Linux Debugging

Accelerated

Dmitry Vostokov
Software Diagnostics Services
Prerequisites

- Debugging at source code level

or

- Basic crash dump analysis
GDB & LLDB, but Why WinDbg?

- The latest choice of a live debugger for Linux
- Second pair of eyes
- Debugging cross platform code
- WSL
Training Goals

- Review fundamentals
- Learn live debugging techniques
- See how software diagnostics is used during debugging
Training Principles

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

2. Accelerated Windows Debugging³ and Accelerated Windows Debugging⁴
Part 1: Fundamentals
Memory Space³
Execution Mode³
Code³
Live Debugging Technique

- Breakpoints
- Inspection
- Tracing

© 2024 Software Diagnostics Services
Pattern³
Debugging Paradigm³
Debugging Paradigm

Time Travel Debugging

Live

Dumps

Logs

© 2024 Software Diagnostics Services
Memory Spacetime
Debugging Paradigm

Idea: Kaluza-Klein Theory of a microscopic 5th dimension
Pattern Mapping

- Software Incident
- Elementary Diagnostics
- Software Diagnostics
- Memory Analysis
- Debugging
- Debugging Implementation
Elementary Diagnostics

 Functional
  • Use-case Deviation

 Non-functional
  • Crash
  • Hang (includes delays)
  • Counter Value (includes resource leaks, CPU spikes)
  • Error Message
Analysis Patterns

- **Memory Analysis catalog**
- **Software Trace and Log Analysis catalog**
Pattern-Oriented Diagnostic Analysis

**Diagnostic Pattern:** a common recurrent identifiable problem together with a set of recommendations and possible solutions to apply in a specific context.

**Diagnostic Problem:** a set of indicators (symptoms, signs) describing a problem.

**Diagnostic Analysis Pattern:** a common recurrent analysis technique and method of diagnostic pattern identification in a specific context.

**Diagnostics Pattern Language:** common names of diagnostic and diagnostic analysis patterns. The same language for any operating system: Windows, Mac OS X, Linux, ...

![Diagram of the diagnostic process]

- **Information Collection (Scripts)**
- **Information Extraction (Checklists)**
- **Problem Identification (Patterns)**
- **Problem Resolution**
  - Troubleshooting Suggestions
  - Debugging Strategy
Unified Debugging Patterns

- Analysis (software diagnostics)
- Architecture/Design of debugging
- Implementation of debugging
- Usage/presentation of debugging (for example, Watch dialog)
Full Debugging Patterns Catalog

Python Debugging for AI, Machine Learning, and Cloud Computing
A Pattern-Oriented Approach
Dmitry Vostokov

© 2024 Software Diagnostics Services
Lightweight Processes (Threads)

GDB Commands

- `info threads`
  Lists threads

- `thread <n>`
  Switches between threads

- `thread apply all bt`
  Lists stack traces from all threads

WinDbg Commands

- `~*k`
  Lists stack traces from all threads

- `~<n>s`
  Switches between threads
Thread Stack Raw Data

**GDB Commands**

\[x/<n>a \langle\text{address}\rangle\]
Prints \(n\) addresses with corresponding symbol mappings if any

**WinDbg Commands**

\[dps \langle\text{address}\rangle L\langle n\rangle\]
Prints \(n\) addresses with corresponding symbol mappings if any
Thread Stack Trace

FunctionD()
#0 0x00007fe9676bf48d in FunctionD ()
#1 0x00007fe9676bf300 in FunctionC ()
#2 0x00000000004005ca in FunctionB ()
#3 0x00000000004005da in FunctionA ()

GDB Commands

(gdb) bt
#0 0x00007fe9676bf48d in FunctionD ()
#1 0x00007fe9676bf300 in FunctionC ()
#2 0x00000000004005ca in FunctionB ()
#3 0x00000000004005da in FunctionA ()
# GDB vs. WinDbg vs. LLDB

## GDB Commands

```
(gdb) bt
#0 0x0000000000007fe9676bf48d in FunctionD ()
#1 0x0000000000000fe9676bf300 in FunctionC ()
#2 0x00000000000000004005ca in FunctionB ()
#3 0x00000000000000004005da in FunctionA ()
```

## WinDbg Commands

```
0:000> k
00 000007fe9676bf300 Module!FunctionD+offset
01 0000000000000004005ca Module!FunctionC+130
02 0000000000000a4005da AppA!FunctionB+220
03 0000000000000000 AppA!FunctionA+110
```

## LLDB Commands

```
(lldb) bt
frame #0: 0x0000000000000020328982a Module`FunctionD + offset
frame #1: 0x000000000203288a9c Module`FunctionC + 130
frame #2: 0x000000000a40da3ea9 AppA`FunctionB + 220
frame #3: 0x000000000a40da3ed9 AppA`FunctionA + 110
```
Thread Stack Trace (no symbols)

User Stack for LWP 10707

Return address FunctionC+130
0x00007fe9676bf300

Return address
0x00000000004005ca

Return address
0x00000000004005da

Symbol file App.sym
FunctionA 22000 - 23000
FunctionB 32000 - 33000

GDB Commands

(gdb) bt
#0 0x00007fe9676bf48d in FunctionD ()
#1 0x00007fe9676bf300 in FunctionC ()
#2 0x00000000004005ca in ?? ()
#3 0x00000000004005da in ?? ()
Exceptions (Seg Fault)

User Space (PID 3604)

App

Stack for LWP 3604 (TID)

ld.so

libpthread.so

Signal 11 (segmentation fault)

Signal 11 (segmentation fault)

Stack for LWP 3605 (TID)

libc.so

GDB Commands

(gdb) x <address>
0x<address>: Cannot access memory at address 0x<address>

WinDbg Commands

0:000> dp <address> L1
<address> ????????`????????

© 2024 Software Diagnostics Services
Exceptions (Runtime)

Stack for LWP 3714 (TID)

ld.so
libpthread.so

libstdc++.so

libc.so

Stack for LWP 3715 (TID)

User Space (PID 3714)

throws exception

App

Signal 6 (abort)

© 2024 Software Diagnostics Services
void main()
{
    foo();
    crash();
}

void foo()
{
    char sz[256] = "Some String";
    bar();
}

void bar()
{
    do();
}

void crash()
{
    // ...
}
Part 2: x64 Disassembly

GDB, LLDB
CPU Registers (x64)

- \textbf{RAX} \supseteq \textbf{EAX} \supseteq \textbf{AX} \supseteq \{\textbf{AH}, \textbf{AL}\}
  \begin{tabular}{|c|c|}
  \hline
  RAX 64-bit & EAX 32-bit \\
  \hline
  \end{tabular}

- ALU: \textbf{RAX}, \textbf{RDX}

- Counter: \textbf{RCX}

- Memory copy: \textbf{RSI} (src), \textbf{RDI} (dst)

- Stack: \textbf{RSP}, \textbf{RBP}

- Next instruction: \textbf{RIP}

- New: \textbf{R8} – \textbf{R15}, \textbf{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)
    # PUSH RIP; &func → RIP
  - `sub $0x30, %rsp`  # RSP−0x30 → RSP
    # make a room for local variables
Memory and Stack Addressing

Stack grows

Lower addresses

RSP-0x20 → ← RBP-0x20
RSP-0x18 → ← RBP-0x18
RSP-0x10 → ← RBP-0x10
RSP-0x8 → ← RBP-0x8
RSP → ← RBP
RSP+0x8 → ← RBP+0x8
RSP+0x10 → ← RBP+0x10
RSP+0x18 → ← RBP+0x18
RSP+0x20 → ← RBP+0x20

Higher addresses
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
- `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)
x64 Function Parameters

- `foo(...);`

- *Left to right via* `RDI, RSI, RDX, RCX, R8, R9, stack*
Part 3: A64 Disassembly
CPU Registers (A64)

- 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
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

© 2024 Software Diagnostics Services
Instructions: memory load (A64)

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

Examples:

1dr  x0, [sp]          // X0 ← value at address SP+0
1dr  x0, [x29, #-8]   // X0 ← value at address X29-0x8
1dp  x29, x30, [sp, #32] // X29 ← value at address SP+32 (0x20)
          // X30 ← value at address SP+40 (0x28)
1dp  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:

```
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)
A64 Function Parameters

- `foo(...);`

- Left to right via `X0 – X7, [SP], [SP+8], [SP+16], ...`
Part 4: x64 Disassembly WinDbg
CPU Registers (x64)

- **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|B)**

**RAX 64-bit**  **EAX 32-bit**
Instructions and Registers (x64)

- 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)
  ; PUSH RIP; RIP ← &func
  sub rsp, 30h ; RSP ← RSP−0x30
  ; make room for local variables

© 2024 Software Diagnostics Services
Memory and Stack Addressing

- 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]

- Higher addresses

Stack grows Higher addresses Lower addresses

© 2024 Software Diagnostics Services
Memory Cell Sizes

- RSP → BYTE PTR [RSP]
- RSP → DWORD PTR [RSP]
- RSP → QWORD PTR [RSP]
Memory Load Instructions (x64)

- 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
  lea r8, [rsp+20h] ; R8 ← address RSP+0x20
  ```
Memory Store Instructions (x64)

- **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 (x64)

- 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)
```
Function Parameters (x64)

- `foo(...);`
- **Left to right via** RDI, RSI, RDX, RCX, R8, R9, stack

Args to Child are **not** parameters

---

**WinDbg Commands**

```
0:000> kv
  # Child-SP   RetAddr   : Args to Child   : Call Site
  ...
```
Part 5: Practice Exercises
Links

- **Applications:**
  Download links are in every exercise.

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

Because of live debugging, due to differences in actual systems and ASLR (Address Space Layout Randomization), when you launch applications, actual addresses and even the number and order of threads in WinDbg, GDB, and LLDB command output may differ from those shown in exercise transcripts.
Exercise UDO

- **Goal:** Download and verify GDB, LLDB, and WinDbg installations
- **Memory Analysis Patterns:** Stack Trace
- \ALD4\Exercise-Linux-UD0.pdf
User Mode Debugging

Exercises UD1 – UD7
Exercise UD1

- **Goal:** Learn how code generation parameters can influence process execution behavior

- **Elementary Diagnostics Patterns:** Crash

- **Memory Analysis Patterns:** Exception Stack Trace; NULL Pointer (Code); Constant Subtrace

- **Debugging Implementation Patterns:** Break-in; Scope; Variable Value; Type Structure; Code Breakpoint

- \ALD4\Exercise-Linux-UD1.pdf
# Code Breakpoints

## GDB Commands
```
break <name>
info break
delete break <number>
```

## WinDbg Commands
```
bp <name>
bl
bc <number>
```

## LLDB Commands
```
break set -name <name>
break list
break delete <number>
```
Exercise UD2

- **Goal:** Learn how to use hardware breakpoints to catch data corruption

- **Elementary Diagnostics Patterns:** Counter Value

- **Memory Analysis Patterns:** Unloaded Module; Memory Leak (Process Heap); Corrupt Structure; Abnormal Value (*from trace analysis patterns*)

- **Debugging Implementation Patterns:** Break-in; Code Breakpoint; Scope; Variable Value; Data Breakpoint

- \ALD4\Exercise-Linux-UD2.pdf
# Data Breakpoints

## GDB Commands

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>watch *&lt;address&gt;</td>
</tr>
<tr>
<td>info break</td>
</tr>
<tr>
<td>delete break &lt;number&gt;</td>
</tr>
</tbody>
</table>

## LLDB Commands

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>watch set expression -- &lt;address&gt;</td>
</tr>
<tr>
<td>watch list</td>
</tr>
<tr>
<td>watch delete &lt;number&gt;</td>
</tr>
</tbody>
</table>
Exercise UD3

- **Goal:** Learn how to navigate parameters, static and local variables, and data structures

- **Elementary Diagnostics Patterns:** Crash

- **Memory Analysis Patterns:** Exception Stack Trace; Stack Overflow (User Mode); String Parameter; Module Variable

- **Debugging Implementation Patterns:** Break-in; Scope; Variable Value; Type Structure

- VALD4\Exercise-Linux-UD3.pdf
Exercise UD4

- **Goal:** Learn how to use conditional breakpoints to log behavior

- **Elementary Diagnostics Patterns:** Use-case Deviation

- **Memory Analysis Patterns:** -

- **Debugging Implementation Patterns:** Break-in; Code Breakpoint; Breakpoint Action

- \ALD4\Exercise-Linux-UD4.pdf
Exercise UD5

- **Goal:** Learn how to debug multiple processes and their deadlock

- **Elementary Diagnostics Patterns:** Crash; Hang

- **Memory Analysis Patterns:** Exception Stack Trace; Constant Subtrace; NULL Pointer (Data); Main Thread; Execution Residue (Unmanaged Space, User); C++ Exception; Hidden Exception (User Space); Handled Exception (User Space); Wait Chain (Mutex Objects); Deadlock (Objects, User Space); Coincidental Symbolic Information; Function Pointer

- **Debugging Implementation Patterns:** Break-in

- VALD4\Exercise-Linux-UD5.pdf
Expected Behavior

Process A
- Acquires Mutex A
- Waits for Mutex B
- Acquires Mutex B
- Releases Mutex B
- Waits for Mutex A
- Acquires Mutex B
- Releases Mutex B
- Releases Mutex A
- Acquires Mutex A

Process B
- Acquires Mutex B
- Releases Mutex B
- Waits for Mutex A
- Releases Mutex A
Deadlock

Process A

- Acquires Mutex A
- Waits for Mutex B

Process B

- Acquires Mutex B
- new_feature()
- Waits for Mutex A
Exercise UD6

- **Goal:** Learn how to recognize when we need kernel-level debugging

- **Elementary Diagnostics Patterns:** Hang; Counter Value

- **Memory Analysis Patterns:** Abnormal Value (*from trace analysis patterns*); Spiking Thread

- **Debugging Implementation Patterns:** Break-in; Code Breakpoint; Data Breakpoint; Code Trace

- \`\`\`ALD4\Exercise-Linux-UD6.pdf`\`\`
Exercise UD7

- **Goal:** Learn how to manipulate threads to debug race conditions

- **Elementary Diagnostics Patterns:** Crash

- **Memory Analysis Patterns:** Exception Stack Trace; NULL Pointer (Code)

- **Debugging Implementation Patterns:** Frozen Thread

- \ALD4\Exercise-Linux-UD7.pdf
Kernel Mode Debugging

Exercises KD8, KD10
Exercise KD0

- **Goal:** Set up Hyper-V kernel debugging environment

- `\ALD4\Exercise-Linux-KD0.pdf`
Exercise KD8

- **Goal:** Learn how to navigate kernel space using KDB

- **Elementary Diagnostics Patterns:** -

- **Memory Analysis Patterns:** Stack Trace; Stack Trace Collection (Unmanaged Space); Stack Trace Collection (CPUs); Module Collection; Execution Residue (Unmanaged Space, Kernel)

- **Debugging Implementation Patterns:** Code Breakpoint

- \ALD4\Exercise-Linux-KD8.pdf
Exercise KD10

- **Goal:** Learn how to configure and build Linux kernel and use GDB for kernel-level debugging

- **Elementary Diagnostics Patterns:** -

- **Memory Analysis Patterns:** Stack Trace

- **Debugging Implementation Patterns:** Code Breakpoint

- \ALD4\Exercise-Linux-KD10.pdf
Managed Debugging

Exercise MD9
Exercise MD9

- **Goal:** Learn how to debug Python code using GDB

- **Elementary Diagnostics Patterns:** Hang

- **Memory Analysis Patterns:** Stack Trace Collection (Unmanaged Space); Stack Trace Collection (Managed Space); Runtime Thread (Python, Linux); Managed Stack Trace (Python); Pointer Cone; Deadlock (Managed Space)

- **Debugging Implementation Patterns:** Break-in

- \ALD4\Exercise-Linux-MD9.pdf
Expected Behavior

Thread 2
- Enter cs1
- Exit cs1
- Enter cs2
- Exit cs2

Thread 3
- Enter cs2
- Waits for cs1
- Enter cs1
- Exit cs1
- Exit cs2
Deadlock

Thread #2

Rlock (cs1) 00007f568666ee40

Thread #2 (owns)

Thread #3 (waiting)

Rlock (cs2) 00007f5686603030

Thread #3

Thread #2 (waiting)
Time Travel Debugging

Exercise TD5
Exercise TD5

- **Goal:** Learn how to find hidden exceptions using Time Travel Debugging

- **Elementary Diagnostics Patterns:** Hang

- **Memory Analysis Patterns:** C++ Exception; Hidden Exception (User Space)

- **Debugging Implementation Patterns:** Instruction Trace

- \ALD4\Exercise-Linux-TD5.pdf
Rust Debugging

Exercise RD11
Exercise RD11

- **Goal:** Learn how WinDbg, GDB, and LLDB can be used to debug Rust applications

- **Elementary Diagnostics Patterns:** Error Message

- **Memory Analysis Patterns:** Stack Trace

- **Debugging Implementation Patterns:** Break-in; Code Breakpoint; Scope; Variable Value; Type Structure

- \ALD4\Exercise-Linux-RD11.pdf
Postmortem Debugging

- **Accelerated Linux Core Dump Analysis, Third Edition** (PDF book + Recording)
- **Accelerated Linux Core Dump Analysis** (Educative course)
Additional Training for Debugging

- Accelerated Linux API for Software Diagnostics
- Accelerated C & C++ for Linux Diagnostics
- Accelerated Linux Disassembly, Reconstruction and Reversing, Second Edition
- Memory Thinking for Rust
- Foundations of Linux Debugging, Disassembling, and Reversing
- Debugging, Disassembly & Reversing in Linux for x64 Architecture (Educative course)
- Foundations of ARM64 Linux Debugging, Disassembling, and Reversing
- Foundations of Linux ARM64: Debug, Disassemble, and Reverse (Educative course)
Resources

- [Software Diagnostics Library](https://SoftwareDiagnosticsLibrary.com)
- [Pattern-Driven Software Problem Solving](https://PatternDrivenProblemSolving.com)
- [Encyclopedia of Crash Dump Analysis Patterns, Third Edition](https://EncyclopediaOfCrashDumpAnalysisPatterns.com)
- [Memory Dump Analysis Anthology (Diagnomicon)](https://MemoryDumpAnalysisAnthology.com)
Q&A

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