# Debugging Malicious Binaries

A debugger is a program that gives you the ability to inspect malicious code at a more granular level.

## 1. General Debugging Concepts

### 1.1 Launching and attaching to processes

Two ways to debug:

- Attaching the debugger to a running process
  - Not able to monitor or control the process's initial actions. (When attached, all of startup and initialization will have already been executed)
- Launching a new process
  - Allows the debugger to monitor every action the process takes
  - Able to monitor processes initial actions

### 1.2 Controlling Process Execution

Important abilities:

- Ability to control execution
- Ability to interrupt execution

Common execution control options:

- **Continue (Run):** Executes all of the instructions until a breakpoint is reached or an exception occurs.
- **Step into and Step over:** Allows you to execute a single command. The difference occurs when you are executing a command that calls a function. Step into, will stop at the start of the function, whereas Step over will execute the entire function and the debugger will pause at the next instruction.
- **Execute till Return (Run until return):** Execute all of the instruction of a given function, until it returns
- **Run to cursor (Run until selection):** Execute instructions until the current cursor position, or until the selected instruction is reached.

### 1.3 Interrupting a Program with breakpoints

Allows you to interrupt the program execution at a very specific location within a program.

Types of breakpoints:

- **Software Breakpoints:** Implemented by replacing the instruction at a breakpoint address with a SW breakpoint instruction (Such as `int`

`3` instruction (having an opcode of `0xCC`)) When a SW instruction is executed, the control is transfered to the debugger. The disadvantage is that a malicious program can look for the instruction and alter the debugger's default behavior

- **Hardware Breakpoints:** Breakpoints through the CPU's breakpoint registers: `DR0 - DR7`. Max of 4 breakpoints, the other registers are used to pass parameters. No instruction is replaced. The CPU decides whether the program should be interrupted.
- **Memory Breakpoints:** Pause the execution when a program accesses the memory, rather than the execution.
- **Conditional Breakpoints:** You specify the conditions that must be satisfied to trigger the breakpoint. (Feature offered by the debugger)

### 1.4 Tracing Program Execution

Tracing is a debugging feature that allows you to record (*log*) specific events while the process is executing.

## 2. Debugging a Binary Using x64dbg

### 2.1 Launching a New Process in x64dbg

You can change the breakpoints for the loaded program in `Options > Preferences > Events`. The default preferences stop the program at the system, TLS and Entry breakpoints.

### 2.2 Attaching to an existing process using x64dbg

When the process is attached, the process is suspended, therefore, you have time to set breakpoint and analyse the code. When you close the debugger, the attached process will terminate. If you don't want the process to terminate, you have to detach a program before closing the debugger.

### 2.3 x64dbg Debugger Interface

The debugger display contains multiple tabs, each tab, contains multiple windows

- **Dissasembly window:** Shows the disassembly of all of the instructions of the debugged program. Display the control flow graph by pressing the G hotkey.
- **Registers window:** Displays the current state of the CPU registers
- **Stack window:** Displays the data contents of the process's runtime stack
- **Dump window:** Displays the standard HEX dump of the memory
- **Memory Map window:** Provides the layout of the process memory and gives you the details of the allocated memory segments in the process.
- **Symbols window:** Left pane displays the loaded modules, the right pane shows a list of the selected module's import and export functions.

- **References window:** Displays the references to the API calls. To display the references, you have to `right click anywhere in the dissasembly window > Search for > Current module > Intermodular calls`
- **Handles window:** Displays the details of all the open handles
- **Threads window:** Displays a list of threads in the current process.

## 2.4 Controlling Process Execution Using x64dbg

| Functionality | Hotkey | Menu |
|---|---|---|
| Run | *f9* | Debugger - run |
| Step into | *f7* | Debugger - Step into |
| Step over | *f8* | Debugger - Step over |
| Run until selection | *f4* | Debugger - run until selection |

## 2.5 Setting a Breakpoint in x64dbg

- **Software breakpoint:** Press *F2* in the address you want the program to stop
- **Hardware breakpoint:** In the dump pane, Right click, select breakpoint, Hardware.
- **Hardware Memory breakpoint:** In the dump pane, Right click, select breakpoint, Hardware write.
- **Memory breakpoint:** In the dump pane Right click on the desired address, select breakpoint, memory

To view and edit all the breakpoints, click on the breakpoints tab.

## 2.6 Debugging 32-bit Malware

Imagine a malware program calls a function that creates a file. To know the name of the file it creates, place a breakpoint at the function call, and analyse the stack once the breakpoint is reached and the program paused.

Let's assume we don't know which object is associated with a handle. To determine the object that is associated with a handle value, we can look it up in the Handles window.

## 2.7 Debugging 64-bit malware

The difference is that we will be dealing with extended registers, 64-bit memory addresses/pointers and slightly different *calling conventions*

In 64-bit architecture, the stack used when a function is called with more than 4 parameters, does not change. The stack space is allocated at the beginning of the function, and does not change until the end of the function. The allocated stack space is used to store local variables and function parameters. The lack of

`push` and `pop` functions make it difficult to determine the number of parameters accepted by the function ant it is also hard to say Wether the memory address is being used as a local variable or a parameter to the function. This can be overcome by reading the API functions documentation.

## 2.8 Debugging a Malicious DLL Using x64dbg

Open the DLL in x64dbg. When it is loaded, a executable will be created in the same location where it is loaded. This will be a host process to load the DLL. After the DLL is loaded, the debugger may pause at the `System breakpoint`, `TLS callback` or `DLL entry point` function.

### 2.8.1 Using rundll32.exe to debug the DLL in x64dbg

Another effective method is to use `rundll32.exe` to debug the DLL. 1. Load `rundll32.exe` from system32 directory into de debugger. 1. Select `debug > change command line` and specify the command line arguments to `rundll32.exe` 1. Select `breakpoints tab > right click inside the breakpoints window and choose the add DLL breakpoint option` 1. Enter the DLL name 1. Close the debugger 1. Re-open the debugger and load `rundll32.exe` again 1. Select Run (*f9*)

### 2.8.2 Debugging a DLL in a specific process

Similar to the previous section

1. Open the debugger
2. Launch the process that the DLL is loaded from
3. Open the Breakpoints tab, right-click in the breakpoints window
4. Add a DLL breakpoint
5. Enter the DLL name
6. Inject the DLL into the host process (Tool like `remoteDLL`)
7. When the DLL is loaded, the debugger will pause

## 2.9 Tracking Execution in x64dbg

Allows to log events as the process executes

- **Trace into (Ctrl + Alt + f7):** The debugger internally traces the program by setting *step into* breakpoint, until a condition is satisfied or the maximum number of steps is reached.
- **Trace over (Ctrl + Alt + f8):** The debugger traces the program by setting *step over* breakpoint, until the condition is satisfied or the maximum number of steps is reached.

Trace conditions:

- **Break Condition:** Specify a condition in this field. If the condition is true, the debugger will pause.
- **Log Text:** Format that will be used to log the trace events in the log file.
- **Log Condition:** Defaults to 1. Determines if the debugger logs the event or not
- **Maximum trace count:** Maximum steps until the debugger gives up. Defaults to `50000`
- **Log File Button:** Specify the full path to the log file where the traces will be saved

### 2.9.1 Instruction Tracing

To perform instruction tracing, set the Trace settings to the following:

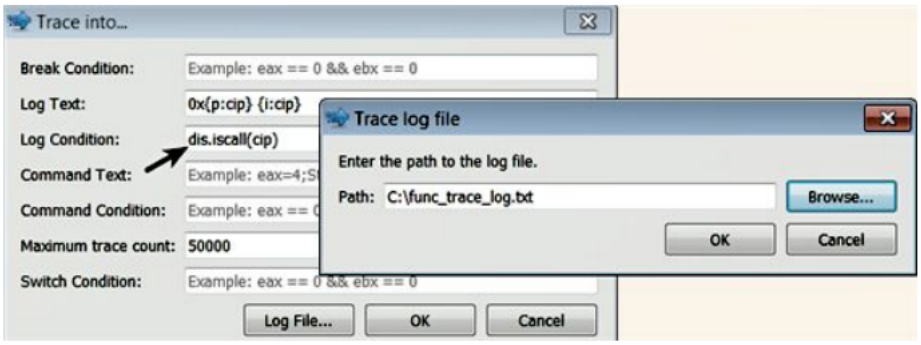


Figure 1: Instruction Tracing

The log text value is in the string format, which specifies the debugger to log the *address* and the *disassembly* of all the traced instructions.

### 2.9.2 Function Tracing

To perform function tracing, set the Trace into settings to the following:

When the debugger reaches the breakpoint, the execution is paused, and the *instructions/functions* till the breakpoint are logged. Then you resume the debugger, the rest of the instructions are executed, but not logged.

## 2.10 Patching in x64dbg

To modify the data in a memory, navigate to the memory address and select the sequence of bytes you want to modify, right click and choose binary, which will bring up a dialog that you can use to modify the data as ASCII, UNICODE, or a sequence of hex bytes.

You can modify a *conditional jump* instruction (`JNE tdss.10001Cf9` with a `nop` instruction, to remove the process restriction.

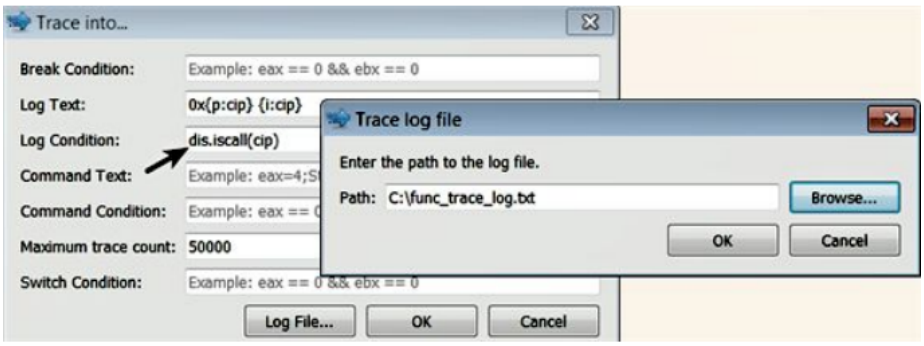


Figure 2: Function Tracing

1. Right-click on the conditional jump instruction
2. Select Assemble
3. Introduce the instruction

Once all the patches you want are created, select `File > patch file` and confirm the patches you have created.

# 3. Debugging Binary Using IDA

## 3.1 Launching a new process in IDA

- Launching the debugger and select `Run > Local Windows debugger`
- Load the executable in IDA, select the correct debugger and pause the program with a breakpoint

## 3.2 Attaching to an existing process using IDA

**When the program has not loaded:**

- Select `Debbuger > Attach > Local Windows Debbuger`. This will list all the running processes

**When the program has loaded:**

- Load the executable associated to a process into IDA before attaching to that process.

## 3.3 IDA's Debugger Interface

- **Dissasembly window:** Synchronized with the current value of the *instruction pointer* register (`eip, rip`.
- **Register window:** Displays and lets you edit the current contents of the CPU's general purpose register.
- **Stack view:** Displays the data contents of the process's runtime stack.

- **Hex view:** Displays the standard HEX dump of the memory.
- **Modules view:** Displays the list of modules loaded into the process memory.
- **Threads view:** Displays the list of threads in the current process.
- **Segments window:** Provides information regarding the allocated memory segments in a process, Displays the information about where the executable and it's sections are loaded in memory.
- **Imports and Exports window:** Lists all the functions imported and exported by the library

## 3.4 Controlling Process Execution using IDA

| Functionality | Hotkey | Menu |
|---|---|---|
| Continue (Run) | *f9* | Debugger - continue process |
| Step into | *f7* | Debugger - Step into |
| Step over | *f8* | Debugger - Step over |
| Run to cursor | *f4* | Debugger - run to cursor |

## 3.5 Setting Breakpoint in IDA

Navigate to the address where you want to set the breakpoint and press *f2* or `right-click > Add breakpoint`.

To set a hardware breakpoint, `right-click on the breakpoint > edit breakpoint`. After selecting the previously created breakpoint, the pop-up window lets you edit the breakpoint in every mode.

## 3.6 Debugging Malware Executables

Suppose we have a malware that calls the `CreateFileW` Windows API To know the name of the file that is passed to the function, pause the program execution on this call. All the parameter that are passed to it will be stored at the stack. Right click on the address and select Follow in hex dump. Now, you can see the contents of the memory for that register, and therefore, the file's name and path. When the function returns, the handle will be returned at the `eax` register.

## 3.7 Debugging a Malicious DLL using IDA

1. Open the DLL file
2. Select `Debugger > Local Windows Debugger > ok`
3. On the pop-up, select the program used to run the DLL, and other parameters.

### 3.7.1 Debugging a DLL in a specific Process

Same as x64dbg

## 3.8 Tracing execution using IDA

IDA allows 3 types of tracing:

To enable tracing, set a breakpoint in the program, right click on the breakpoint and select Enable tracing.

- **Instruction tracing:** Records the execution of each instruction and displays the modified register values. Useful for determining the execution flow of the program, and to know which registers where modified during the execution of each instruction.
- **Function tracing:** Records all of the function calls and then returns. Useful to determine which functions and sub-functions are called during the program execution.
- **Block tracing:** Useful for knowing which blocks of code where executed at runtime.

## 3.9 Debugger Scripting using IDAPython

To use IDAPython for debugger related tasks, load the executable in IDA, and select the debugger. After, open the shell (`File > Script commend > Python`).

**Example:**

This sets a breakpoint at the current cursor location, starts the debugger, waits for the `suspend debugger` event to occur, and then prints the *address* and the *disassembly text* associated with the breakpoint address.

```
idc.add_pbt(idc.get_screen_ea())
idc.start_process('','','')
evt_code = idc.wait_for_next_event(WFNE_SUSP, -1)

if (evt_code > 0) and (evt_code != idc.PROCESS_EXITED):
  evt_ea = idc.get_event_ea()
  print "Breakpoint triggered at: "

hex(evt_ea), idc.generate_disasm_line(evt_ea, 0)
```

And outputs: `Breakpoint Triggered at: 0x117010 push ebp`

# 4. Debugging a .NET Application

Analyzing .NET applications with a program called `dnSpy`. This program decompiles the malicious code and lets you perform static and dynamic code analysis. You can also set breakpoints.