Common in GCC; parameters pushed onto stack.
stdcall	Similar to cdecl, but callee cleans stack.	Return value in EAX.	Standard for Windows API; cleanup by the called function.
fastcall	First two arguments in EDX and ECX; caller may clean stack.	Return value in EAX.	Varies by compiler; more efficient with less stack use.

Table 7: Function Call Conventions and Their Assembly Representations

Visual Studio Version	GCC Version
Push parameters onto stack before call. Stack pointer restored after call.	Move parameters onto stack without altering stack pointer.

Table 8: Calling Conventions in Visual Studio vs. GCC

6.6 Analyzing Switch Statements

The analysis of switch statements in assembly highlights two primary compilation strategies: the **if style** and the use of **jump tables**. Both methods serve to implement switch statements in C, allowing for execution flow to branch based on the value of a variable, but they differ significantly in their assembly representations.

- **If Style**: Suitable for switch statements with fewer or non-contiguous cases. Results in multiple conditional jump instructions, each associated with a case.
- **Jump Table**: Optimal for switch statements with many contiguous cases. Simplifies the decision-making process to a single indexed jump, reducing the overhead of multiple comparisons.
  - A jump table is an array of pointers to the code blocks for each case. This method significantly reduces the number of comparisons needed. Instead of comparing the variable with each case value, the variable's value is used as an index to jump directly to the corresponding code block.

6.7 Disassembling Arrays

Arrays, whether local or global, are represented in assembly by their base addresses and are accessed through indexed addressing modes. The indexing reflects the size of the array elements and their order within the array.

- **Local Arrays**: Defined within a function, such as `a[i]` in the provided example. Their base address is relative to the frame pointer (`ebp` in x86 architecture), and they are typically found on the stack.

- **Global Arrays**: Declared outside any function scope, such as `b[i]` in the example. They are stored in a data segment with a fixed memory address accessible throughout the program.

6.8 Identifying Structs

Structures (structs) in C programming allow for the grouping of variables of different types under a single name, facilitating organized data management. Structures (like arrays) are accessed with a base address used as a starting pointer. It is difficult to determine whether nearby data types are part of the same struct or whether they just happen to be next to each other.

- **Global Variables**: Structs can be declared globally, allowing them to be accessed from anywhere within the program.
- **Memory Allocation**: Memory for structs is dynamically allocated using functions like `malloc`, and the base address of the allocated memory is used to access struct members.

By analyzing the offsets and types of data accessed or manipulated in the disassembled code, one can infer the structure of the original struct. For example, accessing an offset with floating-point instructions suggests the presence of a double type member.

```
struct my_structure {  1
     int x[5];
     char y;
     double z;
};

struct my_structure *gms;  2

void test(struct my_structure *q)
{
     int i;
     q->y = 'a';
     q->z = 15.6;
     for(i = 0; i<5; i++){
           q->x[i] = i;
     }
}

void main()
{
     gms = (struct my_structure *) malloc(
     sizeof(struct my_structure));
     test(gms);
}
```

```
00401050        push    ebp
00401051        mov     ebp, esp
00401053        push    20h
00401055        call    malloc
0040105A        add     esp, 4
0040105D        mov     dword_40EA30, eax
00401062        mov     eax, dword_40EA30
00401067        push    eax  3
00401068        call    sub_401000
0040106D        add     esp, 4
00401070        xor     eax, eax
00401072        pop     ebp
00401073        retn
```

In the C code, we define a structure at (1) made up of an integer array, a character, and a double. In main, we allocate memory for the structure and pass the struct to the test function. The struct gms defined at (2) is a global variable. The base address of this structure is passed to the `sub_401000` (test) function via the push eax at (3).

6.9 Analyzing Linked List Traversal

A linked list is a data structure that consists of a sequence of data records, and each record includes a field that contains a reference (link) to the next record in the sequence. The principal benefit of using a linked list over an array is that the order of the linked items can differ from the order in which the data items are stored in memory or on disk. Therefore, linked lists allow the insertion and removal of nodes at any point in the list.

To recognize a linked list, you must first recognize that some object contains a pointer that points to another object of the same type.

```
struct node
{
   int x;
   struct node * next;
};

typedef struct node pnode;

void main()
{
   pnode * curr, * head;
   int i;

   head = NULL;

   for(i=1;i<=10;i++)
   {
      curr = (pnode *)malloc(sizeof(pnode));
      curr->x = i;
      curr->next  = head;
      head = curr;
   }

   curr = head;

   while(curr)
   {
      printf("%d\n", curr->x);
      curr = curr->next ;
   }
}
```

7 Types of Malware

When performing malware analysis, you will find that you can often speed up your analysis by making educated guesses about what the malware is trying to do and then confirming those hypotheses. To that end, here are the categories that most malware falls into:

Type of Malware	**Description**
Backdoor	Malicious code that installs itself onto a computer to allow the attacker access, enabling command execution with little or no authentication.
Botnet	Similar to a backdoor, allows the attacker access to the system, but infected computers receive instructions from a single command-and-control server.
Downloader	Malicious code that downloads other malicious code, commonly installed by attackers to bring additional malware onto a system.
Information-stealing malware	Collects information from the victim's computer, such as sniffers, password hash grabbers, and keyloggers, typically for accessing online accounts.
Launcher	Used to launch other malicious programs, often employing nontraditional techniques for stealth or greater system access.
Rootkit	Designed to conceal other malicious code, making it difficult for the victim to detect and often paired with backdoors for remote access.
Scareware	Frightens users into buying useless software by pretending to be a security program and claiming to detect malware on the system.
Spam-sending malware	Infects machines to send spam, generating income for attackers by offering spam-sending services.
Worm or virus	Can copy itself and infect additional computers, sometimes combining functionalities like keylogging and spam-sending.

Table 9: Types of Malware and Their Descriptions

Note. Malware often spans multiple categories. For example, a program might have a keylogger that collects passwords and a worm component that sends spam.

Note. Malware can also be classified based on whether the attacker's objective is mass or targeted. **Mass malware**, such as scareware, takes the shotgun approach and is designed to affect as many machines as possible. Of the two objectives, it's the most common, and is usually the less sophisticated and easier to detect and defend against because security software targets it. **Targeted malware** is usually very sophisticated and because it is not widespread, your security products probably won't protect you from it.