

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

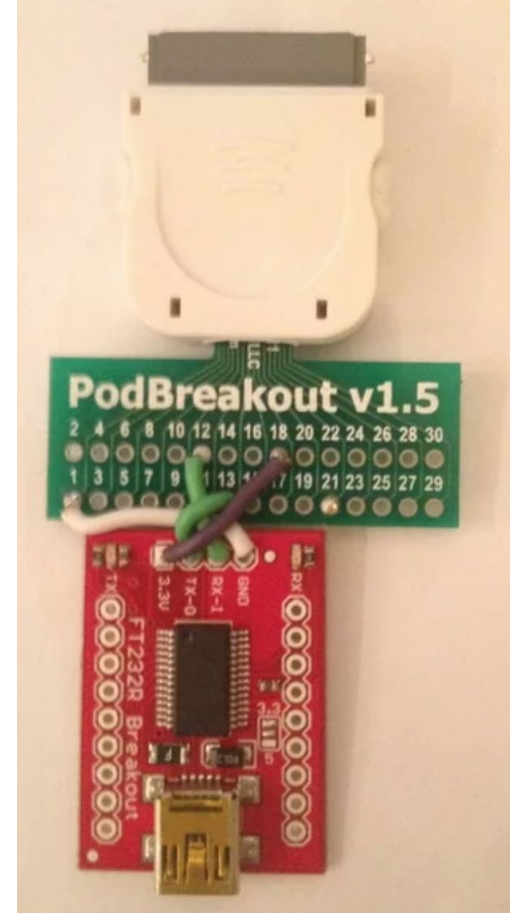


image: <http://www.instructables.com/id/Apple-iOS-SerialUSB-Cable-for-Kernel-Debugging/>

**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

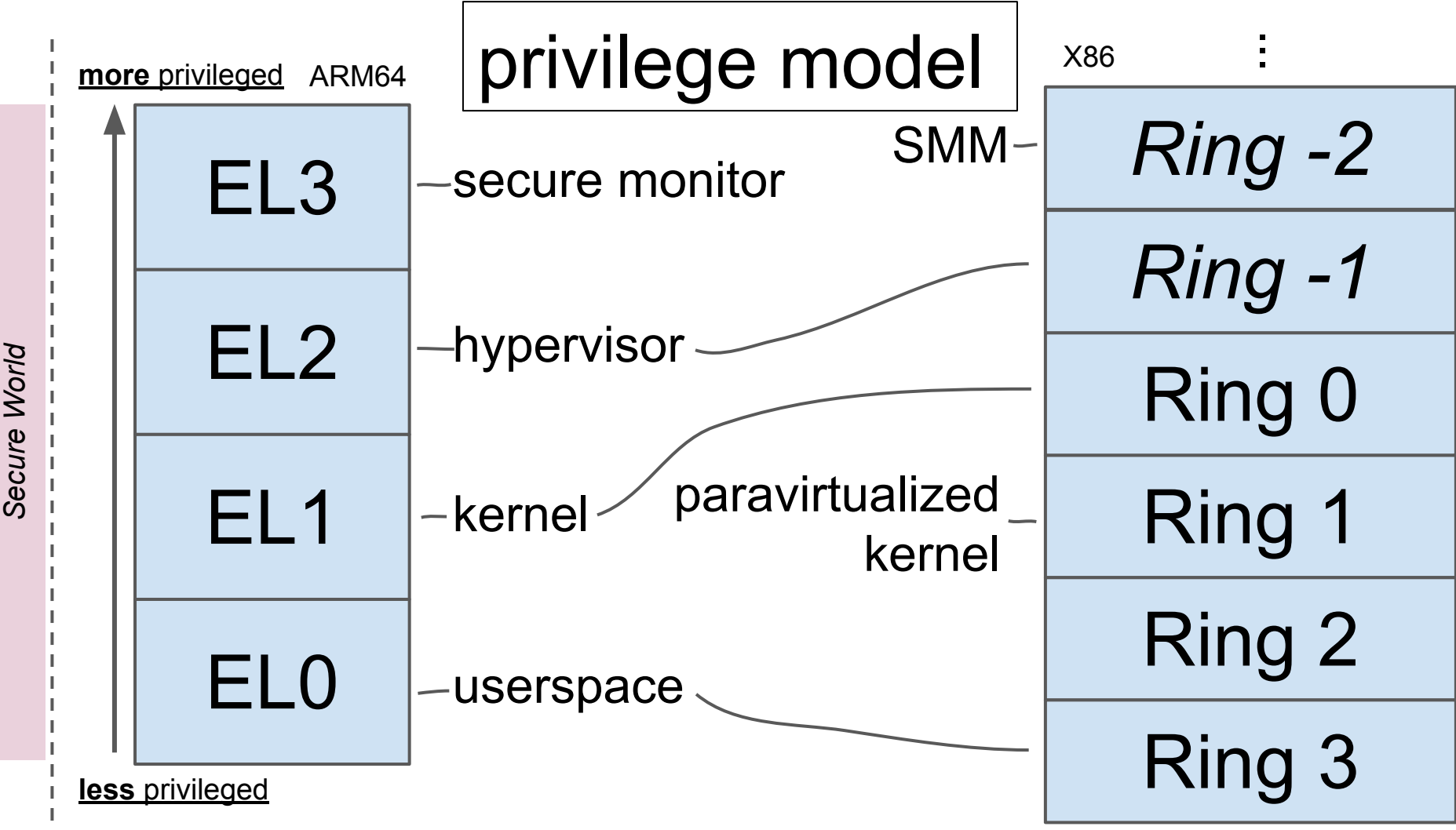
must still be a bit useful

must be reasonably easy to use

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

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

connect with a normal debugging client (eg lldb)



# exception levels in iOS

Exception level restricts:

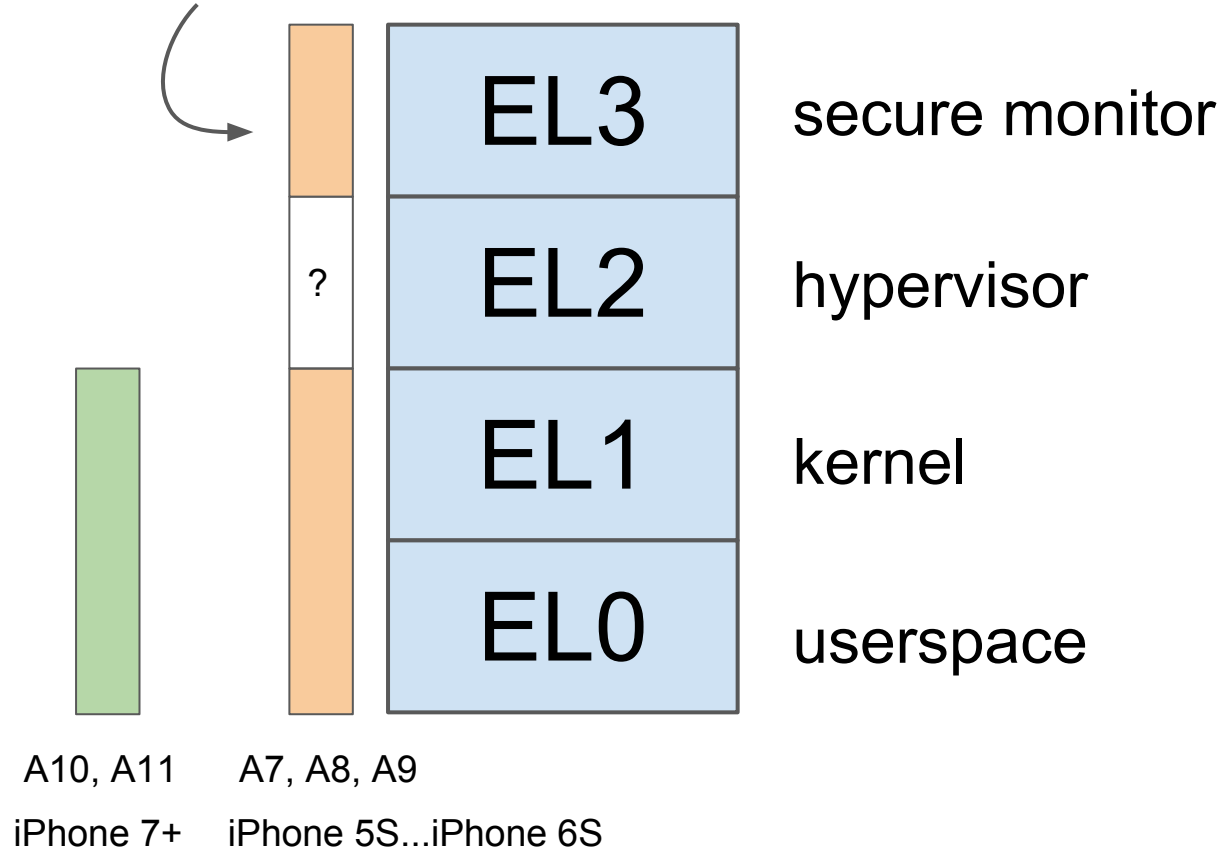
- \* system register access
- \* what instructions may trap

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

KPP runs here on A7-A9

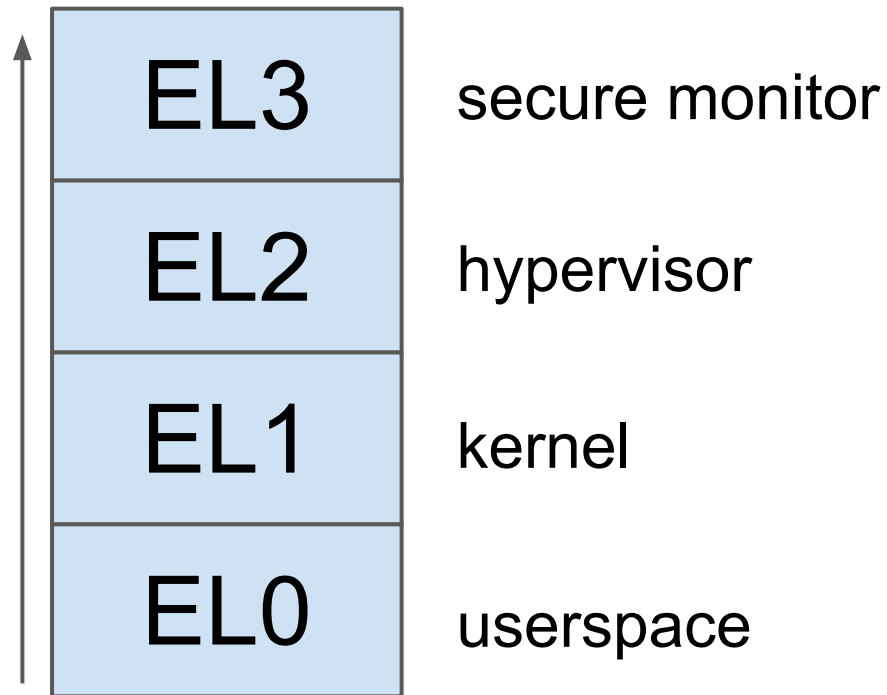


# Exceptions

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



# System Registers

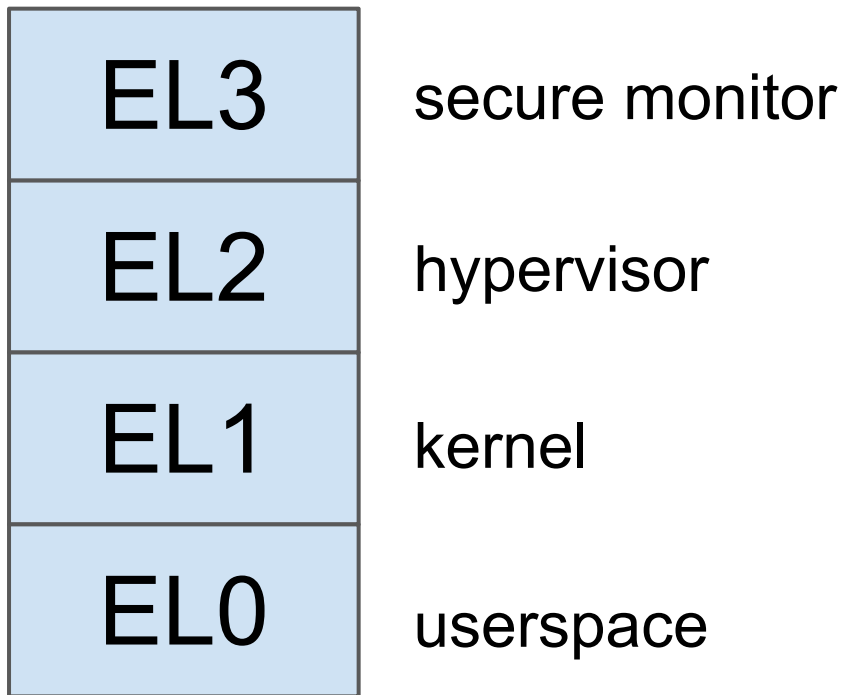
## Vector Base Address Register

Virtual memory address of the start of the exception vector table

**VBAR\_EL1**

for exceptions taken to EL1

**Exception Level 1**



The `_ELx` 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

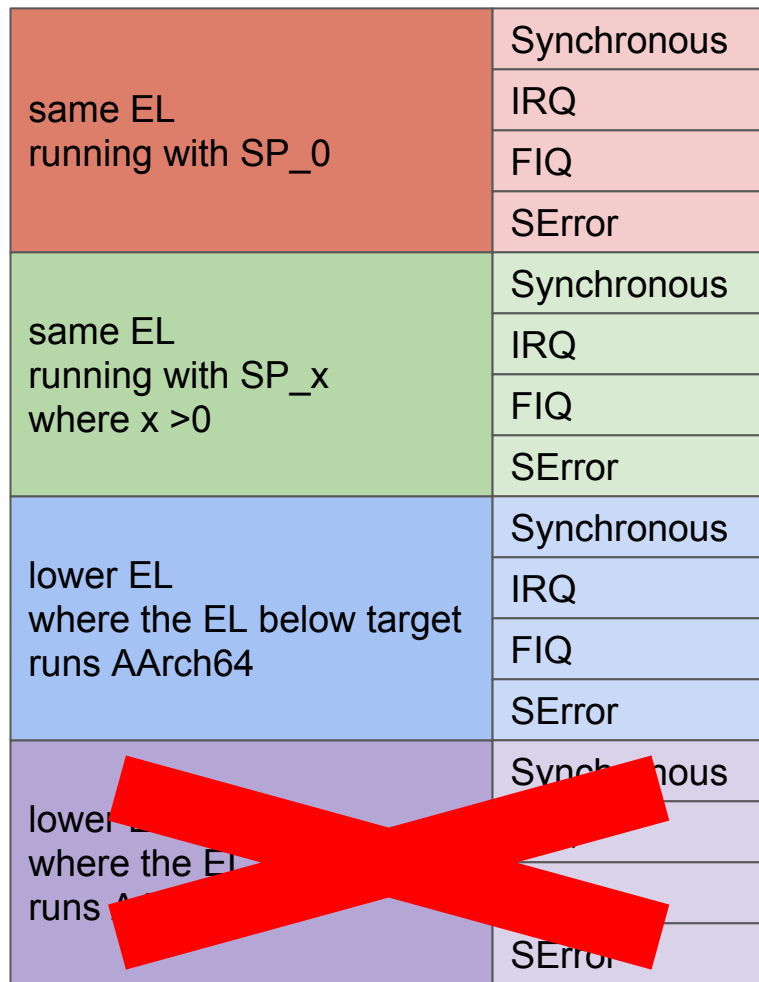
ARM64 XNU VBAR\_EL1 entry for  
synchronous exception from EL0:

locore.s

```
Le10_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



The diagram illustrates the VBAR\_ELx structure, which is a table of pointers to exception handlers. It is organized into four main sections, each with a specific color and a set of four entries: Synchronous, IRQ, FIQ, and SError. The first two sections (red and green) represent the same EL running with different stack pointers (SP\_0 and SP\_x where x > 0). The third section (blue) represents a lower EL where the EL below target runs AArch64. The fourth section (purple) represents a lower EL where the EL below target runs AArch32, but it is crossed out with a large red X, indicating it is not used in iOS 11.

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

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

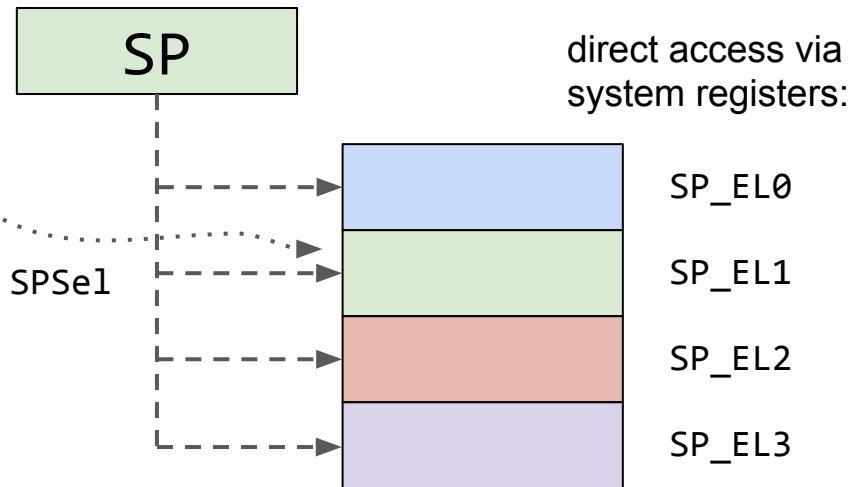
sp is not a simple register. It aliases one of four actual hardware registers.

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

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



when exception is taken, SP aliases the SP for the target EL.  
Setting SPSe1 to 0 switches SP to alias SP\_EL0  
(SPSe1 is a flag, not an index)

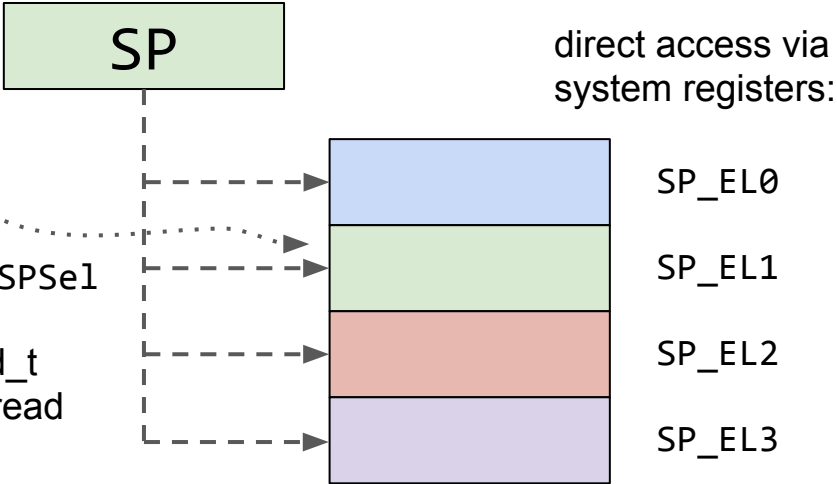
# Exception handling in ARM64 XNU - EL0 SVC

VBAR\_EL1 + 0x400:

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

```
mrs x0, TPIDR_EL1
```

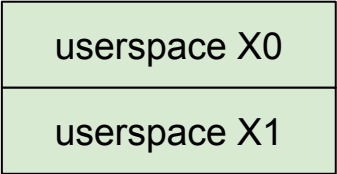
system register containing thread\_t  
pointer for currently executing thread



SPSe1

SP\_EL0  
SP\_EL1  
SP\_EL2  
SP\_EL3

cpu exception stack:



# Exception handling in ARM64 XNU - EL0 SVC

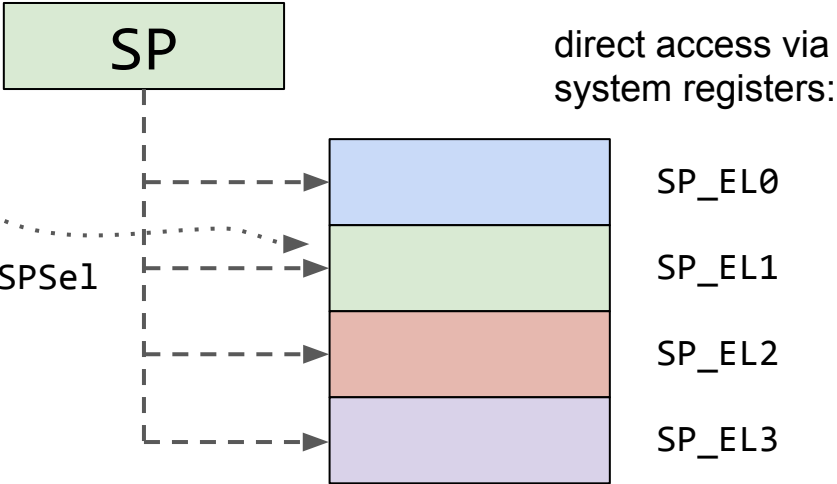
VBAR\_EL1 + 0x400:

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

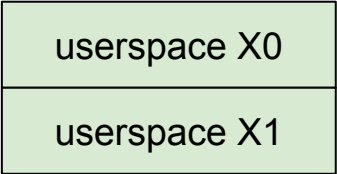
```
mrs x0, TPIDR_EL1
```

```
mrs x1, SP_EL0
```

read what SP was in EL0



cpu exception stack:



## Exception handling in ARM64 XNU - EL0 SVC

VBAR\_EL1 + 0x400:

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

```
mrs x0, TPIDR_EL1
```

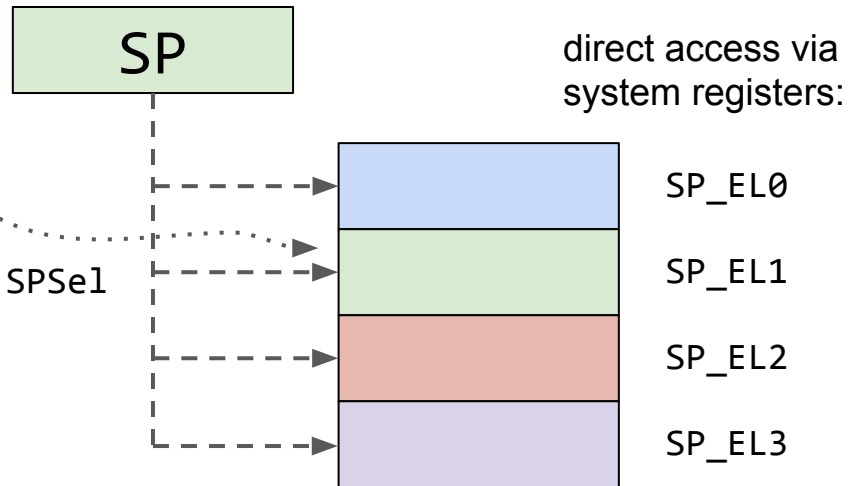
```
mrs x1, SP_EL0
```

```
add x0, x0, ACT_CONTEXT
```

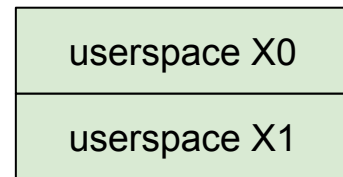
```
DECLARE("ACT_CONTEXT",  
        offsetof(struct thread, machine.contextData));
```

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

```
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);
```



cpu exception stack:



# 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
```

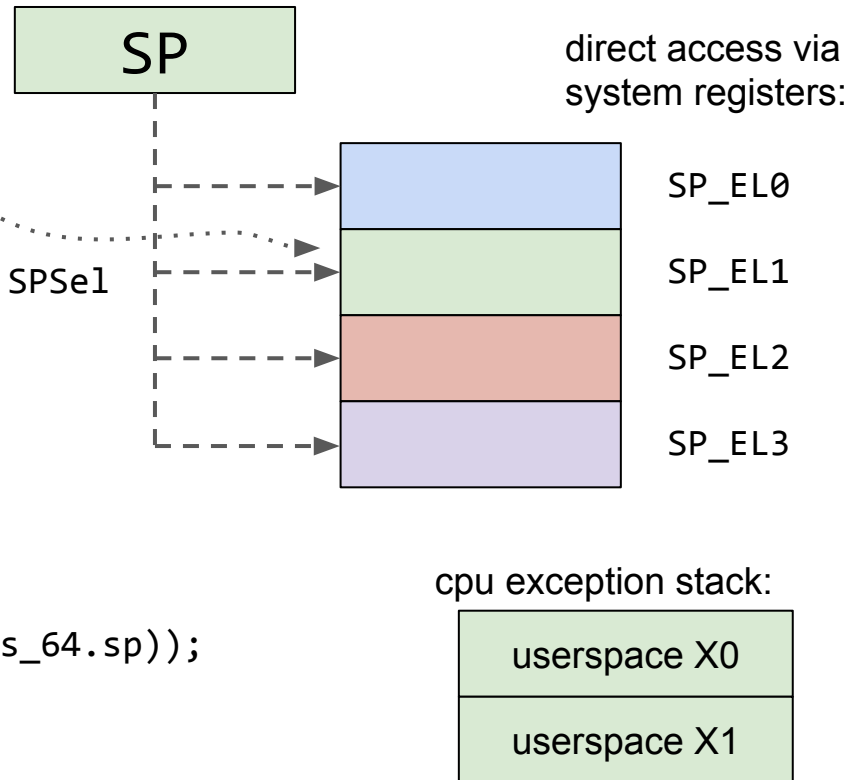
```
add x0, x0, ACT_CONTEXT
```

```
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



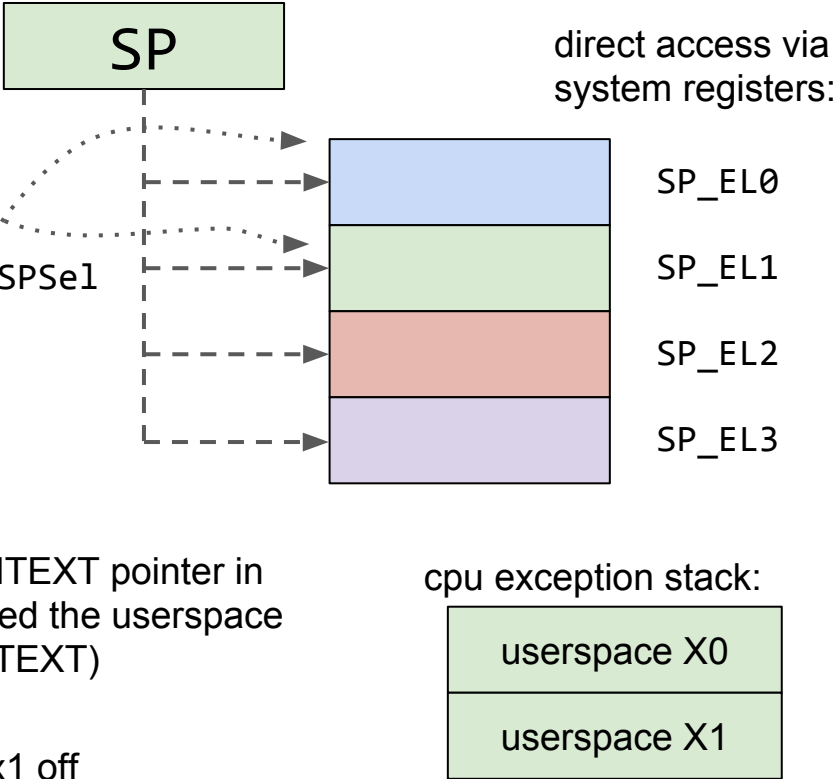
# Exception handling in ARM64 XNU - EL0 SVC

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
```

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



## 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]
```

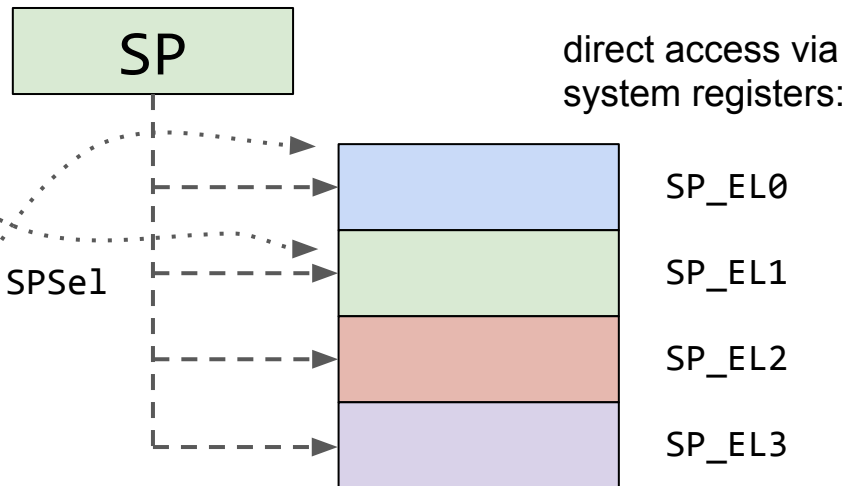
```
str x1, [x0, SS64_SP]
```

```
msr SP_EL0, x0
```

```
ldp x0, x1, [sp], #16
```

```
msr SPSe1, #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
```

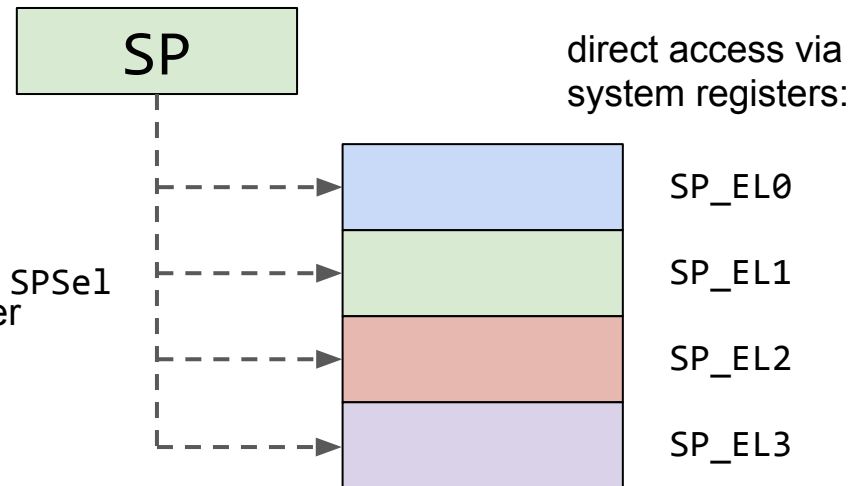
read the thread register

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

```
mov sp, x1
```

pivot to the thread's  
kernelstack

load the thread's kernel stack pointer



```
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]
...
```

saves remaining general purpose  
registers to region pointed to by X0

(for syscall case this is ACT\_CONTEXT)

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

saves NEON registers

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

spill things which aren't real registers

exception link register becomes saved PC

saved program state register becomes saved current program state

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

# 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

← load the second and third arguments for sleh\_synchronous

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  
b1 EXT(sleh\_synchronous)  
POP\_FRAME

first argument (X0) is still the ACT\_CONTEXT (userspace state)

# 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;
    ...

    /* 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;
    }
```

*Diagram annotations:*

- Curved arrow from `uint32_t esr` to `mrs x1, ESR_EL1`
- Curved arrow from `vm_offset_t far` to `mrs x2, FAR_EL1`
- Curved arrow from `panic("Invalid SVC_64 context");` to `handle_svc(state);` with the text "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) {
```

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

...

```
    case ESR_EC_DABORT_EL0:
```

```
        handle_abort(state, esr, far, recover, inspect_data_abort, handle_user_abort);  
        assert(0); /* Unreachable */
```

 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

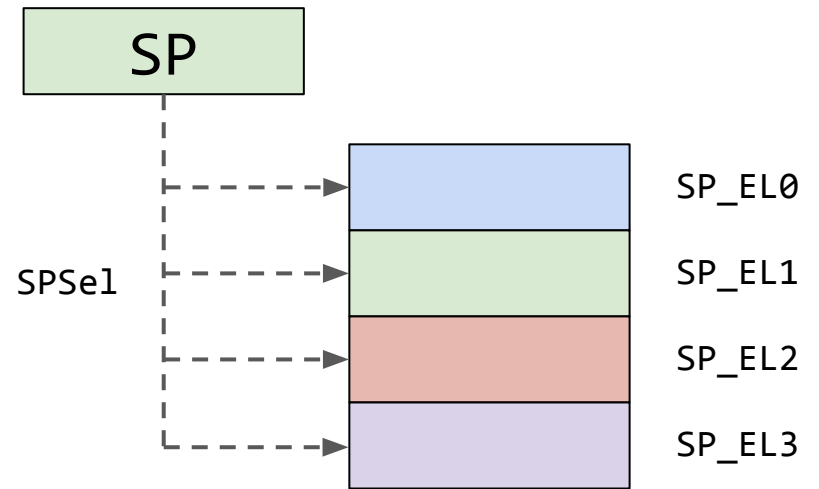
```
void
sleh_synchronous(arm_context_t *context, uint32_t esr, vm_offset_t far)
{
    esr_exception_class_t    class = ESR_EC(esr);
    arm_saved_state_t        *state = &context->ss;

    ...

    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
```

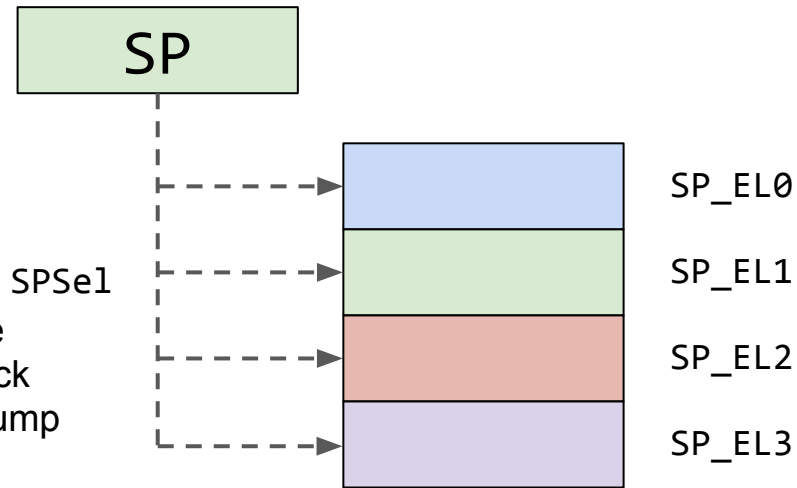
make space on the per-core exception stack for a full register dump, just in case we will panic

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 SPSe1, #0      ← switch to SP_EL0  
                    (which is the thread's kernel stack)  
  
sub sp, sp, ARM_CONTEXT_SIZE  
stp x0, x1, [sp, SS64_X0] ← make space on the  
                             thread's kernel stack  
                             for a full register dump  
  
add x0, sp, ARM_CONTEXT_SIZE  
str x0, [sp, SS64_SP] ← fill in the correct sp  
                       value  
...  
  
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:

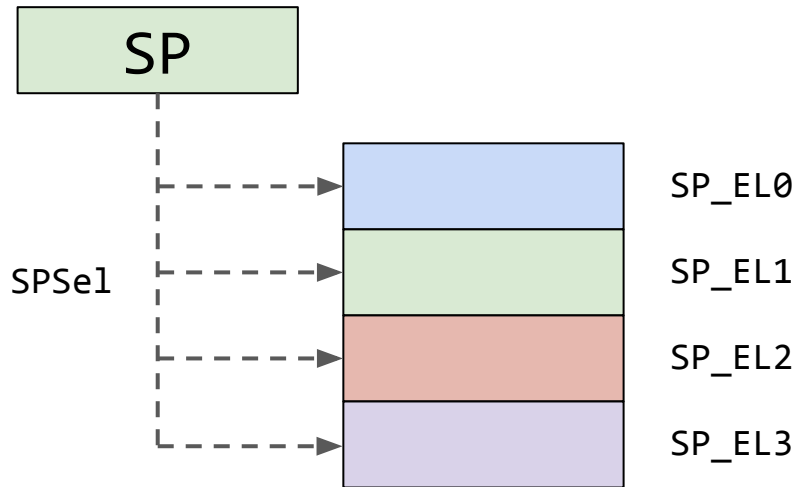
```
mrs x1, TPIDR_EL1
```

```
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

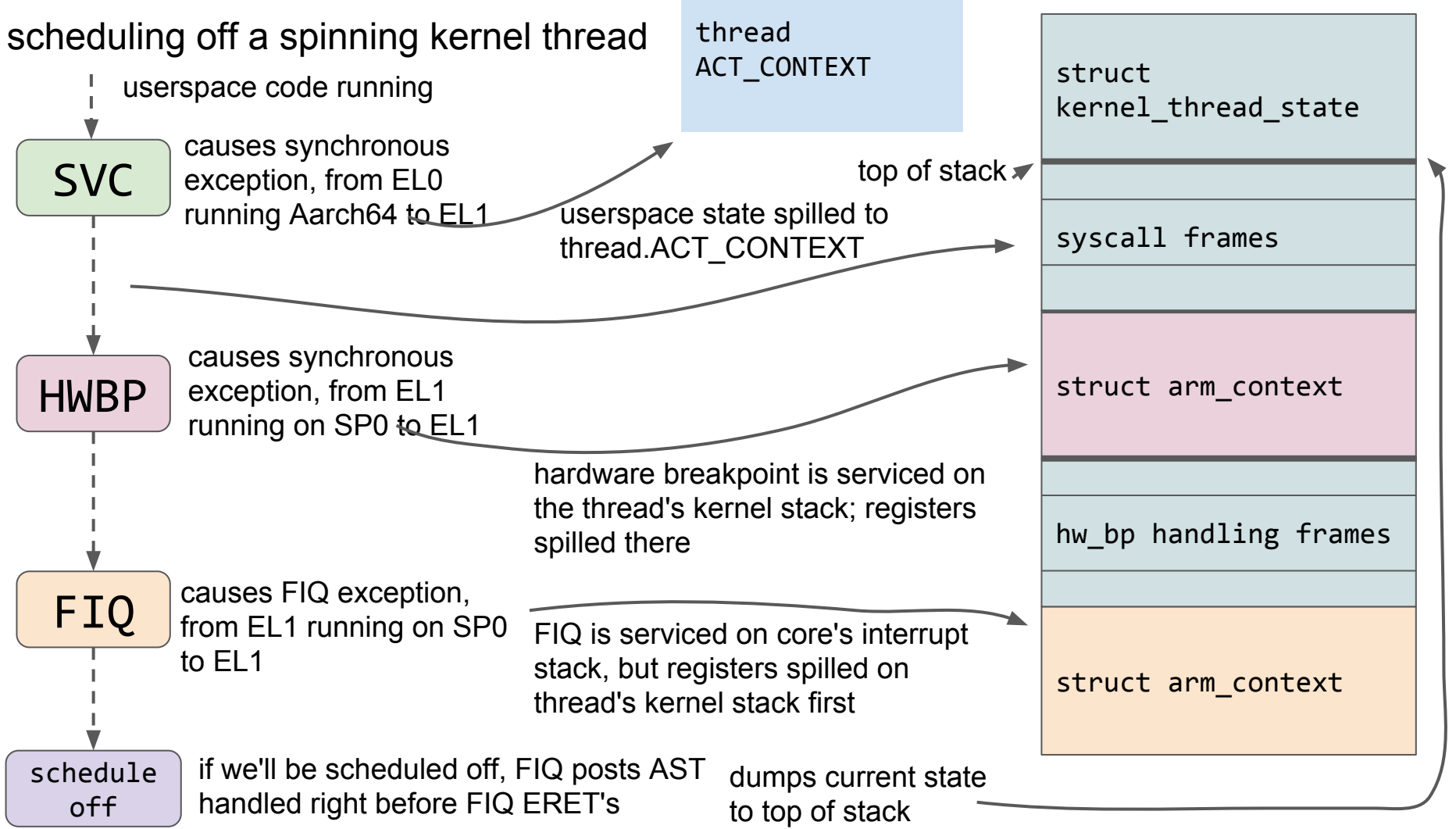
Synchronous EL1\_SP0 to EL1

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



# 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: <ul style="list-style-type: none"><li>• The Local (kernel) Debug Enable bit, <code>MDSCR_EL1.KDE</code>, is 0.</li><li>• The Debug exception mask bit, <code>PSTATE.D</code>, is 1.</li></ul> Otherwise enabled. This means that a debugger must explicitly enable these debug exceptions from $EL_D$ by setting <code>MDSCR_EL1.KDE</code> to 1 and <code>PSTATE.D</code> 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`.

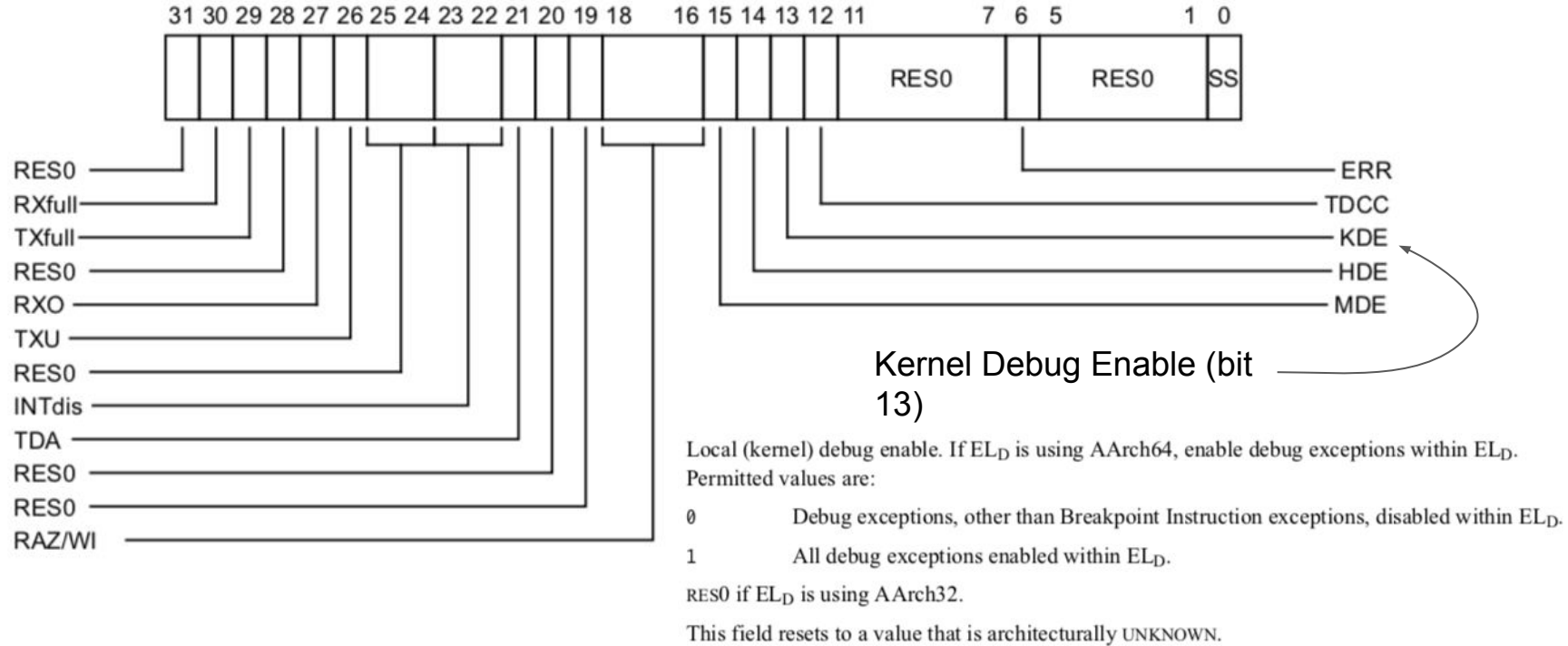
### Software Step exceptions

`MDSCR_EL1.SS`.

# MDSCR\_EL1

global enable bits; per core

## Monitor Debug System Control Register



# Hardware breakpoints

best thought of as 16 pairs of registers

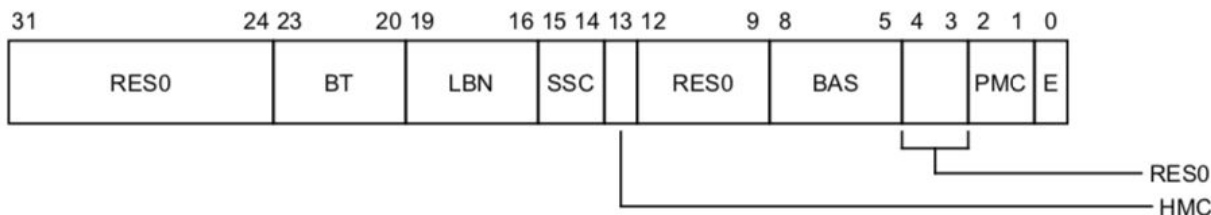
Debug Breakpoint Value Register

Debug Breakpoint Control Register

DBGBVR<1..15>\_EL1

DBGBCR<1..15>\_EL1

addresses where we want hardware  
breakpoints to fire



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

```
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);  
}
```

find the thread's DebugData

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	<p>Disabled if either of the following is true:</p> <ul style="list-style-type: none"><li>• The Local (kernel) debug Enable bit, <code>MDSCR_EL1.KDE</code>, is 0.</li><li>• The exception mask bit, <code>PSTATE.D</code>, is 1.</li></ul> <p>Otherwise enabled.</p> <p>This means that a debugger must explicitly enable these debug exceptions from <math>EL_D</math> by setting <code>MDSCR_EL1.KDE</code> to 1 and <code>PSTATE.D</code> to 0.</p>
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?

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

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


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"

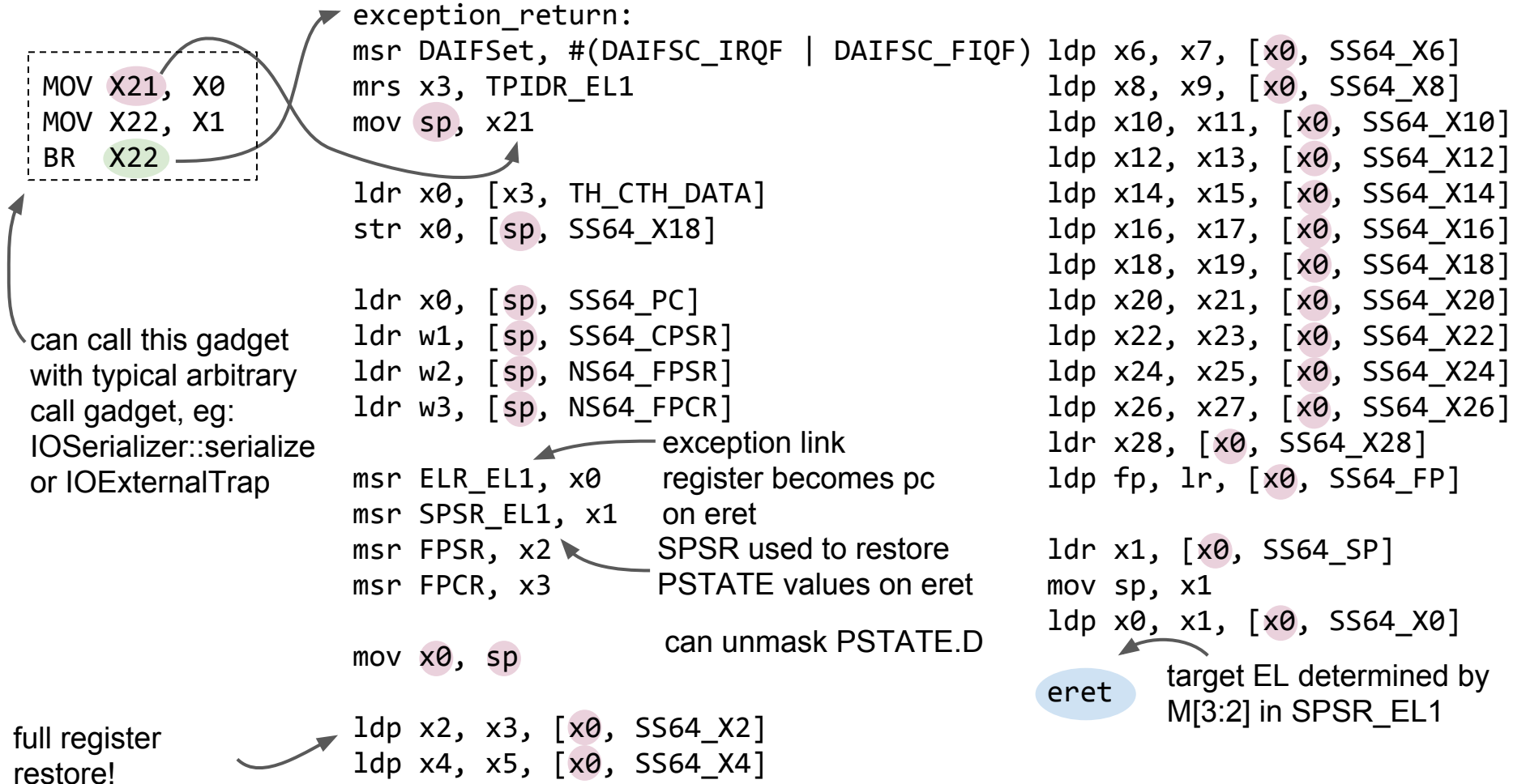


also has its own advantages

very useful for "what exactly does this  
syscall call"



# a better kernel arbitrary call gadget



# what to target?

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
```

saw this earlier

First level synchronous exception handler

gets called by fleh\_dispatch64

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**

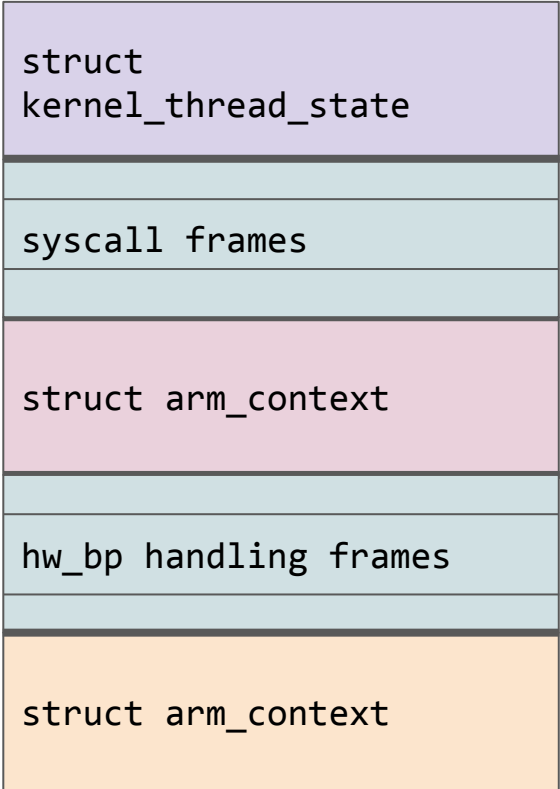
causes FIQ exception, from EL1 running on SP0 to EL1

**schedule off**

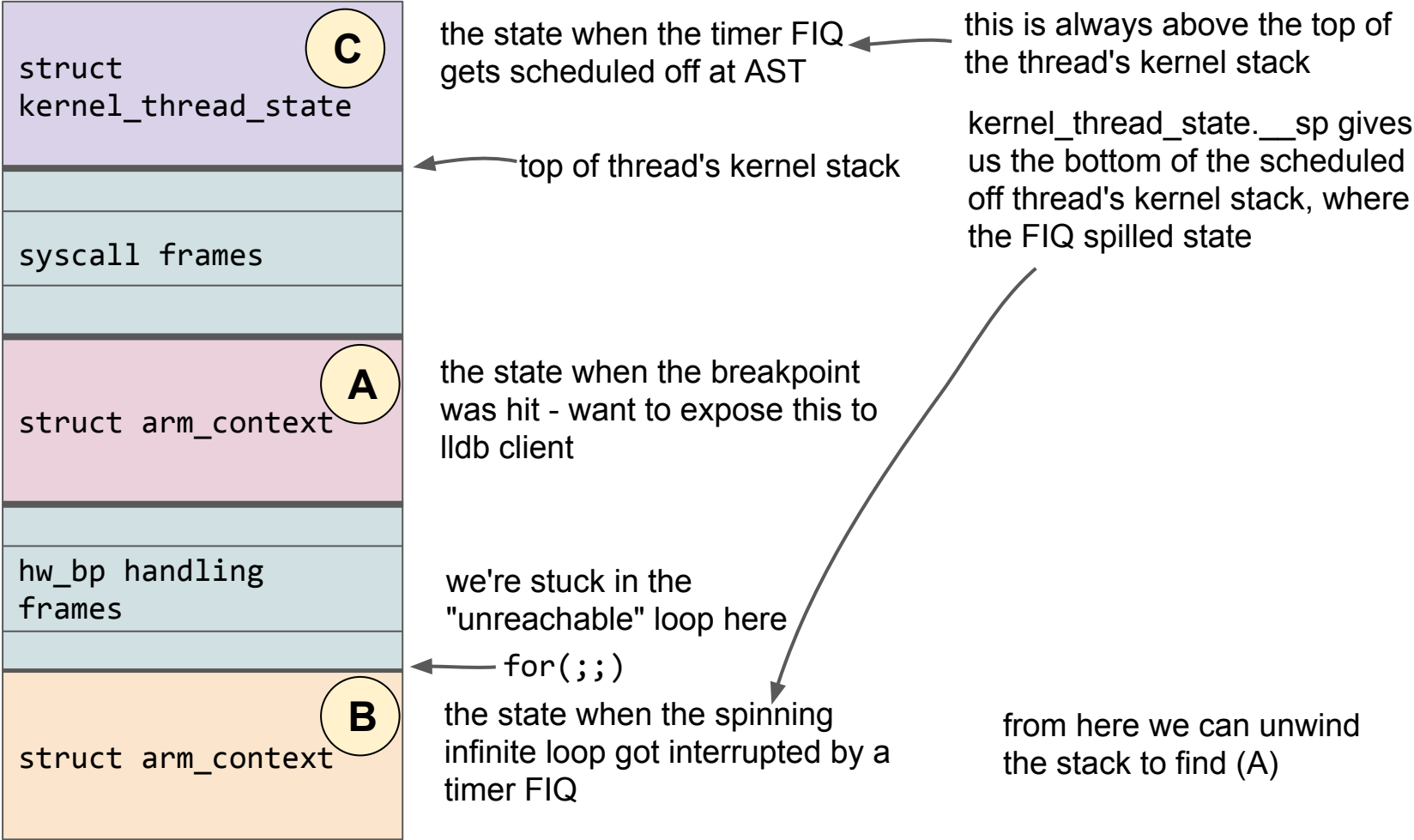
if we'll be scheduled off, FIQ posts AST handled right before FIQ ERET's

we're stuck in the "unreachable" loop here

for(;;) →



# modifying blocked state



unblocking the loop

struct  
kernel\_thread\_state

C

syscall frames

struct arm\_context

A

hw\_bp handling  
frames

struct arm\_context

B

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 loop 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(loop\_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(loop\_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(loop\_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

struct  
kernel\_thread\_state

syscall frames

struct arm\_context

hw\_bp handling  
frames

struct arm\_context

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_RESUMECPU	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:

total length of packet,  
including this header

31

16 15

8 7 6

0

total length

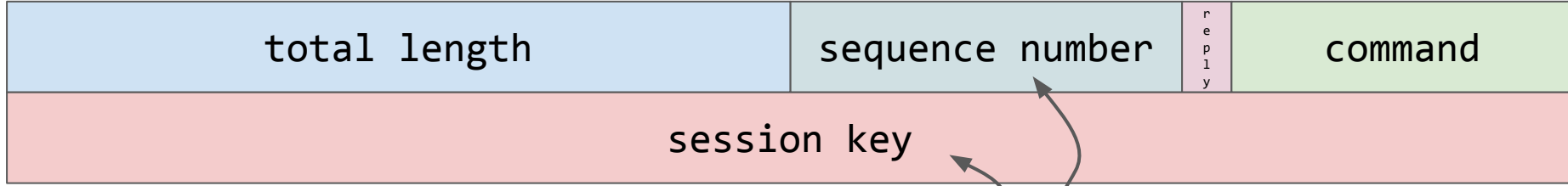
sequence number

r  
e  
p  
l  
y

command

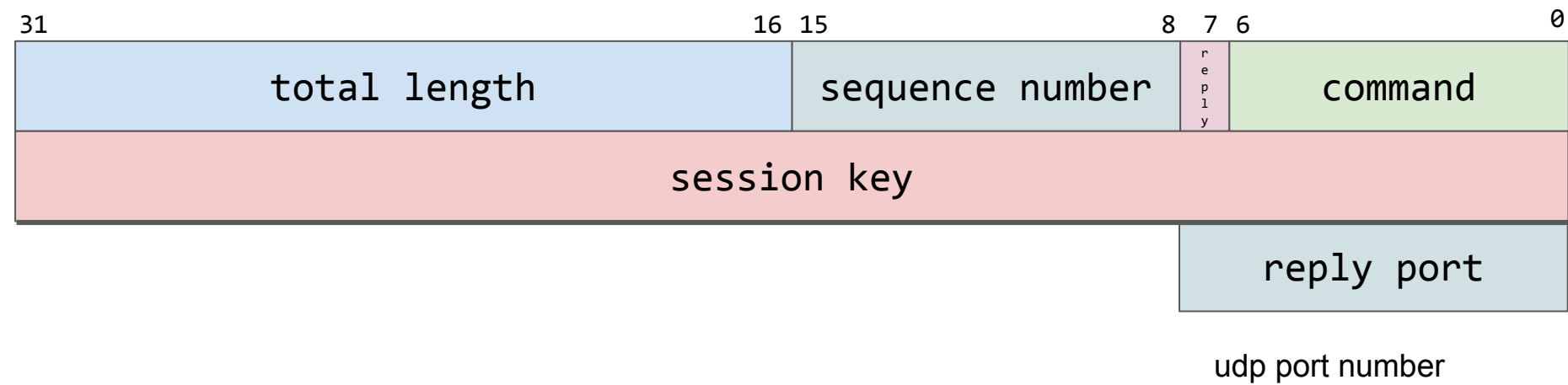
session key

set by the client, just echo back



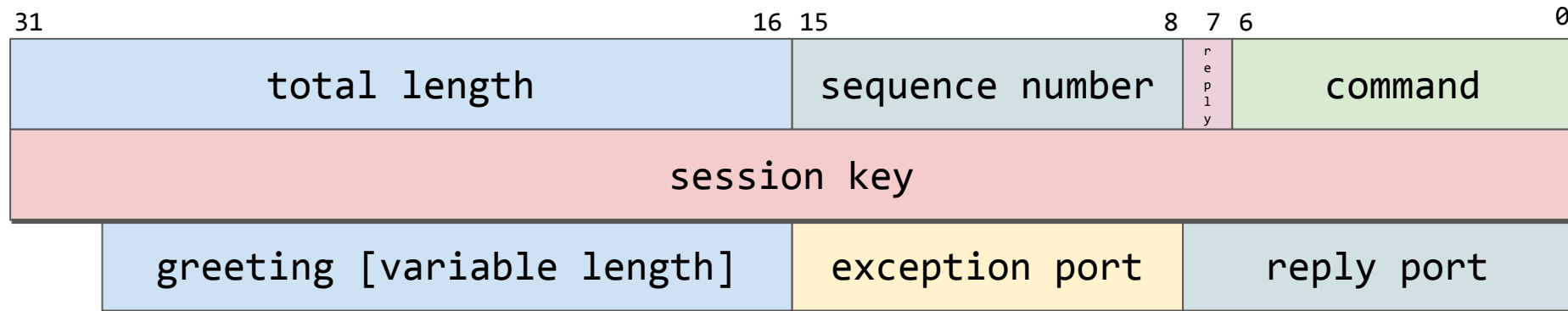
# KDP\_REATTACH

Header:



# KDP\_CONNECT

Header:

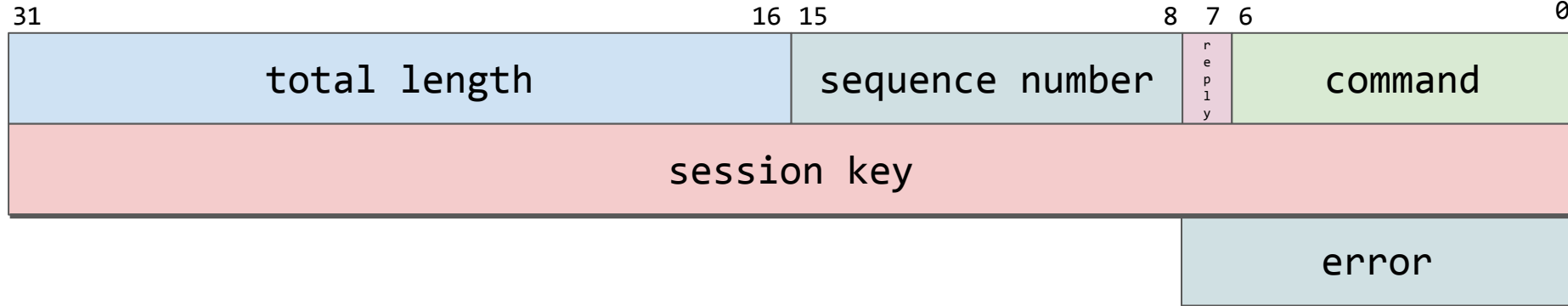


udp port for exception messages  
(we tell the client an event has happened)

lldb actually sends this greeting:

"Greetings from LLDB..."

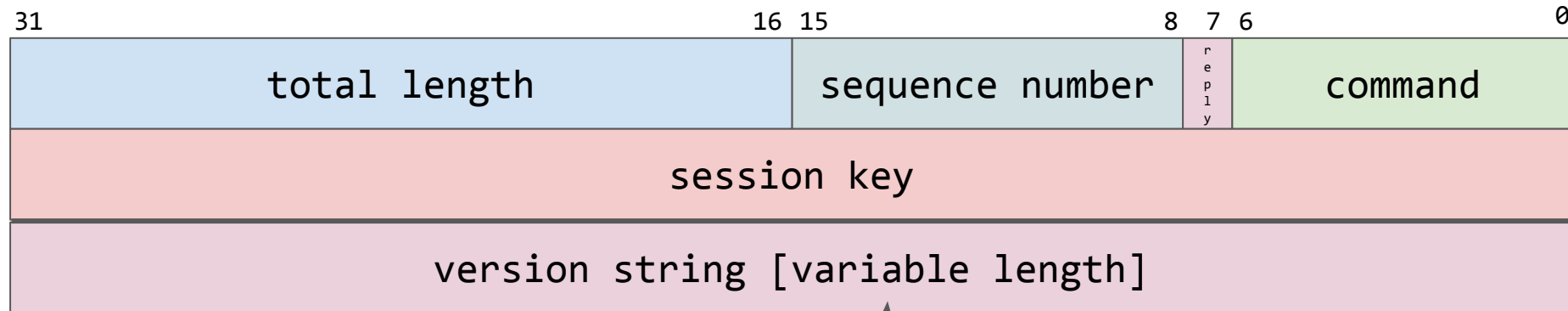
# KDP\_CONNECT\_REPLY



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=0xffffffff00ec04000
```

kernel base address (the 0xfeedfacf)

kernel\_uuid\_string

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=0xfffffffff021804000
Kernel UUID: 5E450F40-E224-33F7-946B-A764D21DF3FC
Load Address: 0xfffffffff021804000
Kernel slid 0x1a800000 in memory.
Loaded kernel file
/Users/ianbeer/prog/ios/iPhone7_firmwares/11.1.2/kernelcache.ip7_11_1_2.uncomp
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)
...
```

kernel version string we built

lldb client computes this from stext

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

. done.

Process 1 stopped

\* thread #1, stop reason = signal SIGSTOP

frame #0: 0xffffffff0218cc474

kernelcache.ip7\_11\_1\_2.uncomp`\_\_\_\_lldb\_unnamed\_symbol149\$\$kernelcache.ip7\_11\_1\_2.uncomp

kernelcache.ip7\_11\_1\_2.uncomp`\_\_\_\_lldb\_unnamed\_symbol149\$\$kernelcache.ip7\_11\_1\_2.uncomp:

-> 0xffffffff0218cc474 <+0>: msr DAIFFSet, #0x3

0xffffffff0218cc478 <+4>: mrs x3, TPIDR\_EL1

0xffffffff0218cc47c <+8>: mov sp, x21

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

Target 0: (kernelcache.ip7\_11\_1\_2.uncomp) stopped.  
(lldb)

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

(lldb) image list

[ 0] 5E450F40-E224-33F7-946B-A764D21DF3FC 0xffffffff021804000

/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:
```

```
0xffffffff021901248 <+0>:  sub    sp, sp, #0x20          ; =0x20
0xffffffff02190124c <+4>:  stp    x29, x30, [sp, #0x10]
0xffffffff021901250 <+8>:  add    x29, sp, #0x10          ; =0x10
0xffffffff021901254 <+12>: str    x0, [sp, #0x8]
0xffffffff021901258 <+16>: adrp   x2, 1211
0xffffffff02190125c <+20>: add    x2, x2, #0x400         ; =0x400
0xffffffff021901260 <+24>: orr    w1, wzr, #0x1
0xffffffff021901264 <+28>: add    x0, sp, #0x8          ; =0x8
0xffffffff021901268 <+32>: bl     0xffffffff021900fbc    ;
___lldb_unnamed_symbol1428$$kernelcache.ip7_11_1_2.uncomp
0xffffffff02190126c <+36>: ldp    x29, x30, [sp, #0x10]
0xffffffff021901270 <+40>: add    sp, sp, #0x20         ; =0x20
0xffffffff021901274 <+44>: ret
```

looking at the source that's  
kalloc\_canblock, a more interesting  
place to put a breakpoint

```
(lldb) break set --address 0xffffffff021900fbc
```

```
(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 = 0xffffffffe027b138e8
- x1 = 0x0000000000000001
- x2 = 0xfffffffff021dbdda8     atm\_manager + 1648
- x3 = 0x0000000000000000
- x4 = 0xffffffffe027b139f8
- x5 = 0x0000000010000003
- x6 = 0xffffffffe004cda9a0
- x7 = 0xffffffffe027b139c8
- x8 = 0x00000000000007fe8
- x9 = 0x00000001029e0000
- x10 = 0x0000000218000000
- x11 = 0x0000000001000000
- x12 = 0x0000000001000000
- x13 = 0x0000000001000000
- x14 = 0xffffffffe027b139a8
- x15 = 0xffffffffe001e3c1b0
- x16 = 0xffffffffe001e3c1b0
- x17 = 0xffffffffe001e3c1b0
- x18 = 0x0000000000000000
- x19 = 0xffffffffe004cda9a0
- x20 = 0x0000000218000000

```
x21 = 0x0000000000000001
x22 = 0xffffffffe027b139f8
x23 = 0x0000000102a4d5ba
x24 = 0x0000000000000027
x25 = 0xffffffffe004cda9e0
x26 = 0x0000000102a4d593
x27 = 0x0000000000000000
x28 = 0x000000000000003f
fp = 0xffffffffe027b13940
lr = 0xfffffffff021984b00
```

```
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol1211$$kernelcache.ip7_11_1_2.uncomp +
688
```

```
sp = 0xffffffffe027b13820
pc = 0xfffffffff021900fbc
```

```
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol1428$$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_symbol1428$$kernelcache.ip7_11_1_2.uncomp
    frame #1: 0xffffffff021984b00
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol1211$$kernelcache.ip7_11_1_2.uncomp
+ 688
    frame #2: 0xffffffff0218e2e80
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol197$$kernelcache.ip7_11_1_2.uncomp
+ 2704
    frame #3: 0xffffffff0218f2458
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol1307$$kernelcache.ip7_11_1_2.uncomp
+ 972
    frame #4: 0xffffffff0219deff8
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol1697$$kernelcache.ip7_11_1_2.uncomp
+ 4388
    frame #5: 0xffffffff0218cc1e0
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol134$$kernelcache.ip7_11_1_2.uncomp +
40
```

```
(lldb) x/1xg $x0
0xffffffffe027b138e8: 0x0000000000000003f
```

we set the breakpoint at kalloc\_canblock, the first argument is a pointer to the size to allocate

```
(lldb) finish
Process 1 stopped
```

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

```
* thread #1, stop reason = step out
```

```
frame #0: 0xfffffffff021984b00
```

```
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol1211$$kernelcache.ip7_11_1_2.uncomp + 688
```

```
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol1211$$kernelcache.ip7_11_1_2.uncomp:
```

```
-> 0xfffffffff021984b00 <+688>: mov    x20, x0
    0xfffffffff021984b04 <+692>: cbz    x20, 0xfffffffff021984b40 ; <+752>
    0xfffffffff021984b08 <+696>: orr    w8, wzr, #0x3
```

```
Target 0: (kernelcache.ip7_11_1_2.uncomp) stopped.
```

```
(lldb) reg r x0
```

```
x0 = 0xffffffffe004f60140
```

looks plausible for a kalloc.64

(lldb) finish

Process 1 stopped

\* thread #1, stop reason = step out

frame #0: 0xffffffff0218e2e80

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:

-> 0xffffffff0218e2e80 <+2704>: cbnz w0, 0xffffffff0218e37ac ; <+5052>

0xffffffff0218e2e84 <+2708>: ldur x8, [x29, #-0x98]

0xffffffff0218e2e88 <+2712>: str x8, [x27]

Target 0: (kernelcache.ip7\_11\_1\_2.uncomp) stopped.

(lldb) x/10xg 0xffffffffe004f60140

0xffffffffe004f60140: 0xdeadbeef00000003 0x0000000000000000

0xffffffffe004f60150: 0x0000000000000000 0x544f4e3e4c4d583c

0xffffffffe004f60160: 0x5f594c4c4145525f 0x4c4d582f3c4c4d58

0xffffffffe004f60170: 0x3f3332213e3c3f3e 0xde00333231234021

0xffffffffe004f60180: 0xffffffffe004e4cae0 0xffffffffe0018e8540

(lldb) x/s 0xffffffffe004f60158

0xffffffffe004f60158: "<XML>NOT\_REALLY\_XML</XML>?<>!23?!@#123"

```
0xffffffffe004f60140: 0xdeadbeef00000003 0x0000000000000000
0xffffffffe004f60150: 0x00000000000000027 0x544f4e3e4c4d583c
0xffffffffe004f60160: 0x5f594c4c4145525f 0x4c4d582f3c4c4d58
0xffffffffe004f60170: 0x3f3332213e3c3f3e 0xde00333231234021
0xffffffffe004f60180: 0xffffffffe004e4cae0 0xffffffffe0018e8540
(lldb) x/s 0xffffffffe004f60158
0xffffffffe004f60158: "<XML>NOT_REALLY_XML</XML>?<>!23?!@#123"
```

what is this structure?

```
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

used to be a  
common target for heap disclosure

(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\_symbol1197\$\$kernelcache.ip7\_11\_1\_2.uncomp + 2708

kernelcache.ip7\_11\_1\_2.uncomp`\_\_\_lldb\_unnamed\_symbol1197\$\$kernelcache.ip7\_11\_1\_2.uncomp:

-> 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\_symbol1197\$\$kernelcache.ip7\_11\_1\_2.uncomp + 2712

kernelcache.ip7\_11\_1\_2.uncomp`\_\_\_lldb\_unnamed\_symbol1197\$\$kernelcache.ip7\_11\_1\_2.uncomp:

-> 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\_symbol1197\$\$kernelcache.ip7\_11\_1\_2.uncomp + 2716

kernelcache.ip7\_11\_1\_2.uncomp`\_\_\_lldb\_unnamed\_symbol1197\$\$kernelcache.ip7\_11\_1\_2.uncomp:

-> 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_symbol1428$$kernelcache.ip7_11_1_2.uncom
p
kernelcache.ip7_11_1_2.uncomp`___lldb_unnamed_symbol1428$$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