# Introduction to Time-Travel Debugging (TTD)

Xusheng Li
xusheng@vector35.com

# About Me

1. Software developer at Vector 35, debugger lead
2. Penn State IST Alumni (advised by Dr. Peng Liu)
3. Paged Out! reviewer
4. Crackmes.ome admin and reviewer

# Debugging

1. Reverse engineering techniques are either static or dynamic
2. Debugging is one of the dynamic analysis techniques
3. Debugging gives you a more “direct” access to the target
4. Debugging at a high level:
    a. Execute the target
    b. Observe the states (registers, memory, threads, etc)
    c. Derive conclusions or hypothesis
    d. Optionally modify the states

# Common Debuggers

1. x64dbg – most popular on Windows
2. GDB – most popular on Linux
3. LLDB – most popular on macOS
4. Binary Ninja debugger – cross-platform

# Debugging (demo)

Regular debugging with Binary Ninja debugger

# Advantages of debugging

1. Ground truth values, not guesses
2. Precise control-flow recovery
3. Defeats certain obfuscation methods “for free”

# Limitations of debugging

1. Debugging only goes forward
    a. there is no way to rewind program execution
    b. If something already happened, there is no way to get back to it
2. Repetitive in nature
    a. If user interactions are required to trigger some code, this has to be repeated every time
    b. If you clicked a wrong button, you may have to restart (this is more often than you think)
    c. Not rocket science, but the irritation adds mental pressure and time consumption
3. Differences in different runs
    a. ASLR can cause the code to be loaded at a different address every time
    b. Certain malware C&C server could limit the number of times an IP address can access it
4. Threats of evasion
    a. You are running the malware and there is a chance when it gets out of your control

# What is TTD?

1. Time-travel debugging (TTD) is the process of recording the execution trace of a target and then replay it during analysis or debugging
2. TTD usually covers two steps:
    a. Record: record the execution trace of the target
    b. Replay: replay the execution trace, to either simulate a debugging experience, or perform certain analysis
    c. TTD is also called “record & replay” in the literature

# TTD (Baic Demo)

TTD demo with Binary Ninja debugger

1. Configure WinDbg/TTD
2. Record a trace
3. Replay the trace in the debugger

# TTD’s advantage

1. TTD records the entire execution, during debugging, the TTD engine just replays the trace as needed
2. Exact state preservation
3. Freedom to go back and forth in execution history
4. We can run deeper analysis to extract information from the trace that is impossible with traditional debugging

# TTD backends

Binary Ninja can interface with multiple TTD backends:

1. WinDbg
2. esReverse (https://eshard.com/esreverse)
3. Undo (https://undo.io/)
4. rr (https://rr-project.org/)
5. GDB
6. PANDA (https://panda.re/)
7. Custom (emulator, etc)

# Quotes on TTD

“Time-travel debugging is a **paradigm shift** in debugging”

“TTD is the future of debugging”

My take: TTD has precise and abundant information about the execution, an important aspect of TTD is to extract and utilize these information

# Advanced Demo 1: TTD.Memory queries

Same as running

dx -g $cursession.TTD.Memory(start,end,"rwe")

Benefits:

1. No need to remember and type the command
2. Much better presentation and navigation
3. More flexible filtering on the results
4. API access to the results

TTD Memory

Query 1

Query Parameters

Start Address: 0x7ff7967a1338

End Address: 0x7ff7967a1366

Access Types: ☐ Read ☐ Write ☑ Execute

Query Memory Events | Clear Results

| Index | Time Start | Access Type | Address | Size | Value | Thread ID | IP |
|---|---|---|---|---|---|---|---|
| 0x0 | 16:18 | E | 0x7ff7967a1338 | 4 | 0x28ec8348 | 6236 | 0x7ff7967a1338 |
| 0x1 | 16:19 | E | 0x7ff7967a133c | 5 | 0x25be8 | 6236 | 0x7ff7967a133c |
| 0x2 | 16:28 | E | 0x7ff7967a1341 | 4 | 0x28c48348 | 6236 | 0x7ff7967a1341 |
| 0x3 | 16:29 | E | 0x7ff7967a1345 | 5 | 0xfffffe72e9 | 6236 | 0x7ff7967a1345 |
| 0x4 | 16:a7 | E | 0x7ff7967a134c | 4 | 0x28ec8348 | 6236 | 0x7ff7967a134c |
| 0x5 | 16:a8 | E | 0x7ff7967a1350 | 5 | 0x797e8 | 6236 | 0x7ff7967a1350 |
| 0x6 | 16:ad | E | 0x7ff7967a1355 | 2 | 0xc085 | 6236 | 0x7ff7967a1355 |
| 0x7 | 16:ae | E | 0x7ff7967a1357 | 2 | 0x2174 | 6236 | 0x7ff7967a1357 |
| 0x8 | 16:af | E | 0x7ff7967a1359 | 9 | 0x3025048b4865 | 6236 | 0x7ff7967a1359 |
| 0x9 | 16:b0 | E | 0x7ff7967a1362 | 4 | 0x8488b48 | 6236 | 0x7ff7967a1362 |

# Advanced Demo 2: TTD.Calls queries

Same as running

dx -g $cursession.TTD.Calls(“library1!symbols”,...)

TTD Calls

Query 1

Query Parameters

Symbols: kernel32!*

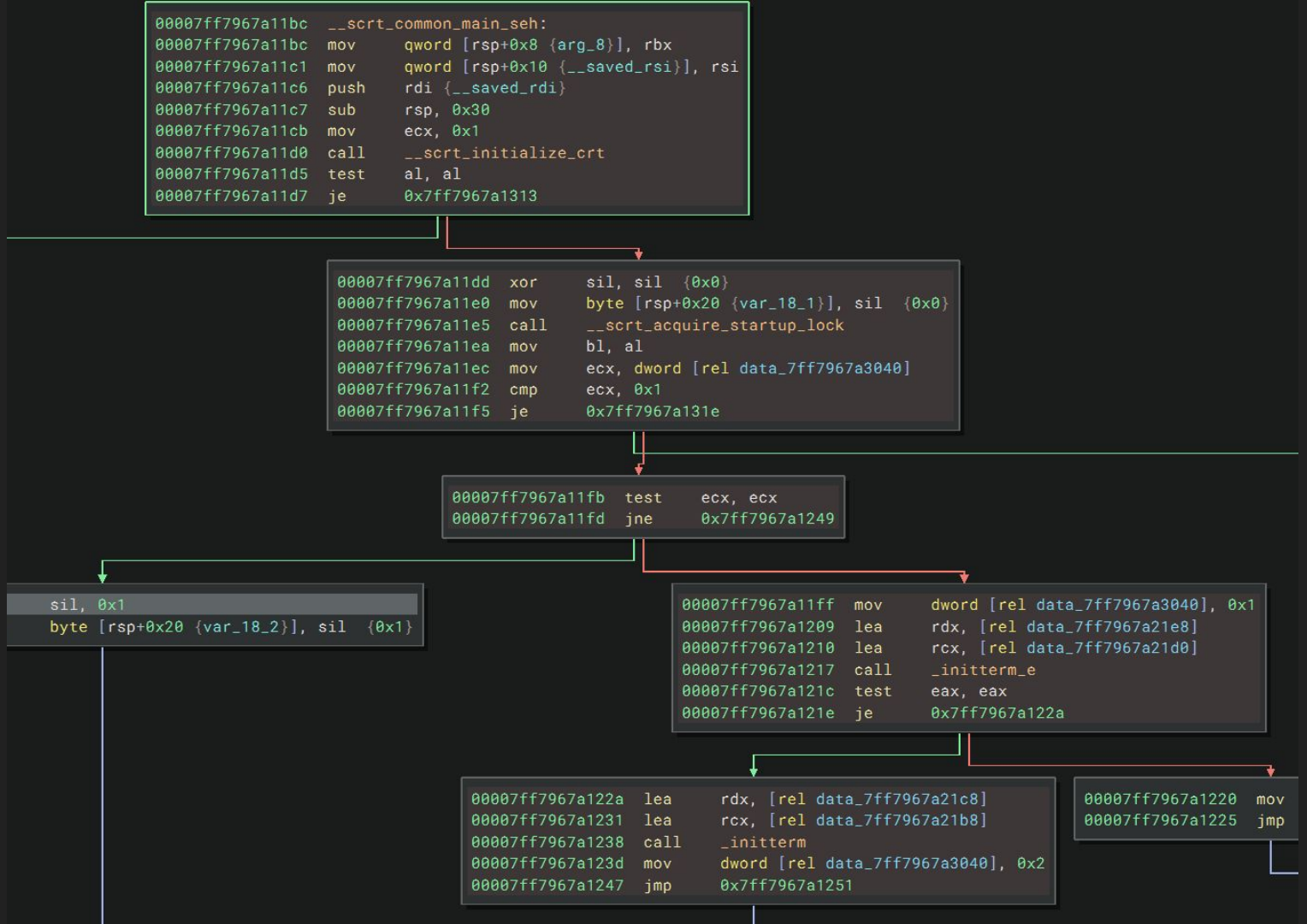
Return Address Range: Start address (hex, optional) to End address (hex, optional)

Query TTD Calls | Clear Results

| Index | Time Start | Time End | Function | Function Address | Return Address | Return Value | Parameters |
|---|---|---|---|---|---|---|---|
| 0 | 10:cc5 | 10:cc7 | KERNEL32!GetLastError | 0x7ff823a08640 | 0x7ff805715549 | 0xbb | {140703219908608, 0, 0, 0} |
| 1 | 10:11c7 | 10:11d6 | KERNEL32!SetLastErrorStub | 0x7ff823a18db0 | 0x7ff8057155d2 | 0xec66c55000 | {187, 2652456814000, 140703219992184, ... |
| 2 | 16:e | Max Position | KERNEL32!BaseThreadInitThunk | 0x7ff823a1e8c0 | 0x7ff82501c34c | 0x0 | {0, 140701358232376, 1015336484864, ... |
| 3 | 16:16 | Max Position | KERNEL32!... | 0x7ff823a76010 | 0x7ff823a1e8d7 | 0x0 | {1015336484864, 140701358232376, ... |
| 4 | 1c:4d | 1c:53 | KERNEL32!AreFileApisANSIStub | 0x7ff823a2f070 | 0x7ff8224ca7e4 | 0x1 | {0, 2652451110912, 922337203685477580... |
| 5 | 1d:a13 | 23:3f5 | KERNEL32!SetUnhandledExceptionFilterStub | 0x7ff823a33600 | 0x7ff7967a11a9 | 0x0 | {140701358233716, 140701358236104, 18, 2} |
| 6 | 2b:11e | 2b:126 | KERNEL32!GetModuleHandleWStub | 0x7ff823a2b450 | 0x7ff7967a181c | 0x7ff7967a0000 | {0, 2652451110912, 922337203685477580... |
| 7 | 4e:128 | 50:39a | KERNEL32!BasepProbeForDllManifest | 0x7ff823a15ae0 | 0x7ff825016c93 | 0xc000008a | {140703680495616, 2652456823184, ... |
| 8 | 4e:13d | 4e:171 | KERNEL32!IsFusionFullySupported | 0x7ff823a15bfc | 0x7ff823a15b2f | 0x1 | {140703680495616, 2652456823184, ... |
| 9 | 6b:12a | 6d:39a | KERNEL32!BasepProbeForDllManifest | 0x7ff823a15ae0 | 0x7ff825016c93 | 0xc000008a | {140703725518848, 2652456817200, ... |
| 10 | 6b:13f | 6b:173 | KERNEL32!IsFusionFullySupported | 0x7ff823a15bfc | 0x7ff823a15b2f | 0x1 | {140703725518848, 2652456817200, ... |
| 11 | ec:1492 | ec:14b6 | KERNEL32!GetStartupInfoWStub | 0x7ff823a2e4c0 | 0x7ff82395db8d | 0x0 | {1015339283856, 0, ... |
| 12 | 109:1587 | 109:159a | KERNEL32!IsProcessorFeaturePresentStub | 0x7ff823a2d9d0 | 0x7ff8239baa81 | 0x1 | {38, 18, 140703726164652, 0} |
| 13 | 109:159d | 109:15b0 | KERNEL32!IsProcessorFeaturePresentStub | 0x7ff823a2d9d0 | 0x7ff8239baa8e | 0x1 | {42, 18, 140703726164652, 0} |
| 14 | 109:15bf | 109:15d2 | KERNEL32!IsProcessorFeaturePresentStub | 0x7ff823a2d9d0 | 0x7ff8239baabd | 0x1 | {39, 10, 140703726164488, 0} |
| 15 | 109:15da | 109:15ed | KERNEL32!IsProcessorFeaturePresentStub | 0x7ff823a2d9d0 | 0x7ff8239baad8 | 0x1 | {40, 14, 140703726164488, 0} |

# Advanced Demo 3: Code coverage visualization

1. First party feature which is similar to lighthouse and BNCov
2. Collects a list of executed instructions from TTD.Memory(start, end, “e”)
3. Uses a render layer to rewrite the line highlight on the fly
4. Many cool possibilities
    a. Annotate the number of times a basic blocks has been take
    b. Use deeper colors to indicate a block has been executed more times
    c. Compare trace and do arithmetics on it


00007ff7967a11bc __scrt_common_main_seh:
00007ff7967a11bc mov qword [rsp+0x8 {arg_8}], rbx
00007ff7967a11c1 mov qword [rsp+0x10 {__saved_rsi}], rsi
00007ff7967a11c6 push rdi {__saved_rdi}
00007ff7967a11c7 sub rsp, 0x30
00007ff7967a11cb mov ecx, 0x1
00007ff7967a11d0 call __scrt_initialize_crt
00007ff7967a11d5 test al, al
00007ff7967a11d7 je 0x7ff7967a1313
00007ff7967a11dd xor sil, sil {0x0}
00007ff7967a11e0 mov byte [rsp+0x20 {var_18_1}], sil {0x0}
00007ff7967a11e5 call __scrt_acquire_startup_lock
00007ff7967a11ea mov bl, al
00007ff7967a11ec mov ecx, dword [rel data_7ff7967a3040]
00007ff7967a11f2 cmp ecx, 0x1
00007ff7967a11f5 je 0x7ff7967a131e
00007ff7967a11fb test ecx, ecx
00007ff7967a11fd jne 0x7ff7967a1249
sil, 0x1
byte [rsp+0x20 {var_18_2}], sil {0x1}
00007ff7967a11ff mov dword [rel data_7ff7967a3040], 0x1
00007ff7967a1209 lea rdx, [rel data_7ff7967a21e8]
00007ff7967a1210 lea rcx, [rel data_7ff7967a21d0]
00007ff7967a1217 call _initterm_e
00007ff7967a121c test eax, eax
00007ff7967a121e je 0x7ff7967a122a
00007ff7967a122a lea rdx, [rel data_7ff7967a21c8]
00007ff7967a1231 lea rcx, [rel data_7ff7967a21b8]
00007ff7967a1238 call _initterm
00007ff7967a123d mov dword [rel data_7ff7967a3040], 0x2
00007ff7967a1247 jmp 0x7ff7967a1251
00007ff7967a1220 mov
00007ff7967a1225 jmp

# TTD Case study

# Future Plans!

There are so many possibilities with TTD

Current integration only scratches the surface of what is possible

Advanced analysis can be built on of them

# Extract API calls and parameters

1. We can run a TTD.Calls() query
2. Apply filters to find calls that originate from a specific module
3. Time travel to it
4. Check the type library information of the called function
5. Dump all of the parameters accordingly

# Find self-modifying code

1. Query all of the executed instructions
2. Find all of the memory writes
3. Do a set intersection
4. You have found self-modifying code
5. Dump the binary on the 1st of such instructions

# Find buffers

1. Iterate over memory read/writes
2. Merge them when appropriate
3. We get buffers!
    a. They could be strings
    b. They could be PE files
    c. etc
4. We amy even run Yara rules or file command on it

# TTD as a capa backend

1. We can extract features from a TTD trace
2. https://github.com/mandiant/capa/issues/1649

# Taint tracking

1. We have memory/register modification information
2. We have semantics of the code (BNIL)
3. These combined can do a decent job on dynamic taint tracking
4. Answers questions like “How does this buffer form”

# Target Instrumentation

1. What if a malware detects it is being recorded?
2. TTD recording is often times believed to be opaque and unexaminable
3. This is not entirely true
4. We can inject ScyllaHide into the process being recorded
5. We could also repurpose CAPE’s debugger for automated instrumentation of the target (https://capev2.readthedocs.io/en/latest/usage/monitor.html)

# TTD internals

1. TTD does NOT record the execution at each instruction for performance reasons
2. If the effect of an instruction is deterministic, it does not need to be recorded
3. TTD uses advanced techniques to analyze the code and only do the recording when the effect of code execution cannot be predicted
4. For example, at a “read” system call, the content of the buffer is not predictable, so it must be recorded into the trace
5. Designing a reliable TTD implementation usually requires decades of time and many engineers

# Takeaways

1. Knows what TTD is and its primary advantage over traditional debugging
2. Know how to use TTD in Binary Ninja debugger
3. Try to leverage TTD more whenever possible

Thank you!

# Q & A

Contact: xusheng@vector35.com