Prerequisites

- Working C or C++ knowledge
- Basic assembly language knowledge
Audience

- Novices
  Learn x64 assembly language

- Experts
  Learn the new pattern approach
Pattern-Oriented RDR

- Complex crashes and hangs (victimware analysis)
- Malware analysis
- Studying new products
Training Goals

- Review fundamentals
- Learn patterns and techniques
Training Principles

- Talk only about what I can show
- Lots of pictures
- Lots of examples
- Original content and examples
Course Idea

- Implicit memory leak resulted from wrong API call parameter

- [www.Debugging.tv](http://www.Debugging.tv) episode 0x31
Schedule Summary

Day 1
- Theory
- Exercise R1

Day 2
- Exercises R2 – R4

Day 3
- Exercises R5 – R6
- Q&A
Part 1: Theory
Computation

Data → CPU → Code

Memory Changes
Disassembly

Data/Code numbers

Data/Code symbolic

488d0d2cceb0000 lea rcx,[CPUx64+0xe2f8 (00000001`3f85e2f8)] ; "Hello World!"

Annotated Disassembly memory analysis pattern
The Problem of Reversing

- Compilation to $\text{Machine Language}_M$
- Decompilation

$\text{Language}_1 \rightarrow \text{Language}_M \rightarrow \text{Language}_2$

$\text{Language}_M \rightarrow ?$
The Solution to Reversing

- Memory Language$^M_M$ Semantics

Language$_1$ $\rightarrow$ Language$_M^M$ $\leftarrow$ Language$_2$

- Decompilation

Understanding of Language$_M^M$
The Reversing Tool

<table>
<thead>
<tr>
<th>RSP</th>
<th>8</th>
<th>10</th>
<th>18</th>
<th>20</th>
<th>28</th>
<th>30</th>
<th>38</th>
<th>40</th>
<th>48</th>
<th>50</th>
</tr>
</thead>
</table>

Memory Cell Diagrams

<table>
<thead>
<tr>
<th>RAX</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

© 2013 Software Diagnostics Services
Re(De)construction

- Time dimension: sequence diagrams
- Space dimension: component diagrams

How does it work temporally and structurally?
ADDR Patterns

- Accelerated
- Disassembly patterns
- De(Re)construction patterns
- Reversing patterns
ADDR Patterns (II)

- Accelerated
- Disassembly patterns
- Decompilation patterns
- Reconstruction patterns
ADDR Schemas

- Function Prologue -> Function Epilogue
- Call Prologue -> Function Call -> Call Epilogue
- Potential Functionality -> Call Skeleton -> Call Path
- Call Parameter -> Function Parameter -> Local Variable
ADDR Implementations

ADDR Pattern Catalogue

- Windows
- Mac OS X
- Linux
Pattern Catalogues

- Elementary Software Diagnostics Patterns
- Memory Analysis Patterns
- Trace and Log Analysis Patterns
- Unified Debugging Patterns
- ADDR Patterns
Pattern Orientation

- Pattern-Driven ADDR
- Pattern-Based ADDR
Part 2: Practice Exercises
Links

- **Memory dumps:**
  Not available in preview version

- **Exercise Transcripts:**
  Not available in preview version
Exercise 0

- **Goal:** Install Debugging Tools for Windows and learn how to set up symbols correctly
Main CPU Registers

Illustrated on memory cell diagrams in `\ADDR\MCD-R1.xlsx`

- **RAX ⊃ EAX ⊃ AX ⊇ {AH, AL}**
- **ALU: RAX, RDX**
- **Counter: RCX**
- **Memory copy: RSI (src), RDI (dst)**
- **Stack: RSP**
- **Next instruction: RIP**
- **New: R8 – R15, Rx(D|W|B)**
Exercise R1

- **Goal:** Review x64 assembly fundamentals; learn how to reconstruct stack trace manually

- **ADDR Patterns:** Universal Pointer, Symbolic Pointer $S^2$, Interpreted Pointer $S^3$, Context Pyramid

- **Memory Cell Diagrams:** Register, Pointer, Stack Frame
1. Top frame from the current RIP\(_1\), RSP\(_1\) (r)
2. Disassemble around the current RIP\(_n\) (u[f] RIP\(_n\))
3. Find out the beginning of the function prologue
4. Check RSP\(_n\) usage (sub, push) and count offsets
5. Get RIP\(_{n+1}\) for the next frame (dps @rsp\(_n\) + offset)
6. Get RSP\(_{n+1}\) for the next frame (RSP\(_n\) +8)
7. ++n
8. goto #2
Exercise R2

- **Goal:** Learn how to map source code to disassembly

- **ADDR Patterns:** Potential Functionality, Function Skeleton, Function Call, Call Path, Local Variable, Static Variable, Pointer Dereference

- **Memory Cell Diagrams:** Pointer Dereference
Exercise R3

- **Goal:** Learn a function structure and associated memory operations

- **ADDR Patterns:** Function Prologue, Function Epilogue, Variable Initialization, Memory Copy

- **Memory Cell Diagrams:** Function Prologue, Function Epilogue
Exercise R4

- **Goal**: Learn how to recognize call and function parameters and track their data flow

- **ADDR Patterns**: Call Prologue, Call Parameter, Call Epilogue, Call Result, Control Path, Function Parameter, Structure Field
Exercise R5

- **Goal:** Master memory cell diagrams as an aid to understanding complex disassembly logic

- **ADDR Patterns:** Last Call, Loop, Memory Copy

- **Memory Cell Diagrams:** Memory Copy
Exercise R6

- **Goal:** Learn how to map code to execution residue and reconstruct past behaviour; recognise previously introduced ADDR patterns in the context of compiled C++ code

- **ADDR Patterns:** Separator Frames, Virtual Call

- **Memory Cell Diagrams:** Virtual Call
Live Debugging Techniques

- **ADDR Patterns:** Component Dependencies, API Trace, Fibre Bundle (trace analysis pattern)

- Some dependencies can be learnt from crash dump stack traces (*example*)

- **Debugging.TV / YouTube**

- Live debugging training: [Accelerated Windows Debugging³](#)
Custom Tracing

- WinDbg logging extension

- www.Debugging.tv episode 0x8 YouTube
Resources

- WinDbg Help / WinDbg.org (quick links)
- DumpAnalysis.org
- Debugging.TV / DebuggingTV YouTube Channel
- Windows Debugging: Practical Foundations
- x64 Windows Debugging: Practical Foundations
- Software Diagnostics Library
- Memory Dump Analysis Anthology

x86 disassembly/reversing:
Hackers Disassembly Uncovered by Kris Kaspersky
Q&A

Please send your feedback using the contact form on PatternDiagnostics.com
Thank you for attendance!