Reversing Disassembly Reconstruction

Accelerated

Version 3.0

Dmitry Vostokov
Software Diagnostics Services
Prerequisites

- Working C or classic C++ knowledge
- Basic assembly language knowledge
- Builds upon the book:
  
  [Practical Foundations of Windows Debugging, Disassembling, Reversing, 2nd Edition](#)
Audience

- Novices
  Improve x64 assembly language knowledge
- Experts
  Learn the new pattern language approach
Pattern-Oriented RDR

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

- Review fundamentals
- Review of x64 disassembly
- 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

- Debugging.TV episode 0x31
Part 1: Theory
Computation

Data → CPU → Code

Memory Changes
Disassembly

Data/Code numbers

Data/Code symbolic

488d0d2cce0000 lea rcx,[CPUx64+0xe2f8 (00000001`3f85e2f8)] ; "Hello World!"

Annotated Disassembly memory analysis pattern
The Problem of Reversing

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

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

- Decompilation

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

- **Memory Language** Semantics

Language_1 \rightarrow Language_M \leftarrow Language_2

- Decompilation

Understanding of Language_M
The Reversing Tool

Memory Cell Diagrams

Idea when reading The Mathematical Structure of Classical and Relativistic Physics: A General Classification Diagram book
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

macOS

Linux

x86

x64

ARM

ARM64
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: x64 Disassembly
x64 CPU Registers

- Illustrated in memory cell diagrams: `\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)**
Instructions and Registers

- **Opcode DST, SRC**

- **Examples:**

  mov rax, 10h ; RAX ← 0x10
  mov r13, rdx ; R13 ← RDX
  add r10, 10h ; R10 ← R10 + 0x10
  imul edx, ecx ; EDX ← EDX * ECX
  call rdx ; RDX already contains
             ; the address of func (&func)
  sub rsp, 30h ; PUSH RIP; &func → RIP
               ; RSP ← RSP–0x30
               ; make room for local variables
Memory and Stack Addressing

Stack grows

Lower addresses

RSP-0x20 → [RSP-0x20]
RSP-0x18 → [RSP-0x18]
RSP-0x10 → [RSP-0x10]
RSP-0x8 → [RSP-0x8]
RSP → [RSP]
RSP+0x8 → [RSP+0x8]
RSP+0x10 → [RSP+0x10]
RSP+0x18 → [RSP+0x18]
RSP+0x20 → [RSP+0x20]

Values

Higher addresses
Memory Cell Sizes

- RSP → BYTE PTR [RSP]
- RSP → DWORD PTR [RSP]
- RSP → QWORD PTR [RSP]
- RSP+0x8 → BYTE PTR [RSP]
- RSP+0x8 → DWORD PTR [RSP]
- RSP+0x8 → QWORD PTR [RSP]
Memory Load Instructions

- **Opcode** DST, PTR [SRC+Offset]
- **Opcode** DST

**Examples:**

- `mov rax, qword ptr [rsp+10h]` ; RAX \←
  ; 64-bit value at address RSP+0x10
- `mov ecx, dword ptr [20]` ; ECX \←
  ; 32-bit value at address 0x20
- `pop rdi` ; RDI \← value at address RSP
  ; RSP \← RSP + 8
- `lea r8, [rsp+20h]` ; R8 \← address RSP+0x20
Memory Store Instructions

- **Opcode** PTR [DST+Offset], SRC
- **Opcode** DST|SRC
- **Examples:**

  mov  qword ptr [rbp-20h], rcx ; 64-bit value at address RBP-0x20
  ; ← RCX
  mov  byte ptr [0], 1           ; 8-bit value at address 0 ← 1
  push rsi
  ; RSP ← RSP - 8
  inc  dword ptr [rcx]
  ; 32-bit value at address RCX ←
  ; 1 + 32-bit value at address RCX
Flow Instructions

- Opcode DST
- Opcode PTR [DST]

Examples:

```plaintext
jmp 00007ff6`9ef2f008 ; RIP ← 0x7ff69ef2f008
                 ; (goto 0x7ff69ef2f008)
jmp qword ptr [rax+10h] ; RIP ← value at address RAX+0x10
call 00007ff6`9ef21400 ; RSP ← RSP - 8
00007ff6`9ef21057: ; value at address RSP ← 0x7ff69ef21057
                    ; RIP ← 0x7ff69ef21400
                    ; (goto 0x7ff69ef21400)
```

© 2023 Software Diagnostics Services
Function Parameters

- x86: Right to left **PUSH**

  Args to Child are parameters

- x64: Left to right **RCX, RDX, R8, R9, stack**

  Args to Child are **not** parameters

---

WinDbg Commands

```
0:000> kv
   # Child-SP   RetAddr   : Args to Child   : Call Site
   ...
```
Function Call and Prolog

; void proc(int p1, long long p2);
mov edx, 2
mov ecx, 1
call proc
addr:

; void proc2();
; void proc(int p1, long long p2) {
;   long long local = 0;
;   proc2();
;
proc:
mov qword ptr [rsp+10h], rdx
mov dword ptr [rsp+8], ecx
sub rsp, 10h
mov qword ptr [rsp+8], 0
call proc2
adr2:
...
Function Epilog and Return

; void proc2();
; void proc(int p1, long long p2) {
;   long long local = 0;
;   proc2();
; }
proc:
  mov qword ptr [rsp+10h], rdx
  mov dword ptr [rsp+8], ecx
  sub rsp, 10h
  mov qword ptr [rsp+8], 0
  call proc2
adr2:
  add rsp, 10h
ret
Part 3: Practice Exercises
Links

- **Memory dumps:**
  Download links are in the exercise R0.

- **Exercise Transcripts:**
  Included in this book.
Exercise R0

- **Goal:** Install WinDbg or Debugging Tools for Windows, or pull Docker image, and check that symbols are set up correctly

- \ADD\Exercise-R0.pdf
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

- \\
- \\
- \\
- \\
- \ADDRT\Exercise-R1.pdf
- \ADDRT\MCD-R1.xlsx
Stack Reconstruction

1. Top frame from the current RIP\(1\), RSP\(1\) (r)
2. Disassemble around the current RIP\(n\) (ub or uf 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\(n\) + offset + 8)
7. ++n
8. goto #2

* If symbols are available, disassemble the function corresponding to RIP\(n\) (uf name)
ADDR: Universal Pointer

- A memory cell value interpreted as a pointer to memory cells
- A memory address that was not specifically designed as a pointer
ADDR: Symbolic Pointer, $S^2$

- A memory cell value associated with a symbolic value from a symbol file or a binary file (exported symbol)
ADDR: Interpreted Pointer, $S^3$

- Interpretation of a memory cell pointer value and its symbol
- Implemented via a typed structure or debugger (extension) command
ADDR: Context Pyramid

- When we move down stack trace frames, we can recover less and less contextual memory information due to register and memory overwrites.
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

- `\ADDR\Exercise-R2.pdf`

- `\ADDR\MCD-R2.xlsx`
A list of function symbols, for example, a list of imported functions, a list of callbacks, a structure or table with function pointers
ADDR: Function Skeleton

- Function calls inside a function body
- Splits a function body into regions
- Helps in understanding a function
ADDR: Function Call

- Simply the call of a function
- Call or jmp instruction
ADDR: Call Path

- Following a sequence of Function Calls
- Example: `call procA, call procC`

```plaintext
... call procA call procB ...
procA:
...
... call procC ...
```
ADDR: Local Variable

- A variable is a memory cell with an address
- A variable with stack region storage
- Usually, a local variable memory cell is referenced by stack pointer or frame pointer registers
ADDR: Static Variable

- A variable is a memory cell with an address
- A variable with non-stack and non-register storage
- Usually, there is a direct memory reference
A pointer is a memory cell that contains the address of another memory cell.

Dereference is a sequence of instructions to get a value from a memory cell referenced by another memory cell.
Exercise R3

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

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

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

- \`\`\ADDRAIN\Exercise-R3.pdf

- \ADDRAIN\MCD-R3.xlsx
ADDR: Function Prologue

- The code emitted by a compiler that is necessary to set up the working internals of a function
- Such code doesn’t have a real counterpart in actual source code
- Example: allocating memory on the stack for all local variables
ADDR: Function Epilogue

- The code emitted by a compiler that is necessary to finish the working internals of a function
- Such code doesn’t have a real counterpart in actual source code
- Example: deallocating memory on the stack for all local variables
ADDR: Variable Initialization

- Code to initialize an individual local variable
- Not part of a function prologue
ADDR: Memory Copy

- Repeated memory move instructions
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

- \ADDRLExercise-R4.pdf
ADDR: Call Prologue

- The code emitted by a compiler that is necessary to set up a function call and its parameters
ADDR: Call Parameter

- Data passed to a function before a function call
ADDR: Call Epilogue

- The code emitted by a compiler to finish a function call and its return results
ADDR: Call Result

- Data returned by a function
ADDR: Control Path

- A possible execution path inside a function consisting of direct and conditional jumps
**ADDR:** Function Parameter

- Data passed to a function inside a function (on the receiver side)
- Such a parameter can be translated to a local variable if passed by stack or copied to a stack location, or it can still be an original register or copied to another register
ADDR: Structure Field

- An offset to the structure memory address
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

- `\ADDR\Exercise-R5.pdf`

- `\ADDR\MCD-R5.xlsx`
ADDR: Last Call

- A function possibly called before the current instruction pointer
ADDR: Loop

- An unconditional jump to the previous code address
Virtual Function Call
class Base
{
    public: int method (int i) { return i; }
} myBase;

class Derived : public Base
{
    public: int method (int i) { return ++i; }
} myDerived;

myDerived.method(0);
myDerived.Base::method(0);

Base *pMyBase = &myDerived;
pMyBase->method(0);
class Base
{
public:
    int method (int i) { return i; }
    virtual int vmethod (int i) { return i; }
} myBase;

class Derived : public Base
{
public:
    int method (int i) { return ++i; }
    virtual int vmethod (int i) override { return ++i; }
} myDerived;

Base *pMyBase = &myDerived;
pMyBase->method(0);
pMyBase->vmethod(0);
pMyBase->Base::vmethod(0);
class Base {
public:
  virtual void vmethod1 () {}
  virtual void vmethod2 () {}
} myBase;

class Derived : public Base {
public:
  void vmethod2 () override {}
} myDerived;
VPTR and Class Memory Layout

Base *pMyBase = &myBase;
pMyBase->vmethod2();

myBase:  vptr (Base: : `vftable')

pMyBase = &myDerived;
pMyBase->vmethod2();

myDerived:  vptr (Derived `vftable')

0
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 classic C++ code

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

- **Memory Cell Diagrams:** Virtual Call

- `\ADDR\Exercise-R6.pdf`

- `\ADDR\MCD-R6.xlsx`
ADDR: Separator Frames

- Frames that divide a stack trace into separate analysis units
ADDR: Virtual Call

- A call through virtual function table structure field
- Usually involves a double Pointer Dereference
Live Debugging Techniques

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

- Some dependencies can be learned from crash dump stack traces

- [Debugging.TV](https://www.debugging.tv) / [YouTube](https://www.youtube.com)

- Live debugging training: [Accelerated Windows Debugging](https://www.accelerated-windows-debugging.com)
Memory Analysis Patterns

- Regular Data
- Injected Symbols
- Execution Residue
- Rough Stack Trace
- Annotated Disassembly
- Historical Information
Resources

- WinDbg Help / WinDbg.org (quick links)
- DumpAnalysis.org / SoftwareDiagnostics.Institute
- PatternDiagnostics.com
- Debugging.TV / YouTube.com/DebuggingTV / YouTube.com/PatternDiagnostics
- Practical Foundations of Windows Debugging, Disassembling, Reversing, Second Edition
- Accelerated Windows API for Software Diagnostics
- Accelerated C & C++ for Windows Diagnostics
- Memory Dump Analysis Anthology (Diagnomicon)
Q&A

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