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

© 2022 Software Diagnostics Services

Training Goals

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

© 2022 Software Diagnostics Services

Training Principles

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

© 2022 Software Diagnostics Services

Schedule Summary

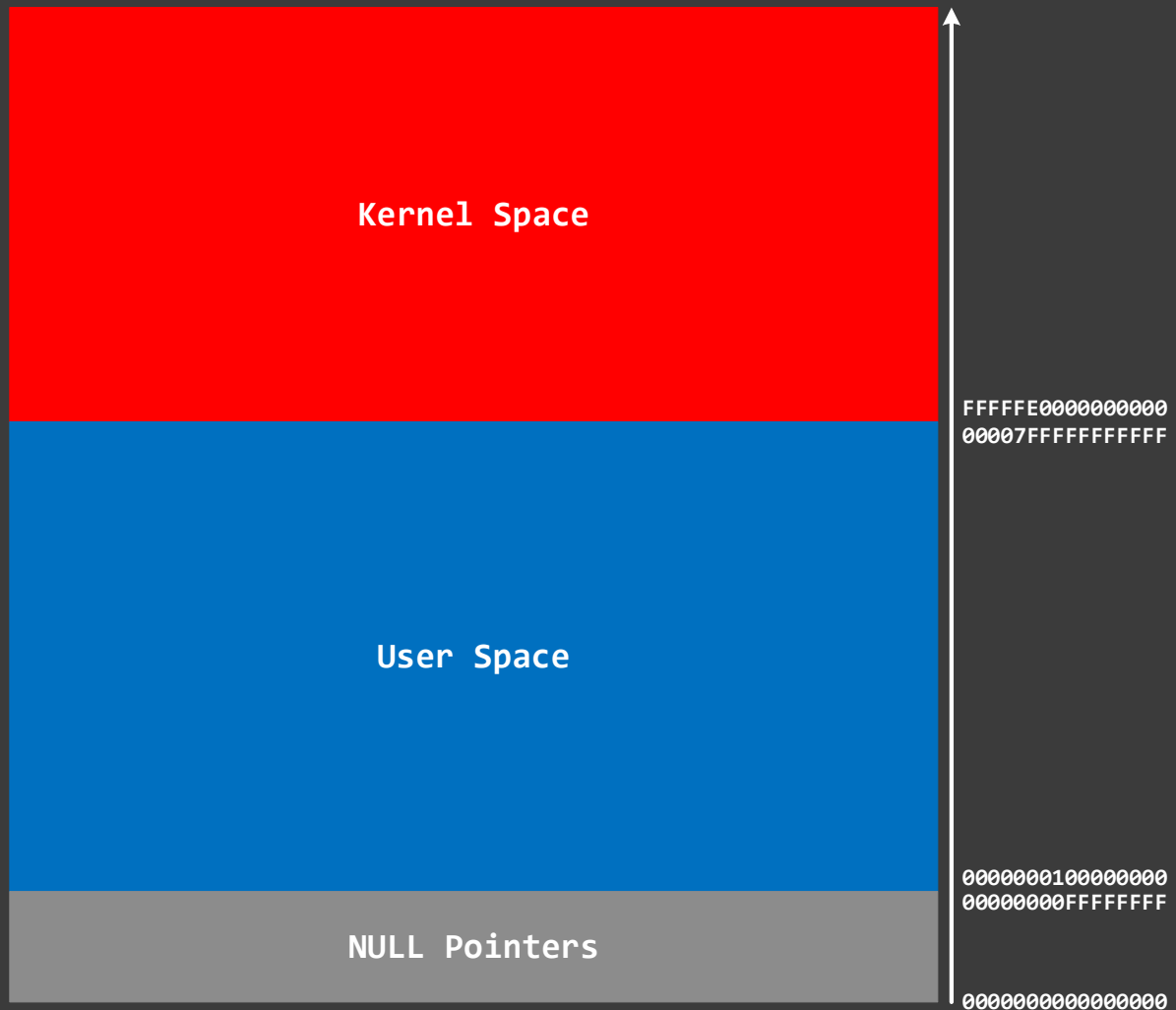
- Day 1
  - Analysis fundamentals (30 minutes)
  - Core dump collection methods (10 minutes)
  - Core dump analysis (1 hour 20 minutes)
- Day 2
  - ARM64 disassembly (30 minutes)
  - Core dump analysis (1 hour 30 minutes)

© 2022 Software Diagnostics Services

Part 1: Fundamentals

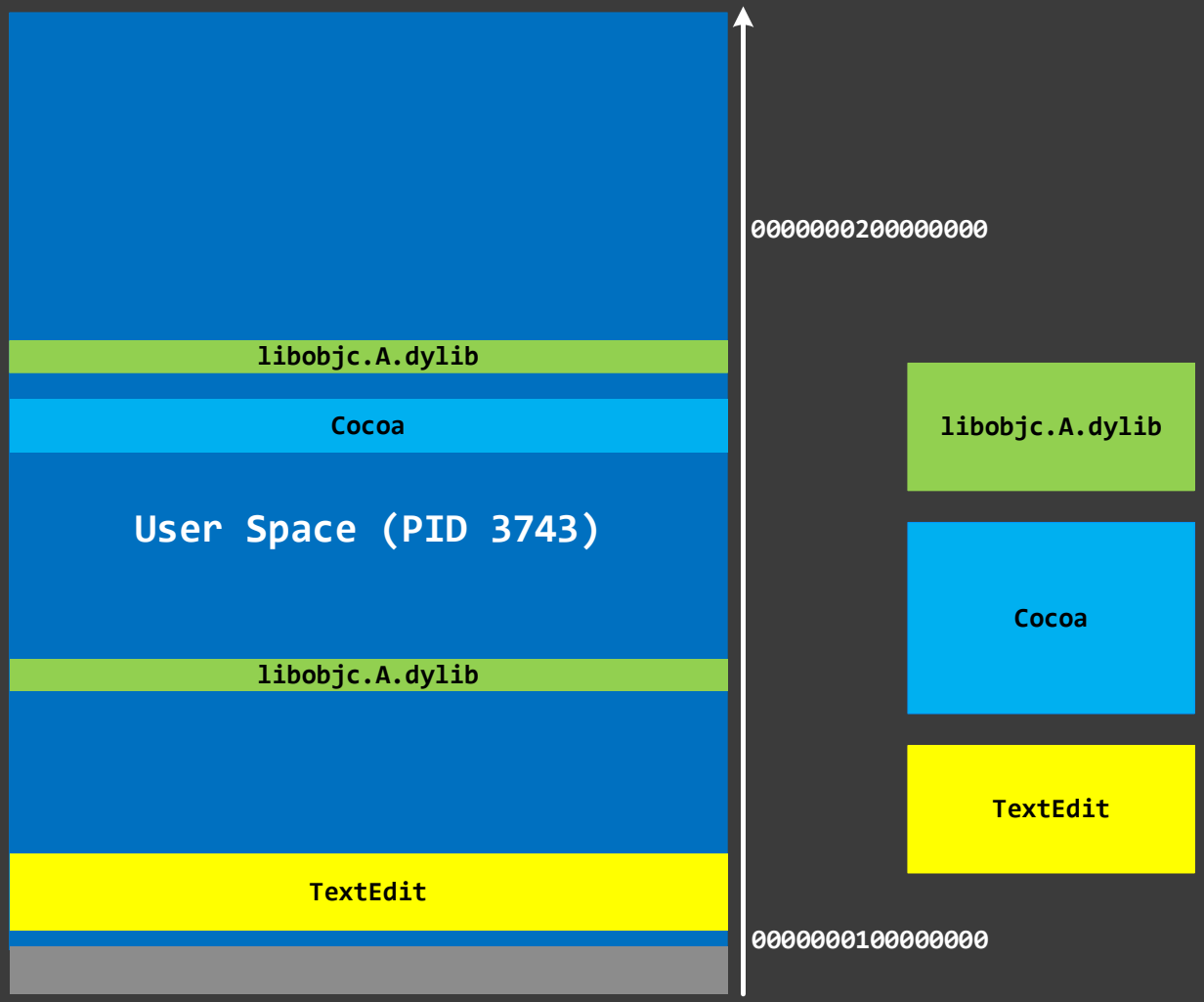
© 2022 Software Diagnostics Services

Memory/Kernel/User Space



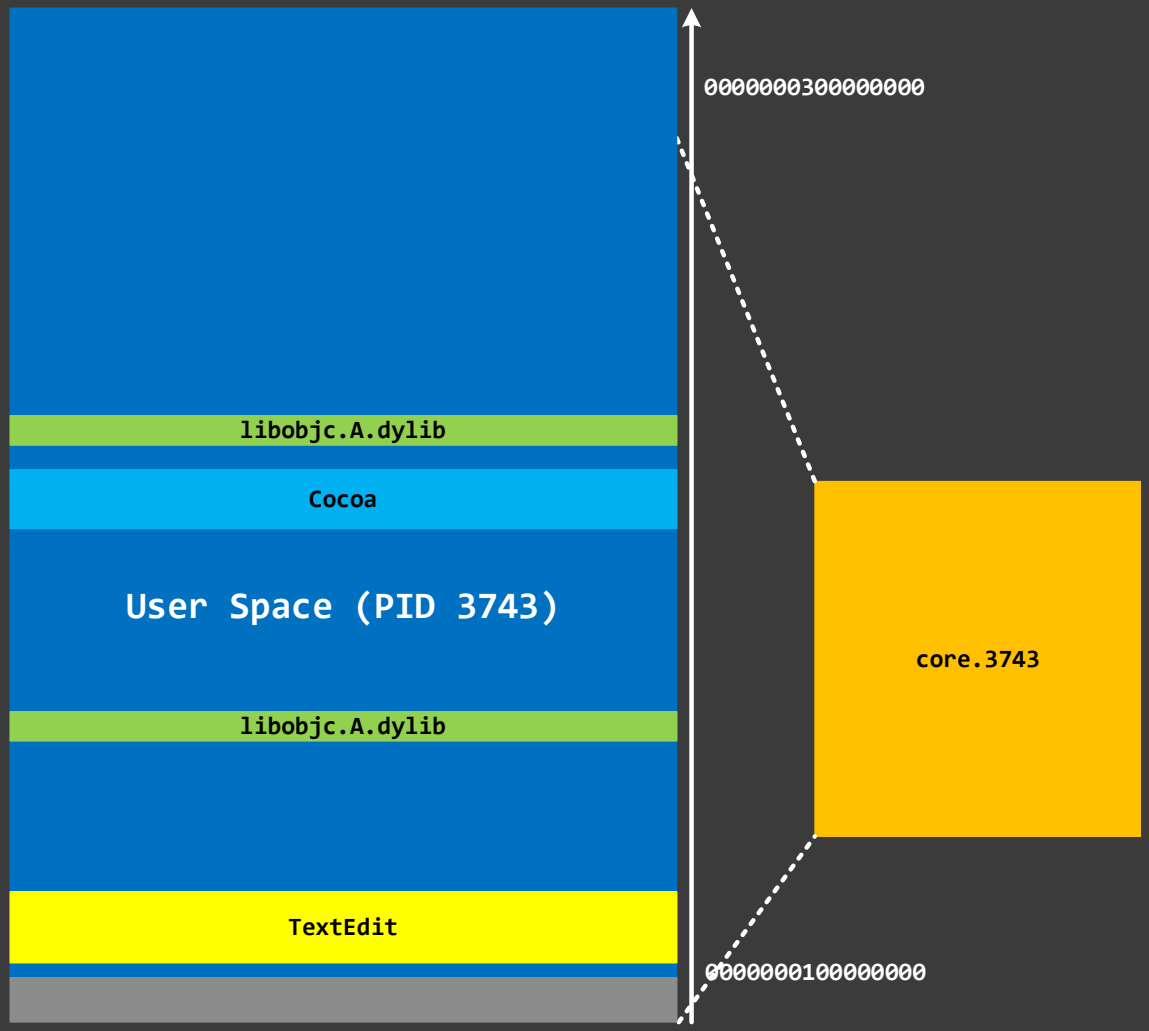
© 2022 Software Diagnostics Services

App/Process/Library



© 2022 Software Diagnostics Services

Process Memory Dump



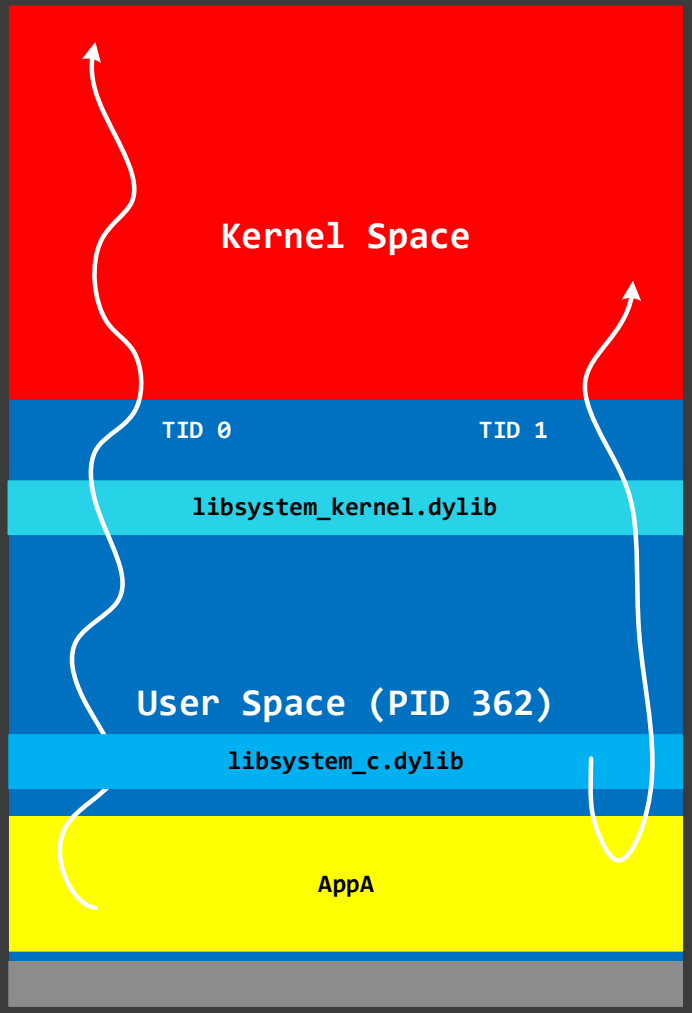
LLDB Commands

image list
Lists dynamic libraries

image dump sections
Lists memory regions

© 2022 Software Diagnostics Services

Process Threads



LLDB Commands

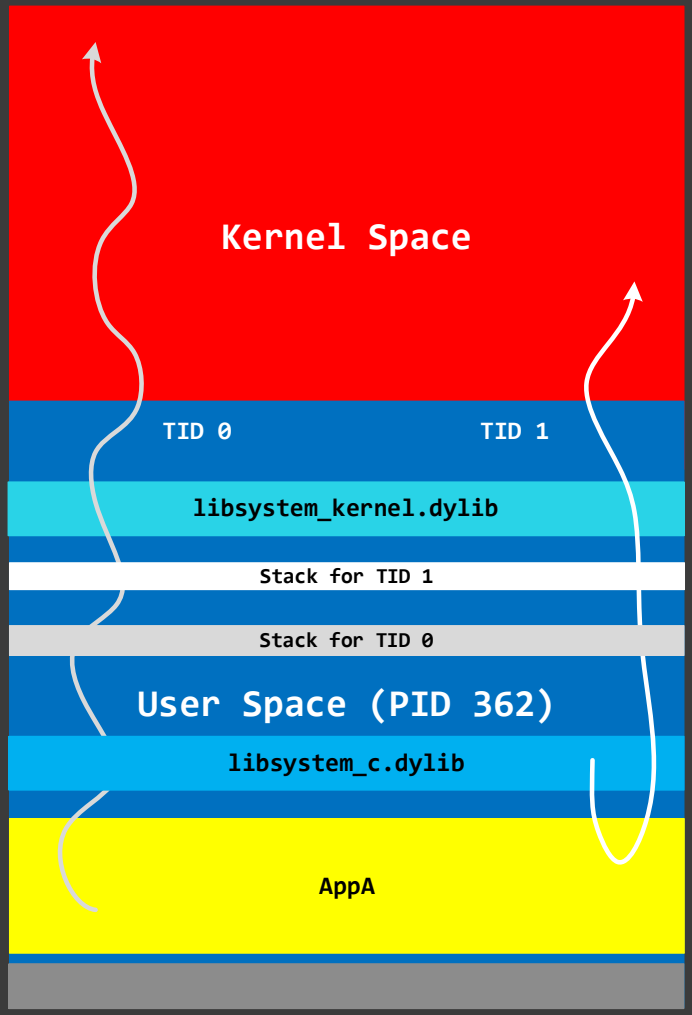
thread list
Lists threads

thread select <n>
Switches between threads

thread backtrace all
Lists stack traces from all threads

© 2022 Software Diagnostics Services

Thread Stack Raw Data

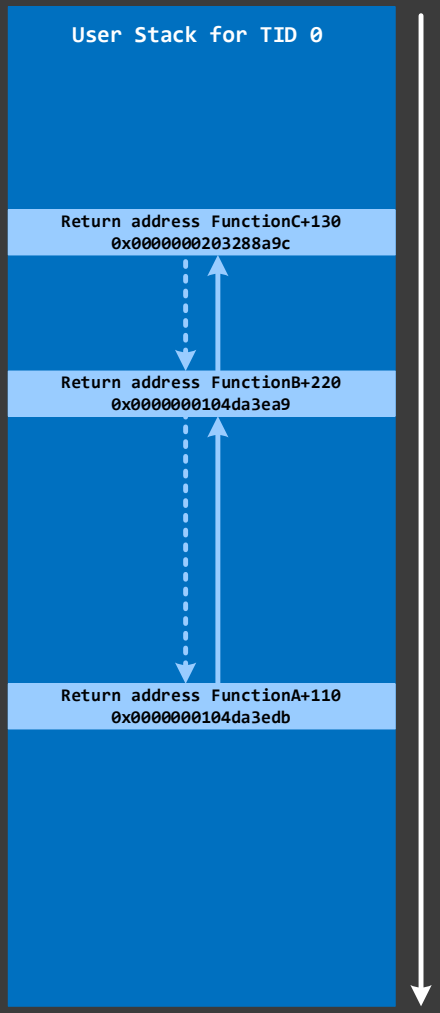


LLDB Commands

x/<n>a <address>
Prints n addresses with corresponding symbol mappings if any

© 2022 Software Diagnostics Services

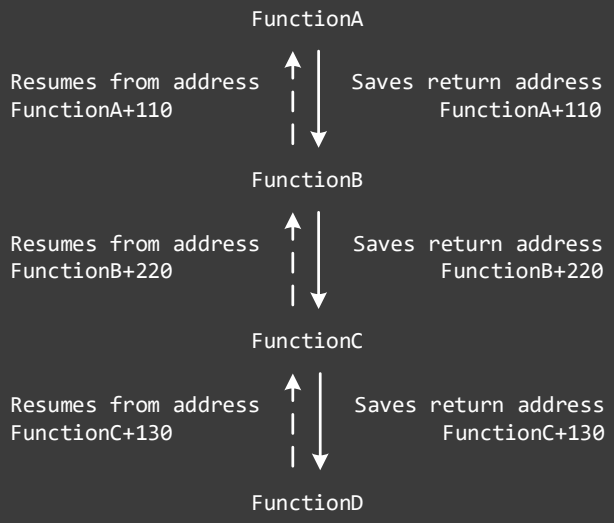
Thread Stack Trace




```
FunctionA()
{
  ...
  FunctionB();
  ...
}

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

FunctionC()
{
  ...
  FunctionD();
  ...
}
```



LLDB Commands

```
(lldb) bt
frame #0: 0x000000020328982a Module`FunctionD + offset
frame #1: 0x0000000203288a9c Module`FunctionC + 130
frame #2: 0x0000000104da3ea9 AppA`FunctionB + 220
frame #3: 0x0000000104da3edb AppA`FunctionA + 110
```

© 2022 Software Diagnostics Services

GDB and LLDB vs. WinDbg

GDB Commands

```
(gdb) bt
#0 0x000000020328982a in FunctionD ()
#1 0x0000000203288a9c in FunctionC ()
#2 0x0000000104da3ea9 in FunctionB ()
#3 0x0000000104da3edb in FunctionA ()
```

LLDB Commands

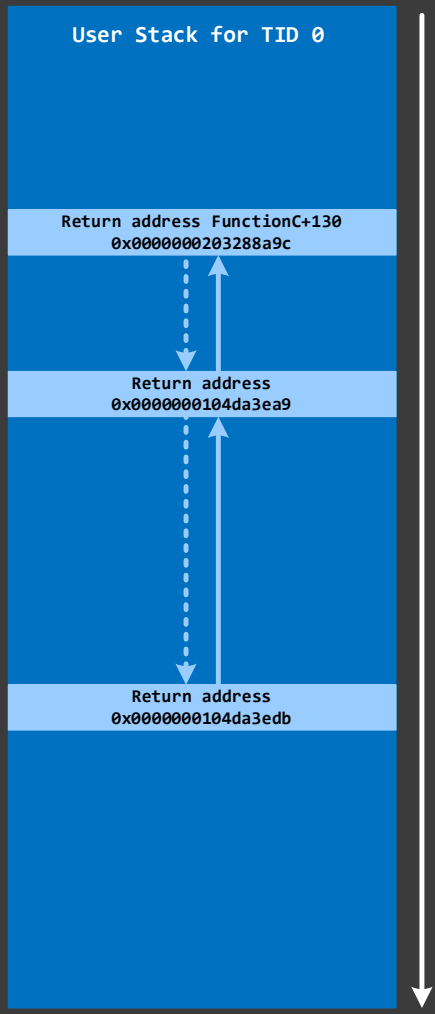
```
(lldb) bt
frame #0: 0x000000020328982a Module`FunctionD + offset
frame #1: 0x0000000203288a9c Module`FunctionC + 130
frame #2: 0x0000000104da3ea9 AppA`FunctionB + 220
frame #3: 0x0000000104da3edb AppA`FunctionA + 110
```

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
```

© 2022 Software Diagnostics Services

Thread Stack Trace (no dSYM)



Symbol file AppA.dSYM

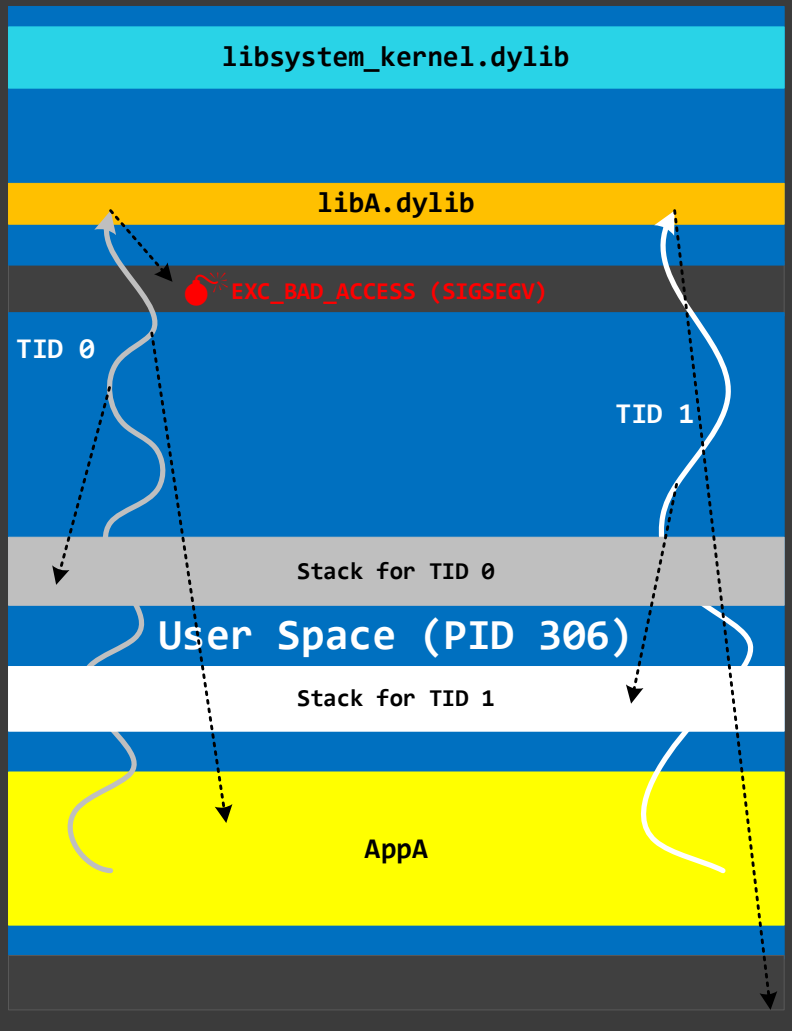
FunctionA 22000 - 23000
FunctionB 32000 - 33000

LLDB Commands

```
(lldb) bt
frame #0: 0x00007fff885e982a Module`FunctionD + offset
frame #1: 0x00007fff83288a9c Module`FunctionC + 130
frame #2: 0x0000000104da3ea9 AppA + 32220
frame #3: 0x0000000104da3edb AppA + 22110
```

© 2022 Software Diagnostics Services

Exceptions (Access Violation)

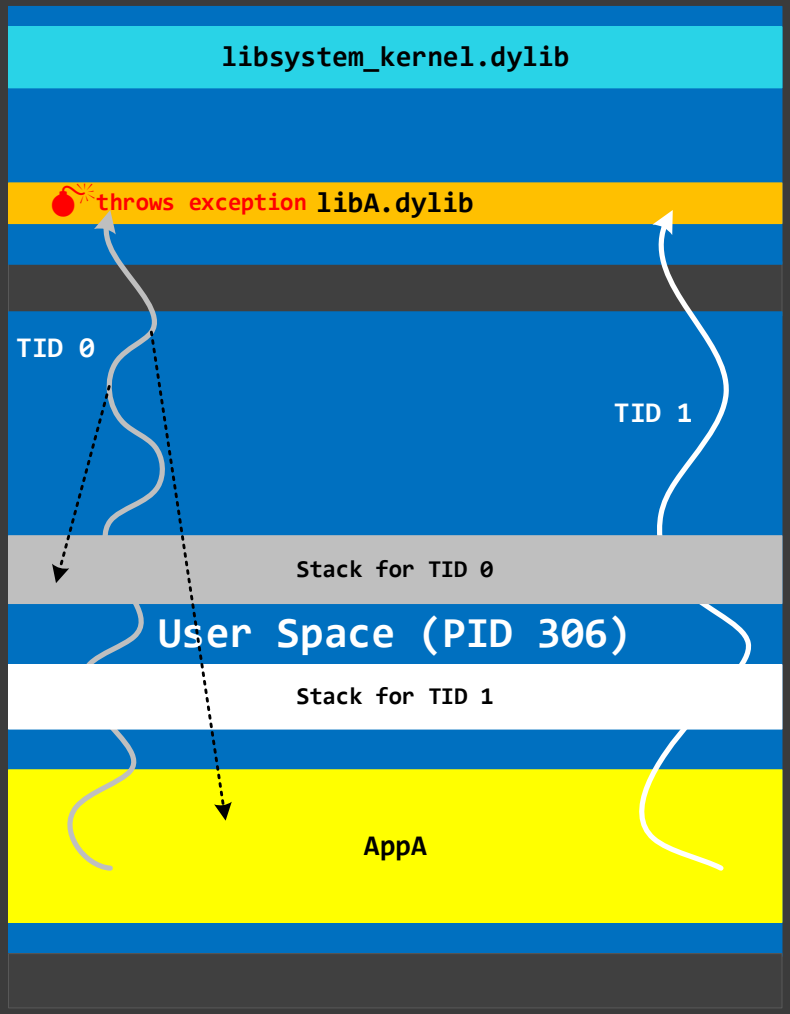


LLDB Commands

```
(lldb) x <address>
error: core file does not contain 0x<address>
```

© 2022 Software Diagnostics Services

Exceptions (Runtime)



© 2022 Software Diagnostics Services

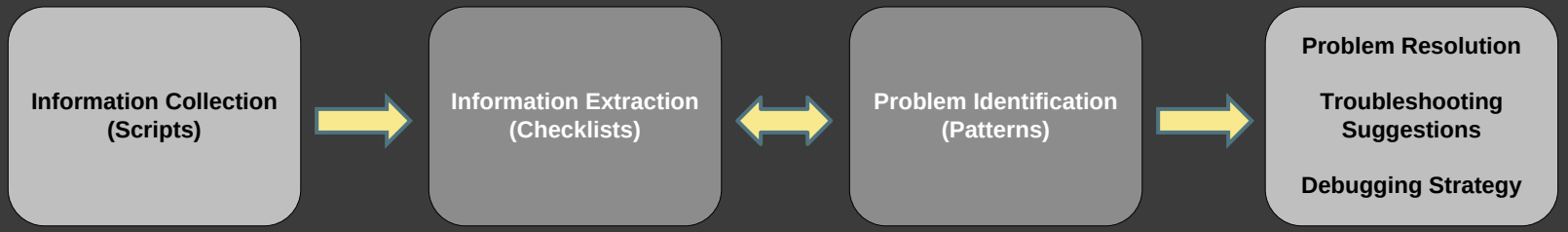
Pattern-Oriented Diagnostic Analysis

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, ...



© 2022 Software Diagnostics Services

Part 2: Core Dump Collection

© 2022 Software Diagnostics Services

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
```

© 2022 Software Diagnostics Services

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…

© 2022 Software Diagnostics Services

Part 3: ARM64 Disassembly

© 2022 Software Diagnostics Services

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**)

LLDB Commands

```
register read
```

© 2022 Software Diagnostics Services

Instructions: registers (ARM64)

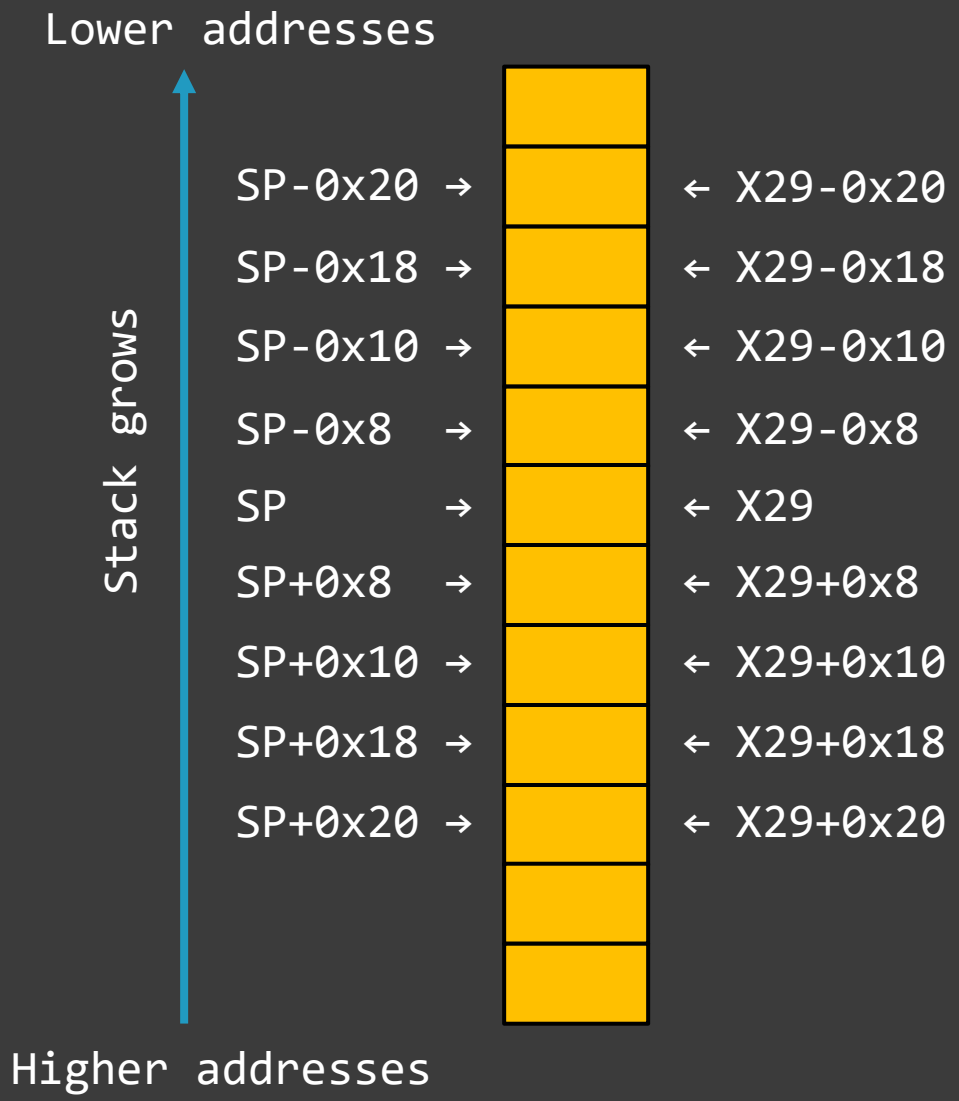
- `Opcode DST, SRC, SRC2`
- Examples:

```
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
```

© 2022 Software Diagnostics Services

Memory and Stack Addressing



© 2022 Software Diagnostics Services

Instructions: memory load (ARM64)

- `Opcode DST, DST2, [SRC, Offset]`
- `Opcode DST, DST2, [SRC], Offset ; Postincrement`
- Examples:

```
ldr   x0, [sp]                ; X0 ← value at address SP+0
ldr   x0, [x29, #-0x8]        ; X0 ← value at address X29-0x8
ldp   x29, x30, [sp, #0x20]   ; X29 ← value at address SP+0x20
                              ; X30 ← value at address SP+0x28
ldp   x29, x30, [sp], #0x10   ; X29 ← value at address SP+0
                              ; X30 ← value at address SP+0x8
                              ; SP ← SP+0x10
```

© 2022 Software Diagnostics Services

Instructions: memory store (ARM64)

- `Opcode SRC, SRC2, [DST, Offset]`
- `Opcode 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
```

© 2022 Software Diagnostics Services

Instructions: flow (ARM64)

- `Opcode DST, SRC`
- Examples:

```
adrp  x0, 2              ; x0 ← PC&0xFFFFFFFFFFFFF000 + 0x1000*2

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

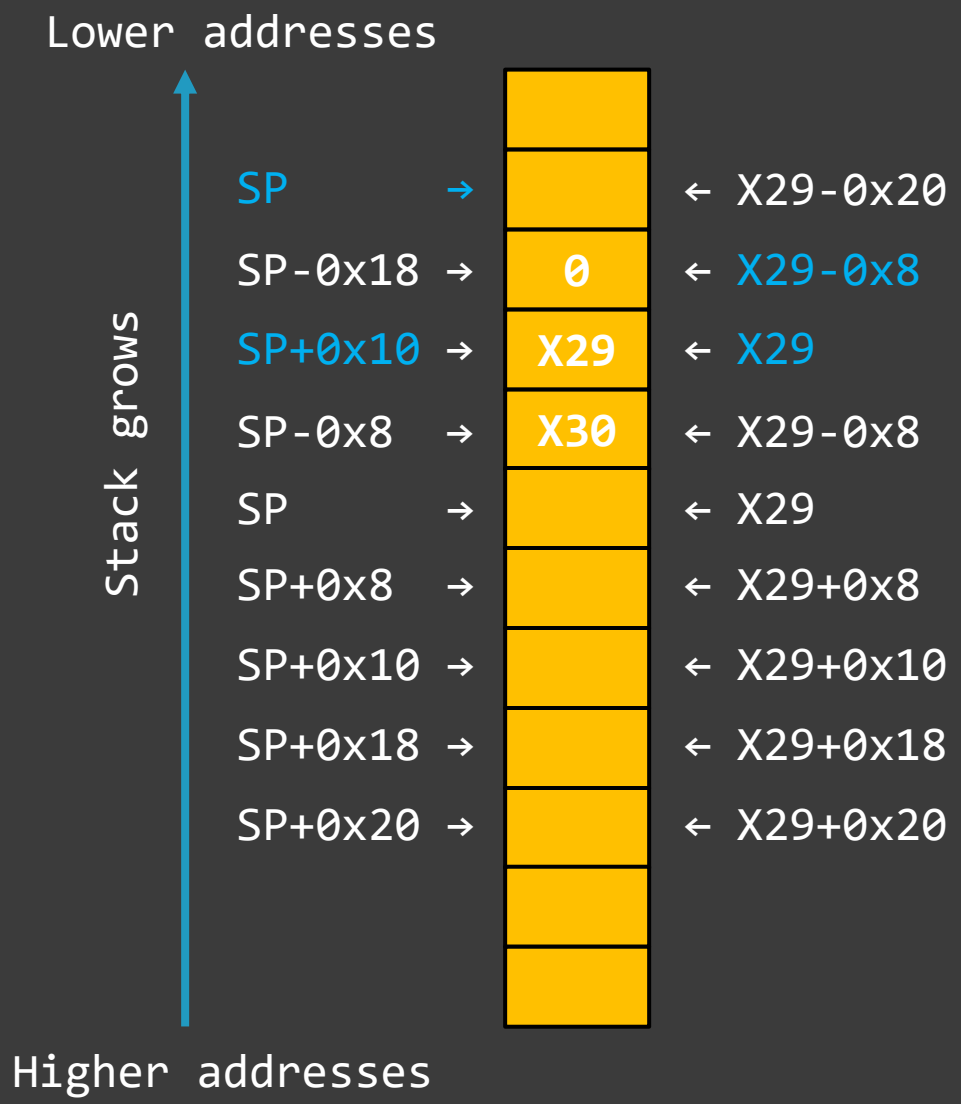
0x10493fc14:             ; PC == 0x10493fc14
bl    0x10493ff74        ; LR ← PC+4 (0x10493fc18)
                         ; PC ← 0x10493ff74
                         ; (goto 0x10493ff74)
```

© 2022 Software Diagnostics Services

Function Call and Prolog

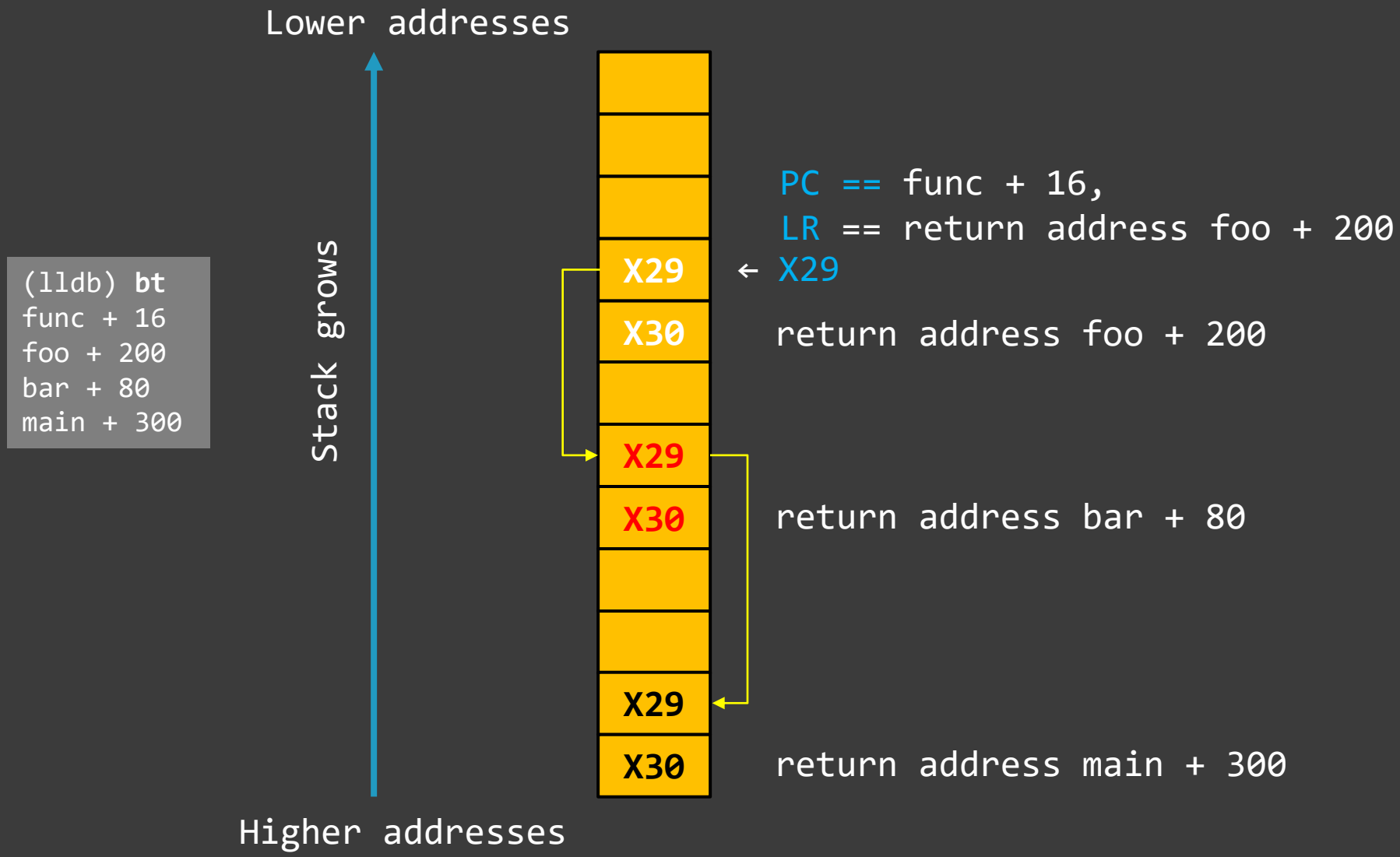
```
; void proc(int p1, long p2);
mov  w0, #0x1
mov  x1, #0x2
bl   proc


; 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
...
```



© 2022 Software Diagnostics Services

Stack Trace Reconstruction



© 2022 Software Diagnostics Services

Pointer Authentication

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

Problem of reading real addresses:

- ```
  (lldb) x/a 0x00006000a000000
  error: core file does not contain 0xa000000
  (lldb) x/a 0x000060a000000000
  error: core file does not contain 0x20000000
  ```

Solution via Typed Memory:

- ```
  (lldb) p/x *(long *)0x00006000a000000
  (long) $1 = 0x0000000000000000
  (lldb) parray/x 10 (long *)0x00006000a000000
  ```

© 2022 Software Diagnostics Services
  ```

Part 4: Practice Exercises

© 2022 Software Diagnostics Services

Links

- Memory Dumps:

Included in Exercise X0

- Exercise Transcripts:

Included in this book

© 2022 Software Diagnostics Services

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

© 2022 Software Diagnostics Services

Process Core Dumps

Exercises X1 – X12

© 2022 Software Diagnostics Services

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

© 2022 Software Diagnostics Services

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

© 2022 Software Diagnostics Services

Exercise X3

- **Goal:** Learn how to identify spiking threads
- **Patterns:** Spiking Thread
- \AMCDA-Dumps\Exercise-X3.pdf

© 2022 Software Diagnostics Services

Exercise X4

- **Goal:** Learn how to identify heap regions and heap corruption
- **Patterns:** Dynamic Memory Corruption (Process Heap)
- \AMCDA-Dumps\Exercise-X4.pdf

© 2022 Software Diagnostics Services

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

© 2022 Software Diagnostics Services

Exercise X7

- **Goal:** Learn how to identify active threads
- **Patterns:** Missing Thread; Active Thread; Near Exception
- \AMCDA-Dumps\Exercise-X7.pdf

© 2022 Software Diagnostics Services

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

© 2022 Software Diagnostics Services

Exercise X9

- **Goal:** Learn how to identify heap leaks
- **Patterns:** Heap Leak; Execution Residue; Module Hint
- \AMCDA-Dumps\Exercise-X9.pdf

© 2022 Software Diagnostics Services

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

© 2022 Software Diagnostics Services

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

© 2022 Software Diagnostics Services

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

© 2022 Software Diagnostics Services

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)

© 2022 Software Diagnostics Services

Further Training

Accelerated macOS (M2)
Disassembly, Reconstruction, and Reversing

© 2022 Software Diagnostics Services

Q&A

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

© 2022 Software Diagnostics Services

Thank you for attendance!

© 2022 Software Diagnostics Services