June 2018

build your own iOS kernel debugger

@i41nbeer

bio

VR and exploit dev with Google Project Zero

Mostly XNU nowadays

Demo

history

KDP support was there in the iOS kernel

bootrom/kernel bug to set boot args

soldering iron to connect some wires to 30 pin dock connector breakout board

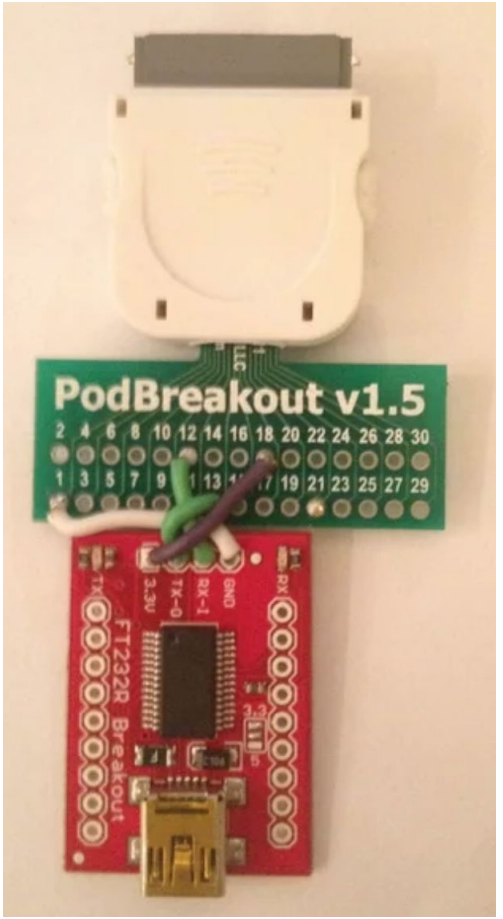


image: http://www.instructables.com/id/Apple-iOS-SerialUSB-Cable-for-Kernel-Debugging/

now:

ARM64 iOS kernel KDP won't work

Kernel text not (supposed to be) modifiable; no breakpoint instructions

exception vector for EL1 breakpoints doesn't work

no real serial port

sleh.c

we'll reach this code if a hardware breakpoint fires while in the kernel

```
void
sleh_synchronous(arm_context_t *context, uint32_t esr, vm_offset_t far)
{
  esr_exception_class_t       class = ESR_EC(esr);
  arm_saved_state_t           *state = &context->ss;

...

  switch (class) {
...
  case ESR_EC_BKPT_REG_MATCH_EL1:
    if (FSC_DEBUG_FAULT == ISS_SSDE_FSC(esr)) {
      kprintf("Hardware Breakpoint Debug exception from kernel.  Hanging here (by design).\n");
      for (;;);
      __unreachable_ok_push
      DebuggerCall(EXC_BREAKPOINT, &context->ss);
      break;
      __unreachable_ok_pop
    }
    panic("Unsupported Class %u event code. state=%p class=%u esr=%u far=%p",
          class, state, class, esr, (void *)far);
    assert(0); /* Unreachable */
    break;
```

What effect does this actually have?

ideas

we'll look at the relevant manual pages

if we could get a hw breakpoint to fire in EL1

then we could cause that core to enter an infinite loop

what does it mean for us to infinite loop here?

crucially: will the scheduler still schedule us off the core?

can we stop it?

if we do get scheduled off, what state is stored and where?
can we modify it (safely)

what could we do if we can?

could we build a debugger with that?
find all the state we need and modify it in response to debugger commands?

This talk & tool

how to get to that infinite loop, and back out again!

use that to build a kernel debugger for all iOS devices

prerequisites

must work on stock devices

must work with regular equipment (no fancy cables)

must not require KPP/KTRR defeat

part of the motivation is to show that if you can implement a kernel debugger without modifying code, you can certainly do whatever your malware/APT implant/whatever wants to

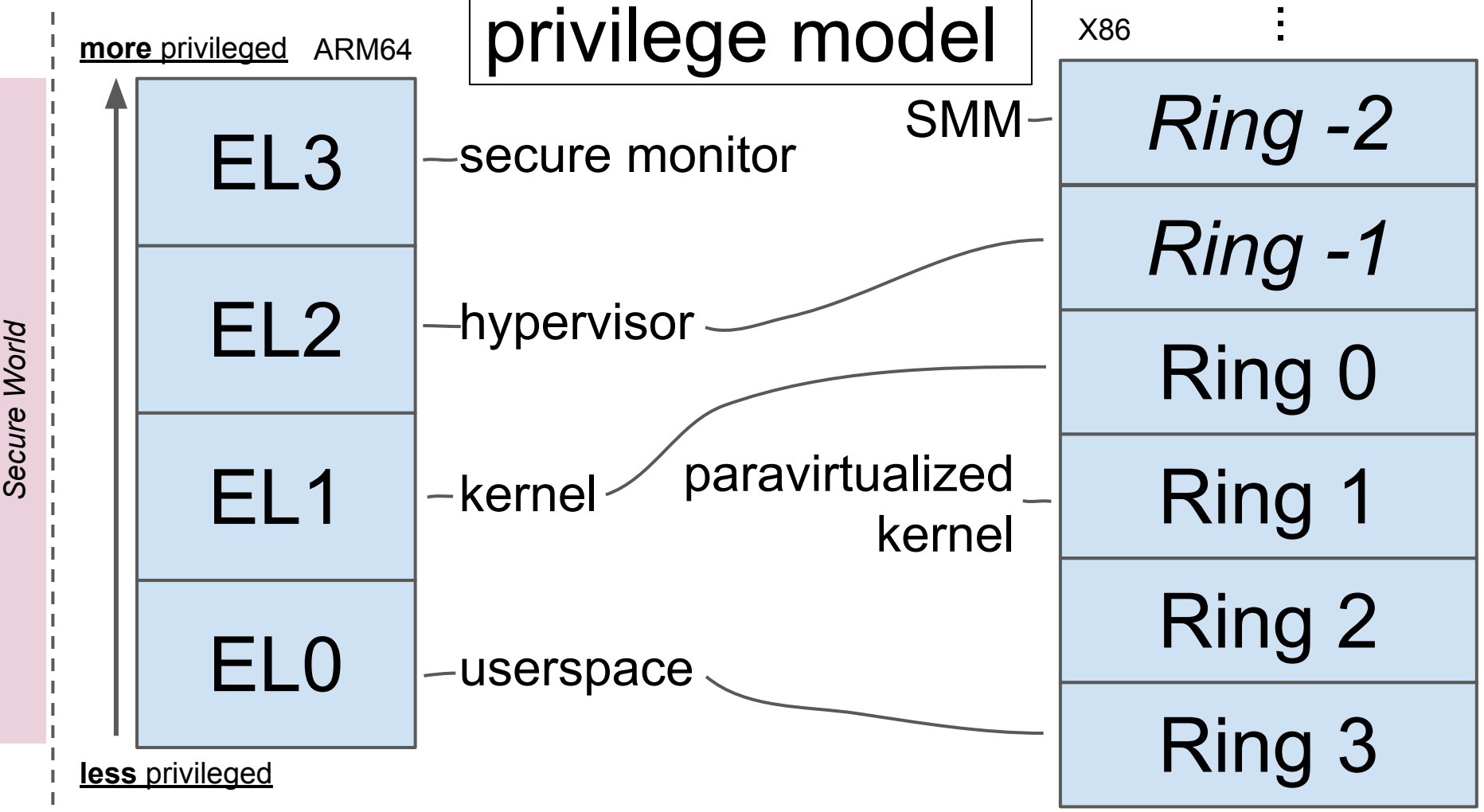
must still be a bit useful

at least: set breakpoint, view and modify register & memory state when hit, continue

must be reasonably easy to use

connect with a normal debugging client (eg lldb)

privilege model



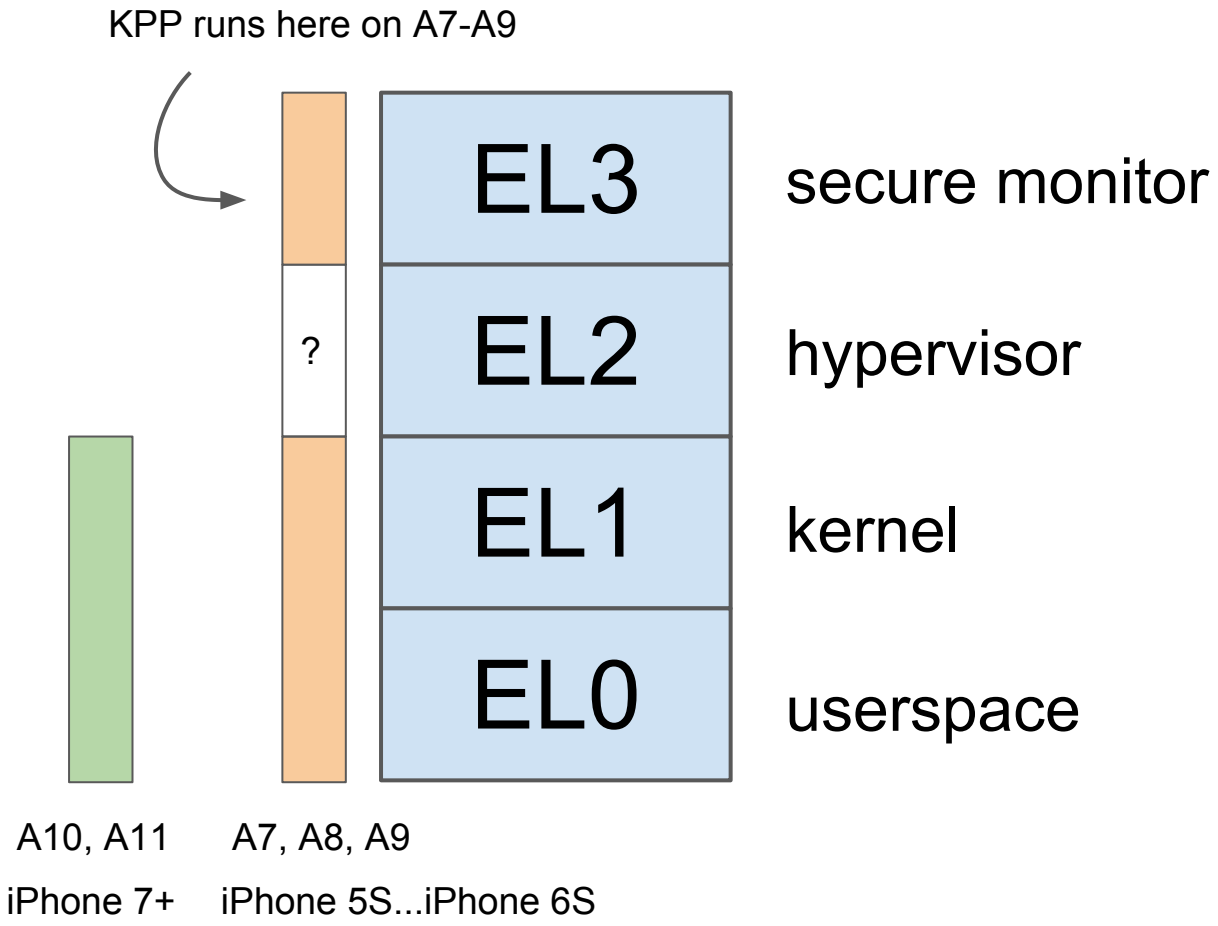
exception levels in iOS

Exception level restricts:
* system register access
* what instructions may trap

For an OS to do anything interesting it must transition between these levels

exceptions are the only thing which cause transitions

fundamental to understand them

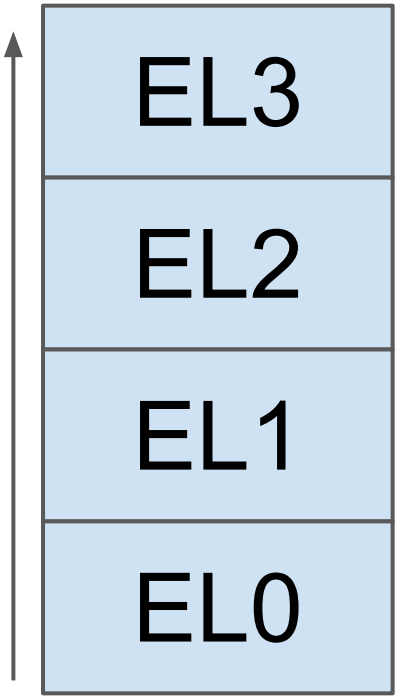


Exceptions

Cause transitions upwards

(or sometimes to the same level)

Synchronous	syscall, memory abort, trapped instruction, breakpoint, watchpoint...
IRQ	hardware events
FIQ	hardware events (iOS: hardware timers)
SError	KPP: secure monitor error You are panicking if you get one of these



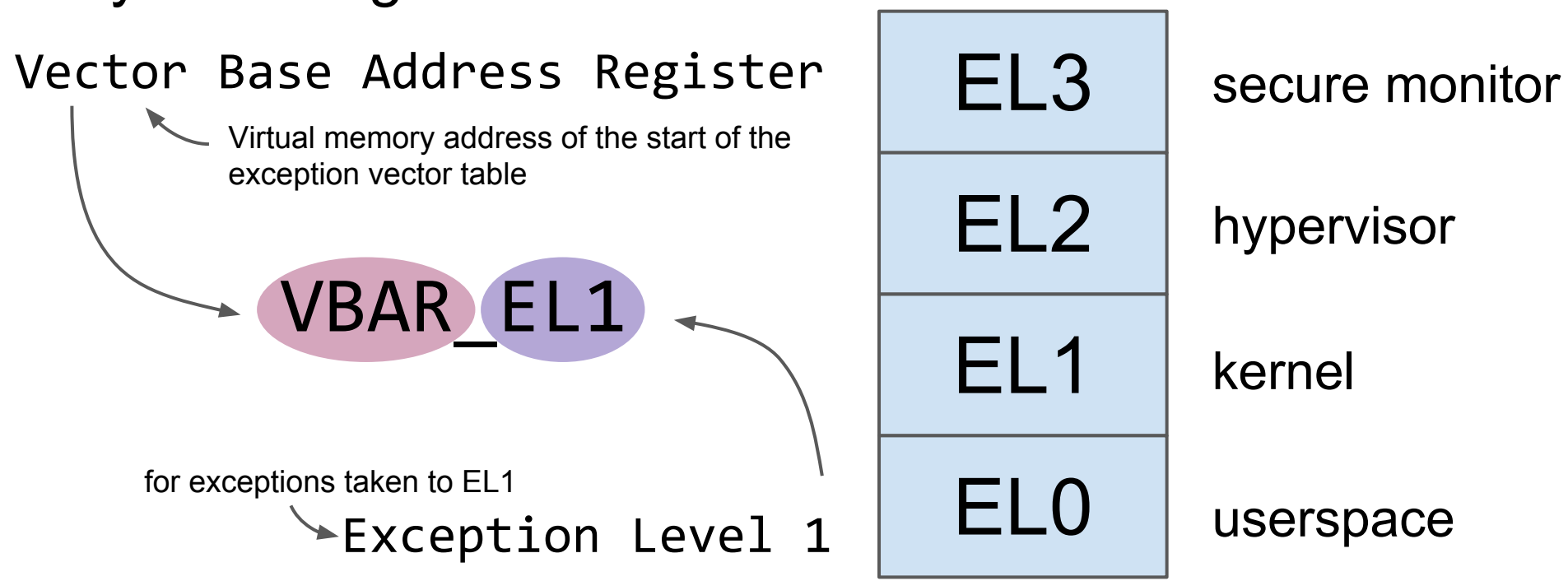
secure monitor

hypervisor

kernel

userspace

System Registers



The _EL*x* suffix is the lowest exception level with access to the register (typically read/write, sometimes only read)

System Registers

read from a system register:

```
mrs x0, TPIDR_EL1
```

write to a system register:

```
msr SP_EL0, x0
```

not to be confused with intel MSR (model-specific registers.)

VBAR_ELx

source	type
same EL running with SP_0	Synchronous
	IRQ
	FIQ
	SError
same EL running with SP_x where x >0	Synchronous
	IRQ
	FIQ
	SError
lower EL where the EL below target runs AArch64	Synchronous
	IRQ
	FIQ
	SError
lower EL where the EL below target runs AArch32	Synchronous
	IRQ
	FIQ
	SError

VBAR_ELx doesn't point to an array of pointers!

It's an array of 16 0x80 byte code chunks

ARM64 XNU VBAR_EL1 entry for synchronous exception from EL0:

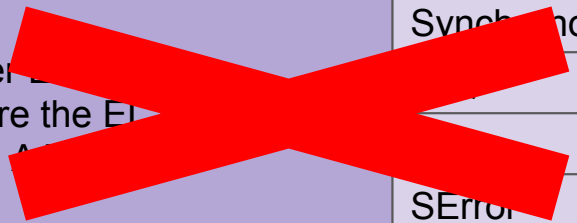
locore.s

```
Lel0_synchronous_vector_64:
EL0_64_VECTOR
mrs  x1, TPIDR_EL1
ldr x1, [x1, TH_KSTACKPTR]
mov sp, x1
adrp x1, fleh_synchronous@page
add x1, x1, fleh_synchronous@pageoff
b fleh_dispatch64
```

this is from the era before speculative execution side channels - it's a little different now - find me after the talk...

VBAR_ELx

same EL running with SP_0	Synchronous
	IRQ
	FIQ
	SError
same EL running with SP_x where x >0	Synchronous
	IRQ
	FIQ
	SError
lower EL where the EL below target runs AArch64	Synchronous
	IRQ
	FIQ
	SError
lower EL where the EL below target runs AArch32	Synchronous
	IRQ
	FIQ
	SError



in iOS 11, this is empty

32-bit really is gone!

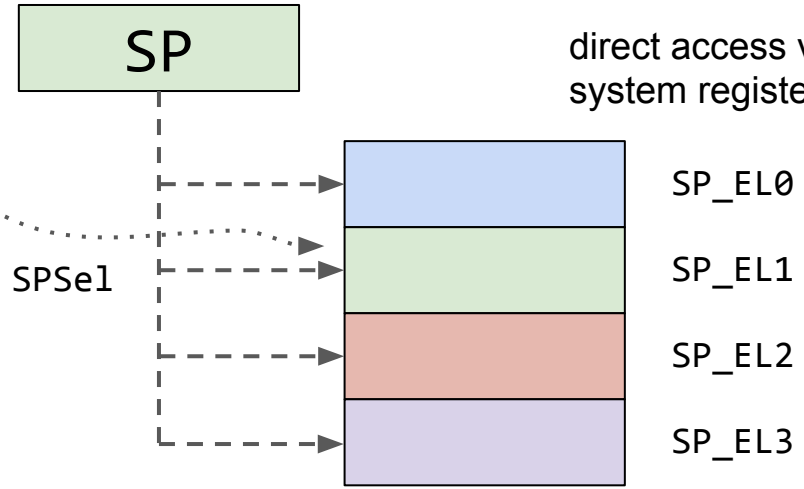
Exception handling in ARM64 XNU - EL0 SVC

again, this is from a more innocent time, there's another step now...

```
VBAR_EL1 + 0x400:

 stp x0, x1, [sp, #-16]!
```

SP_EL1 is the cpu core's exception stack pointer - not per-thread
set in start_cpu

this is just temporarily spilling two registers



sp is not a simple register. It aliases one of four actual hardware registers.

when an exception is taken sp will alias the SP_ELx for the EL which the exception was taken to.

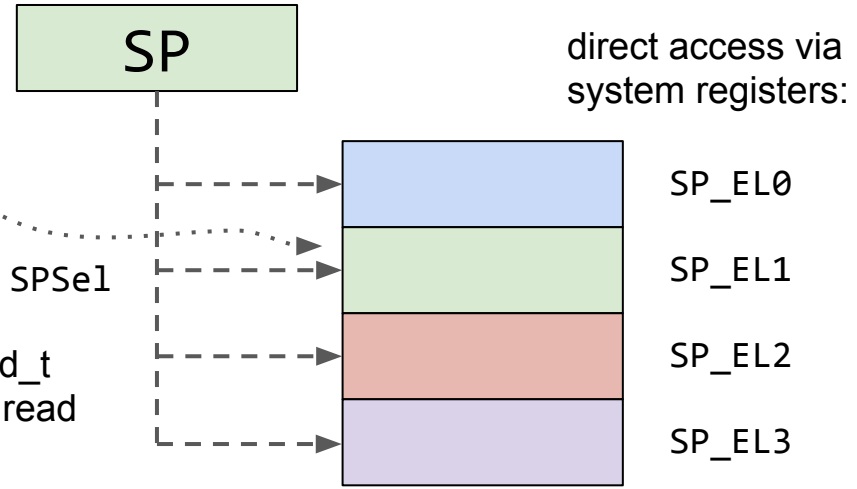
generally all code, regardless of EL actually runs on SP_EL0 most of the time

this makes handling nested exceptions easier

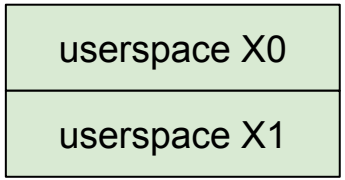
when exception is taken, SP aliases the SP for the target EL.
Setting SPSel to 0 switches SP to alias SP_EL0
(SPSel is a flag, not an index)

Exception handling in ARM64 XNU - EL0 SVC

```
VBAR_EL1 + 0x400:

  stp x0, x1, [sp, #-16]!

  mrs x0, TPIDR_EL1
```

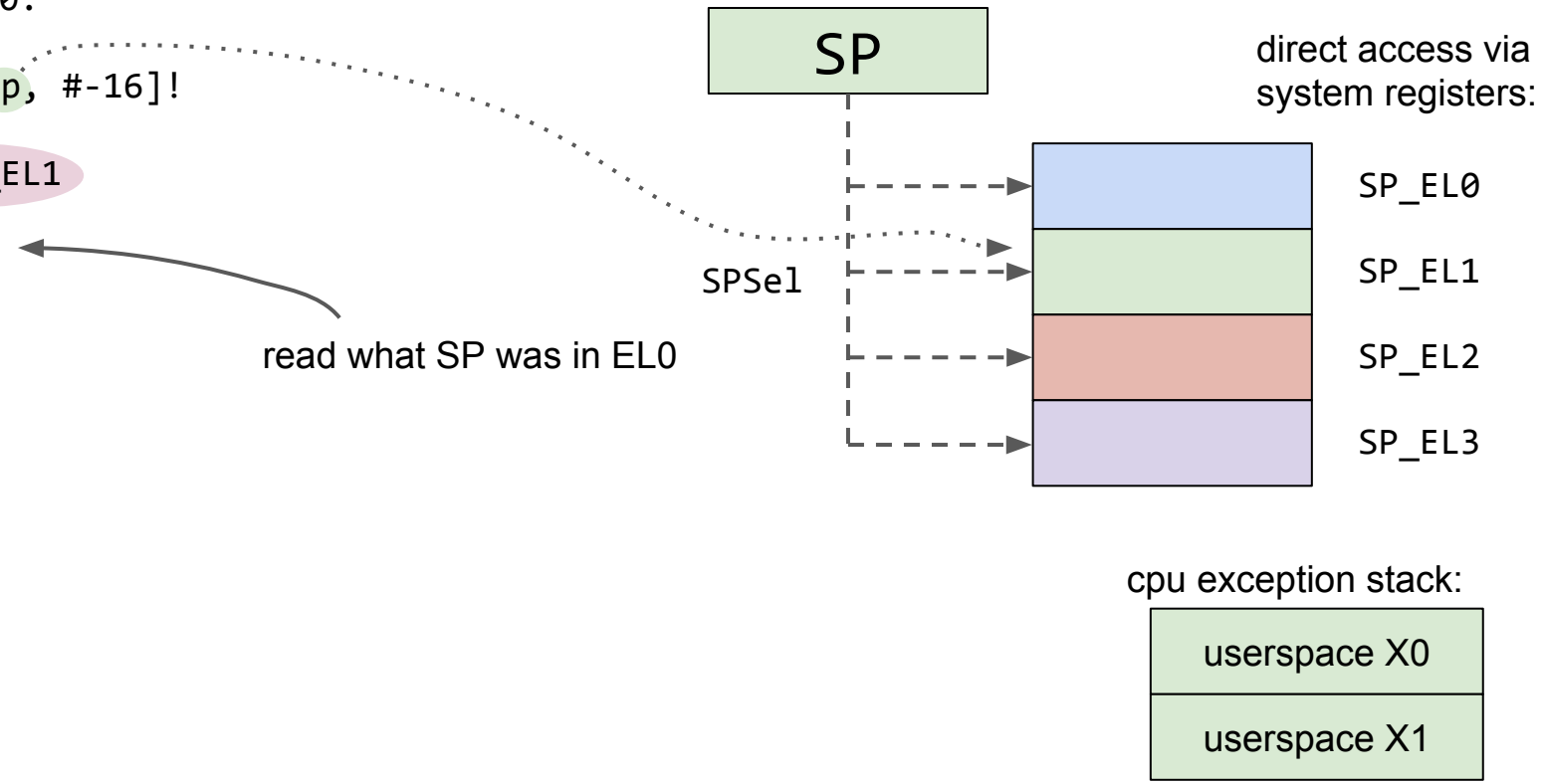


cpu exception stack:



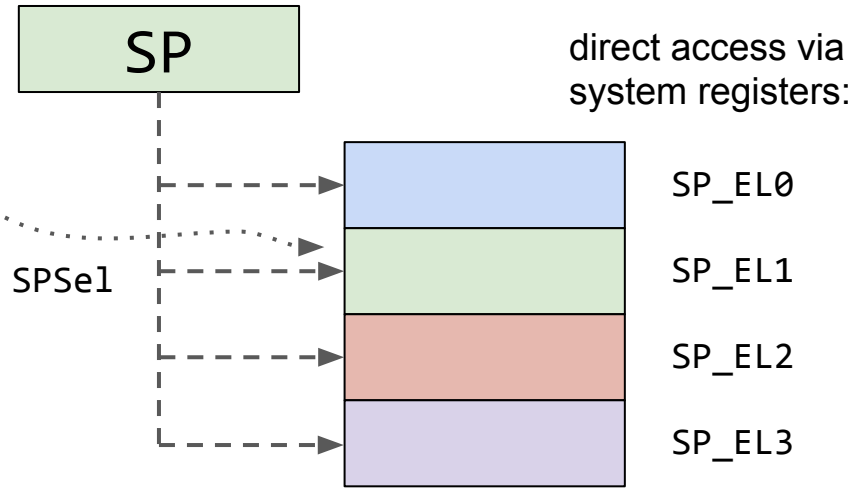
Exception handling in ARM64 XNU - EL0 SVC

```
VBAR_EL1 + 0x400:

  stp x0, x1, [sp, #-16]!

  mrs x0, TPIDR_EL1

  mrs x1, SP_EL0
```



Exception handling in ARM64 XNU - EL0 SVC

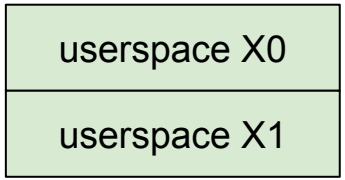
```
VBAR_EL1 + 0x400:

 stp x0, x1, [sp, #-16]!

 mrs x0, TPIDR_EL1

 mrs x1, SP_EL0

 add x0, x0, ACT_CONTEXT
```




```
DECLARE("ACT_CONTEXT",
          offsetof(struct thread, machine.contextData));
```

```
struct machine_thread {
  arm_context_t *contextData; /* allocated user context */
```

cpu exception stack:




```
machine_thread_create:
/* If this isn't a kernel thread, we'll have userspace state. */
thread->machine.contextData = (arm_context_t *)zalloc(user_ss_zone);
```

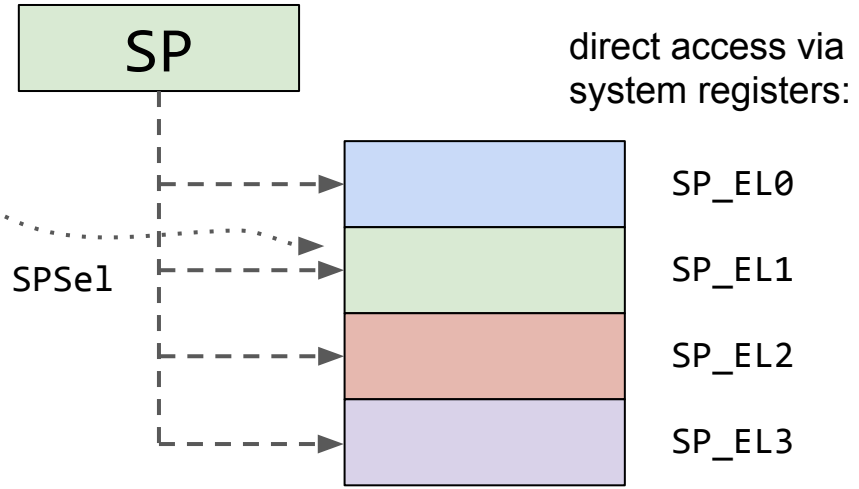
arm_context_t

```
struct arm_context {
  struct arm_saved_state ss;
  struct arm_neon_saved_state ns;
};

struct arm_saved_state {
  arm_state_hdr_t ash;
  union {
    struct arm_saved_state32 ss_32;
    struct arm_saved_state64 ss_64;
  } uss;
} __attribute__((aligned(16)));
```

```
struct arm_saved_state64 {
     uint64_t    x[29];
     uint64_t    fp;
     uint64_t    lr;
     uint64_t    sp;
     uint64_t    pc;
     uint32_t    cpsr;
     uint32_t  reserved;
     uint64_t  far;
     uint32_t  esr;
     uint32_t  exception;
};
```

Exception handling in ARM64 XNU - EL0 SVC

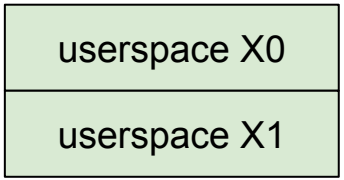
```
VBAR_EL1 + 0x400:

 stp x0, x1, [sp, #-16]!

 mrs x0, TPIDR_EL1

 mrs x1, SP_EL0

 add x0, x0, ACT_CONTEXT

 ldr x0, [x0]

 str x1, [x0, SS64_SP]
```




```
 DECLARE("SS64_SP", offsetof(arm_context_t, ss.ss_64.sp));
```

we've saved the userspace stack pointer to the thread's userspace saved context area

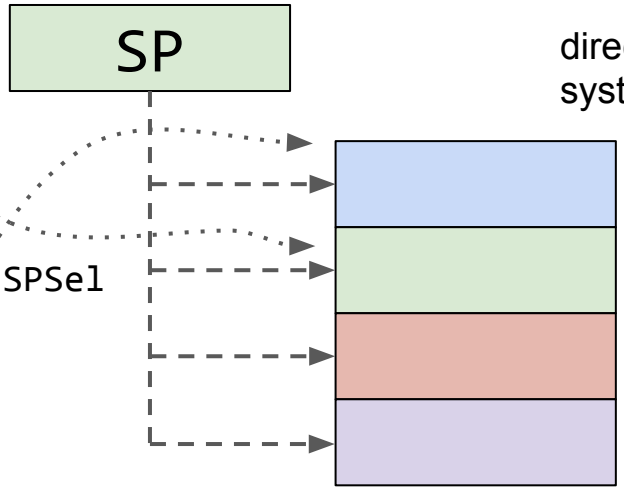
cpu exception stack:



Exception handling in ARM64 XNU - EL0 SVC

VBAR_EL1 + 0xxx?

```
stp x0, x1, [sp, #-16]!
mrs x0, TPIDR_EL1
mrs x1, SP_EL0
add x0, x0, ACT_CONTEXT
ldr x0, [x0]
str x1, [x0, SS64_SP]
msr SP_EL0, x0
ldp x0, x1, [sp], #16
```

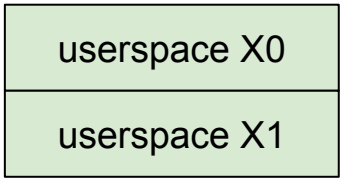


direct access via system registers:

save the ACT_CONTEXT pointer in SP_EL0 (we've saved the userspace value in ACT_CONTEXT)

pop the userspace values of x0 and x1 off the cpu exception stack

cpu exception stack:

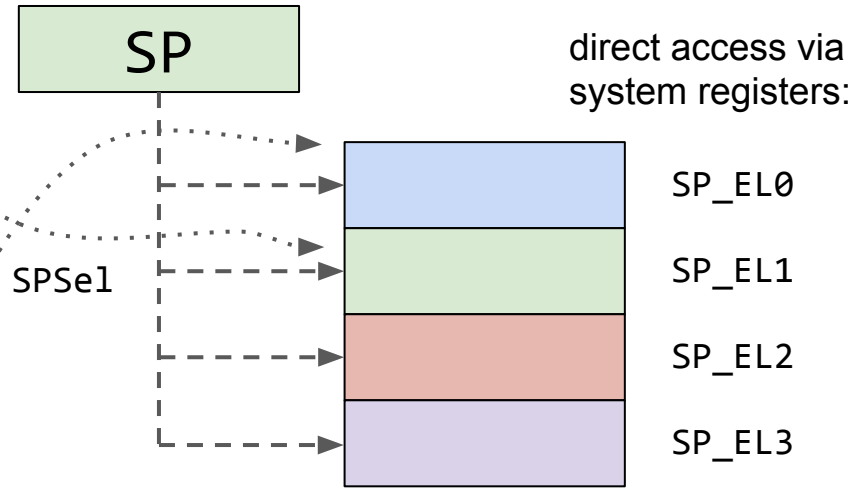


Exception handling in ARM64 XNU - EL0 SVC

VBAR_EL1 + 0xxx?

```
stp x0, x1, [sp, #-16]!
mrs x0, TPIDR_EL1
mrs x1, SP_EL0
add x0, x0, ACT_CONTEXT
ldr x0, [x0]
str x1, [x0, SS64_SP]
msr SP_EL0, x0
ldp x0, x1, [sp], #16
msr SPSel, #0
```

switch SP to alias SP_EL0



are now "running on SP0" for the purposes of another exception happening now

Exception handling in ARM64 XNU - EL0 SVC

skip forwards a bit:...

```
mov x0, sp

mrs x1, TPIDR_EL1
```

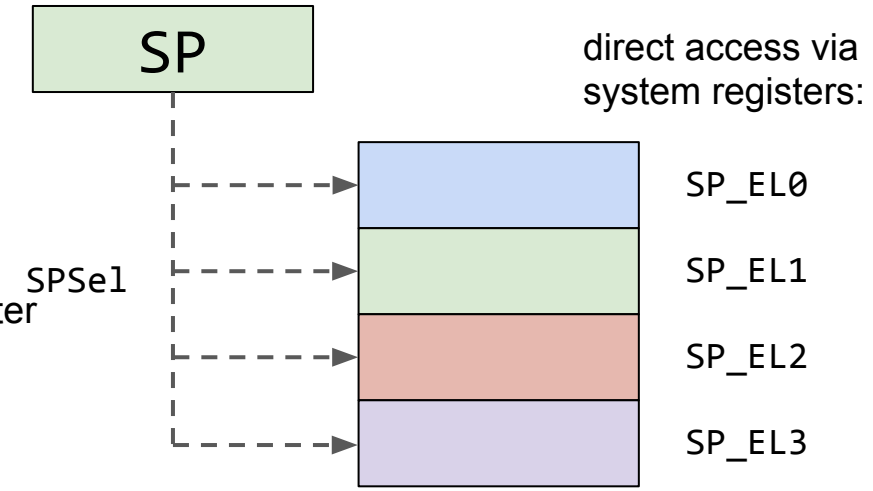
read the thread register

```
ldr x1, [x1, TH_KSTACKPTR]
```

load the thread's kernel stack pointer

```
mov sp, x1
```

pivot to the thread's kernelstack




```
DECLARE("TH_KSTACKPTR",
        offsetof(struct thread, machine.kstackptr));

machine_stack_attach:
thread->machine.kstackptr = stack + kernel_stack_size - sizeof(struct thread_kernel_state);
```

Exception handling in ARM64 XNU - EL0 SVC

skip forwards a bit:

```
SPILL_REGISTERS:

stp x2, x3, [x0, SS64_X2]
stp x4, x5, [x0, SS64_X4]
...

stp q0, q1, [x0, NS64_Q0]
stp q2, q3, [x0, NS64_Q2]
...

mrs lr, ELR_EL1
mrs x23, SPSR_EL1
mrs x24, FPSR
mrs x25, FPCR

str lr, [x0, SS64_PC]
str w23, [x0, SS64_CPSR]
str w24, [x0, NS64_FPSR]
str w25, [x0, NS64_FPCR]
```

saves remaining general purpose registers to region pointed to by X0

(for syscall case this is ACT_CONTEXT)

saves NEON registers

spill things which aren't real registers

exception link register becomes saved PC
saved program state register becomes saved current program state

low level exception handling in XNU for ARM64

again, this is from a more innocent time, there's another step now...

`stub in vector table`	`locore.s` hand-written assembly * switch to SP0 * switch away from per-cpu exception stack to thread kernel stack * jump to fleh_dispatch

`fleh_dispatch`
 	
`first level exception handler dispatcher`
	`locore.s`
	hand-written assembly
	* spill register state
	* indirect jump to fleh

`FLEH` `first level exception handler`	`locore.s` hand-written assembly * load regs for a c function call
`SLEH` `second level exception handler`	`sleh.c` c code * handle the exception

last chunk of assembly code before we call into C

```
fleh_synchronous:
     mrs       x1, ESR_EL1
     mrs       x2, FAR_EL1

     and       w3, w1, #(ESR_EC_MASK)
     lsr       w3, w3, #(ESR_EC_SHIFT)
     mov       w4, #(ESR_EC_IABORT_EL1)
     cmp       w3, w4
     b.eq Lfleh_sync_load_lr

Lvalid_link_register:

     PUSH_FRAME
     bl        EXT(sleh_synchronous)
     POP_FRAME
```

load the second and third arguments for sleh_synchronous

first argument (X0) is still the ACT_CONTEXT (userspace state)

sleh.c

mrs x1, ESR_EL1

mrs x2, FAR_EL1

```
void
sleh_synchronous(arm_context_t *context, uint32_t esr, vm_offset_t far)
{
  esr_exception_class_t       class = ESR_EC(esr);
  arm_saved_state_t           *state = &context->ss;

...

  /* Inherit the interrupt masks from previous context */
  if (SPSR_INTERRUPTS_ENABLED(get_saved_state_cpsr(state)))
    ml_set_interrupts_enabled(TRUE);

  switch (class) {
  case ESR_EC_SVC_64:
    if (!is_saved_state64(state) ||
        !PSR64_IS_USER(get_saved_state_cpsr(state)))
    {
      panic("Invalid SVC_64 context");
    }

    handle_svc(state);
    break;
```

this is the syscall handler

sleh.c

```
void
sleh_synchronous(arm_context_t *context, uint32_t esr, vm_offset_t far)
{
  esr_exception_class_t       class = ESR_EC(esr);
  arm_saved_state_t           *state = &context->ss;

...

  switch (class) {
...
  case ESR_EC_DABORT_EL0:
    handle_abort(state, esr, far, recover, inspect_data_abort, handle_user_abort);
    assert(0); /* Unreachable */
```

this is a data-abort ("segfault" for non-instruction load/store)

eventually this gets turned into a mach message sent to the task's exception ports, and it's that state which will be modified

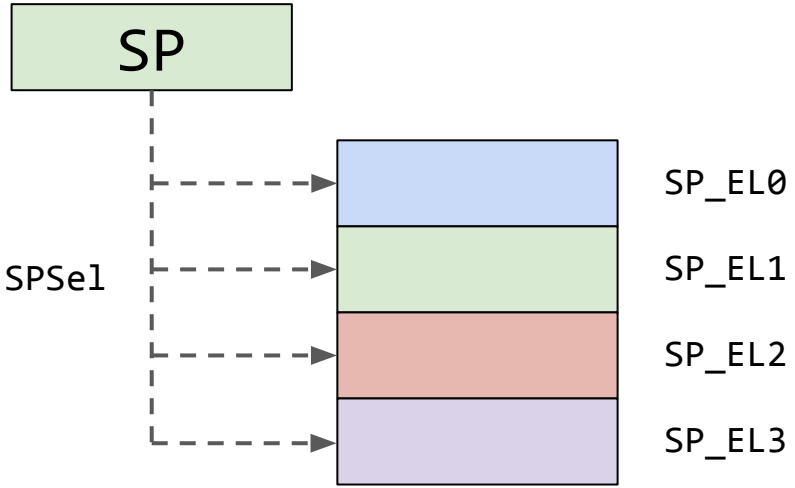
if this thread does survive the abort (a registered exception handler returns success) it will return to userspace directly via thread_exception_return

sleh.c

we'll reach this code if a hardware breakpoint fires while in the kernel

```
void
sleh_synchronous(arm_context_t *context, uint32_t esr, vm_offset_t far)
{
  esr_exception_class_t       class = ESR_EC(esr);
  arm_saved_state_t           *state = &context->ss;

...

  switch (class) {
...
  case ESR_EC_BKPT_REG_MATCH_EL1:
    if (FSC_DEBUG_FAULT == ISS_SSDE_FSC(esr)) {
      kprintf("Hardware Breakpoint Debug exception from kernel.  Hanging here (by design).\n");
      for (;;);
      __unreachable_ok_push
      DebuggerCall(EXC_BREAKPOINT, &context->ss);
      break;
      __unreachable_ok_pop
    }
    panic("Unsupported Class %u event code. state=%p class=%u esr=%u far=%p",
          class, state, class, esr, (void *)far);
    assert(0); /* Unreachable */
    break;
```

VBAR_EL1 differences for SYNC **EL1**_SP0 to EL1



this means a synchronous exception which originated in the kernel, like a hardware breakpoint exception!
cpu will switch SP to alias SP_EL1 for us, so we're on the core's exception stack now

first difference:
we could be here due to the kernel's stack pointer being wrong (eg stack overflow, stack buffer overflow)
so we should probably try to detect that first:

```
sub sp, sp, ARM_CONTEXT_SIZE

stp x0, x1, [sp, SS64_X0]

mrs x1, ESR_EL1
```

make space on the per-core exception stack for a full register dump, just in case we will panic

do some checking to see if this could be a problem with the thread's kernel stack
let's assume this is okay

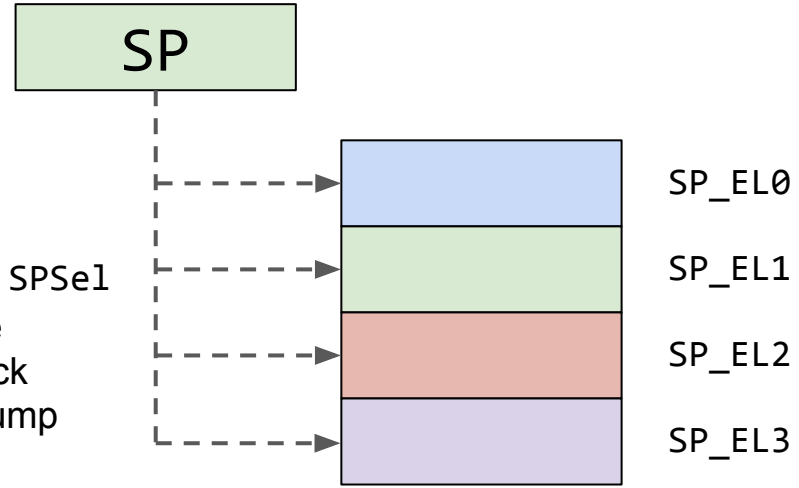
VBAR_EL1 differences for SYNC **EL1**_SP0 to EL1

```
msr SPSel, #0
```
switch to SP_EL0
(which is the thread's kernel stack)

```
sub sp, sp, ARM_CONTEXT_SIZE
```
make space on the thread's kernel stack for a full register dump

```
stp x0, x1, [sp, SS64_X0]

add x0, sp, ARM_CONTEXT_SIZE
```
fill in the correct sp value

```
str x0, [sp, SS64_SP]

...

mov x0, sp
```
set X0 to the base of that register save area

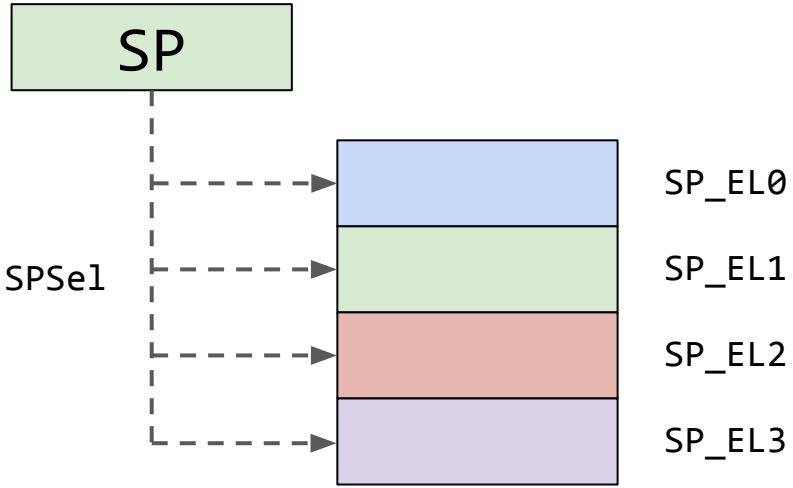


VBAR_EL1 differences for FIQ
EL1_SP0 to EL1

same as SYNC at first; still sets up a new frame on the SP_EL0 stack to hold the spilled state, but:

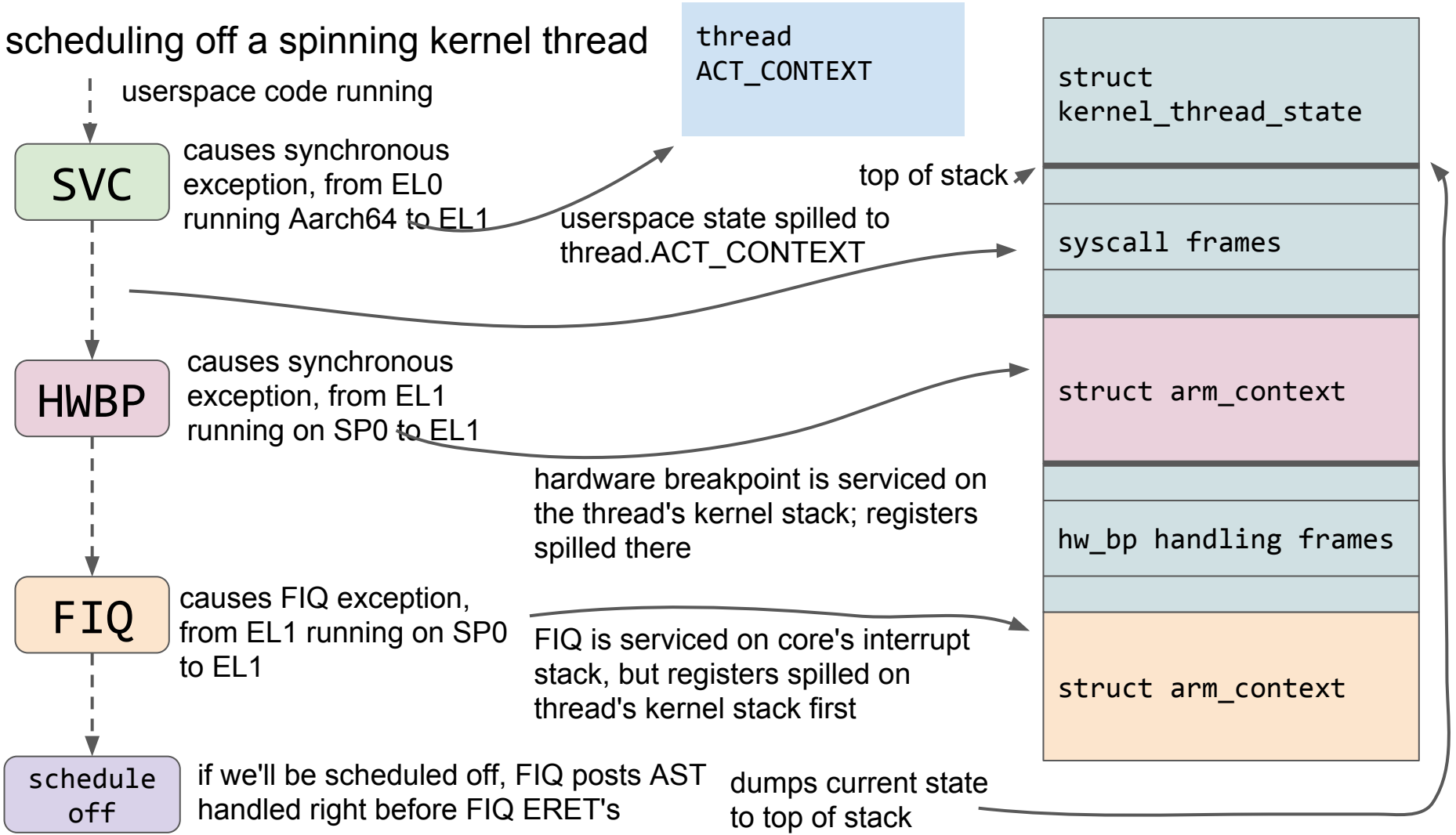
```
mrs x1, TPIDR_EL1

ldr x1, [x1, ACT_CPUDATAP]

ldr x1, [x1, CPU_ISTACKPTR]

mov sp, x1
```

switches to the per-core interrupt stack rather than staying on the kernel stack



Register spill destinations

Synchronous EL0 to EL1	userspace state saved to thread's ACT_CONTEXT
Synchronous EL1_SP0 to EL1	kernel state saved to new frame on thread's kernel stack
FIQ EL1_SP0 to EL1	kernel state saved to new frame on thread's kernel stack


scheduling off a spinning kernel thread
userspace code running
SVC
causes synchronous exception, from EL0 running Aarch64 to EL1
thread ACT_CONTEXT
top of stack
struct kernel_thread_state
userspace state spilled to thread.ACT_CONTEXT
syscall frames
HWBP
causes synchronous exception, from EL1 running on SP0 to EL1
struct arm_context
hardware breakpoint is serviced on the thread's kernel stack; registers spilled there
hw_bp handling frames
FIQ
causes FIQ exception, from EL1 running on SP0 to EL1
FIQ is serviced on core's interrupt stack, but registers spilled on thread's kernel stack first
struct arm_context
schedule off
if we'll be scheduled off, FIQ posts AST handled right before FIQ ERET's
dumps current state to top of stack

how to get hardware breakpoints to fire

read the manual :)

D2.4 Enabling debug exceptions from the current Exception level and Security state

Table D2-5 Whether debug exceptions are enabled from the current Exception level

Current Exception level	Breakpoint Instruction exceptions	All other debug exceptions
Any Exception level that is higher than EL_D[a]	Enabled	Disabled
EL_D	Enabled	Disabled if either of the following is true: • The Local (kernel) Debug Enable bit, MDSCR_EL1.KDE, is 0. • The Debug exception mask bit, PSTATE.D, is 1. Otherwise enabled. This means that a debugger must explicitly enable these debug exceptions from EL_D by setting MDSCR_EL1.KDE to 1 and PSTATE.D to 0.
Any Exception level that is lower than EL_D	Enabled	Enabled

a. This includes EL3. EL3 is always higher than EL_D.

Note

PSTATE.D is set to 1 at reset and on exception entry.

structure of hardware breakpoint registers

D2.2 The debug exception enable controls

The enable controls for each debug exception are as follows:

Breakpoint Instruction exceptions

None. Breakpoint Instruction exceptions are always enabled.

Breakpoint exceptions

MDSCR_EL1.MDE, plus an enable control for each breakpoint, DBGBCR<n>_EL1.E.

Watchpoint exceptions

MDSCR_EL1.MDE, plus an enable control for each watchpoint, DBGWCR<n>_EL1.E.

Vector Catch exceptions

MDSCR_EL1.MDE.

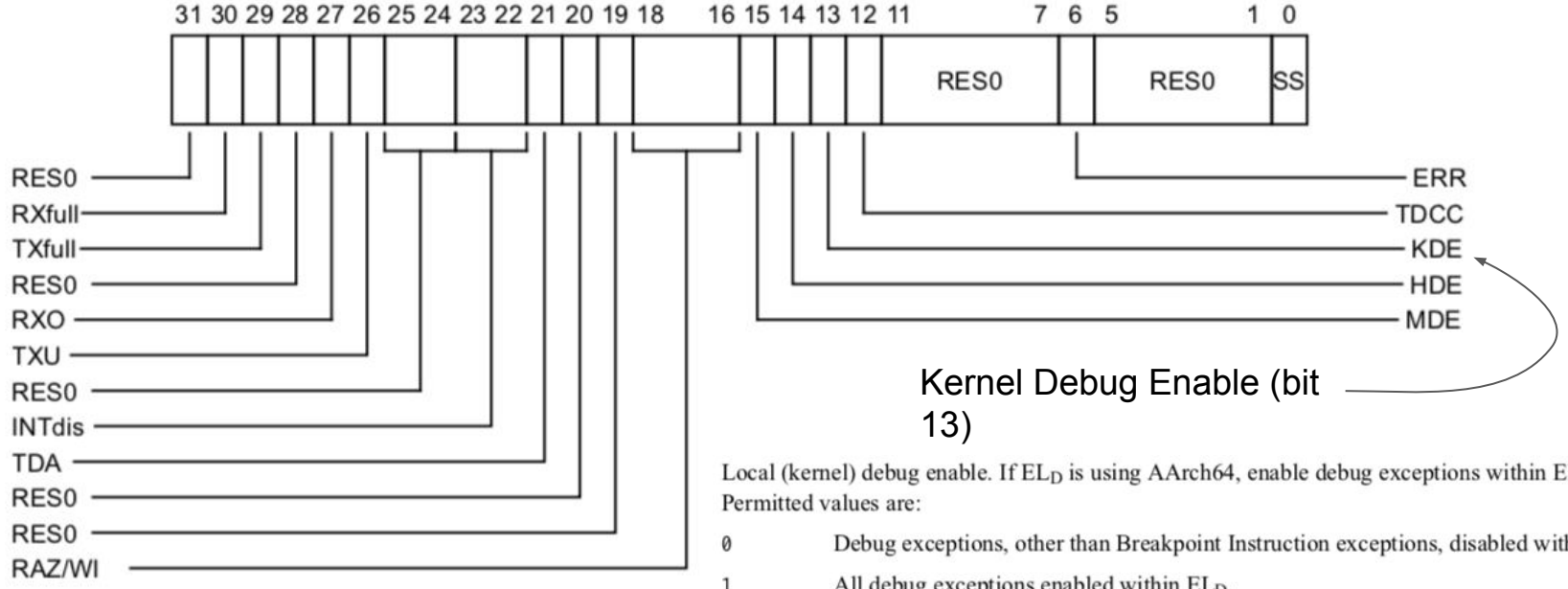
Software Step exceptions

MDSCR_EL1.SS.

MDSCR_EL1

global enable bits; per core

Monitor Debug System Control Register



Kernel Debug Enable (bit 13)

Local (kernel) debug enable. If EL_D is using AArch64, enable debug exceptions within EL_D. Permitted values are:

0	Debug exceptions, other than Breakpoint Instruction exceptions, disabled within EL_D.
1	All debug exceptions enabled within EL_D.

RES0 if EL_D is using AArch32.

This field resets to a value that is architecturally UNKNOWN.

Hardware breakpoints

best thought of as 16 pairs of registers

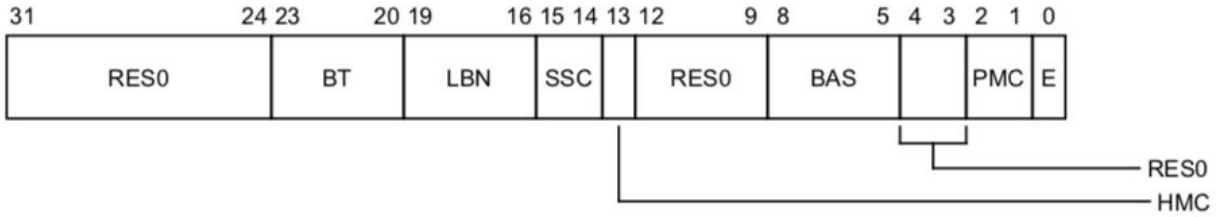
Debug Breakpoint Value Register

DBGBVR<1..15>_EL1

addresses where we want hardware breakpoints to fire

Debug Breakpoint Control Register

DBGBCR<1..15>_EL1



PMC: privilege mode control:

Privilege mode control. Determines the Exception level or levels at which a Breakpoint debug event for breakpoint n is generated. This field must be interpreted along with the SSC and HMC fields, and there are constraints on the permitted values of the {HMC, SSC, PMC} fields. For more information see the DBGBCR<n>_EL1.SSC description.

For more information on the operation of the SSC, HMC, and PMC fields, see *Execution conditions for which a breakpoint generates Breakpoint exceptions* on page D2-1956.

This field resets to a value that is architecturally UNKNOWN.

setting hardware breakpoints

supported by the thread_set_state API

```
 struct arm64_debug_state state = {0};
  for (int i = 0; i < MAX_BREAKPOINTS; i++) {
    if (breakpoints[i] == 0) {
      continue;
    }
    state.bvr[i] = breakpoints[i];
#define BCR_BAS_ALL (0xf << 5)
#define BCR_E (1 << 0)
    state.bcr[i] = BCR_BAS_ALL | BCR_E; // enabled
  }
  kern_return_t err = thread_set_state(mach_thread_self(),
                                       ARM_DEBUG_STATE64,
                                       (thread_state_t)&state,
                                       sizeof(state)/4);
```

the kernel side of thread_set_state does check these flags, need some help from the kernel r/w...

setting hardware breakpoints

find the thread's DebugData

```
uint64_t DebugData = rk64(thread_t_addr + ACT_DEBUGDATA_OFFSET);

 for (int i = 0; i < MAX_BREAKPOINTS; i++) {
   if (breakpoints[i] == 0) {
     continue;
   }

   uint32_t bcr = rk32(DebugData + offsetof(struct arm_debug_aggregate_state, ds64.bcr[i]));

   bcr |= ARM_DBG_CR_MODE_CONTROL_ANY;

   wk32(DebugData + offsetof(struct arm_debug_aggregate_state, ds64.bcr[i]), bcr);
 }
```

read the current bcr value for this BP

set the flag to fire the bp in all ELs

actually set it in the thread's DebugData object

exception masking

Table D2-5 Whether debug exceptions are enabled from the current Exception level

Current Exception level	Breakpoint Instruction exceptions	All other debug exceptions
Any Exception level that is higher than EL_D[a]	Enabled	Disabled
EL_D	Enabled	Disabled if either of the following is true: • The Local (kern… ebug Enable bit, MDSCR_EL1.KDE, is 0. • …ception mask bit, PSTATE.D, is 1. …herwise enabled. This means that a debugger must explicitly enable these debug exceptions from EL_D by setting MDSCR_EL1.KDE to 1 and PSTATE.D to 0.
Any Exception level that is lower than EL_D	Enabled	Enabled

a. This includes EL3. EL3 is always higher than EL_D.

Note

PSTATE.D is set to 1 at reset and on exception entry.

XNU never clears PSTATE.D

How to unmask PSTATE.D

if we can't unmask PSTATE.D, nothing will work

can we ROP on every syscall entry?

probably; this was way too much effort in the end...

can we move the VBAR?

KPP tries to check this.
KTRR tries to make sure there is nothing executable at EL1 which can write to this system register.

can we just accept this as a limitation and force a different way of calling syscalls?

I hope this isn't giving up, just pragmatic...
If you have a better trick, please let me know!

syscall wrapping

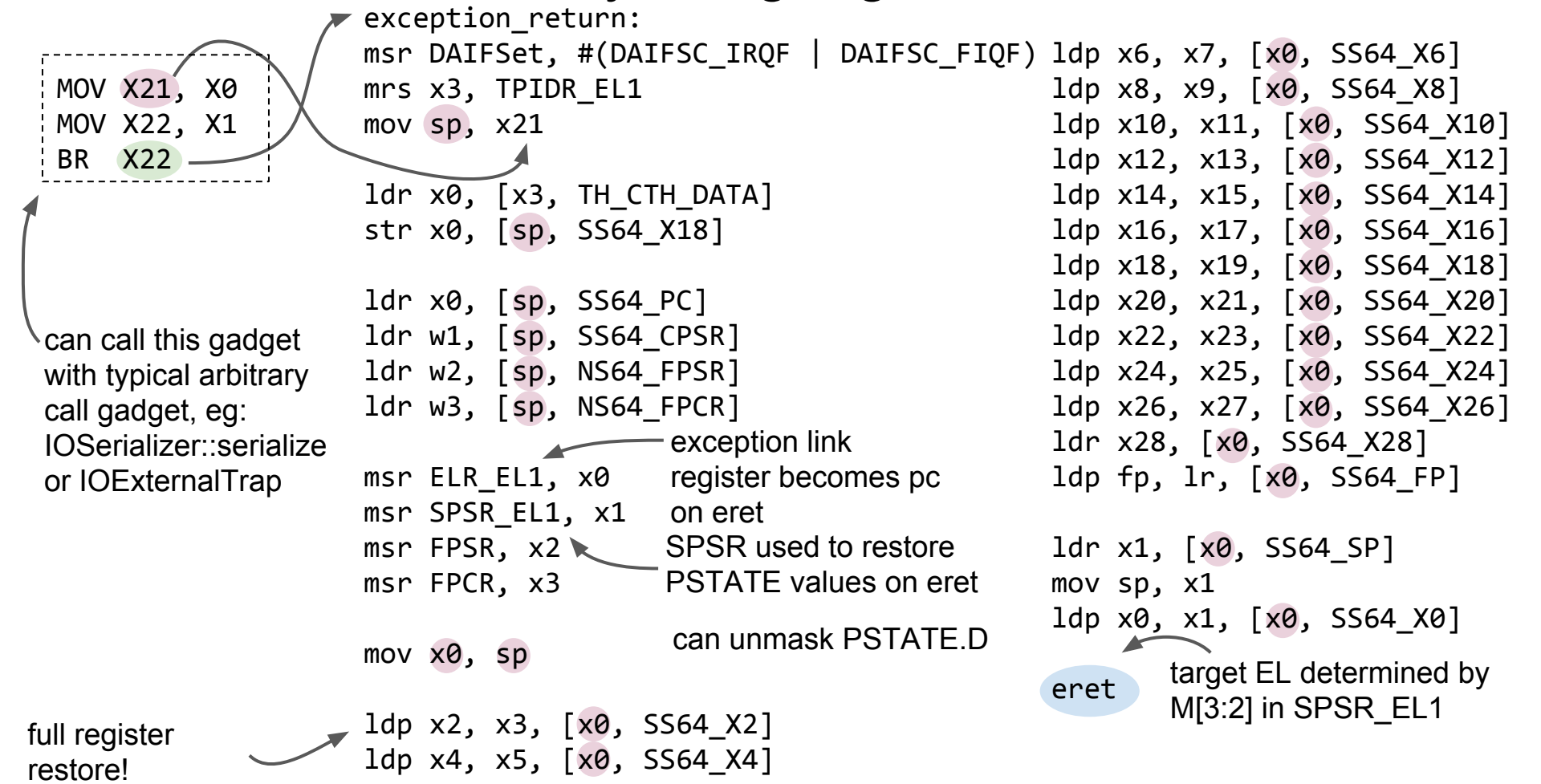
major limitation of this debugger

not useful for full system debugging
"what calls this"

also has its own advantages

very useful for "what exactly does this
syscall call"

a better kernel arbitrary call gadget



what to target?

saw this earlier
First level synchronous exception handler

gets called by fleh_dispatch64

```
fleh_synchronous:
mrs     x1, ESR_EL1
mrs     x2, FAR_EL1
and     w3, w1, #(ESR_EC_MASK)
lsr     w3, w3, #(ESR_EC_SHIFT)
mov     w4, #(ESR_EC_IABORT_EL1)
cmp     w3, w4
b.eq  Lfleh_sync_load_lr
Lvalid_link_register:

PUSH_FRAME
bl      EXT(sleh_synchronous)
POP_FRAME

b     exception_return_dispatch
```

ERET to here

can point this to the real ACT_CONTEXT

expected register state
x21: pointer to arm_context to restore in exception_return_dispatch

x0: pointer to arm_context to pass to SLEH

can point this to a buffer we control with the arguments for the wrapped syscall

sp: top of thread's kernel stack

point this to the thread's real kernel stack

life of a debuggable syscall:

simple arbitrary call primitive allocates struct arm_context in kernel memory with correct arguments for target syscall

userspace syscall args
arm_context {
x16: syscall number
x0: arg0
}

full ERET state
arm_context {
spsr: unmasked D
}

SVC

syscall wrapper calls arbitrary call gadget

calls ERET gadget
erets from EL1 to EL1, unmasking PSTATE.D

HWBP

causes synchronous exception, from EL1 running on SP0 to EL1

FIQ

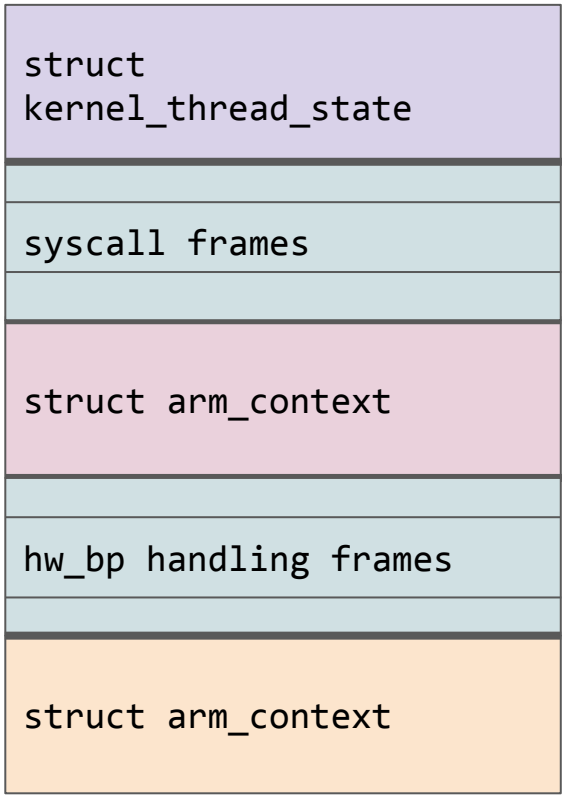
causes FIQ exception, from EL1 running on SP0 to EL1

we're stuck in the "unreachable" loop here

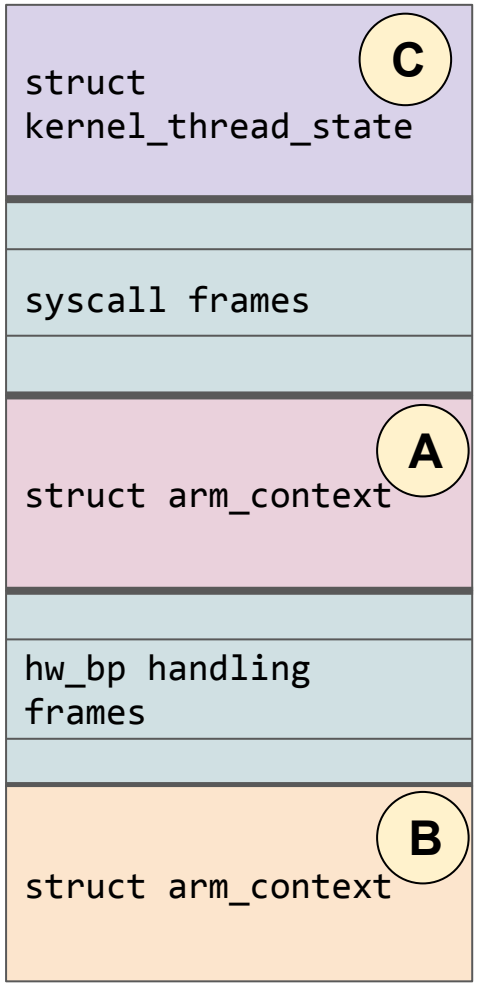
for(;;)

schedule off

if we'll be scheduled off, FIQ posts AST handled right before FIQ ERET's



modifying blocked state



the state when the timer FIQ gets scheduled off at AST

this is always above the top of the thread's kernel stack

top of thread's kernel stack

kernel_thread_state.__sp gives us the bottom of the scheduled off thread's kernel stack, where the FIQ spilled state

the state when the breakpoint was hit - want to expose this to lldb client

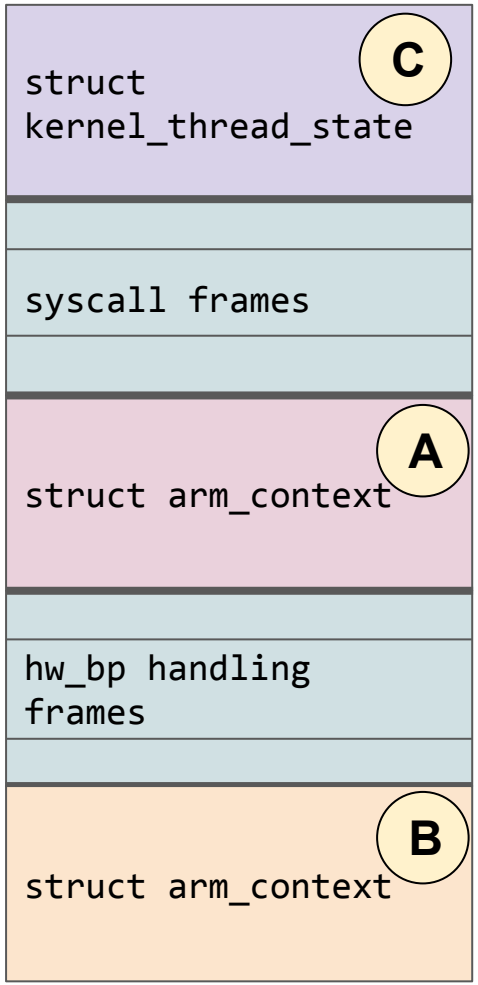
we're stuck in the "unreachable" loop here

`for(;;)`

the state when the spinning infinite loop got interrupted by a timer FIQ

from here we can unwind the stack to find (A)

unblocking the looper



need to safely return from the synchronous exception handler for the hardware breakpoint

can just jump to the epilog of the SLEH; it will handle returning for us

minor problem: although PSTATE.D will be unmasked when we return, when the scheduler puts the looper back on the core the debug systems registers won't be reloaded until the thread returns to userspace

can fix with a return gadget instead:

eret loads this into ELR_EL1

```
wk64(looper_saved_state + offsetof(arm_context_t, ss.ss_64.pc),
     ksym(KSYMBOL_ARM_DEBUG_SET));
```

unblock by eret'ing to arm_debug_set, which loads all the hw bp system registers

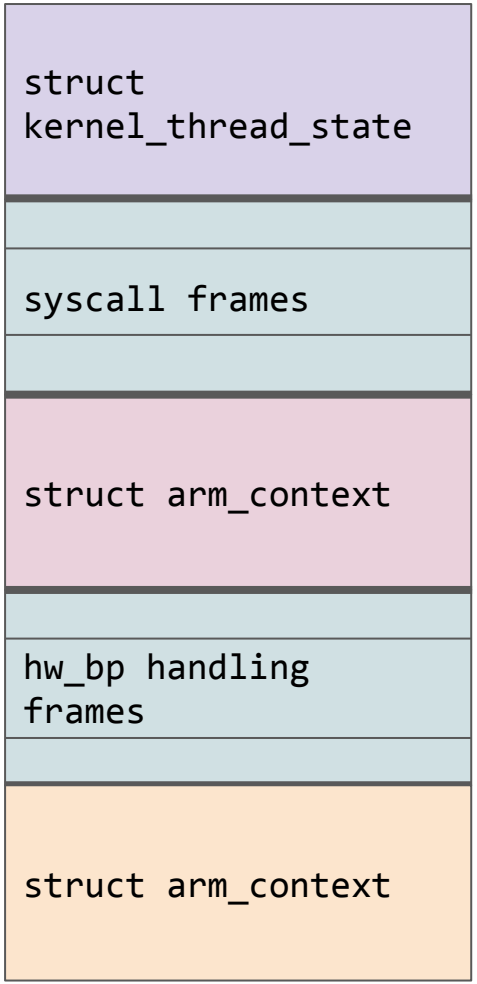
```
wk64(looper_saved_state + offsetof(arm_context_t, ss.ss_64.x[0]),
     thread_get_debug_area(debugee_thread_port));
```

first arg to arm_debug_set is the thread's debug state

```
wk64(looper_saved_state + offsetof(arm_context_t, ss.ss_64.lr),
     ksym(KSYMBOL_SLEH_SYNC_EPILOG));
```

point the lr to the SLEH epilog

structure of the monitor thread



monitor thread pins itself to the same core as the debugee

finds the target thread's kernel stack and its kernel_thread_state

from there can find the sp and the FIQ state

send an exception message to the lldb client and enter a command processing loop, exposing this area as the current register state

when command loop exits, check the exit reason. If it's continue, fix up the FIQ state as we saw earlier, then exit the monitor thread!

when scheduler schedules the debugee, it will now continue!

does this PC match the expected infinite loop instruction?

yes? unwind the stack to find the state spilled by the hw bp sync exception

connecting lldb via KDP

pretty simple client - server protocol over UDP

server listens on port 41139

client also listens on a port for exception messages from the server

"real" KDP has to send the UDP packets itself

we can just do it in another userspace thread

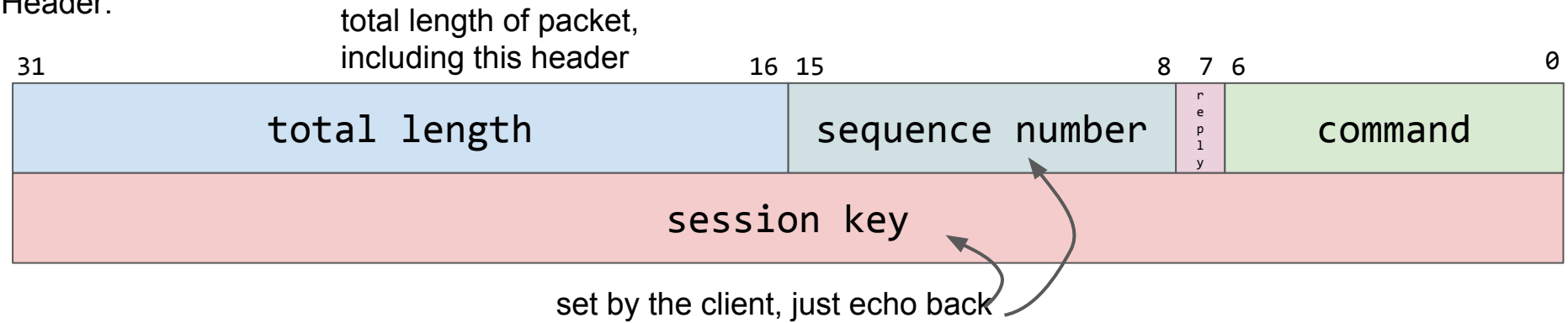
list of (implemented) kdp commands

KDP_CONNECT	0	KDP_BREAKPOINT64_SET	22
KDP_REATTACH	18	KDP_BREAKPOINT64_REMOVE	23
KDP_VERSION	3	KDP_RESUMECPUS	12
KDP_HOSTINFO	2	KDP_READREGS	7
KDP_DISCONNECT	1	KDP_WRITEREGS	8
KDP_KERNELVERSION	24	KDP_KERNEL_CONTINUE (KDP_READIOPORT)	27
KDP_READMEM64	20	KDP_KERNEL_SINGLE_STEP (KDP_WRITEIOPORT)	28
KDP_WRITEMEM64	21		

They pretty much all do what you'd expect

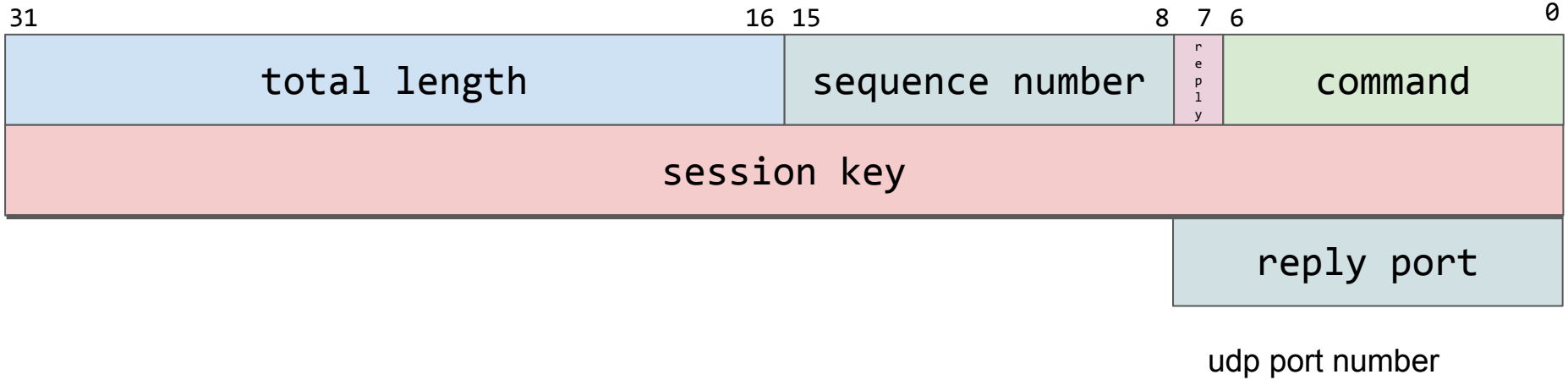
KDP packet structure

Header:



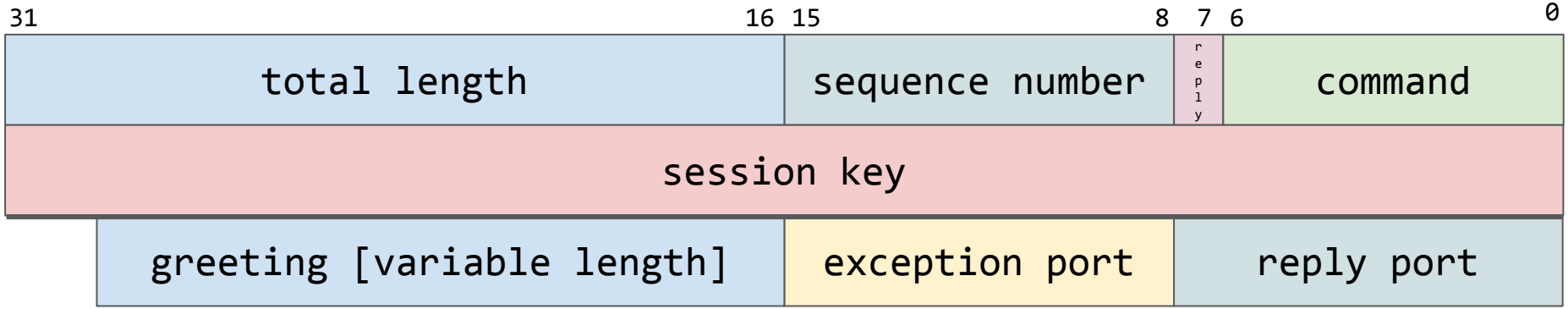
KDP_REATTACH

Header:



KDP_CONNECT

Header:

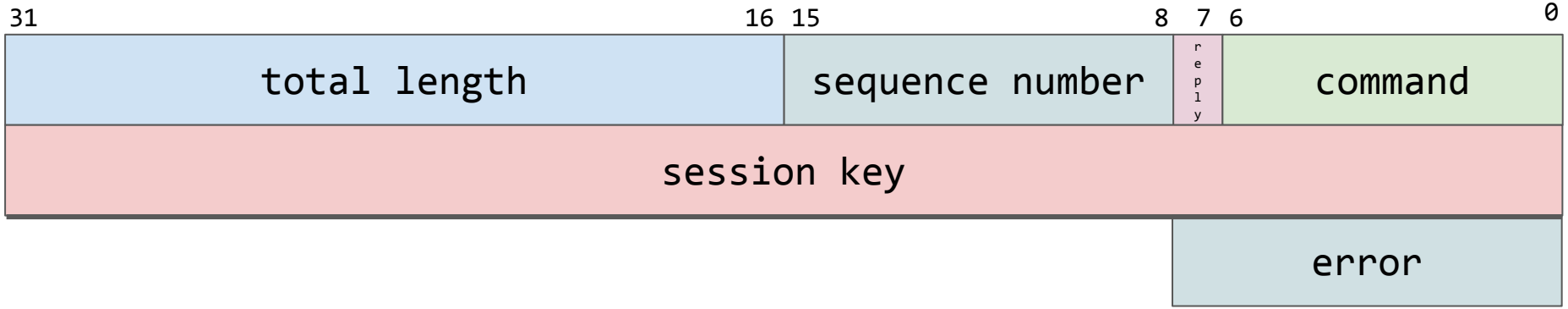


udp port for exception messages
(we tell the client an event has happened)

lldb actually sends this greeting:

`"Greetings from LLDB..."`

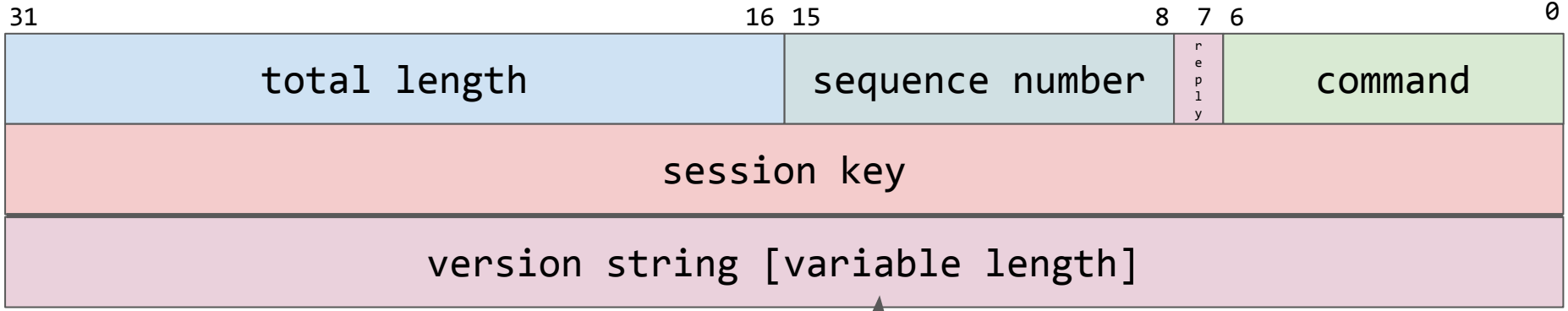
KDP_CONNECT_REPLY



lots of messages use this structure for reply messages

KDP_KERNELVERSION_REPLY

The reply to a kernelversion request packet



global variable version

```
Darwin Kernel Version 17.2.0:Fri Sep 29 18:14:50 PDT 2017;
root:xnu-4570.20.62~4/RELEASE_ARM64_T8010;UUID=5E450F40-E224-33F7-946
B-A764D21DF3FC;stext=0xfffffff00ec04000
```

kernel_uuid_string

kernel base address (the `0xfeedfacf`)

LLDB uses this to parse the loaded kernel, find loaded kexts etc

example session

this is the stock MacOS lldb you get with xcode

```
$ lldb kernelcache.ip7_11_1_2.uncomp
```

uncompressed kernel cache from IPSW, extract with joker

```
(lldb) target create "kernelcache.ip7_11_1_2.uncomp"
Current executable set to 'kernelcache.ip7_11_1_2.uncomp' (arm64).
(lldb) kdp-remote 172.20.10.11
```

iPhone's IP

```
Version: Darwin Kernel Version 17.2.0: Fri Sep 29 18:14:50 PDT 2017;
root:xnu-4570.20.62~4/RELEASE_ARM64_T8010; UUID=5E450F40-E224-33F7-946B-A764D21DF3FC;
stext=0xffffff021804000
Kernel UUID: 5E450F40-E224-33F7-946B-A764D21DF3FC
Load Address: 0xffffff021804000
Kernel slid 0x1a800000 in memory.
Loaded kernel file
/Users/ianbeer/prog/ios/iPhone7_firmwares/11.1.2/kernelcache.ip7_11_1_2.uncomp
```

kernel version string we built

lldb client computes this from stext

```
Loading 165 kext modules warning: Can't find binary/dSYM for com.apple.kec.corecrypto
(B3028F6D-3547-37E1-B166-DB8972637087)
.warning: Can't find binary/dSYM for com.apple.kec.Libm
(51AFA03E-8041-3D11-BD40-A6D1AED1C667)
.warning: Can't find binary/dSYM for com.apple.kec.pthread
(422770EA-D9A0-3B84-B683-15A6910AB51E)
.warning: Can't find binary/dSYM for com.apple.iokit.IOSlowAdaptiveClockingFamily
(1D16EC28-554A-3C74-B14A-AA62B624EDF1)
...
```

for MacOS kernel debug we get some of these; nothing for iOS

```
. done.
Process 1 stopped
* thread #1, stop reason = signal SIGSTOP
    frame #0: 0xfffffff0218cc474
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol49$$kernelcache.ip7_11_1_2.uncomp
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol49$$kernelcache.ip7_11_1_2.uncomp:
->  0xfffffff0218cc474 <+0>: msr    DAIFSet, #0x3
    0xfffffff0218cc478 <+4>: mrs    x3, TPIDR_EL1
    0xfffffff0218cc47c <+8>: mov    sp, x21
Target 0: (kernelcache.ip7_11_1_2.uncomp) stopped.
(lldb)
```

this is a lie, we're not actually stopped here. The initial connection stopped state is faked.

can't single step yet, but can set a breakpoint and continue

```
(lldb) image list
[  0] 5E450F40-E224-33F7-946B-A764D21DF3FC 0xfffffff021804000
/Users/ianbeer/prog/ios/iPhone7_firmwares/11.1.2/kernelcache.ip7_11_1_2.uncomp
(lldb)
```

this is only going to find names with symbols

```
(lldb) image lookup -rn kalloc
1 match found in
/Users/ianbeer/prog/ios/iPhone7_firmwares/11.1.2/kernelcache.ip7_11_1_2.uncomp:
        Address: kernelcache.ip7_11_1_2.uncomp[0xffffffff007101248]
(kernelcache.ip7_11_1_2.uncomp.__TEXT_EXEC.__text + 234056)
        Summary: kernelcache.ip7_11_1_2.uncomp`kalloc_external
```

```
(lldb) disassemble --name kalloc_external
kernelcache.ip7_11_1_2.uncomp`kalloc_external:
    0xfffffff021901248 <+0>:  sub    sp, sp, #0x20             ; =0x20
    0xfffffff02190124c <+4>:  stp    x29, x30, [sp, #0x10]
    0xfffffff021901250 <+8>:  add    x29, sp, #0x10            ; =0x10
    0xfffffff021901254 <+12>: str    x0, [sp, #0x8]
    0xfffffff021901258 <+16>: adrp   x2, 1211
    0xfffffff02190125c <+20>: add    x2, x2, #0x400            ; =0x400
    0xfffffff021901260 <+24>: orr    w1, wzr, #0x1
    0xfffffff021901264 <+28>: add    x0, sp, #0x8              ; =0x8
    0xfffffff021901268 <+32>: bl     0xfffffff021900fbc        ;
___lldb_unnamed_symbol428$$kernelcache.ip7_11_1_2.uncomp
    0xfffffff02190126c <+36>: ldp    x29, x30, [sp, #0x10]
    0xfffffff021901270 <+40>: add    sp, sp, #0x20             ; =0x20
    0xfffffff021901274 <+44>: ret
```

looking at the source that's kalloc_canblock, a more interesting place to put a breakpoint

```
(lldb) break set --address 0xfffffff021900fbc
```

```
(lldb) command alias kc process plugin packet send -c 27
(lldb) command alias ks process plugin packet send -c 28
(lldb) kc
9b5d080000000000
(lldb) c
Process 1 resuming
```

this is the only hack you have to do in the client; my fake debug server needs to know when its setting/removing a breakpoint for a single-step/continue.

our breakpoint was hit! :)

```
Process 1 stopped
* thread #1, stop reason = breakpoint 1.1
    frame #0: 0xffffffff021900fbc
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol428$$kernelcache.ip7_11_1_2.uncomp
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol428$$kernelcache.ip7_11_1_2.uncomp:
->  0xffffffff021900fbc <+0>: sub    sp, sp, #0x60             ; =0x60
    0xffffffff021900fc0 <+4>: stp    x26, x25, [sp, #0x10]
    0xffffffff021900fc4 <+8>: stp    x24, x23, [sp, #0x20]
Target 0: (kernelcache.ip7_11_1_2.uncomp) stopped.
(lldb)
```

```
(lldb) reg r
General Purpose Registers:
        x0 = 0xffffffe027b138e8
        x1 = 0x0000000000000001
        x2 = 0xffffffff021dbdda8  atm_manager + 1648
        x3 = 0x0000000000000000
        x4 = 0xffffffe027b139f8
        x5 = 0x0000000010000003
        x6 = 0xffffffe004cda9a0
        x7 = 0xffffffe027b139c8
        x8 = 0x0000000000007fe8
        x9 = 0x00000001029e0000
       x10 = 0x0000000218000000
       x11 = 0x0000000001000000
       x12 = 0x0000000001000000
       x13 = 0x0000000001000000
       x14 = 0xffffffe027b139a8
       x15 = 0xffffffe001e3c1b0
       x16 = 0xffffffe001e3c1b0
       x17 = 0xffffffe001e3c1b0
       x18 = 0x0000000000000000
       x19 = 0xffffffe004cda9a0
       x20 = 0x0000000218000000
```

```
       x21 = 0x0000000000000001
       x22 = 0xffffffe027b139f8
       x23 = 0x0000000102a4d5ba
       x24 = 0x0000000000000027
       x25 = 0xffffffe004cda9e0
       x26 = 0x0000000102a4d593
       x27 = 0x0000000000000000
       x28 = 0x000000000000003f
        fp = 0xffffffe027b13940
        lr = 0xffffffff021984b00
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol1211$$kernelcache.ip7_11_1_2.uncomp +
688
        sp = 0xffffffe027b13820
        pc = 0xffffffff021900fbc
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol428$$kernelcache.ip7_11_1_2.uncomp
      cpsr = 0x20400104

(lldb)
```

```
(lldb) bt
* thread #1, stop reason = breakpoint 1.1
  * frame #0: 0xffffffff021900fbc
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol428$$kernelcache.ip7_11_1_2.uncomp
    frame #1: 0xffffffff021984b00
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol1211$$kernelcache.ip7_11_1_2.uncomp
+ 688
    frame #2: 0xffffffff0218e2e80
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol197$$kernelcache.ip7_11_1_2.uncomp
+ 2704
    frame #3: 0xffffffff0218f2458
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol307$$kernelcache.ip7_11_1_2.uncomp
+ 972
    frame #4: 0xffffffff0219deff8
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol1697$$kernelcache.ip7_11_1_2.uncomp
+ 4388
    frame #5: 0xffffffff0218cc1e0
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol34$$kernelcache.ip7_11_1_2.uncomp +
40
```

```
(lldb) x/1xg $x0
0xffffffe027b138e8: 0x000000000000003f
```

we set the breakpoint at kalloc_canblock, the first argument is a pointer to the size to allocate

```
(lldb) finish
Process 1 stopped
* thread #1, stop reason = step out
    frame #0: 0xffffffff021984b00
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol1211$$kernelcache.ip7_11_1_2.u
ncomp + 688
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol1211$$kernelcache.ip7_11_1_2.u
ncomp:
->  0xffffffff021984b00 <+688>: mov    x20, x0
    0xffffffff021984b04 <+692>: cbz    x20, 0xffffffff021984b40   ; <+752>
    0xffffffff021984b08 <+696>: orr    w8, wzr, #0x3
Target 0: (kernelcache.ip7_11_1_2.uncomp) stopped.
(lldb) reg r x0
      x0 = 0xffffffe004f60140
```

lldb client handles the logic of setting the BP in the right place for this

looks plausible for a kalloc.64

```
(lldb) finish
Process 1 stopped
* thread #1, stop reason = step out
    frame #0: 0xfffffff0218e2e80
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol197$$kernelcache.ip7_11_1_2.uncomp +
2704
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol197$$kernelcache.ip7_11_1_2.uncomp:
->  0xfffffff0218e2e80 <+2704>: cbnz   w0, 0xfffffff0218e37ac    ; <+5052>
    0xfffffff0218e2e84 <+2708>: ldur   x8, [x29, #-0x98]
    0xfffffff0218e2e88 <+2712>: str    x8, [x27]
Target 0: (kernelcache.ip7_11_1_2.uncomp) stopped.
(lldb) x/10xg 0xffffffe004f60140
0xffffffe004f60140: 0xdeadbeef00000003 0x0000000000000000
0xffffffe004f60150: 0x0000000000000027 0x544f4e3e4c4d583c
0xffffffe004f60160: 0x5f594c4c4145525f 0x4c4d582f3c4c4d58
0xffffffe004f60170: 0x3f3332213e3c3f3e 0xde00333231234021
0xffffffe004f60180: 0xffffffe004e4cae0 0xffffffe0018e8540
(lldb) x/s 0xffffffe004f60158
0xffffffe004f60158: "<XML>NOT_REALLY_XML</XML>?<>!23?!@#123"
```

```
0xffffffe004f60140: 0xdeadbeef00000003 0x0000000000000000
0xffffffe004f60150: 0x0000000000000027 0x544f4e3e4c4d583c
0xffffffe004f60160: 0x5f594c4c4145525f 0x4c4d582f3c4c4d58
0xffffffe004f60170: 0x3f3332213e3c3f3e 0xde00333231234021
0xffffffe004f60180: 0xffffffe004e4cae0 0xffffffe0018e8540
(lldb) x/s 0xffffffe004f60158
0xffffffe004f60158: "<XML>NOT_REALLY_XML</XML>?<>!23?!@#123"
```

what is this structure?

used to be a
common target for heap disclosure

```
struct vm_map_copy {
     int            type;
#define VM_MAP_COPY_ENTRY_LIST      1
#define VM_MAP_COPY_OBJECT          2
#define VM_MAP_COPY_KERNEL_BUFFER   3
     vm_object_offset_t offset;
     vm_map_size_t      size;
     union {
          struct vm_map_header    hdr;      /* ENTRY_LIST */
          vm_object_t             object;   /* OBJECT */
          uint8_t                 kdata[0]; /* KERNEL_BUFFER */
     } c_u;
};
```

this kalloc call was in vm_map_copyin_kernel_buffer

```
(lldb) ks
9c7b080000000000
(lldb) s
Process 1 stopped
* thread #1, stop reason = EXC_BREAKPOINT (code=1, subcode=0x1)
    frame #0: 0xffffffff0218e2e84
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol197$$kernelcache.ip7_11_1_2.uncom
p + 2708
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol197$$kernelcache.ip7_11_1_2.uncom
p:
->  0xffffffff0218e2e84 <+2708>: ldur   x8, [x29, #-0x98]
    0xffffffff0218e2e88 <+2712>: str    x8, [x27]
    0xffffffff0218e2e8c <+2716>: add    x14, sp, #0x58            ; =0x58
Target 0: (kernelcache.ip7_11_1_2.uncomp) stopped.
```

```
(lldb) ks
9c81080000000000
(lldb) s
Process 1 stopped
* thread #1, stop reason = EXC_BREAKPOINT (code=1, subcode=0x1)
    frame #0: 0xffffffff0218e2e88
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol197$$kernelcache.ip7_11_1_2.uncom
p + 2712
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol197$$kernelcache.ip7_11_1_2.uncom
p:
->  0xffffffff0218e2e88 <+2712>: str    x8, [x27]
    0xffffffff0218e2e8c <+2716>: add    x14, sp, #0x58            ; =0x58
    0xffffffff0218e2e90 <+2720>: mov    w5, #0x10000000
Target 0: (kernelcache.ip7_11_1_2.uncomp) stopped.
```

```
(lldb) ks
9c87080000000000
(lldb) s
Process 1 stopped
* thread #1, stop reason = EXC_BREAKPOINT (code=1, subcode=0x1)
    frame #0: 0xffffffff0218e2e8c
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol197$$kernelcache.ip7_11_1_2.uncom
p + 2716
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol197$$kernelcache.ip7_11_1_2.uncom
p:
->  0xffffffff0218e2e8c <+2716>: add    x14, sp, #0x58            ; =0x58
    0xffffffff0218e2e90 <+2720>: mov    w5, #0x10000000
    0xffffffff0218e2e94 <+2724>: movk   w5, #0x3
Target 0: (kernelcache.ip7_11_1_2.uncomp) stopped.
```

```
(lldb) kc
9b8d080000000000
(lldb) c
Process 1 resuming
Process 1 stopped
* thread #1, stop reason = breakpoint 1.1
    frame #0: 0xffffffff021900fbc
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol428$$kernelcache.ip7_11_1_2.uncom
p
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol428$$kernelcache.ip7_11_1_2.uncom
p:
->  0xffffffff021900fbc <+0>: sub    sp, sp, #0x60             ; =0x60
    0xffffffff021900fc0 <+4>: stp    x26, x25, [sp, #0x10]
    0xffffffff021900fc4 <+8>: stp    x24, x23, [sp, #0x20]
Target 0: (kernelcache.ip7_11_1_2.uncomp) stopped.
(lldb)
```

Conclusion

built a working, useful same-machine kernel debugger

minimal feature set, enough for my current purposes

KPP/KTRR: if you can single step a kernel thread with them there, you can probably steal whatsapp/wechat/etc messages, log GPS etc.

release

Was supposed to be released already; now very soon!

initial version only supports 11.1.2