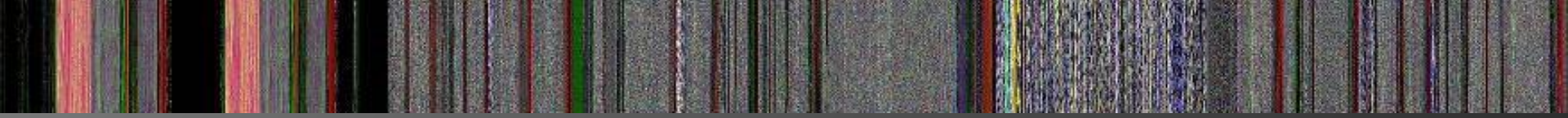


Linux Reversing Disassembly Reconstruction

Accelerated

Dmitry Vostokov
Software Diagnostics Services

Prerequisites

- Working C or classic C++ knowledge
- Basic assembly language knowledge
- Builds upon these books:

Practical Foundations of Linux Debugging, Disassembling, Reversing

Practical Foundations of ARM64 Linux Debugging, Disassembling, Reversing

Audience

- Novices

Improve x64 and ARM64 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
- Learn patterns and techniques

Training Principles

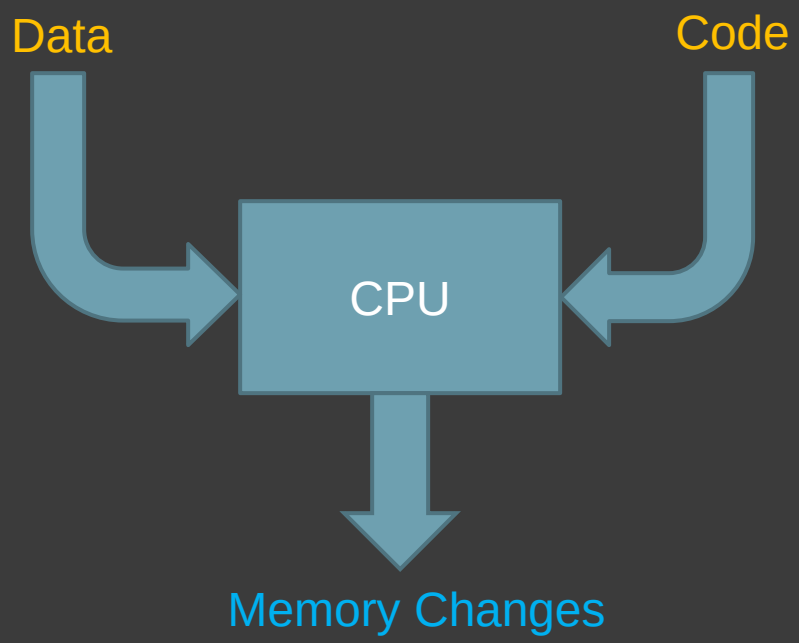
- Talk only about what I can show
- Lots of pictures
- Lots of examples
- Original content and examples

Course Idea

- Practical Foundations books for Linux (x64 and ARM64)
- Accelerated Linux Core Dump Analysis, Second Revised Edition (x64 and ARM64)
- Accelerated Disassembly, Reconstruction and Reversing, Second Edition, Revised and Extended (Windows x64)

Part 1: Theory

Computation



Disassembly

Data/Code numbers

Data/Code symbolic

```
48 8d 05 a1 b4 07 00    lea    0x7b4a1(%rip),%rax        # 0x47d004
48 89 05 36 68 0a 00    mov    %rax,0xa6836(%rip)        # 0x4a83a0 <name>

         e0 53 00 91    add    x0, sp, #0x14
         e0 0f 00 f9    str    x0, [sp, #24]
```

Annotated Disassembly memory analysis pattern

The Problem of Reversing

- Compilation to **M**achine Language$_M$

Language$_1$  Language$_M$  Language$_2$

- Decompilation

Language$_M$  ?

The Solution to Reversing

- **M**emory Language$_M$ Semantics

Language$_1$  Language$_M$  Language$_2$

- Decompilation

Understanding of Language$_M$

The Reversing Tool

RSP				
8				
10				
18				
20				
28				
30				
38				
40				
48				
50				

Memory Cell Diagrams

RAX							

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

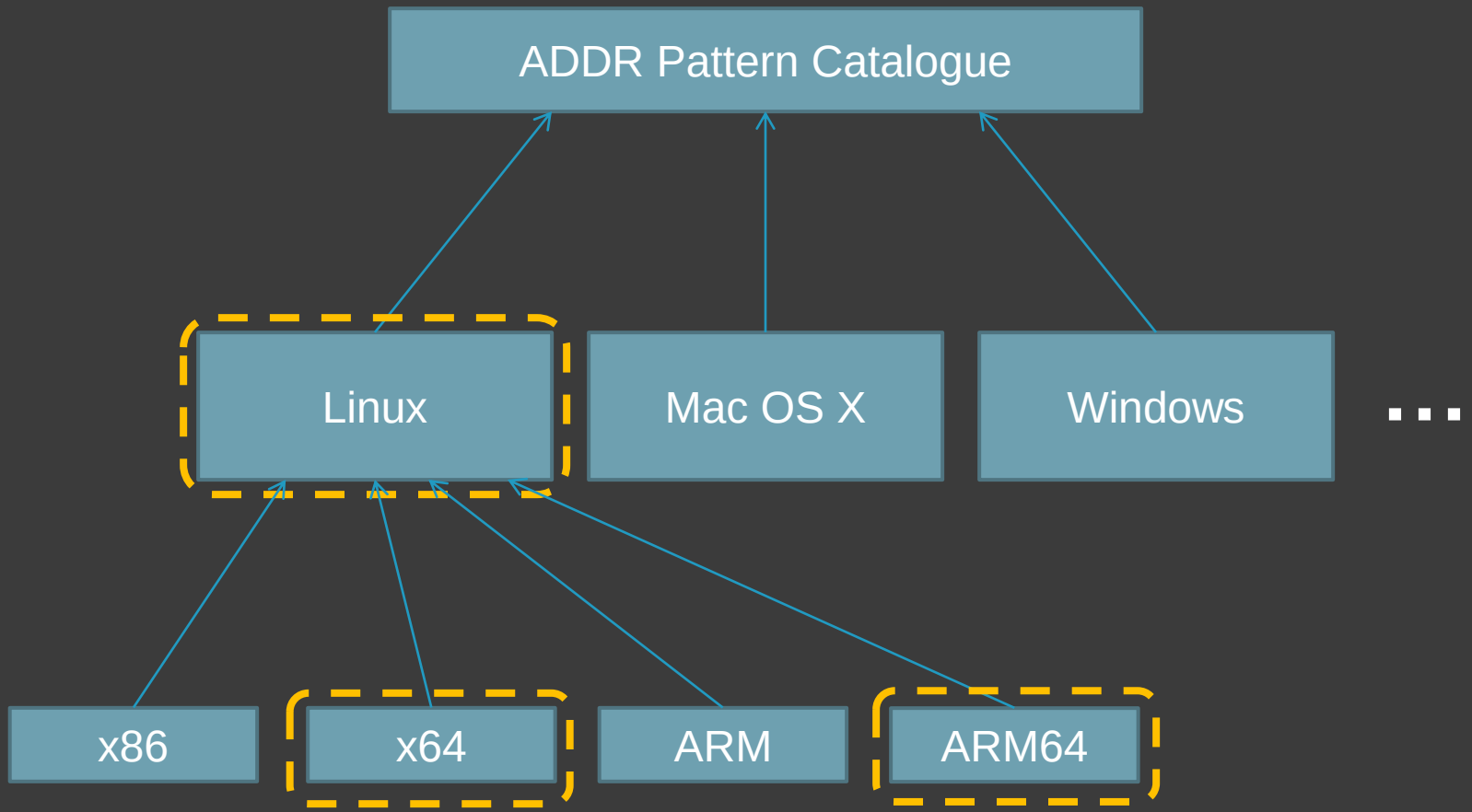
ADDR Patterns (II)

- 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



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:

Download links are in the exercise R0.

- Exercise Transcripts:

Included in this book.

Exercise R0

- **Goal:** Install GDB and check if GDB loads a core dump correctly
- \ADDR-Linux\Exercise-R0-x64-GDB.pdf
- \ADDR-Linux\Exercise-R0-ARM64-GDB.pdf

Main CPU Registers (x64)

Illustrated in memory cell diagrams: \ADDR-Linux\MCD-R1-x64.xlsx

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

Main CPU Registers (ARM64)

Illustrated in memory cell diagrams: \ADDR-Linux\MCD-R1-ARM64.xlsx

- X0 – X28, W0 – W28
- Stack: SP, FP (X29)
- Next instruction: PC
- Link register: LR (X30)
- Zero register: ZXR, WXR

Exercise R1

- **Goal:** Review x64 and ARM64 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
- \ADDR-Linux\Exercise-R1-x64-GDB.pdf
- \ADDR-Linux\MCD-R1-x64.xlsx
- \ADDR-Linux\Exercise-R1-ARM64-GDB.pdf
- \ADDR-Linux\MCD-R1-ARM64.xlsx

Stack Reconstruction (x64)

1. Top frame from the current RIP_1, RSP_1 (**info reg**)
2. Disassemble around the current RIP_n (**disass** 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 (**x/a** 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 (**disass name**)
  If symbols are not available, disassemble backwards until the function prologue is found

Stack Reconstruction (ARM64)

1. Top frame from the current PC_1, SP_1 (**info reg**)
2. Get PC_{n+1} for the next frame (**x/a** SP_n + 8)
3. Get SP_{n+1} for the next frame (**x/gx** SP_n)
4. ++n
5. goto #2

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:** Function Skeleton, Function Call, Call Path, Local Variable, Static Variable, Pointer Dereference
- **Memory Cell Diagrams:** Pointer Dereference
- \ADDR-Linux\Exercise-R2-x64-GDB.pdf
- \ADDR-Linux\MCD-R2-x64.xlsx
- \ADDR-Linux\Exercise-R2-ARM64-GDB.pdf
- \ADDR-Linux\MCD-R2-ARM64.xlsx

ADDR: Function Skeleton

- Function calls (or branch and links) inside a function body
- Splits a function body into regions
- Helps in understanding a function

ADDR: Function Call

- Simply the call of (or branch and link to) a function
- Call (bl, blr) or unconditional jmp (b) instructions

ADDR: Call Path

- Following a sequence of Function Calls
- Example: call procA, call procC (or bl procA, bl procC)

```
…
call procA
call procB
…

procA:
…
call procC
…
```

```
…
bl procA
bl procB
…

procA:
…
bl 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

ADDR: Pointer Dereference

- A pointer is a memory cell that contains the address of (references) 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 a function structure and associated memory operations
- **ADDR Patterns:** Function Prologue, Function Epilogue, Variable Initialization, Memory Copy
- **Memory Cell Diagrams:** Function Prologue, Function Epilogue
- \ADDR-Linux\Exercise-R3-x64-GDB.pdf
- \ADDR-Linux\MCD-R3-x64.xlsx
- \ADDR-Linux\Exercise-R3-ARM64-GDB.pdf
- \ADDR-Linux\MCD-R3-ARM64.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
- \ADDR-Linux\Exercise-R4-x64-GDB.pdf
- \ADDR-Linux\Exercise-R4-ARM64-GDB.pdf

ADDR: Call Prologue

- The code emitted by a compiler that is necessary to set up a function call (or branch and link) and its parameters

ADDR: Call Parameter

- Data passed to a function before a function call (or branch and link)

ADDR: Call Epilogue

- The code emitted by a compiler to finish a function call (or branch and link) and processing of 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 or branches

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

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-Linux\Exercise-R5-x64-GDB.pdf
- \ADDR-Linux\MCD-R5-x64.xlsx
- \ADDR-Linux\Exercise-R5-ARM64-GDB.pdf
- \ADDR-Linux\MCD-R5-ARM64.xlsx

ADDR: Last Call

- A function possibly called (or branched and linked to) before the current instruction pointer

ADDR: Loop

- An unconditional jump or branch to the previous code address

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:** Virtual Call

- **Memory Cell Diagrams:** Virtual Call

- \ADDR-Linux\Exercise-R6-x64-GDB.pdf
- \ADDR-Linux\MCD-R6-x64.xlsx

- \ADDR-Linux\Exercise-R6-ARM64-GDB.pdf
- \ADDR-Linux\MCD-R6-ARM64.xlsx

ADDR: Virtual Call

- A call (or branch and link) through virtual function table structure field
- Usually involves a double Pointer Dereference

Additional ADDR Patterns

ADDR: Potential Functionality

- A list of function symbols, for example, a list of imported functions, a list of callbacks, a structure or table with function pointers

ADDR: Structure Field

- An offset to the structure memory address

ADDR: Separator Frames

- Frames that divide a stack trace into separate analysis units

Live Debugging Techniques

- **ADDR Patterns:** Component Dependencies, API Trace, Fibre Bundle (trace analysis pattern)
- Some dependencies can be learnt from crash dump stack traces
- Debugging.TV / YouTube
- Live debugging training: Accelerated Linux Debugging[4]

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 Linux Debugging, Disassembling, Reversing
- Practical Foundations of ARM64 Linux Debugging, Disassembling, Reversing
- Accelerated Linux Disassembly, Reconstruction, and Reversing (WinDbg Version)
- Memory Dump Analysis Anthology (Diagnomicon)

Q&A

Please send your feedback using the contact form on PatternDiagnostics.com

Thank you for attendance!