MALWARE ANALYSIS

Debugging

Dr. Zeyad Safaa Younus Saffawi

Debugger

- A **debugger** is a software (or sometimes hardware) tool used to test and examine the execution of another program.
- During software development, programs often contain errors. A debugger helps identify and fix these errors.
- Normally, you only see the **input and output**, but not how the program actually produces the result.
  - A debugger gives insight into what happens **inside the program while it's running**, such as memory values and register states.
  - **Disassembler,** which provides a **static snapshot** before execution,
  - **Debugger** offers a **dynamic view** of the program during execution.
  - It can show how **memory addresses change** as the program runs.
  - This is especially important in **malware analysis**, as it lets you monitor every **register, memory location, and function argument**.
  - You can even **modify variables** or **program behavior** in real time.
- Two common debugger tools are **OllyDbg** and **WinDbg**.

Debugger Tools

- **OllyDbg**
  - popular for binary code analysis when source code is unavailable.
  - Most popular for malware analysis
  - User-mode debugging only
  - IDA Pro has a built-in debugger, but it's not as easy to use or powerful as Ollydbg
- **Windbg**
  - WinDbg (Windows Debugger) is a more powerful tool developed by Microsoft that can operate in both user-mode and kernel-mode debugging .
  - This capability (user-mode and kernel-mode debugging)allows it to debug the Windows operating system itself (e.g., analyzing device drivers or blue screen of death memory dumps), which OllyDbg cannot do.

Source-Level vs. Assembly-Level Debuggers

- **Source-Level Debuggers**
  - Commonly used by software developers during coding.
  - Allow setting **breakpoints** on specific lines of source code.
  - Enable stepping through the program **line by line**.
  - Let you easily view and analyze variable values and program flow.
- **Assembly-Level (Low-Level) Debuggers**
  - Work directly with **assembly instructions** instead of high-level code.
  - Allow stepping through **each instruction**, setting breakpoints at specific assembly lines, and examining memory.
  - Do **not require access to source code**, which is why they are widely used in **malware analysis**.
- *In short:*
  - **Source-level** = for developers with source code.
  - **Assembly-level** = for analysts (e.g., malware) without source code.

Kernel vs. User-Mode Debugging

- **User-Mode Debugging**
  - The debugger and the program being debugged run on **the same system**.
  - You are debugging a **single executable**, isolated from others by the operating system.
  - Easier to perform and safer because it doesn't affect the whole system.
  - Common tools: **OllyDbg**, **IDA Pro (built-in debugger)**, and **WinDbg (user-mode)**.
- **Kernel-Mode Debugging**
  - More complex and powerful — used to debug **the operating system kernel** or drivers.
  - Typically requires **two systems because there is only one kernel per computer**:
    - One runs the **debugged code (target)**.
    - The other runs the **debugger (host)**.
  - Necessary because if the kernel stops at a breakpoint, the **entire system halts**.
  - The OS must be specially configured for kernel debugging and connected to the debugger machine.
  - **WinDbg** is the main tool supporting kernel debugging.
  - It *is possible* (though rare) to debug the kernel on the same system — a tool called **SoftICE** once provided this feature, but it was discontinued in **2007**.
- **User-mode** → Debugging applications, simple and isolated.
- **Kernel-mode** → Debugging the OS or drivers, complex and risky, requires two machines.

Kernel-Mode Debugging

- Windows 7 advanced boot options
  - Press F8 during startup
  - Debugging mode

Using a Debugger — Key Concepts

Starting vs attaching

- **Start under debugger**: is a method used to run a program with a debugger from the very beginning, allowing developers to pause execution on the first line of code, inspect variables, and trace the program logic step by step from the beginning to find errors
  - This gives the developer full control from the first instruction, allowing inspection of initial setup, static initializers, and the complete execution path
- **Attach to running process**: is a method used to connect a debugger to a program during the executing
  - When the attachment is successful, all threads pause; useful to inspect a live process's current state, memory, and registers.
  - This is useful for analyzing long-running services, inspecting the state of an application at a specific moment of failure, or analyzing live malware processes.

Single-stepping

- **Single-stepping** is a debugging technique that execute one instruction at a time to observe changes in registers and memory.

  - allowing developers to closely monitor program flow, inspect variables, and find errors by seeing exactly how code behaves sequentially.
  - Great for small code regions; **too slow** for large code.

Step-into vs Step-over

- **Single step**: Executes one instruction at a time.
- **Step-over**:
  - Executes the function call and returns immediately without entering the function body.
  - Useful to skip over functions you don’t want to analyze in detail.
  - Risk: you might miss important actions inside the function.
- **Step-into**:
  - Enters the function and stops at its first instruction.
  - Used when you want to analyze the internal behavior of the function step by step.

Pausing Execution with Breakpoints

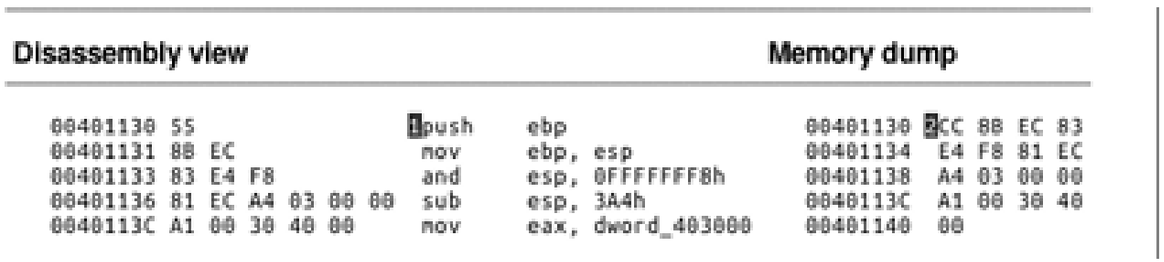
- **Pausing execution** with **breakpoints** is a fundamental debugging technique that stops a running program at a specified point in the code, allowing the developer to inspect the program's state, variables, and execution flow.
- **Example**:
  - If you don’t know where a function call goes, set a breakpoint on the call instruction.
  - Then watch the EAX register value to determine the target address.
- **Benefits**:
  - **Stepping** helps in tracking the execution path, instruction by instruction.
  - **Breakpoints** let you pause at critical points and inspect memory, registers, and execution path.

Introduction to Breakpoints

- Breakpoints are the fundamental mechanism used by analysts to control the execution flow of a program. They are categorized into three primary types:
- **Software Execution Breakpoints:** Rely on modifying the instruction code.
- **Hardware Execution Breakpoints:** Use dedicated CPU registers.
- **Conditional Breakpoints:** Logic-based pauses that trigger only when specific criteria are met.

Software Execution Breakpoints

- This is the default mechanism for most debuggers when a user sets a breakpoint.
- **The Mechanism:** The debugger overwrites the first byte of the target instruction with **0xCC**.
- **INT 3 Instruction:** 0xCC is the machine code for the INT 3 interrupt, specifically designed for use with debuggers.
- **Control Flow:** When the processor hits 0xCC, the OS generates an exception and transfers execution control to the debugger.
- **User Interface:** The debugger masks the 0xCC byte, displaying the original instruction (e.g., 0x55) to the user despite what is actually in memory.



Failure Points of Software Breakpoints

- Software breakpoints are "intrusive" because they alter the binary. This makes them detectable by malware:
- **Code Integrity Checks:** If the malware performs a checksum (e.g., CRC) on its own memory, it will detect the 0xCC byte and realize it is being debugged.
- **Self-Modifying Code:** If the malware overwrites its own instructions during execution, the 0xCC byte may be erased, causing the breakpoint to fail.
- **Read Scanned Memory:** If another part of the code reads the memory containing the breakpoint, it will see the debugger's byte instead of the original code.

Hardware Execution Breakpoints

- These breakpoints utilize the hardware capabilities of the CPU rather than modifying the code bytes.
- **Debug Registers:** Uses four hardware registers (**DR0 through DR3**) to store the specific addresses of the breakpoints.
- **Control Register (DR7):** Stores metadata and control info, defining whether to break on:
  - Execution (Instruction fetch).
  - Read/Write access (Data).
- **Key Advantage:** Since no code bytes are changed, they are invisible to software integrity checks.
- **Limitation:** A maximum of only 4 hardware breakpoints can be active at once.

Hardware Breakpoint Protection

- Malware can attempt to overwrite the DR0-DR3 registers to clear breakpoints.
- **General Detect Flag (in DR7):**
  - When enabled, it triggers a breakpoint *before* any MOV instruction that attempts to modify a Debug Register.
  - **Limitation:** It specifically detects MOV instructions; other methods of altering registers might still bypass this check.

Conditional Breakpoints

- A conditional breakpoint triggers only if a pre-defined logical condition is true.
- **Example:** Setting a breakpoint on GetProcAddress, but only pausing execution if the parameter being passed is "RegSetValue".
- **Implementation:** These are usually implemented as software breakpoints.
- **Behind the Scenes:** The debugger receives the break every time, checks the condition, and if it's false, automatically resumes execution without notifying the user.
- **Performance Hit:** Frequently hit conditional breakpoints can significantly slow down the program.

Understanding Exceptions

- Exceptions are events that interrupt normal execution, allowing a debugger to gain control.
- **Triggers:**
  - Breakpoints (INT 3).
  - Invalid memory access (Access Violation).
  - Division by zero.
- **The Exception Chain:**
  - **First-Chance Exception:** The debugger is notified first. It can choose to handle it or pass it to the program.
  - **Second-Chance Exception:** If neither the debugger nor the program's handlers resolve the issue, the debugger gets a final chance before the program crashes.

Common Exceptions in Debugging

- **INT 3:** The standard software breakpoint exception.
- **Single-Stepping:** Implemented via the **Trap Flag** in the Flags Register. The CPU executes one instruction and immediately generates an exception.
- **Memory-Access Violation:** Occurs when code tries to access a protected or invalid memory address.
- **Privilege Violation:** Attempting to execute Ring 0 (Kernel mode) instructions while in Ring 3 (User mode).

The following chart lists the exceptions that can be generated by the Intel 80286, 80386, 80486, and Pentium processors:

Exception (dec/hex)	Description
0 00h	Divide error: Occurs during a DIV or an IDIV instruction when the divisor is zero or a quotient overflow occurs.
1 01h	Single-step/debug exception: Occurs for any of a number of conditions: - Instruction address breakpoint fault - Data address breakpoint trap - General detect fault - Single-step trap - Task-switch breakpoint trap
2 02h	Nonmaskable interrupt: Occurs because of a nonmaskable hardware interrupt.
3 03h	Breakpoint: Occurs when the processor encounters an INT 3 instruction.

Modifying Execution in Practice

- Analysts use debuggers to alter the program's behavior in real-time:
- **Skipping Functions:** Change the Instruction Pointer (EIP/RIP) to point to the instruction immediately following a function call to avoid executing it.
- **Testing Functions:** Forcing a function to run in isolation by manually setting up its parameters on the stack and jumping to its start address. (Note: This often corrupts the stack).

Case Study: Malware Environmental Logic

- Sophisticated malware uses environmental checks to decide its payload:
- Real Virus: operation depends on language setting of a computer
- **Simplified Chinese:** Malware uninstalls itself (avoids infecting its own region).
- **English:** Displays a pop-up message: "Your luck's no good."
- **Japanese or Indonesian:** Triggers destructive behavior, such as overwriting the hard drive with random data.

Case Study: Malware Environmental Logic

- Break at 1; Change Return Value

```
00411349   call    GetSystemDefaultLCID
0041134F  1mov     [ebp+var_4], eax
00411352   cmp     [ebp+var_4], 409h        409 = English
00411359   jnz     short loc_411360
0041135B   call    sub_411037
00411360   cmp     [ebp+var_4], 411h        411 = Japanese
00411367   jz      short loc_411372
00411369   cmp     [ebp+var_4], 421h        421 = Indonesian
00411370   jnz     short loc_411377
00411372   call    sub_41100F
00411377   cmp     [ebp+var_4], 0C04h       C04 = Chinese
0041137E   jnz     short loc_411385
00411380   call    sub_41100A
```

References

- Sikorski, M., & Honig, A. (2012). *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press.
- Bowne, S. (n.d.). *CNIT 126 8: Debugging* [PowerPoint slides]. SlideShare. https://www.slideshare.net/slideshow/cnit-126-8-debugging/72827533
- Bowne, S. (n.d.). *CNIT 126: Ch 8 Debugging* [Video]. YouTube. https://www.youtube.com/watch?v=a0hhndBIRKk