Introduction to Malware Cheat Sheet

Malware Definition and Types:

- **Malware (Malicious Software):** Software designed to cause harm to a computer system, network, or its users without their consent.
- **Types:**
  - **Viruses:** Require a host program to spread and replicate.
  - **Worms:** Self-replicating and can spread autonomously across networks.
  - **Trojans:** Disguise themselves as legitimate software to gain access.
  - **Ransomware:** Encrypts files and demands a ransom for their decryption.
  - **Spyware:** Secretly monitors user activity and collects information.
  - **Adware:** Displays unwanted advertisements.
  - **Rootkits:** Conceal malicious processes and files from detection.
  - **Keyloggers:** Record keystrokes.
  - **Backdoors:** Provide unauthorized remote access.
  - **Bots:** Software robots controlled remotely, often in botnets.

Malware Analysis:

- The process of examining malware samples to understand their functionality, behavior, origin, and potential impact.

Forensic Importance of Malware Analysis:

- **Incident Response:** Understanding how malware breached a system and its actions helps in containment, eradication, and recovery.
- **Attribution:** Identifying the threat actor or group behind the malware.
- **Evidence Collection:** Malware and its artifacts can serve as crucial digital evidence.
- **Vulnerability Assessment:** Analyzing exploited vulnerabilities can improve security posture.
- **Threat Intelligence:** Contributing to the understanding of emerging threats.

Introduction to Different Analysis Techniques:

- **Static Analysis:** Examining the malware code and structure without executing it.

- **Dynamic Analysis:** Executing the malware in a controlled environment to observe its behavior.
- **Hybrid Analysis:** Combining static and dynamic analysis techniques.

Malware Behavior:

- **Persistence Mechanisms:** How malware ensures it runs after a system reboot (e.g., registry keys, startup folders, scheduled tasks).
- **Communication:** How malware communicates with command and control (C2) servers.
- **Data Exfiltration:** How malware steals sensitive information.
- **Lateral Movement:** How malware spreads within a network.
- **System Modification:** Changes made to the operating system or installed applications.
- **Defense Evasion:** Techniques used to avoid detection by security software.

Setting up Malware Analysis Laboratory:

- **Isolated Network:** Preventing malware from spreading to production systems.
- **Virtual Machines (VMs):** Providing a safe and easily restorable environment for execution.
- **Snapshotting Capabilities:** Allowing rollback to a clean state after analysis.
- **Analysis Tools:** Installing necessary software for static and dynamic analysis.
- **Internet Simulation:** Controlled network environment to observe network communication.

Static Analysis:

- **Hashing:** Calculating cryptographic hashes (MD5, SHA-1, SHA-256) to uniquely identify malware samples.
- **Finding Strings:** Extracting human-readable text from the malware binary, which can reveal URLs, IP addresses, function names, and other indicators.
- **Decoding Obfuscated Strings Using FLOSS (FireEye Labs Obfuscated String Solver):** Automatically identifying and decoding obfuscated strings.
- **PE Files Headers and Sections (Portable Executable):** Understanding the structure of Windows executable files:
  - **Headers:** Contain metadata about the file (e.g., entry point, size, timestamps).
  - **Sections:** Contain different parts of the program code and data (.text, .data, .rdata, .rsrc, etc.).
- **PE View:** A tool for examining the PE file structure.
- **Linked Libraries and Functions:** Identifying external libraries and functions the malware uses, revealing its capabilities.
- **Dependency Walker:** Visualizes the dependency tree of Windows modules.

- **CFF Explorer (PE Editor):** A more advanced tool for examining and modifying PE files.
- **Resource Hacker:** Allows viewing and modifying resources within PE files (e.g., icons, dialogs, strings).
- **Malware Signature and Clam AV Virus Signature:** Understanding how antivirus software identifies malware based on unique patterns or sequences of bytes.
- **YARA Signatures:** Rule-based language for creating patterns to identify malware families based on textual or binary characteristics.

Dynamic Analysis:

- **Sandboxes:** Isolated environments that allow safe execution and monitoring of malware behavior (e.g., Cuckoo Sandbox, Any.Run).
- **Running and Monitoring a Malware:** Executing the malware within a sandbox or controlled VM and observing its actions.
- **Process Monitor (ProcMon):** Windows Sysinternals tool for monitoring real-time file system, registry, and process activity.
- **Process Explorer (ProcExp):** Windows Sysinternals tool providing detailed information about running processes.
- **RegShot:** Compares registry snapshots before and after malware execution to identify changes.
- **Faking a Network (INetSim):** Simulating network services to observe how malware interacts with the internet.
- **Using Wireshark for Packet Analysis:** Capturing and analyzing network traffic generated by the malware to understand its communication patterns.

Assembly and Reverse Engineering Cheat Sheet

Introduction to x86 Assembly and CPU Registers:

- **x86 Assembly:** Low-level programming language representing machine code instructions for Intel x86 processors.
- **CPU Registers:** Small, high-speed storage locations within the CPU used to hold data and control information during program execution.
  - **General-Purpose Registers:**
    - `EAX` (Accumulator): Used for arithmetic operations and function return values.
    - `EBX` (Base): Often used as a base pointer for memory addressing.
    - `ECX` (Counter): Used as a loop counter.
    - `EDX` (Data): Used in I/O operations and with `EAX` for larger arithmetic.
    - **16-bit equivalents:** `AX` , `BX` , `CX` , `DX` .
    - **8-bit lower/upper halves:** `AL` , `AH` , `BL` , `BH` , `CL` , `CH` , `DL` , `DH` .
  - **Segment Registers:** Define memory segments (less common in modern flat memory model).
    - `CS` (Code Segment): Points to the segment containing instructions.
    - `DS` (Data Segment): Points to the segment containing data.
    - `SS` (Stack Segment): Points to the segment containing the stack.
    - `ES` , `FS` , `GS` (Extra Segment Registers): Additional segment pointers.
  - **Pointer Registers:**
    - `EIP` (Instruction Pointer): Holds the address of the next instruction to be executed. (`IP` in 16-bit).
    - `ESP` (Stack Pointer): Points to the top of the stack. (`SP` in 16-bit).
    - `EBP` (Base Pointer): Often used to reference local variables on the stack. (`BP` in 16-bit).
  - **Flags Register (`EFLAGS`):** Contains status flags (e.g., Zero Flag, Carry Flag, Sign Flag) and control flags.

Overview of the Stack:

- **LIFO (Last-In, First-Out) Data Structure:** Used for temporary storage of data during function calls.
- **Stack Pointer (`ESP`):** Points to the current top of the stack (lowest memory address in the stack segment).
- **Push Operation:** Decrements `ESP` and places data onto the stack.
- **Pop Operation:** Retrieves data from the stack and increments `ESP` .

- **Function Calls:**
  - Arguments are often pushed onto the stack before a call.
  - The return address (address of the instruction after the call) is pushed onto the stack.
  - Local variables are allocated on the stack within a function's stack frame.
- **Stack Frames:** Dedicated area on the stack for each function call, containing arguments, return address, local variables, and saved registers.

IDA Pro with its Functions and Features:

- **Disassembler:** Converts machine code into human-readable assembly language.
- **Decompiler (F5):** Attempts to convert assembly code into higher-level pseudo-C code, aiding in understanding program logic.
- **Interactive Disassembly:** Allows renaming variables and functions, adding comments, defining data types, and creating cross-references.
- **Graph View:** Visual representation of code flow, making it easier to understand program structure.
- **Debugger:** Allows step-by-step execution of the program, examining registers and memory.
- **Plugins and Scripts (IDC, Python):** Extend IDA's functionality for specialized tasks.
- **Signature Analysis (FLIRT):** Identifies known library functions, simplifying analysis.
- **Hex View:** Allows direct examination and modification of the raw binary data.
- **Cross-References:** Shows where variables, functions, and code locations are used.

Understanding of C Code Construct in Assembly:

- **Variables:** Local variables are typically allocated on the stack. Global variables reside in the data segment.
- **Control Flow:**
  - `if/else` : Translated into conditional jump instructions (`je` , `jne` , `jg` , `jl` , etc.).
  - `for/while` **loops:** Implemented using comparison and jump instructions to repeat code blocks.
  - `switch` **statements:** Often compiled into jump tables for efficient branching.
- **Functions:**
  - Arguments passed via registers or the stack.
  - Return address pushed onto the stack.
  - Stack frame setup (`push ebp` , `mov ebp, esp`).
  - Local variables allocated by adjusting the stack pointer (`sub esp, <size>`).
  - Return value placed in `EAX` .
  - Stack frame teardown (`mov esp, ebp` , `pop ebp` , `ret`).

- **Pointers:** Represented as memory addresses. Dereferencing a pointer involves accessing the data at that address.
- **Arrays:** Contiguous blocks of memory. Accessing elements involves calculating offsets.
- **Structures:** Contiguous blocks of memory where members are accessed at specific offsets.

Analyzing Malicious Windows Programs:

- **Identifying Entry Point:** The first instruction executed when the program runs (often found in the PE header).
- **Tracing API Calls:** Monitoring calls to Windows API functions to understand the program's interactions with the OS (e.g., file operations, network communication, registry modifications).
- **Analyzing Control Flow:** Understanding the sequence of instructions and how the program makes decisions.
- **Identifying Malicious Behavior:** Looking for patterns associated with malware (e.g., network connections to suspicious IPs, file encryption routines, persistence mechanisms).
- **Unpacking:** Dealing with packed or obfuscated malware that hides its true code.
- **Reverse Engineering Algorithms:** Understanding how malware implements its malicious functionality.

Live Memory Analysis using Volatility:

- **Volatility:** An open-source memory forensics framework for extracting digital artifacts from volatile memory (RAM) dumps.
- **Profiles:** Volatility requires a profile that matches the operating system and architecture of the memory dump.
- **Key Plugins:**
    - `pslist` / `pstree` : Lists running processes.
    - `psscan` : Scans memory for process structures (can find hidden or terminated processes).
    - `dlllist` : Lists loaded DLLs for a process.
    - `handles` : Lists open handles for a process (files, registry keys, etc.).
    - `netscan` : Scans for network connections and listening ports.
    - `registry` : Allows interaction with the registry hives present in memory.
    - `filescan` : Scans for file objects in memory.
    - `procdump` : Dumps the executable image of a process from memory.
    - `malfind` : Identifies injected code or hidden modules.
    - `yarascan` : Scans memory for YARA signatures.
    - `apihooks` : Detects API hooking by malware.

- **Analyzing Artifacts:** Examining process information, network connections, loaded modules, and other memory artifacts to identify malicious activity that might not be visible on disk.

Debugging Cheat Sheet

Difference between Source Level vs. Assembly Level Debugger:

- **Source Level Debugger:**
  - Operates on the original source code (e.g., C++, Python, Java).
  - Allows stepping through code line by line in the source language.
  - Displays variables and data structures in their original types.
  - Easier to understand program flow at a higher level.
  - Requires access to the source code and debug symbols.
  - Examples: GDB (with source), Visual Studio Debugger, PyCharm Debugger.
- **Assembly Level Debugger:**
  - Operates directly on the disassembled machine code (assembly language).
  - Allows stepping through individual assembly instructions.
  - Displays CPU registers, memory contents, and flags.
  - Provides a low-level view of program execution.
  - Useful when source code is unavailable or when analyzing low-level behavior.
  - Examples: OllyDbg, Immunity Debugger, WinDbg.

Kernel Mode vs. User Mode Debugger:

- **User Mode Debugger:**
  - Debugs applications running in user mode, which has restricted access to system resources.
  - Cannot directly access kernel-level data structures or code.
  - Most common type of debugger for application development and malware analysis.
  - Examples: GDB, Visual Studio Debugger, OllyDbg, Immunity Debugger.
- **Kernel Mode Debugger:**
  - Debugs the operating system kernel and device drivers, which have privileged access to system resources.
  - Allows examining kernel data structures, tracing system calls, and analyzing kernel-level issues (e.g., BSODs, driver bugs).
  - Requires special setup and privileges.
  - Example: WinDbg (can also debug user mode).

Debugger Common Features:

- **Stepping:** Executing the program one instruction (assembly level) or one line (source level) at a time.
  - **Step Over:** Executes the current function call without stepping into its instructions.
  - **Step Into:** Steps into the instructions of the current function call.
  - **Step Out:** Executes the remaining instructions of the current function and returns to the caller.
- **Breakpoints:** Specific locations in the code where program execution will pause, allowing inspection of the program state.
  - **Software Breakpoints:** Inserted by the debugger by modifying the code (e.g., replacing an instruction with an interrupt instruction).
  - **Hardware Breakpoints:** Supported by the CPU, allowing breakpoints on code execution, data access, or data write without modifying the code. Limited number available.
  - **Conditional Breakpoints:** Break only when a specific condition is met (e.g., a variable reaches a certain value).
- **Watch Variables:** Monitoring the values of specific variables as the program executes.
- **Call Stack:** Shows the sequence of function calls that led to the current point of execution.
- **Memory View:** Allows examining the contents of memory at specific addresses.
- **Register View:** Displays the current values of CPU registers.
- **Disassembly View:** Shows the disassembled machine code (in assembly level debuggers).
- **Expression Evaluation:** Allows evaluating expressions involving variables and operators.

Breakpoints:

- **Purpose:** Pause program execution at a point of interest for inspection.
- **Types:** Software, Hardware, Conditional (as described above).
- **Setting Breakpoints:** Typically done by clicking in the code editor margin or using a debugger command.
- **Managing Breakpoints:** Enabling, disabling, deleting breakpoints.

Exceptions:

- **Definition:** Events that occur during program execution that disrupt the normal flow of instructions (e.g., division by zero, invalid memory access).
- **Debugger Handling:** Debuggers can be configured to break when specific exceptions occur, allowing analysis of the error state.
- **First-Chance Exceptions:** The debugger intercepts the exception before the program's exception handlers.

- **Second-Chance Exceptions:** The debugger intercepts the exception if the program's handlers don't handle it.
- **Ignoring Exceptions:** Debuggers can be configured to ignore certain exceptions.

Modification of Program Execution:

- **Changing Variable Values:** Altering the contents of variables during debugging to test different scenarios or bypass certain conditions.
- **Modifying Memory:** Directly changing values in memory.
- **Patching Instructions:** Replacing assembly instructions to alter program behavior (often used in reverse engineering and malware analysis).
- **Forcing Return Values:** Specifying the return value of a function before it actually returns.
- **Skipping Instructions:** Bypassing certain parts of the code.

Working with OllyDbg and Immunity Debugger:

- **OllyDbg (Windows, x86):** A user-mode assembly level debugger popular for malware analysis.
  - User-friendly interface with disassembly, registers, memory, and stack views.
  - Powerful breakpoint features (software, hardware, conditional).
  - Plugin support for extending functionality.
  - Tracing capabilities to record instruction execution.
- **Immunity Debugger (Windows, x86):** Another user-mode assembly level debugger, built on top of OllyDbg with a focus on exploit development.
  - Includes Python scripting capabilities for automation and custom commands (PyCommands).
  - Integrates well with exploit development tools.
  - Shares many core features with OllyDbg.

Kernel Debugging with WinDbg (Windows):

- **Purpose:** Debugging the Windows kernel and device drivers.
- **Setup:** Requires a separate debugging machine or a virtual machine setup with kernel debugging enabled. Communication is typically via serial cable, named pipes, or network.
- **Commands:** Uses a command-line interface with a wide range of commands for inspecting kernel data structures, tracing execution, setting breakpoints in kernel code, and analyzing crashes (BSODs).
- **Key Features:**
  - Examining kernel objects (processes, threads, drivers).

- Setting breakpoints in kernel-mode code and driver routines.
- Stepping through kernel-mode execution.
- Analyzing crash dumps (.dmp files) to diagnose BSODs.
- Inspecting memory in kernel space.
- Using symbols (.pdb files) to resolve kernel function and variable names.
- Extensibility through extensions.

Behaviours of Malware Cheat Sheet

Common Behavior of Malware:

- **Persistence:** Mechanisms to ensure the malware runs after system restarts (e.g., registry keys, startup folders, scheduled tasks).
- **Communication:** Establishing connections with Command and Control (C2) servers for instructions, data exfiltration, or updates.
- **Data Exfiltration:** Stealing sensitive information (credentials, personal data, financial information).
- **Lateral Movement:** Spreading to other systems within a network.
- **Privilege Escalation:** Gaining higher-level access to the system.
- **Defense Evasion:** Techniques to avoid detection and analysis by security software and analysts.
- **System Modification:** Altering system configurations, files, or functionality.
- **Resource Consumption:** Overloading system resources (CPU, memory, network) to cause denial of service or slow down the system.
- **Destructive Actions:** Corrupting or deleting files, rendering the system unusable.
- **Spreading/Replication:** Copying itself to other systems or files.

Process Injection:

- Injecting malicious code into the address space of a legitimate running process.
- Allows malware to hide its activity within a trusted process.
- Can bypass some security software that monitors specific malware processes.
- Techniques include:
  - **CreateRemoteThread:** Creating a new thread in the target process and running malicious code.
  - **WriteProcessMemory:** Writing malicious code into the target process's memory.
  - **Thread Hijacking:** Taking control of an existing thread in the target process.

Process Replacement (Process Hollowing/RunPE):

- Creating a new legitimate process in a suspended state.
- Unmapping the legitimate code from the process's memory.
- Writing malicious code into the process's memory.
- Resuming the process, causing it to execute the malicious code instead of the original legitimate code.

Hook Injection:

- Modifying the normal execution flow of a program by intercepting function calls (hooks).
- Malware can install hooks in various parts of the system (e.g., user-mode APIs, kernel-mode functions).
- Allows malware to monitor user input, intercept network traffic, or modify system behavior.
- Techniques involve altering function pointers or using API hooking frameworks.

Data Encoding:

- Transforming data into a different format to make it less readable or to bypass simple detection mechanisms.
- Common encoding schemes include Base64, hexadecimal, and simple XOR.
- Often used for C2 communication, storing configuration data, or hiding malicious payloads.
- Requires decoding to understand the actual data.

Anti-Disassembly:

- Techniques used to make it harder for analysts to disassemble and understand the malware's code.
- Examples:
  - **Junk Code:** Inserting irrelevant instructions to clutter the disassembly listing.
  - **Opaque Predicates:** Conditional jumps where the outcome is always the same, confusing disassemblers' control flow analysis.
  - **Self-Modifying Code:** Altering the malware's code during runtime, making static analysis difficult.
  - **Control Flow Obfuscation:** Making the program's execution path difficult to follow.

Anti-Debugging:

- Techniques used to detect and evade debugging environments, hindering dynamic analysis.
- Examples:
  - **API Checks:** Calling functions like `IsDebuggerPresent` or `CheckRemoteDebuggerPresent` .
  - **Timing Checks:** Measuring the time taken for certain operations, which might be longer under a debugger.
  - **Breakpoint Detection:** Checking for software breakpoints.
  - **Exception Handling:** Using exceptions to detect debugger presence.
  - **Hardware Breakpoint Detection:** Attempting to set more hardware breakpoints than available.

- **Emulator/Sandbox Detection:** Checking for artifacts or behaviors common in virtualized or sandboxed environments.

Anti-Virtual Machine Techniques:

- Methods used by malware to detect if it's running in a virtual machine and alter its behavior or terminate to avoid analysis.
- Examples:
  - **Registry Checks:** Looking for specific registry keys associated with virtualization software (e.g., VMware, VirtualBox).
  - **File Checks:** Checking for the presence of VM-specific files or directories.
  - **Process Checks:** Looking for VM-related processes.
  - **Hardware Checks:** Detecting virtualized hardware characteristics (e.g., MAC addresses, serial numbers).
  - **Timing Anomalies:** Detecting differences in instruction execution speed in VMs.
  - **Mouse/Keyboard Activity:** Checking for human interaction, which might be absent in automated analysis environments.

Packers and Unpacking:

- **Packers:** Software used to compress and/or encrypt executable files to reduce their size and hinder static analysis. The original code is typically hidden within the packed file.
- **Unpacking:** The process of reversing the packing mechanism to reveal the original malicious code. This often involves dynamic analysis to observe the malware unpacking itself in memory.
- **Common Packers:** UPX, PECompact, ASPack.
- **Unpacking Techniques:** Manual analysis using debuggers to find the original entry point (OEP) after unpacking, or using automated unpacking tools.

Other Platform Malware Cheat Sheet

Introduction to Linux Malwares:

- **Less Prevalent than Windows Malware:** Due to Linux's diverse distributions, permission model, and lower desktop market share.
- **Growing Threat:** Increasing use of Linux in servers, cloud environments, and IoT devices makes it a more attractive target.
- **Types:** ELF executables, shell scripts, rootkits, web server malware, ransomware targeting servers.
- **Focus:** Often targets server infrastructure for data theft, botnet recruitment, or disruption.

Linux Binary Architecture (ELF - Executable and Linkable Format):

- **Header:** Contains metadata about the file (e.g., entry point, program headers, section headers).
- **Program Headers:** Describe segments (loadable regions) of the file for execution.
- **Section Headers:** Describe sections containing code, data, symbols, etc. (used during linking and debugging).
- **Sections:** `.text` (executable code), `.data` (initialized data), `.bss` (uninitialized data), `.rodata` (read-only data), symbol table, relocation table, etc.
- **Dynamic Linking:** Relies on shared libraries (`.so` files).

Analysis of Linux Malware:

- **Static Analysis:**
  - **Hashing:** Identify known malware.
  - **Strings:** Look for readable text (URLs, commands, function names).
  - **File Format Analysis (ELF):** Use tools like `readelf` , `objdump` , `file` to examine headers, sections, and symbols.
  - **Dependencies:** Use `ldd` to list required libraries.
  - **Disassembly:** Use tools like `objdump -d` or IDA Pro (Linux support) to view assembly code (x86, x64, ARM).
  - **YARA Rules:** Create signatures to identify Linux malware families.
- **Dynamic Analysis:**
  - **Sandboxing:** Use tools like Cuckoo Sandbox (Linux support) or custom chroot environments.
  - **System Call Tracing:** Use `strace` to monitor system calls made by the malware.

- Network Analysis: Use `tcpdump` or Wireshark to capture network traffic.
- Process Monitoring: Use tools like `ps`, `top`, `htop`.
- Memory Analysis: Use tools like Volatility (Linux profiles) to examine memory dumps.

Android Architecture:

- **Linux Kernel:** Underlying operating system.
- **Hardware Abstraction Layer (HAL):** Interfaces between the kernel and hardware.
- **Android Runtime (ART/Dalvik):** Executes applications.
- **Libraries:** C/C++ libraries used by various Android components.
- **Application Framework:** Provides APIs for application development (Activity Manager, Content Providers, etc.).
- **Applications:** User-installed and system applications.

Android Permissions:

- **Mechanism for Restricting Access:** Applications must declare permissions they need to access sensitive resources or perform certain actions (e.g., internet access, reading contacts, sending SMS).
- **User Consent:** Users are typically prompted to grant permissions during installation or runtime (depending on the Android version and permission type).
- **Types of Permissions:** Normal, Dangerous, Signature, SignatureOrSystem. Dangerous permissions require explicit user consent.
- **Malware Exploitation:** Malware often requests excessive or unnecessary permissions to carry out malicious activities.

Types of Android Malware:

- **SMS Trojans:** Send premium SMS messages without user consent.
- **Spyware:** Steal personal information (contacts, SMS, call logs, location).
- **Banker Trojans:** Steal banking credentials.
- **Ransomware:** Encrypt files or lock devices and demand ransom.
- **Adware:** Display intrusive advertisements.
- **Rooting Malware:** Attempts to gain root privileges on the device.
- **Botnets:** Recruit devices into botnets for various malicious purposes.
- **Mobile Ransomware:** Locks the device or encrypts files, demanding payment.

...lysis and Reverse Engineering of Android Malware:

...cquisition: Obtaining the APK (Android Package Kit) file.

...tatic Analysis:

- **APK Disassembly:** Use tools like `apktool` to decompile the APK and access the Dalvik bytecode (smali).
- **Manifest Analysis (AndroidManifest.xml):** Examine requested permissions, components, entry points, and intent filters.
- **Code Review (Smali):** Analyze the disassembled code for malicious logic.
- **String Analysis:** Look for suspicious URLs, IP addresses, or keywords.
- **Resource Analysis:** Examine images, layouts, and other resources for indicators.
- **Signature Analysis:** Check the app's signature.
- **YARA Rules:** Create signatures for Android malware families.

...ynamic Analysis:

- **Emulators/Simulators:** Use Android emulators (e.g., Android Studio Emulator, Genymotion) for safe execution.
- **Real Devices (with caution):** Use isolated test devices.
- **Sandboxing:** Utilize Android sandbox environments.
- **Log Analysis (Logcat):** Monitor system logs for malicious activity.
- **Network Analysis (tcpdump, Wireshark):** Capture network traffic.
- **Behavioral Analysis:** Observe the app's actions during runtime.

...everse Engineering Tools:

- `apktool` : Decompiles and rebuilds APKs.
- `dex2jar` : Converts Dalvik bytecode (`.dex`) to Java bytecode (`.jar`).
- `jd-gui` , `Luyten` : Java bytecode decompilers.
- `AndroGuard` : Comprehensive Android reverse engineering and analysis tool.
- `Frida` : Dynamic instrumentation toolkit for injecting code and monitoring function calls.
- IDA Pro (with Dex plugin): Professional disassembler with Android support.