Linux Reversing Disassembly Reconstruction

Accelerated

Second Edition

Dmitry Vostokov
Software Diagnostics Services
Prerequisites

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

- Novices
  Improve x64 (x86_64, AMD64) and A64 (AArch64, 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

- Accelerated Linux Core Dump Analysis, Third Edition (x64 and A64)

- Accelerated Disassembly, Reconstruction and Reversing, Second Edition, Revised and Extended (Windows x64)
Part 1: Theory
Computation

Data → CPU → Code

Memory Changes
Disassembly

Data/Code numbers

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 $\text{Machine Language}_M$

$\text{Language}_1 \xrightarrow{\text{}} \text{Language}_M \xleftarrow{\text{}} \text{Language}_2$

- Decompilation

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

- Memory Language \(_M\) Semantics

Language\(_1\) \(\rightarrow\) Language\(_M\) \(\leftarrow\) Language\(_2\)

- Decompilation

Understanding of Language\(_M\)
The Reversing Tool

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

Memory Cell Diagrams

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 $\rightarrow$ Function Epilogue
- Call Prologue $\rightarrow$ Function Call $\rightarrow$ Call Epilogue
- Potential Functionality $\rightarrow$ Call Skeleton $\rightarrow$ Call Path
- Call Parameter $\rightarrow$ Function Parameter $\rightarrow$ Local Variable
ADDR Implementations

ADDR Pattern Catalogue

Linux
macOS
Windows

x86
x64
ARM
A64

GCC
Clang
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
CPU Registers (x64)

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

- **RAX** $\supset$ **EAX** $\supset$ **AX** $\supseteq\{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)**
Instructions: registers (x64)

- **Opcode SRC, DST # default AT&T flavour**

- **Examples:**

  mov $0x10, %rax       # 0x10 → RAX
  mov %rsp, %rbp        # RSP → RBP
  add $0x10, %r10       # R10 + 0x10 → R10
  imul %ecx, %edx       # ECX * EDX → EDX
  callq *%rdx           # RDX already contains
                         # the address of func (&func)
  sub $0x30, %rsp       # RSP−0x30 → RSP
                         # make a room for local variables
Memory and Stack Addressing

Stack grows

Higher addresses

Lower addresses

RSP → RSP
RSP - 0x8 → RSP
RSP - 0x18 → RSP
RSP - 0x20 → RSP

← RBP
← RBP - 0x8
← RBP - 0x18
← RBP - 0x20
← RBP - 0x10
← RBP - 0x10
← RBP - 0x18
← RBP + 0x8
← RBP + 0x10
← RBP + 0x10
Instructions: memory load (x64)

- Opcode Offset(SRC), DST

- Opcode DST

Examples:

- `mov 0x10(%rsp), %rax`  # value at address RSP+0x10 → RAX
- `mov -0x10(%rbp), %rcx`  # value at address RBP-0x10 → RCX
- `add (%rax), %rdx`  # RDX + value at address RAX → RDX
- `pop %rdi`  # value at address RSP → RDI
- `pop %rdi`  # RSP + 8 → RSP
- `lea 0x20(%rbp), %r8`  # address RBP+0x20 → R8
Instructions: memory store (x64)

- **Opcode SRC, Offset(DST)**
- **Opcode SRC|DST**

**Examples:**

- `mov %rcx, -0x20(%rbp)`  # RCX → value at address RBP-0x20
- `addl $1, (%rax)`  # 1 + 32-bit value at address RAX → # 32-bit value at address RAX
- `push %rsi`  # RSP - 8 → RSP
  # RSI → value at address RSP
- `inc (%rcx)`  # 1 + value at address RCX → # value at address RCX
Instructions: flow (x64)

- **Opcode DST**

- **Examples:**

  jmp 0x10493fc1c  # 0x10493fc1c → RIP
                  # (goto 0x10493fc1c)

  call 0x10493ff74 # RSP - 8 → RSP

  0x10493fc14:     # 0x10493fc14 → value at address RSP
                  # 0x10493ff74 → RIP
                  # (goto 0x10493ff74)
# void proc(int p1, long p2);
mov $0x1, %edi
mov $0x2, %rsi
call proc
addr:

# void proc2();
# void proc(int p1, long p2) {
#  long local = 0;
#  proc2();
# }
proc:
push %rbp
mov %rsp, %rbp
sub $0x8, %rsp
mov $0, -0x8(%rbp)
call proc2
adr2:
...
Function Epilog and Return (x64)

```c
# void proc2();
# void proc(int p1, long p2) {
#     long local = 0;
#     proc2();
# }
proc:
push %rbp
mov  %rsp, %rbp
sub  $0x8, %rsp
mov  $0, -0x8(%rbp)
call proc2
adr2:
...
leaveq  # GCC
retq

adr2:
...
add $0x8, %rsp  # Clang
pop  $rbp
retq
```

Stack grows

Lower addresses

```
RSP-0x10 → RBP-0x20
RSP-0x8 → adr2 → RBP-0x18
RSP → 0 → RBP-0x8
RSP+0x8 → RBP → RBP
RSP+0x10 → addr → RBP+0x8
RSP → RBP
RSP+0x20 → RBP+0x18
RSP+0x28 → RBP+0x20
RSP+0x30 → RBP+0x28
```

Higher addresses

© 2023 Software Diagnostics Services
Stack Trace Reconstruction (x64)

(gdb) bt
func + 16
foo + 200
bar + 80
main + 300

Stack grows

Lower addresses

Return address: foo + 200
RIP == func + 16
RBP

Return address: bar + 80

Return address: main + 300

Part 3: A64 Disassembly
CPU Registers (A64)

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

- X0 – X28, W0 – W28
- Stack: SP, X29 (FP)
- Next instruction: PC
- Link register: X30 (LR)
- Zero register: XZR, WZR
- 64-bit floating point registers D0 – D31
- 128-bit Q0 – Q31
Instructions: registers (A64)

- **Opcode** DST, SRC, SRC₂

**Examples:**

- `mov x0, #16`  // X₀ ← 16 (0x10)
- `mov x29, sp`  // X₂₉ ← SP
- `add x1, x2, #16`  // X₁ ← X₂+16 (0x10)
- `mul x1, x2, x3`  // X₁ ← X₂*X₃
- `blr x8`  // X₈ already contains
  // the address of func (&func)
  // LR ← PC+4; PC ← &func
- `sub sp, sp, #48`  // SP ← SP-48 (-0x30)
  // make a room for local variables
Memory and Stack Addressing

Stack grows

Lower addresses

- SP-0x20 → ← X29-0x20
- SP-0x18 → ← X29-0x18
- SP-0x10 → ← X29-0x10
- SP-0x8 → ← X29-0x8
- SP → ← X29
- SP+0x8 → ← X29+0x8
- SP+0x10 → ← X29+0x10
- SP+0x18 → ← X29+0x18
- SP+0x20 → ← X29+0x20

Higher addresses
Instructions: memory load (A64)

- Opcode DST, DST₂, [SRC, Offset]
- Opcode DST, DST₂, [SRC], Offset // Postincrement

Examples:

```plaintext
ldr  x0, [sp]  // X0 ← value at address SP+0
ldr  x0, [x29, #-8]  // X0 ← value at address X29-0x8
ldp  x29, x30, [sp, #32]  // X29 ← value at address SP+32 (0x20)
                          // X30 ← value at address SP+40 (0x28)
ldp  x29, x30, [sp], #16  // X29 ← value at address SP+0
                          // X30 ← value at address SP+8
                          // SP ← SP+16 (0x10)
```
Instructions: memory store (A64)

- **Opcode** SRC, SRC₂, [DST, Offset]

- **Opcode** SRC, SRC₂, [DST, Offset]! // Preincrement

**Examples:**

```assembly
str x0, [sp, #16]    // x0 → value at address SP+16 (0x10)
str x0, [x29, #-8]    // x0 → value at address X29-8
stp x29, x30, [sp, #32] // x29 → value at address SP+32 (0x20)  
                          // x30 → value at address SP+40 (0x28)
stp x29, x30, [sp, #-16]! // SP ← SP-16 (-0x10)  
                          // x29 → set value at address SP  
                          // x30 → set value at address SP+8
```
Instructions: flow (A64)

- Opcode DST, SRC

- Examples:

  adrp x0, 0x420000 // x0 ← 0x420000

  b 0x10493fc1c // PC ← 0x10493fc1c
  // (goto 0x10493fc1c)

  br x17 // PC ← the value of X17

  0x10493fc14: // PC == 0x10493fc14

  bl 0x10493ff74 // LR ← PC+4 (0x10493fc18)
  // PC ← 0x10493ff74
  // (goto 0x10493ff74)
Function Call and Prolog (A64)

GCC

// void proc(int p1, long p2);
mov w0, #0x1
mov x1, #0x2
bl proc
addr:

// void proc2();
// void proc(int p1, long p2) {
//   long local = 0;
//   proc2();
// }
proc:
stp x29, x30, [sp, #-32]!
mov x29, sp
str zxr, [x29, #16]
bl proc2
adr2:

SP → X29 ← X29
SP-0x18 → X30 ← X29-0x18
SP+0x10 → 0 ← X29+16
SP-0x8 →       ← X29-0x8
SP →          ← X29
SP+0x8 →          ← X29+0x8
SP+0x10 →          ← X29+0x10
SP+0x18 →          ← X29+0x18
SP+0x20 →          ← X29+0x20

SP      →
SP-0x8  →
SP-0x18 →
SP      →
SP+0x8  →
SP+0x10 →
SP+0x18 →
SP+0x20 →

SP      →
SP-0x8  →
SP-0x18 →
SP      →
SP+0x8  →
SP+0x10 →
SP+0x18 →
SP+0x20 →

SP      →
SP-0x8  →
SP-0x18 →
SP      →
SP+0x8  →
SP+0x10 →
SP+0x18 →
SP+0x20 →

SP      →
SP-0x8  →
SP-0x18 →
SP      →
SP+0x8  →
SP+0x10 →
SP+0x18 →
SP+0x20 →

SP      →
SP-0x8  →
SP-0x18 →
SP      →
SP+0x8  →
SP+0x10 →
SP+0x18 →
SP+0x20 →

SP      →
SP-0x8  →
SP-0x18 →
SP      →
SP+0x8  →
SP+0x10 →
SP+0x18 →
SP+0x20 →

SP      →
SP-0x8  →
SP-0x18 →
SP      →
SP+0x8  →
SP+0x10 →
SP+0x18 →
SP+0x20 →

SP      →
SP-0x8  →
SP-0x18 →
SP      →
SP+0x8  →
SP+0x10 →
SP+0x18 →
SP+0x20 →

SP      →
SP-0x8  →
SP-0x18 →
SP      →
SP+0x8  →
SP+0x10 →
SP+0x18 →
SP+0x20 →

SP      →
SP-0x8  →
SP-0x18 →
SP      →
SP+0x8  →
SP+0x10 →
SP+0x18 →
SP+0x20 →
Function Epilog and Return (A64)

GCC

// void proc(int p1, long p2);
mov  w0, #0x1
mov  x1, #0x2
bl   proc
addr:

// void proc2();
// void proc(int p1, long p2) {
//   long local = 0;
//   proc2();
// }
proc:
stp  x29, x30, [sp, #0x32]!
mov  x29, sp
str  zxr, [x29, #0x16]
bl   proc
adr2:
...
ldp  x29, x30, [sp], #0x32
ret
Function Call and Prolog (A64)

Clang

```c
// void proc(int p1, long p2);
mov w0, #0x1
mov x1, #0x2
bl proc
addr:

// void proc2();
// void proc(int p1, long p2) {
//   long local = 0;
//   proc2();
// }
proc:
sub sp, sp, #0x20
stp x29, x30, [sp, #16]
add x29, sp, #0x10
str zxr, [x29, #-8]
bl proc2
adr2:
...
```

### Stack grows

- SP → SP - 0x8 → SP - 0x18 → SP + 0x8 → SP + 0x10 → SP + 0x18 → SP + 0x20 → ...
- ← X29 → ← X29 - 0x8 → ← X29 - 0x18 → ← X29 + 0x8 → ← X29 + 0x10 → ← X29 + 0x18 → ← X29 + 0x20 → ...

### Lower addresses
- SP → X29 - 0x20
- SP - 0x18 → ← X29 - 0x8
- SP + 0x10 → ← X29
- SP - 0x8 → ← X29 - 0x8
- SP → ← X29
- SP + 0x8 → ← X29 + 0x8
- SP + 0x10 → ← X29 + 0x10
- SP + 0x18 → ← X29 + 0x18
- SP + 0x20 → ← X29 + 0x20
Function Epilog and Return (A64)

Clang

// void proc(int p1, long p2);
mov  w0, #0x1
mov  x1, #0x2
bl   proc
addr:

// void proc2();
// void proc(int p1, long p2) {
//   long local = 0;
//   proc2();
// }
proc:
sub  sp, sp, #0x20
stp  x29, x30, [sp, #16]
add  x29, sp, #0x10
str  zxr, [x29, #0x10]
bl   proc2
adr2:
...
ldp  x29, x30, [sp, #16]
add  sp, sp, #0x20
ret

Clang
Stack grows

0x0 → X29
0x10 → X29
0x18 → X29
0x20 → X29
0x28 → X29
0x30 → X29
0x38 → X29
0x40 → X29

Lower addresses

SP   → X29-0x10
SP+0x8 → 0   ← X29-0x8
SP+0x10 → X29  ← X29
SP+0x18 → X30  ← X29+0x8
SP   → X29  ← X29
SP+0x28 → X29+0x18  ← X29+0x18
SP+0x30 → X29+0x20  ← X29+0x20
SP+0x38 → X29+0x28  ← X29+0x28
SP+0x40 → X29+0x30  ← X29+0x30

Higher addresses
Stack Trace Reconstruction (A64)

(gdb) bt
func + 16
foo + 200
bar + 80
main + 300

PC == func + 16,
LR == return address foo + 200

return address foo + 200

return address bar + 80

return address main + 300
Part 4: Practice Exercises
Links

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

- **Exercise Transcripts:**
  Included in this book.
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
Exercise R1

- **Goal:** Review x64 and AArch64 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

- \`\`\ADDRLinux\Exercise-R1-x64-GDB.pdf\`
- \`\ADDRLinux\MCD-R1-x64.xlsx\`

- \`\ADDRLinux\Exercise-R1-ARM64-GDB.pdf\`
- \`\ADDRLinux\MCD-R1-ARM64.xlsx\`
Stack Reconstruction (x64)

1. Top frame from the current RIP₁, RSP₁ (info reg)
2. Disassemble around the current RIPₙ (disass RIPₙ)*
3. Find out the beginning of the function prologue*
4. Check RSPₙ usage (sub, push) and count offsets
5. Get RIPₙ₊₁ for the next frame (x/a RSPₙ + offset)
6. Get RSPₙ₊₁ for the next frame (RSPₙ + offset + 8)
7. ++n
8. goto #2

* If symbols are available, disassemble the function corresponding to RIPₙ (disass name)
If symbols are not available, disassemble backwards until the function prologue is found
Stack Reconstruction (A64)

1. Top frame from the current PC$_1$, X29$_1$ (info reg)
2. Get PC$_{n+1}$ for the next frame ($x/a$ X29$_n$ + 8)
3. Get X29$_{n+1}$ for the next frame ($x/gx$ X29$_n$)
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
...  
```
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

- \ADDRLinux\Exercise-R4-x64-GDB.pdf

- \ADDRLinux\Exercise-R4-ARM64-GDB.pdf
ADDRESS: 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

- **Files:**
  - 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, Fiber Bundle (trace analysis pattern)

- Some dependencies can be learnt from crash dump stack traces

- Debugging.TV / YouTube

- Live debugging training: Accelerated Linux Debugging

© 2023 Software Diagnostics Services
Memory Analysis Patterns

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

- DumpAnalysis.org / SoftwareDiagnostics.Institute
- PatternDiagnostics.com
- Debugging.TV / YouTube.com/DebuggingTV / YouTube.com/PatternDiagnostics
- A64 Instruction Set Architecture
- A64 Base Instructions
- GDB Pocket Reference
- Accelerated Linux Core Dump Analysis, Third Edition
- Debugging, Disassembly & Reversing in Linux for x64 Architecture
- Foundations of Linux Debugging, Disassembling, and Reversing
- Foundations of ARM64 Linux Debugging, Disassembling, and Reversing
- Memory Dump Analysis Anthology (Diagnomicon) articles in volumes 1, 7, 9A cover GDB
Q&A

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