Time Travel Debugging (TTD) Lab Instructions

Instructor: Xusheng Li (xusheng@vector35.com)

Lab Overview

This lab will introduce you to Time Travel Debugging (TTD) for Windows using Binary Ninja. You will learn how to set up TTD, record execution traces, navigate through recorded traces, and use advanced TTD features for debugging and analysis.

Prerequisites: - Binary Ninja installed on Windows - Administrative privileges (required for TTD recording) - Sample executable for debugging (helloworld.exe provided)

Part 1: Setting Up TTD

Objective

Configure WinDbg/TTD integration in Binary Ninja for this lab environment.

Instructions

Note: For this lab, we will use a special setup that points to a pre-installed version of WinDbg on the system. In a normal setup, you would typically use Binary Ninja's built-in automatic installation by navigating to **Menu → Debugger → "Install WinDbg/TTD"**, which downloads and installs the necessary components automatically.

Lab-Specific Configuration

1. Launch Binary Ninja
2. Open Settings by navigating to **Edit → Settings** (or press **Ctrl+,**)
3. In the Settings search bar at the top, type: `debugger.x64dbgEngPath`
4. Click on the **debugger.x64dbgEngPath** setting to edit it
5. Set the value to: `C:\Program Files\Windows Kits\WinDbg\amd64`
    - This points Binary Ninja to the pre-installed WinDbg debugging engine on the lab machine
6. Restart Binary Ninja and it should report no error

Checkpoint: Confirm that TTD is properly configured before proceeding.

Part 2: Recording a Trace

Objective

Learn how to record TTD traces of program execution.

Exercise 2.1: Understanding the Target Program

Before recording a trace, let's understand what the program does.

1. Open a command prompt or terminal in the lab directory
2. Run `helloworld.exe` without any arguments:

   ```
   helloworld.exe
   ```

3. Observe the output - what does the program print?
4. Now run it with some command-line arguments:

   ```
   helloworld.exe arg1 arg2 test
   ```

5. Compare the output - what changed?

Reference: The source code `helloworld.c` is provided in the lab directory. Review it to understand: - What the program prints - How it processes command-line arguments - What return value it uses

Exercise 2.2: Recording from Binary Ninja

1. In Binary Ninja, open `helloworld.exe` from the lab directory
2. Navigate to **Menu → Debugger → "Record TTD Trace"**
3. Configure the recording settings:
   - **Executable Path:** Should already be populated with the path to `helloworld.exe`
   - **Working Directory:** Should already be set to the directory containing helloworld.exe
   - **Command-line Arguments:** Enter some arguments of your choice (e.g., `student debug test` or `arg1 arg2 arg3`)
     - Think about what arguments would be interesting to track during debugging
   - **Trace Output Location:** By default, this will be set to the same directory as the executable (the lab directory). You can keep this default or choose a different location if preferred.
   - **Start Recording Deferred:** Leave unchecked
   - **Include Child Processes:** Leave unchecked (unless needed)
4. Click **Start Recording**
5. If a UAC (User Account Control) prompt appears, accept it
   - **Note:** In this lab environment, you are logged in as Administrator, so the UAC prompt may not appear. However, if you use TTD as a normal user on your own system, you will need to accept the UAC prompt for administrative elevation.
6. Let the program run to completion
7. A dialog will pop up indicating that the recording has completed. Click **OK** to dismiss it.
8. Note the location of the generated `.run` trace file

Questions

1. What does helloworld.exe print to the console?
2. How does the program handle command-line arguments?
3. What file extension does a TTD trace use?
4. Why does TTD recording require administrative privileges?
5. How large is your trace file compared to the original executable?

Checkpoint: Ensure you have successfully recorded at least one trace file before continuing.

Part 3: Loading and Basic Navigation

Objective

Load a TTD trace and learn basic navigation controls.

Exercise 3.1: Loading a Trace

1. Open the original executable (`helloworld.exe`) in Binary Ninja (if you haven't already)
2. Open the Debugger sidebar
3. In the Debugger sidebar, locate the debug adapter dropdown and select **DBGENG_TTD**
4. Click the **Launch** button
    - The Debug Adapter Settings dialog will appear
5. In the **Trace File Path** field, enter the full path to your `.run` file
    - Example: `C:/Users/user/Documents/helloworld01.run` (or wherever your trace file was saved)
6. Click **OK** to start the debugging session

Exercise 3.2: Understanding Navigation Controls

Once the TTD trace is loaded, TTD-specific buttons will appear in the debugger control buttons widget.

Tip: Hover over the buttons to see their names and keyboard shortcuts.

The TTD debugger provides both forward and reverse navigation controls:

Forward Controls (Standard Debugging)

- **Continue (F9):** Run forward to next breakpoint
- **Step Over (F8):** Execute current line, don't enter functions
- **Step Into (F7):** Execute and enter function calls
- **Step Return (Ctrl+F9):** Run until current function returns

Reverse Controls (Time Travel - shown in RED)

- **Reverse Continue (Shift+F9):** Run backward to previous breakpoint
- **Step Over Backward (Shift+F8):** Step backward over instructions
- **Step Into Backward (Shift+F7):** Step backward into function calls
- **Step Return Backward (Ctrl+Shift+F9):** Run backward to function entry

Additional Navigation

- **Navigate to Timestamp (Shift+G):** Opens a dialog that allows you to jump directly to any timestamp in the trace

Exercise 3.3: Basic Navigation Tasks

Complete these tasks using the navigation controls:

1. **Task A:** Start debugging and continue to the main function
    - First, navigate to the `main` function in the disassembly view
    - Press **F2** to set a breakpoint at the beginning of `main`
    - A red mark should appear on the left to indicate the breakpoint is set
    - Use Continue (F9) to run to the breakpoint
2. **Task B:** Step through the first 10 instructions in main using Step Over (F8)
3. **Task C:** Use reverse navigation (Shift+F8) to return to the entry point of main
4. **Task D:** Set a breakpoint on a function call (e.g., printf)
5. **Task E:** Continue forward (F9) to hit the breakpoint
6. **Task F:** Use Reverse Continue (Shift+F9) to go backward to the previous breakpoint or beginning
7. **Task G:** Resume the target until it hits the end of the trace
    - Use Continue (F9) repeatedly or remove breakpoints and continue
    - When you reach the end, check the **Stack Trace** view to see the final call stack
    - Note the final program state and position
8. **Task H:** Resume backwards to go to the start of the trace
    - Use Reverse Continue (Shift+F9) to navigate backward
    - Observe how the execution state rewinds to the beginning
9. **Task I:** Use Navigate to Timestamp (Shift+G) to jump to a specific position in the trace
10. **Task J:** Practice using both forward and reverse Step Into commands to trace through function calls

Questions

1. What visual indicator shows you're using reverse navigation?
2. What if you issued a wrong command? For example, when you want a single step but you resumed the target? How is it different in regular debugging and TTD?
3. What is the difference between step into and step over? What is the difference between reverse step into and reverse step over?
4. Can you modify program state during TTD replay? Why or why not?

Part 4: Advanced TTD Usage

Objective

Utilize advanced TTD features for detailed program analysis.

Exercise 4.1: TTD Events Widget

The Events widget displays thread lifecycle, module loading, and exceptions.

1. Open the **TTD Events** sidebar widget
2. Explore the different tabs:
    - **Threads:** View thread creation and termination events
    - **Modules:** See when DLLs were loaded/unloaded
    - **Exceptions:** Review any exceptions that occurred
3. Identify:
    - When was the main thread created?
    - What modules were loaded at startup?
    - Were any exceptions raised during execution?
4. **Double-click** events to navigate to their occurrence in the trace

Exercise 4.2: Code Coverage Analysis

TTD can highlight which instructions were actually executed.

1. Generate coverage information:
    - Navigate to **Debugger → TTD Analysis**
    - In the dialog that appears, click **Run Analysis**
    - Wait for the analysis to complete (this caches the coverage information)
2. Enable code coverage visualization:
    - Click the **hamburger menu** (three horizontal lines) at the upper-right corner of the disassembly view
    - Navigate to **Render Layers → TTD Coverage**
    - If TTD Coverage is already checked but you don't see highlighting, try unchecking it and checking it again
3. Observe:
    - Executed instructions highlighted in **red**
    - Unexecuted code paths remain unhighlighted
4. Analysis tasks:
    - Identify any unreachable code
    - Find conditional branches that were not taken
    - Understand actual program flow vs. possible flow

Note: Code coverage analysis is cached for performance on large traces.

Exercise 4.3: TTD Calls Widget

The Calls widget allows you to query all invocations of specific functions. In this exercise, you will learn how to use the return address range filter and analyze calls to printf.

Part A: Understanding Return Address Range Filtering

1. Open the **TTD Calls** sidebar widget

2. First query - without return address range:

    - In the **Function** field, enter: `kernel32!*`
    - Leave the **Return Address Range** fields empty
    - Click **Query**
    - Note the total number of results returned

- Observe that these are ALL calls to kernel32 functions from anywhere in the process

3. Find the helloworld module address range:
   - Open the **Modules** sidebar widget
   - Locate the `helloworld.exe` module in the list
   - Note the **Start Address** and **End Address** of the module
   - Example: Start might be `0x00400000`, End might be `0x00408000`

4. Second query - with return address range:
   - Return to the **TTD Calls** widget
   - In the **Function** field, enter: `kernel32!*` (same as before)
   - In the **Return Address Range Start** field, enter the start address of helloworld.exe
   - In the **Return Address Range End** field, enter the end address of helloworld.exe
   - Click **Query**
   - Note the total number of results returned

5. Compare the results:
   - How many results did you get without the return address range?
   - How many results did you get with the return address range?
   - **Question:** Why is there a difference in the number of results? What does the return address range filter do?

Part B: Analyzing Printf Calls

6. Query printf implementation:
   - In the **Function** field, enter: `ucrtbase!_stdio_common_vfprintf`
     - This is the actual printf implementation in the MSVC toolchain
   - Leave the **Return Address Range** fields empty
   - Click **Query**
   - Note the total number of results returned
7. Analyze the results:
   - How many calls to `_stdio_common_vfprintf` were made?
   - **Question:** How does this number relate to the command-line arguments you passed when recording the trace? Think about what helloworld.exe prints based on the arguments.
8. Time travel to a printf call:
   - **Double-click** the first result to time-travel to that function call
   - Observe the disassembly at the call site
9. Find the format string:
   - Look at the **Registers** view (open the Registers sidebar widget if needed)
   - Examine the register values at this point
   - **Hint:** One of the registers should point to a string (the format string)
   - In the x64 calling convention, function arguments are typically passed in `rcx`, `rdx`, `r8`, `r9`
   - **Question:** Which register contains a pointer to the format string? What is the format string being used?
10. Additional analysis:
    - Navigate to each of the other printf calls (double-click each result)
    - For each call, identify what is being printed

- Verify this matches the expected output from helloworld.exe

Exercise 4.4: TTD Memory Widget

The Memory widget tracks memory operations across execution.

Note: You can create TTD memory queries directly from the disassembly view using the context menu. For example, select a few instructions, then **right-click → Debugger → TTD Memory Access → Execute** to query when those instructions were executed.

1. Open the **TTD Memory** sidebar widget
2. Method 1: Manual query using the widget:
   - Identify an interesting variable or string in your program's disassembly
   - Note its address (e.g., `0x00401000`)
   - Configure a memory query:
     - **Address Range:** Enter start and end addresses (e.g., `0x00401000-0x00401100`)
     - **Operation Type:** Select Read, Write, or Execute
     - Click **Query**
3. Method 2: Quick query using context menu:
   - In the disassembly view, select a few instructions of interest
   - Right-click on the selection
   - Navigate to **Debugger → TTD Memory Access → Execute**
   - This automatically creates a query for when those instruction addresses were executed
4. Review results showing:
   - Positions where memory was accessed
   - Type of access (read/write/execute)
   - Context of the access
5. **Double-click** a result to jump to that memory access in the trace

Exercise 4.5: Performance Considerations

Important Warning: TTD queries execute synchronously and block the UI.

1. Test query performance:
   - Try a narrow query: `kernel32!CreateFileA`
   - Compare with a broad wildcard: `*!*`
2. Observe:
   - UI becomes unresponsive during query execution
   - No cancellation mechanism is available
   - Broader queries and larger address ranges take significantly longer
3. **Best Practices:**
   - Start with specific queries before using wildcards
   - Limit memory query address ranges
   - Be patient with large traces

Lab Summary and Deliverables

What You Learned

1. **Setup:** Installing and configuring TTD in Binary Ninja
2. **Recording:** Creating TTD traces from executables
3. **Navigation:** Using forward and reverse debugging controls
4. **Advanced Analysis:** Using Calls, Memory, and Events widgets
5. **Performance Considerations:** Understanding query performance and best practices

Deliverables (Optional)

Note: Submitting a lab report is completely optional. The exercises and questions are designed to help you follow along and learn TTD effectively. If you choose to create a report, it can serve as a useful reference for your own learning.

If you would like to document your work, consider including:

1. **Screenshots:**
   - TTD installation confirmation
   - Recorded trace file properties
   - Navigation controls demonstration
   - Each advanced widget (Calls, Memory, Events) with query results
   - Code coverage visualization
2. **Written Answers:**
   - Responses to all questions in Parts 2-4
   - Observations about performance with different query types
   - Additional findings or interesting discoveries

Additional Resources

- Binary Ninja Debugger Documentation
- WinDbg Time Travel Debugging Overview
- Binary Ninja API Reference
- Awesome TTD - Curated List of TTD Resources

Next Steps

After completing this lab: - Practice TTD with more complex malware samples - Explore TTD for vulnerability research - Integrate TTD analysis into your reverse engineering workflow - Experiment with advanced Python scripting for automated analysis

Congratulations on completing the TTD Lab!