macOS
Core Dump Analysis
Accelerated
Version 3.0

Dmitry Vostokov
Software Diagnostics Services
Prerequisites

Basic macOS troubleshooting

LLDB Commands
We use these boxes to introduce LLDB commands used in practice exercises
Training Goals

- Review fundamentals
- Learn how to collect core dumps
- Learn how to analyze core dumps
Training Principles

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

- **Day 1**
  - Analysis fundamentals (30 minutes)
  - Core dump collection methods (10 minutes)
  - Core dump analysis (1 hour 20 minutes)

- **Day 2**
  - ARM64 disassembly (30 minutes)
  - Core dump analysis (1 hour 30 minutes)
Part 1: Fundamentals
Memory/Kernel/User Space

- **Kernel Space**
- **User Space**
- **NULL Pointers**
Process Threads

Kernel Space

TID 0  TID 1
libsystem_kernel.dylib

User Space (PID 362)
libsystem_c.dylib

AppA

LLDB Commands

**thread list**
Lists threads

**thread select <n>**
Switches between threads

**thread backtrace all**
Lists stack traces from all threads
Thread Stack Raw Data

Kernel Space

User Space (PID 362)

libsystem_kernel.dylib

Stack for TID 0

Stack for TID 1

AppA

libsystem_c.dylib

TID 0

TID 1

LLDB Commands

`x/<n>a <address>`
Prints n addresses with corresponding symbol mappings if any
Thread Stack Trace

User Stack for TID 0

Return address FunctionC+130 0x000000020328a9c

Return address FunctionB+220 0x0000000104da3ea9

Return address FunctionA+110 0x0000000104da3edeb

FunctionA()
{
  ...
  FunctionB();
  ...
}
FunctionB()
{
  ...
  FunctionC();
  ...
}
FunctionC()
{
  ...
  FunctionD();
  ...
}
FunctionD()

Saves return address

Resumes from address FunctionC+130 0x000000020328a9c
Resumes from address FunctionB+220 0x0000000104da3ea9
Resumes from address FunctionA+110 0x0000000104da3edeb

LLDB Commands

(lldb) bt
frame #0: 0x000000020328982a Module`FunctionD + offset
frame #1: 0x000000020328a9c Module`FunctionC + 130
frame #2: 0x0000000104da3ea9 AppA`FunctionB + 220
frame #3: 0x0000000104da3edeb AppA`FunctionA + 110
GDB and LLDB vs. WinDbg

**GDB Commands**

```
(gdb) bt
#0 0x000000020328982a in FunctionD ()
#1 0x0000000203288a9c in FunctionC ()
#2 0x000000104da3ea9 in FunctionB ()
#3 0x000000104da3edb in FunctionA ()
```

**LLDB Commands**

```
(lldb) bt
frame #0: 0x000000020328982a Module`FunctionD + offset
frame #1: 0x0000000203288a9c Module`FunctionC + 130
frame #2: 0x000000104da3ea9 AppA`FunctionB + 220
frame #3: 0x000000104da3edb AppA`FunctionA + 110
```

**WinDbg Commands**

```
0:000> kn
00 0000000203288a9c Module!FunctionD+offset
01 0000000104da3ea9 Module!FunctionC+130
02 0000000104da3edb AppA!FunctionB+220
03 0000000000000000 AppA!FunctionA+110
```
Thread Stack Trace (no dSYM)

User Stack for TID 0

Symbol file AppA.dSYM
FunctionA 22000 - 23000
FunctionB 32000 - 33000

LLDB Commands

(lldb) bt
frame #0: 0x00007fff885e982a Module`FunctionD + offset
frame #1: 0x00007fff885e982a Module`FunctionC + 130
frame #2: 0x00007fff885e982a Module`FunctionB + 130
frame #3: 0x00007fff885e982a Module`FunctionA + 32220
frame #4: 0x00007fff885e982a Module`AppA + 22110
Exceptions (Access Violation)

User Space (PID 306)

AppA
libsystem_kernel.dylib
libA.dylib

TID 0
EXC_BAD_ACCESS (SIGSEGV)
NULL pointer 0x0

TID 1

Stack for TID 0
Stack for TID 1

LLDB Commands

(lldb) x <address>
error: core file does not contain 0x<address>
Exceptions (Runtime)
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: macOS, Linux, Windows, ...
Part 2: Core Dump Collection
Enabling Collection

- Temporary for the current terminal session:
  
  ```
  % ulimit -c unlimited
  ```

- Add entitlements:
  
  ```
  % /usr/libexec/PlistBuddy -c "Add :com.apple.security.get-task-allow bool true" tmp.entitlements
  ```

- Sign code:
  
  ```
  % codesign -s -f --entitlements tmp.entitlements YourApp
  ```

- Set permissions
  
  ```
  % sudo chmod 1777 /cores
  ```
Generation Methods

- **Command line:**
  
  `% sudo gcore PID`
  
  `% sudo kill -s SIGQUIT PID`
  
  `% sudo kill -s SIGABRT PID`

- **GUI:**
  
  Utilities \ Activity Monitor
  
  View \ Send Signal to Process…
Part 3: ARM64 Disassembly
CPU Registers (ARM64)

- X0 – X28, W0 – W28
- Stack: SP, X29 (FP)
- Next instruction: PC
- Link register: X30 (LR)
- Zero register: XZR, WZR
- 128-bit V0 – V31 (Q0 – Q31)
Instructions: registers (ARM64)

- **Opcodes**: DST, SRC, SRC₂

**Examples:**

- `mov x0, #0x10`  
  \( X₀ ← 0x10 \)

- `mov x29, sp`  
  \( X₂⁹ ← SP \)

- `add x1, x2, #0x10`  
  \( X₁ ← X₂+0x10 \)

- `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, #0x30`  
  \( SP ← SP-0x30 \)  
  \( \) make a room for local variables
Memory and Stack Addressing

Stack grows

Higher addresses

Lower addresses

SP $-0x20 \rightarrow \leftarrow X29-0x20$

SP $-0x18 \rightarrow \leftarrow X29-0x18$

SP $-0x10 \rightarrow \leftarrow X29-0x10$

SP $-0x8 \rightarrow \leftarrow X29-0x8$

SP $\rightarrow \leftarrow X29$

SP $+0x8 \rightarrow \leftarrow X29+0x8$

SP $+0x10 \rightarrow \leftarrow X29+0x10$

SP $+0x18 \rightarrow \leftarrow X29+0x18$

SP $+0x20 \rightarrow \leftarrow X29+0x20$
Instructions: memory load (ARM64)

- **Opcode** DST, DST₂, [SRC, Offset]
- **Opcode** DST, DST₂, [SRC], Offset ; Postincrement

**Examples:**

1. `ldr x0, [sp]` ; X₀ ← value at address SP+0
2. `ldr x0, [x29, #0x8]` ; X₀ ← value at address X29-0x8
3. `ldp x29, x30, [sp, #0x20]` ; X29 ← value at address SP+0x20
   ; X30 ← value at address SP+0x28
4. `ldp x29, x30, [sp], #0x10` ; X29 ← value at address SP+0
   ; X30 ← value at address SP+0x8
   ; SP ← SP+0x10
Instructions: memory store (ARM64)

- **Opcodes**
  - SRC, SRC₂, [DST, Offset]
  - SRC, SRC₂, [DST, Offset]! ; Preincrement

- **Examples:**
  - `str x0, [sp, #0x10]` ; `x0 → value at address SP+0x10`
  - `str x0, [x29, #-0x8]` ; `x0 → value at address X29-0x8`
  - `stp x29, x30, [sp, #0x20]` ; `x29 → value at address SP+0x20`
    ; `x30 → value at address SP+0x28`
  - `stp x29, x30, [sp, #-0x10]!` ; `SP ← SP-0x10`
    ; `x29 → set value at address SP`
    ; `x30 → set value at address SP+0x8`
Instructions: flow (ARM64)

- **Opcode DST, SRC**

- **Examples:**

  adr
  
x0, 2 ; x0 ← PC&0xFFFFFFFFFFFFFFFF000 + 0x1000*2

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

  0x10493fc14:
  ; PC == 0x10493fc14

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

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

; void proc2();
; void proc(int p1, long p2) {
;   long local = 0;
;   proc2();
; }
proc:
sub sp, sp, #0x20
stp x29, x30, [sp, #0x10]
add x29, sp, #0x10
str zxr, [x29, #-0x8]
bl proc2
...

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

; void proc2();
; void proc(int p1, long p2) {
;   long local = 0;
;   proc2();
; }
proc:
sub sp, sp, #0x20
stp x29, x30, [sp, #0x10]
add x29, sp, #0x10
str zxr, [x29, #-0x8]
bl proc2
...
Stack Trace Reconstruction

```
return address foo + 200
← X29
Stack grows

PC == func + 16,
LR == return address foo + 200
← X29
return address foo + 200

return address bar + 80

return address main + 300

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

Lower addresses

Higher addresses
Pointer Authentication

- $0x823d80018ea0308c$ ($0x000000018ea0308c$) Module\`func + 92
- (lldb) x/a $0x823d80018ea0308c$
  $0x18ea0308c$: $0x0b000269320107e8$

Problem of reading real addresses:

- (lldb) x/a $0xa000000$
  error: core file does not contain $0xa000000$
- (lldb) x/a $0xa00060000a000000$
  error: core file does not contain $0x20000000$

Solution via Typed Memory:

- (lldb) p/x *(long *)$0x0000000000000000$
  (long) $\$1 = 0x0000000000000000$
- (lldb) parray/x 10 (long *)$0x0000000000000000$
Part 4: Practice Exercises
Links

- **Memory Dumps:**
  Included in Exercise X0

- **Exercise Transcripts:**
  Included in this book
Exercise X0

- **Goal**: Install Xcode and check if LLDB loads a core dump correctly, compare the core dump backtrace with a diagnostic report

- **Patterns**: Stack Trace; Incorrect Stack Trace

- \AMCDA-Dumps\Exercise-X0.pdf
Process Core Dumps

Exercises X1 – X12
Exercise X1

- **Goal:** Learn how to list stack traces, disassemble functions, follow function calls, check backtrace correctness, dump data, get environment

- **Patterns:** Manual Dump (Process); Stack Trace; Stack Trace Collection; Annotated Disassembly; Paratext; Not My Version; Environment Hint

- [AMCDA-Dumps\Exercise-X1.pdf](AMCDA-Dumps\Exercise-X1.pdf)
Exercise X2

- **Goal:** Learn how to identify multiple exceptions, find problem CPU instructions

- **Patterns:** Multiple Exceptions (User Mode); NULL Pointer (Data); NULL Pointer (Code)

- \AMCDA-Dumps\Exercise-X2.pdf
Exercise X3

- **Goal:** Learn how to identify spiking threads

- **Patterns:** Spiking Thread

- `\AMCDA-Dumps\Exercise-X3.pdf`
Exercise X4

- **Goal:** Learn how to identify heap regions and heap corruption

- **Patterns:** Dynamic Memory Corruption (Process Heap)

- \AMCDA-Dumps\Exercise-X4.pdf
Exercise X5

- **Goal:** Learn how to identify stack corruption

- **Patterns:** Truncated Stack Trace; Local Buffer Overflow; Execution Residue; Self-Diagnosis (User Mode)

- \AMCDA-Dumps\Exercise-X5.pdf

© 2022 Software Diagnostics Services
Exercise X6

- **Goal:** Learn how to identify stack overflow, stack boundaries, reconstruct stack trace
- **Patterns:** Stack Overflow; Execution Residue
- AMCDA-Dumps\Exercise-X6.pdf
Exercise X7

- **Goal:** Learn how to identify active threads

- **Patterns:** Missing Thread; Active Thread; Near Exception

- \AMCDA-Dumps\Exercise-X7.pdf
Exercise X8

- **Goal:** Learn how to identify runtime exceptions, past execution residue and stack traces, identify handled exceptions

- **Patterns:** C++ Exception; Execution Residue; Coincidental Symbolic Information; Handled Exception

- \`\`\`AMCDA-Dumps\Exercise-X8.pdf`\`\`
Exercise X9

- **Goal:** Learn how to identify heap leaks

- **Patterns:** Heap Leak; Execution Residue; Module Hint

- \AMCDA-Dumps\Exercise-X9.pdf
Exercise X10

- **Goal:** Learn how to identify contention wait chains, synchronization issues, advanced disassembly, dump arrays

- **Patterns:** Double Free; High Contention; Wait Chain; Critical Region; Self-Diagnosis

- \AMCDA-Dumps\Exercise-X10.pdf
Exercise X11

- **Goal:** Learn how to identify synchronization wait chains, deadlocks, hidden and handled exceptions

- **Patterns:** Wait Chains; Deadlock; Execution Residue; Handled Exception

- \AMCDA-Dumps\Exercise-X11.pdf
Exercise X12

- **Goal**: Learn how to dump memory for post-processing, get the list of functions and module variables, load symbols, inspect arguments and local variables, list symbols, inspect types, search memory

- **Patterns**: Module Variable

- \AMCDA-Dumps\Exercise-X12.pdf
Resources

- DumpAnalysis.org / SoftwareDiagnostics.Institute / PatternDiagnostics.com
- Debugging.TV / YouTube.com/DebuggingTV / YouTube.com/PatternDiagnostics
- Accelerated Linux Core Dump Analysis, Second Edition (ARM64, WinDbg)
- Accelerated Linux Core Dump Analysis, Third Edition (ARM64, GDB)
- Accelerated Linux Disassembly, Reconstruction and Reversing (ARM64, GDB)
- The LLDB Debugger
- A64 Instruction Set Architecture
- A64 Base Instructions
- GDB to LLDB Command Map
- WinDbg and LLDB Commands
- LLDB Cheat Sheet
- Enable core dumps on Mac OS X Monterey M1 Pro
- PAC it up: Towards Pointer Integrity using ARM Pointer Authentication
- iOS Crash Dump Analysis, Second Edition
- Advanced Apple Debugging & Reverse Engineering: Exploring Apple code through LLDB, Python, and DTrace, Third Edition
- Foundations of ARM64 Linux Debugging, Disassembling, Reversing (Apress)
- Practical Foundations of macOS Debugging, Disassembling, Reversing (forthcoming)
Further Training

Accelerated macOS (M2) Disassembly, Reconstruction, and Reversing
Q&A

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