Malware Analysis

Debugging

COMPILED BY: DHARMESH DAVE | ASST. PROF. | NATIONAL FORENSIC SCIENCES UNIVERSITY

Introduction

- A debugger is a piece of software or hardware used to **test or examine the execution** of another program.
- Debuggers give you insight into what a program is doing while it is executing. Debuggers are designed to allow developers to measure and control the internal state and execution of a program.
- Debuggers provide information about a program that would be difficult, if not impossible, to get from a disassembler. Disassemblers offer a snapshot of what a program looks like immediately prior to execution of the first instruction.

Introduction

- Debuggers provide a **dynamic view of a program** as it runs.
- For example, debuggers can show the values of memory addresses as they change throughout the execution of a program.
- Debuggers also let you **change anything** about program execution at any time.

Source-Level vs. Assembly-Level Debuggers

- Source-level-debugger is usually built into integrated development environments (IDEs).
- Source-level debuggers allow you to set breakpoints, which stop on lines of source code, in order to examine internal variable states and to step through program execution one line at a time.
- Assembly-level debuggers, sometimes called low-level debuggers, operate on assembly code instead of source code.
- As with a source-level debugger, you can use an assembly-level debugger to step through a program one instruction at a time, set breakpoints to stop on specific lines of assembly code, and examine memory locations.

Kernel vs. User-Mode Debugging

- It is more challenging to debug kernel-mode code than to debug user-mode code because you usually need **two different systems for kernel mode**.
- In user mode debudding, the debugger is running on the same system as the code being debugged. When debugging in user mode, you are debugging a single executable, which is separated from other executables by the OS.
- Kernel debugging is performed on two systems because there is only one kernel; if the kernel is at a breakpoint, no applications can be running on the system. One system runs the code that is being debugged, and another runs the debugger. Additionally, the OS must be configured to allow for kernel debugging, and you must connect the two machines.

Using Debugger

- When you start the program and it is loaded into memory, it **stops running immediately prior to the execution of its entry point**. At this point, you have complete control of the program.
- You can also attach a debugger to a program that is already running. All the program's threads are paused, and you can debug it.

Single-Stepping

- Single-step means that you run a **single instruction and then return control** to the debugger. Single-stepping allows you to see everything going on within a program.
- It is possible to single-step through an entire program, but you **should not do it for complex programs** because it can take such a long time. But one must be selective about which code to analyze.

Stepping-Over vs. Stepping-Into

- When single-stepping through code, the debugger stops after every instruction. However, while you are generally concerned with what a program is doing, you may not be concerned with the functionality of each call.
- For example, if your program calls LoadLibrary, you probably don't want to step through every instruction of the LoadLibrary function.
- **To control the instructions that you see in your debugger, you can stepover or step-into instructions**. When you step-over call instructions, you bypass them.
- If you step-over a call, the next instruction you will see in your debugger will be the instruction after the function call returns.

Stepping-Over vs. Stepping-Into

- **Stepping-over** allows you to **significantly decrease the amount of instructions you need to analyze**, at the risk of missing important functionality if you step-over the wrong functions.
- Additionally, certain function calls never return, and if your program calls a function that never returns and you stepover it, the debugger will never regain control.
- When this happens (and it probably will), restart the program and step to the same location, but this time, step-into the function.

Pausing Execution with Breakpoints

- Breakpoints are used to pause execution and allow you to examine a program's state. When a program is paused at a breakpoint, it is referred to as broken.
- Breakpoints are needed because you can't access registers or memory addresses while a program is running, since these values are constantly changing.
- We set a breakpoint on the call to **CreateFileW, and then look at the values on the stack** when the breakpoint is triggered.
- Luckily, we can use a debugger to simplify this task because encryption routines are often separate functions that transform the data.
- You can use several different types of breakpoints, including software execution, hardware execution, and conditional breakpoints.

Pausing Execution with Breakpoints

```
004010D0 sub esp, 0CCh
004010D6 mov eax, dword_403000
004010DB xor eax, esp
004010DD mov [esp+0CCh+var_4], eax
004010E4 lea eax, [esp+0CCh+buf]
004010E7 call GetData
004010EC lea eax, [esp+0CCh+buf]
004010EF call EncryptData
004010F4 mov ecx, s
004010FA push 0 ; flags
004010FC push 0C8h ; len
00401101 lea eax, [esp+0D4h+buf]
00401105 push eax ; buf
00401106 push ecx ; s
00401107 call ds:Send
```

Software Execution Breakpoints

- When you set a breakpoint without any options, most popular debuggers set a **software execution breakpoint by default**.
- The debugger implements a software breakpoint **by overwriting** the first byte of an instruction with **0xCC**, the instruction for **INT 3**, the breakpoint interrupt designed for use with debuggers.
- When the 0xCC instruction is executed, the OS generates an exception and transfers control to the debugger.

Software Execution Breakpoints

Disassembly view		Memory dump
00401130 55	**push ebp**	00401130 **CC** 8B EC 83
00401131 8B EC	mov ebp, esp	00401134 E4 F8 81 EC
00401133 83 E4 F8	and esp, 0FFFFFFF8h	00401138 A4 03 00 00
00401136 81 EC A4 03 00 00	sub esp, 3A4h	0040113C A1 00 30 40
0040113C A1 00 30 40 00	mov eax,	00401140 00
dword_403000		

- The debugger's memory dump will show the original 0x55 byte, but if a program is reading its own code or an external program is reading those bytes, the 0xCC value will be shown.
- If these bytes change during the execution of the program, the breakpoint will not occur. For example, if you set a breakpoint on a section of code, and that code is self-modifying or modified by another section of code, your breakpoint will be erased.

Software Execution Breakpoints Limitation

- If any other code is reading the memory of the function with a breakpoint, it will read the 0xCC bytes instead of the original byte. Also, any code that verifies the integrity of that function will notice the discrepancy.

Hardware Execution Breakpoints

- The x86 architecture supports **hardware execution breakpoints** through dedicated hardware registers.
- Every time the processor executes an instruction, there is hardware to detect if the instruction pointer is equal to the breakpoint address.
- Unlike software breakpoints, with hardware breakpoints, it doesn't matter which bytes are stored at that location.
- For example, if you set a breakpoint at address 0x00401234, the processor will break at that location, regardless of what is stored there.

Hardware Execution Breakpoints

- **Hardware breakpoints** have another advantage over software breakpoints in that **they can be set to break on access rather than on execution**.
- For example, you can set a hardware breakpoint to break whenever a certain memory location is read or written.
- If you're trying to determine what the value stored at a memory location signifies, you could set a hardware breakpoint on the memory location. Then, when there is a write to that location, the debugger will break, regardless of the address of the instruction being executed. (You can set access breakpoints to trigger on reads, writes, or both.)

Hardware Execution Breakpoints Limitation

- Unfortunately, hardware execution breakpoints have one major drawback: only **four hardware registers** store breakpoint addresses.
- One further drawback of hardware breakpoints is that they are **easy to modify** by the running program.
- There are eight debug registers in the chipset, but **only six are used**. The first four, DR0 through DR3, store the address of a breakpoint. The debug **control register (DR7)** stores information on whether the values in DR0 through DR3 are enabled and whether they represent read, write, or execution breakpoints.

Hardware Execution Breakpoints Solution

- Malicious programs can modify these registers, often to interfere with debuggers. Thankfully, x86 chips have a feature to protect against this.
- By setting the **General Detect flag in the DR7 register**, you will trigger a breakpoint to occur prior to executing any **mov instruction** that is accessing a debug register.

Conditional Breakpoints

- Conditional breakpoints are software breakpoints that will **break only if a certain condition is true**.
- If you want to break only if the parameter being passed to GetProcAddress is RegSetValue
- Conditional breakpoints are implemented as software breakpoints that the debugger always receives. The **debugger evaluates the condition, and if the condition is not met, it automatically continues execution** without alerting the user.
- Breakpoints take much longer to run than ordinary instructions, and your program will slow down considerably if you set a conditional breakpoint on an instruction that is accessed often

References

- Practical Malware Analysis by Michael Sikorski