KTRW
The journey to build a debuggable iPhone
Brandon Azad
Probe address: KongSWD-
Probe type: kong
Probe firmware: 0.52
Probe tckrate: 19500000

Listening on port 8000 for CPU0, CPU1
Listening on port 8002 for IOP
Listening on port 8003 for AE2
Listening on port 8004 for ISP
Detected HSP C0

KongSWD-- CPU0:Run CPU1:PowerOff IOP:Reset AE2:Reset ISP:PowerOff

NO CPU > cpu cpu0

KongSWD-- CPU0:Run CPU1:PowerOff IOP:Reset AE2:Reset ISP:PowerOff

CPU0 > halt
CPU0: ASTRIS_ERR_OK
r0: 0x00000001 r1: 0xbfff080 r2: 0x00000004 r3: 0x00000000
r4: 0x00000001 r5: 0xbfff080 r6: 0xbfff084 r7: 0xbff60700
r8: 0xbfff080 r9: 0x80000000 r10: 0x00000001 r11: 0xbff60700
ip: 0x3f200020 sp: 0xbff606f4 lr: 0xbff6e709 pc: 0xbff6e768
cpsr: 0x80000073 N--------FT Supervisor
0xbff6e768 0xbff28 it cs

KongSWD-- CPU0:Halt CPU1:PowerOff IOP:Reset AE2:Reset ISP:PowerOff

CPU0 > save ~/Desktop/iBoot 0xbff00000 0x50000

....................
327680 bytes received in 9.527 sec, 34395 bytes per second

KongSWD-- CPU0:Halt CPU1:PowerOff IOP:Reset AE2:Reset ISP:PowerOff

CPU0 >

https://pbs.twimg.com/media/DyP1Q_qX0AA838S?format=png&name=large
I do not use dev-fused devices
I do not use dev-fused devices

But they sure would make my security research easier...
Goal: Build my own "home-brewed dev phone"

- Patch kernel memory (__TEXT_EXEC)
- Breakpoints, watchpoints
- Use with LLDB / IDA Pro
- Can update iOS version
- Only parts you can get at an Apple store
EPIC JAILBREAK: Introducing checkm8 (read "checkmate"), a permanent unpatchable bootrom exploit for hundreds of millions of iOS devices.

Most generations of iPhones and iPads are vulnerable: from iPhone 4S (A5 chip) to iPhone 8 and iPhone X (A11 chip).

axi0mX/ipwndfu
open-source jailbreaking tool for many iOS devices - axi0mX/ipwndfu
github.com

4:15 AM - 27 Sep 2019
7,477 Retweets 16,436 Likes

1/ The last iOS device with a public bootrom exploit until today was iPhone 4, which was released in 2010. This is possibly the biggest news in iOS jailbreak community in years. I am releasing my exploit for free for the benefit of iOS community. I am also releasing a laboratory exploit to test iOS 13.1.2. If you want to contribute to the jailbreak community, please help by reporting bugs and seeking dependencies. I am already working on an open-source jailbreak tool for iPhone 8, iPhone X, and iPhone 11. Please stay tuned for more updates.

2/ I have been working on a jailbreak tool for iPhone 8, iPhone X, and iPhone 11. The tool is still in development, but I am hoping to release it soon. I am currently working on fixing some bugs and improving the tool's performance. If you have any questions or comments, please let me know.

3/ I am excited to announce that I have successfully jailbroken my iPhone 8 with my new tool. The tool is still in development, but I am hoping to release it soon. I am currently working on fixing some bugs and improving the tool's performance. If you have any questions or comments, please let me know.

4/ I am excited to announce that I have successfully jailbroken my iPhone 8 with my new tool. The tool is still in development, but I am hoping to release it soon. I am currently working on fixing some bugs and improving the tool's performance. If you have any questions or comments, please let me know.
KTRR
Goal: Build my own "home-brewed dev phone"

- Patch kernel memory (__TEXT_EXEC)
- Breakpoints, watchpoints
- Use with LLDB / IDA Pro
- Can update iOS version
- Only parts you can get at an Apple store
low addresses

kernelcache

high addresses

iPhone10,1 16G77 kernelcache

Read/write data
low addresses  

kernelcache

high addresses

---

Strings  Vtables  Page tables  __pinst

---

iPhone10,1 16G77 kernelcache
low addresses

Protected by KTRR

high addresses

rw-  r--  r-x  r--  r-x  rw-

iPhone10,1 16G77 kernelcache
KTRR (Kernel Text Readonly Region)

Strong form of $W^X$ (write-xor-execute) protection

Apple A10 and later

All writes to memory in the KTRR region fail

All instruction fetches from outside the KTRR region fail
DRAM
AMCC
Memory controller
CPU cores
MMUs
L2 cache
DRAM
AMCC
AMCC

Memory controller

L2 cache

DRAM

CPU core

MMUs

Memory controller

AMCC
WRITE
0x839a6c000
WRITE 0x839a6c000

CPU core

DRAM

L2 cache

AMCC

Memory controller

MMUs
WRITE 0x839a6c000 OK
WRITE
0x803280000
WRITE 0x803280000
AMCC
Memory controller
L2 cache
DRAM
CPU core
0x80320000
WRITE
MMUs
AMCC
Memory controller
CPU core

0x839a6c000

MMUs

DRAM

EXEC

AMCC

Memory controller
MMU registers lose state during core sleep
FFFF007E4000  LowResetVectorBase

FFFF007E4000  MSR  #0, c1, c0, #4
FFFF007E4004  MSR  #6, #0xF

...
LowResetVectorBase

MSR #0, c1, c0, #4
MSR #6, #0xF

ADRL X17, _rorgn_begin
LDR X17, [X17]
CBZ X17, Lskip_ktrr

ADRL X19, _rorgn_end
LDR X19, [X19]
CBZ X19, Lskip_ktrr

MSR ARM64_REG_KTRR_LOWER_EL1, X17
SUB X19, X19, #4,LSL#12
MSR ARM64_REG_KTRR_UPPER_EL1, X19
MOV X17, #1
MSR ARM64_REG_KTRR_LOCK_EL1, X17

Lskip_ktrr ; CODE XREF: LowResetVectorBase+8C↑j
LowResetVectorBase

MSR #0, c1, c0, #4
MSR #6, #0xF

ADRL X17, _rorgn_begin
LDR X17, [X17]
CBZ X17, Lskip_ktrr
ADRL X19, _rorgn_end
LDR X19, [X19]
CBZ X19, Lskip_ktrr
MSR ARM64_REG_KTRR_LOWER_EL1, X17
SUB X19, X19, #4,LSL#12
MSR ARM64_REG_KTRR_UPPER_EL1, X19
MOV X17, #1
MSR ARM64_REG_KTRR_LOCK_EL1, X17

Lskip_ktrr ; CODE XREF: LowResetVectorBase+8C↑j

MMU KTRR registers
Breaking KTRR
iOS 10.1.1: The Yalu KTRR bypass

Luca Todesco found that Apple accidentally left an MSR TTBR1_EL1 instruction executable

Use that instruction to set a new page table base

KTRR is still initialized (so no new kernel code), but readonly pages could now be made read/write
low addresses

Protected by KTRR

high addresses

rw-  r--  r-x  r--  r-x  rw-

iPhone10,1 16G77 kernelcache
low addresses

Protected by KTRR

high addresses

rw-  rw-  r-x  rw-  r-x  rw-

iPhone10,1 16G77 kernelcache
iOS 11.1.2: Build your own iOS kernel debugger

Ian Beer found that self-hosted debugging could be enabled using ROP

Built an iOS kernel debugger with breakpoints for iOS 11.1.2 that works with LLDB

KTRR is still fully enabled, but existing instructions can be executed in arbitrary order
Attempts to bypass KTRR

iBoot bug: no

TLB "corruption": no

L2 cache "corruption": no
"kernel" : "Darwin Kernel Version 18.2.0: Mon Nov 12 20:32:02 PST 2018; root:xnu-4903.232.2~1\RELEASE_ARM64_T8015",
"incident" : "47933C7-BAA2-4D77-8ABD-FD6C3B36B40C",
"crashReporterKey" : "ce81d5ec16fcde03b706d6a939240745a65782a2",
"date" : "2019-08-30 14:37:52.07 -0700",
"panicString" : "Attempting to forcibly halt cpu 1\ncpu 1 failed to halt with error -5: halt not supported for this configuration
Debugger synchronization timed out; waited 100000000 nanoseconds
panic(cpu 0 caller 0xffffffff00b5c96bc): "$WDT timeout: CPU 1 failed to respond"
@/BuildRoot/Library/Caches/com.apple.xbs/Sources/AppleARMPMPlatform/AppleARMPMPlatform-700.230.2/AppleARMPMWatchDogTimer.cpp: 501
Debugger message: panic
Memory ID: 0xff
OS version: 16C101
Kernel version: Darwin Kernel Version 18.2.0: Mon Nov 12 20:32:02 PST 2018; root:xnu-4903.232.2~1/
RELEASE_ARM64_T8015
KernelCache UUID: EE1D833E1AF3B367B764DA9719881315
Kernel UUID: 94463A80-7B38-3176-8872-0B8E344C7138
iBoot version: iBoot-4513.230.10
secure boot?: YES
Paniclog version: 11
Kernel slide: 0x0000000000000000
Kernel identifying: 0xffffffff-00000000-00000000-00000000-F
panic-base-2019-08-30-143759.i...

"kernel" : "Darwin Kernel Version 18.2.0: Mon Nov 12 20:32:02 PST 2018; root:xnu-4903.232.2~1/RELEASE_ARM64_T8015",
"incident" : "479333C7-BAA2-4D77-8ABD-FD6C3B36B40C",
"crashReporterKey" : "ce81d5ec16fcde03b706d6a939240745a65782a2",
"date" : "2019-08-30 14:37:52.07 -0700",
"panicString" : "Attempting to forcibly halt cpu 1\ncpu 1 failed to halt with error -5: halt not supported for this configuration\nDebugger synchronization timed out; waited 10000000 nanoseconds\npanic(cpu 0
caller 0xffffffff00b5c96bc): "\nBuildRoot/Library/Caches/com.apple.xbs/Sources/AppleARMP/AppleARMP/AppleARMP Platform-700.230.2/AppleARMPWatchDogTimer.cpp:501\nDebugger message: panic\nMemory ID: 0xff\nOS version: 16C101\nKernel version: Darwin Kernel Version 18.2.0: Mon Nov 12
20:32:02 PST 2018; root:xnu-4903.232.2~1/
RELEASE_ARM64_T8015\nKernelCache UUID: EE1D833E1AF3B367B764DA9719881315\nKernel UUID: 94463A80-7B38-3176-8872-0B8E344C7138\niBoot version: iBoot-4513.230.10\nsecure boot?: YES\nPaniclog version: 11\nKernel slide: 0-0000000000000000\nHardware UUID: 0xffffffff9c-00-1000"
panic-base-2019-08-30-143759

"kernel" : "Darwin Kernel Version 18.2.0: Mon Nov 12 20:32:02 PST 2018; root:xnu-4903.232.2~1\RELEASE_ARM64_T8015",
"incident" : "47933C7-BAA2-4D77-8ABD-FD6C3B36B40C",
"crashReporterKey" : "ce81d5ec16fcde03b706d6a939240745a65782a2",
"date" : "2019-08-30 14:37:52.07 -0700",
"panicString" : "Attempting to forcibly halt cpu 1\ncpu 1 failed to halt with error -5: halt not supported for this configuration\nDebugger synchronization timed out; waited 10000000 nanoseconds\npanic(cpu 0 caller 0xffffffff00b5c96bc): "WDT timeout: CPU 1 failed to respond)"@\nBuildRoot/Library/Caches/com.apple.xbs/Sources/AppleARMPlatform/AppleARMPlatform-700.230.2/AppleARMWatchDogTimer.cpp: 501\nDebugger message: panic
nMemory ID: 0xff
nOS version: 16C101
nKernel version: Darwin Kernel Version 18.2.0: Mon Nov 12 20:32:02 PST 2018; root:xnu-4903.232.2~1\nRELEASE_ARM64_T8015\nKernelCache UUID: EE1D833E1AF3B367B764DA9719881315\nKernel UID: 94463A80-7B38-3176-8872-0B8E344C7138\niBoot version: iBoot-4513.230.10\nsecure boot?: YES\nPaniclog version: 11\nKernel slide: 0:0000000000000000\nKernel address: 0xffffffff00000000\n"
Last login: Mon Dec 16 15:05:24 on ttys025
bazad@bazad-macbookpro ~ % cd ~/Developer/source/xnu
bazad@bazad-macbookpro ~/Developer/source/xnu (git)-[master] % grep -RF "Attempting to forcibly halt" .
./osfmk/arm/model_dep.c: paniclog_append_noflush("Attempting to forcibly halt cpu %d\n", cpu);
bazad@bazad-macbookpro ~/Developer/source/xnu (git)-[master] %
ml_dbgwrap_halt_cpu(int cpu_index, uint64_t timeout_ns)
{
    cpu_data_t *cdp = cpu_datap(cpu_index);
    if ((cdp == NULL) || (cdp->coresight_base[CORESIGHT_UTT] == 0))
        return DBGWRAP_ERR_UNSUPPORTED;

    /* Only one cpu is allowed to initiate the halt sequence, to prevent cpu's from cross-halting
     * each other. The first cpu to request a halt may then halt any and all other cpu's besides itself. */
    int curcpu = cpu_number();
    if (cpu_index == curcpu)
        return DBGWRAP_ERR_SELF_HALT;
(halt_from_cpu != (uint32_t)curcpu))
  return DBGWRAP_ERR_INPROGRESS;

volatile dbgwrap_reg_t *dbgWrapReg = (volatile dbgwrap_reg_t *)
(cdp->coresight_base[CORESIGHT_UTT] + DBGWRAP_REG_OFFSET);

if (ml_dbgwrap_cpu_is_halted(cpu_index))
  return DBGWRAP_WARN_ALREADY_HALTED;

/* Clear all other writable bits besides dbgHalt; none of the power-down or reset
bits must be set. */
*dbgWrapReg = DBGWRAP_DBGHALT;

if (timeout_ns != 0) {

(halt_from_cpu ≠ (uint32_t)currcpu))
return DBGWRAP_ERR_INPROGRESS;

volatile dbgwrap_reg_t *dbgWrapReg = (volatile dbgwrap_reg_t *)
(sizeof[CORESIGHT_UTT] + DBGWRAP_REG_OFFSET);

if (ml_dbgwrap_cpu_is_halted(cpu_index))
return DBGWRAP_WARN_ALREADY_HALT;

/* Clear all other writable bits besides dbgHalt; none of the power-down or reset
bits must be set. */
*dbgWrapReg = DBGWRAP_DBGHALT;

if (timeout_ns ≠ 0) {
CoreSight?
```c
dbgwrap_status_t
ml_dbgwrap_halt_cpu_with_state(int cpu_index,
    uint64_t timeout_ns, dbgwrap_thread_state_t *state) {
    cpu_data_t *cdp = cpu_datap(cpu_index);
    ...
    /* Ensure memory-mapped coresight registers can be written */
    (*((volatile uint32_t *)(cdp->coresight_base[CORESIGHT_ED]
        + ARM_DEBUG_OFFSET_DBGLAR)) = ARM_DBG_LOCK_ACCESS_KEY;
    ...
    for (unsigned int i = 0; i < ...; ++i) {
        instr = (0xD51U << 20) | ... | i; // msr DBGDTR0, x<i>
        ml_dbgwrap_stuff_instr(cdp, instr, ...);
        state->x[i] = ml_dbgwrap_read_dtr(cdp, ...);
    }
    ...
    return status;
```
dbgwrap_status_t
ml_dbgwrap_halt_cpu_with_state(int cpu_index,
    uint64_t timeout_ns, dbgwrap_thread_state_t *state) {
    cpu_data_t *cdp = cpu_datap(cpu_index);

    /* Ensure memory-mapped coresight registers can be written */
    *((volatile uint32_t *)(cdp->coresight_base[CORESIGHT_ED] + ARM_DEBUG_OFFSET_DBGLAR)) = ARM_DBG_LOCK_ACCESS_KEY;

    for (unsigned int i = 0; i < ...; ++i) {
        instr = (0xD51U << 20) | ... | i; // msr DBGDTR0, x<i>
        ml_dbgwrap_stuff_instr(cdp, instr, ...);
        state->x[i] = ml_dbgwrap_read_dtr(cdp, ...);
    }

    return status;
dbgwrap_status_t
ml_dbgwrap_halt_cpu_with_state(int cpu_index,
                                  uint64_t timeout_ns, dbgwrap_thread_state_t *state) {
    cpu_data_t *cdp = cpu_datap(cpu_index);

    /* Ensure memory-mapped coresight registers can be written */
    *((volatile uint32_t *)(cdp->coresight_base[CORESIGHT_ED]
                                + ARM_DEBUG_OFFSET_DBGLAR)) = ARM_DBG_LOCK_ACCESS_KEY;

    for (unsigned int i = 0; i < ...; ++i) {
        instr = (0xD51U << 20) | ... | i; // msr DBGDTR0, x<i>
        ml_dbgwrap_stuff_instr(cdp, instr, ...);
        state->x[i] = ml_dbgwrap_read_dtr(cdp, ...);
    }

    return status;
}
ml_dbgwrap_status_t ml_dbgwrap_halt_cpu_with_state(int cpu_index, uint64_t timeout_ns, dbgwrap_thread_state_t* state) {
    cpu_data_t* cdp = cpu_datap(cpu_index);
    ...
    /* Ensure memory-mapped coresight registers can be written */
    *(volatile uint32_t*)((cdp->coresight_base[CORESIGHT_ED] + ARM_DEBUG_OFFSET_DBGLAR)) = ARM_DBG_LOCK_ACCESS_KEY;
    ...
    for (unsigned int i = 0; i < ...; ++i) {
        instr = (0xD51U << 20) | ... | i; // msr DBGDTR0, x<i>
        ml_dbgwrap_stuff_instr(cdp, instr, ...);
        state->x[i] = ml_dbgwrap_read_dtr(cdp, ...);
    }
    ...
    return status;
}
// Set EDLR to unlock the CoreSight registers.

uint32_t edlsr = kernel_call_7(1ldr_w0_x0_ret, 1, (ed_mmi0 + 0xFB4));
INFO(" edlsr = %x", edlsr);

kernel_call_7(str_w0_x1_ret, 2, 0xc5acce55, (ed_mmi0 + 0xFB0));
edlsr = kernel_call_7(1ldr_w0_x0_ret, 1, (ed_mmi0 + 0xFB4));
INFO(" edlsr = %x", edlsr);

}

// Try to call ml_dbgwrap_halt_cpu_with_state.

uint64_t dbgwrap_thread_state = kernel_vm_allocate(0x4000);

uint32_t halt_status = kernel_call_7(ml_dbgwrap_halt_cpu_with_state, 3, 2, 10000000, 
dbgwrap_thread_state);
INFO("halt_status = %x", halt_status);

arm_thread_state64_t state = {};

kernel_read(dbgwrap_thread_state, &state, sizeof(state));

for (int i = 0; i < sizeof(state.__x) / sizeof(state.__x[0]); i++) {
    printf(" x[%d] = %016llx\n", i, state.__x[i]);
}

printf(" fp = %016llx\n", state.__fp);
printf(" lr = %016llx\n", state.__lr);
printf(" sp = %016llx\n", state.__sp);
printf(" pc = %016llx\n", state.__pc);
printf(" cpsr = %08x\n", state.__cpsr);
sleep(1);
// Set EDLR to unlock the CoreSight registers.
uint32_t edlrs = kernel_call(1, (ed_mmio + 0xFB4));

rstack(0x400);

trap_halt_cpu_with_state, 3,
1, (ed_mmio + 0xFB0));

locate(0x400);
Kernel mode
Reset vector!
We've halted execution in the reset vector, before the MMU has been turned on!
We've halted execution in the reset vector, before the MMU has been turned on!

How do we use this?
We've halted execution in the reset vector, before the MMU has been turned on! How do we use this? What exactly is this CoreSight thing anyway?
H8.4 Memory-mapped accesses to the external debug interface

Support for memory-mapped access to the external debug interface is **optional**. When memory-mapped access to the external debug interface is supported, the external debug interface is accessed as a little-endian memory-mapped peripheral.

If the external debug interface is **CoreSight** compliant, then an **optional** Software Lock can be implemented for memory-mapped accesses to each component.

The Software Lock is **optional** and deprecated. If ARMv8.4-Debug is implemented, the Software Lock is not implemented. If it is not implemented, the behavior is as if it is unlocked. The Software Locks are controlled by EDLSR and EDLAR, PMLSR and PMLAR, and CTILSR and CTILAR. See *Management registers and CoreSight compliance* on page K2-7237.

With the exception of these registers and the effect of the Software Lock, the behavior of the memory-mapped accesses is the same as for other accesses to the external debug interface.

--- **Note** ---

The recommended memory-mapped accesses to the external debug interface are not compatible with the memory-mapped interface defined in ARMv7. In particular:

- The memory map is different.
- Memory-mapped accesses do not behave differently to Debug Access Port accesses when OSLR.OSLK == 1, meaning that the OS Lock is locked.

The following sections give more information about these memory-mapped accesses:
Nailgun Attack

Break the privilege isolation in ARM devices

Overview

Processors nowadays are consistently equipped with debugging features to facilitate the program debugging and analysis. Specifically, the ARM debugging architecture involves a series of CoreSight components and debug registers to aid the system debugging, but the security of the debugging features is under-examined since it normally requires physical access to use these features in the traditional debugging model.

The idea of Nailgun Attack is to misuse the debugging architecture with the inter-processor debugging model. In the inter-processor debugging model, a processor...
CoreSight External Debug Interface

On-chip debug architecture

Per-CPU debug registers accessible via MMIO

Extensively documented in the ARMv8 manual

Can set breakpoints/watchpoints, execute instructions, etc

The memory-mapped version of the debug registers used by Ian Beer in "Build your own iOS kernel debugger"
Use External Debug to single-step LowResetVectorBase and skip KTRR lockdown
LowResetVectorBase

MSR  #0, c1, c0, #4
MSR  #6, #0xF

ADRL  X17, _rorgn_begin
LDR  X17, [X17]
CBZ  X17, Lskip_ktrr
ADRL  X19, _rorgn_end
LDR  X19, [X19]
CBZ  X19, Lskip_ktrr
MSR  ARM64_REG_KTRR_LOWER_EL1, X17
SUB  X19, X19, #4,LSL#12
MSR  ARM64_REG_KTRR_UPPER_EL1, X19
MOV  X17, #1
MSR  ARM64_REG_KTRR_LOCK_EL1, X17
Lskip_ktrr ; CODE XREF: LowResetVectorBase+8C↑j
Challenges

We can halt a CPU core, and we can execute instructions on it to modify state, but how do we resume execution after making our modifications?

We are using one CPU core to hijack another as it executes the reset vector, so how do we handle subsequent core resets?
344 } while ((dbgwrap & (1 << 28)) == 0);

345 // The CPU is now halted in Debug state at reset.
346 DEBUG_TRACE(1, "Halted CPU %u in Debug State", cpu_id);
347 DEBUG_TRACE(1, "DBGWRAP = %llx", (dbgwrap = kernel_ioread64(dbgwrap_reg)));
349 print_edprsr();

350 // TODO: If this operation actually prevents the CPU from resetting, we should observe no
351 // panic later.

352 // Restart.
353 DEBUG_TRACE(1, "Restarting CPU %u", cpu_id);
354 kernel_iowrite64(dbgwrap_reg, (1 << 38) | (1 << 29) | (1 << 26));
356 DEBUG_TRACE(1, "DBGWRAP = %llx", kernel_ioread64(dbgwrap_reg));
358 print_edprsr();
359 kernel_iowrite64(dbgwrap_reg, (1 << 29) | (1 << 26));
360 DEBUG_TRACE(1, "DBGWRAP = %llx", kernel_ioread64(dbgwrap_reg));
362 sample_cpus();

364 for (int i = 0; i < 20; i++) {
365     print_edprsr();
366     sleep(1);
367 }
0x70E4000 LowResetVectorBase

0x70E4004 MSR #0, c1, c0, #4

0x70E4004 MSR #6, #0xF

...  
0x70E4080 ADRL X17, _rorgn_begin
0x70E4088 LDR X17, [X17]
0x70E408C CBZ X17, Lskip_ktrr

0x70E4090 ADRL X19, _rorgn_end
0x70E4098 LDR X19, [X19]
0x70E409C CBZ X19, Lskip_ktrr

0x70E40A0 MSR ARM64_REG_KTRR_LOWER_EL1, X17
0x70E40A4 SUB X19, X19, #4,LSL#12

0x70E40A8 MSR ARM64_REG_KTRR_UPPER_EL1, X19
0x70E40AC MOV X17, #1

0x70E40B0 MSR ARM64_REG_KTRR_LOCK_EL1, X17
0x70E40B4 Lskip_ktrr ; CODE XREF: LowResetVectorBase+8C↑j
LowResetVectorBase

MSR #0, c1, c0, #4
MSR #6, #0xF

... 

ADRL X17, _rorgn_begin
LDR X17, [X17]
CBZ X17, Lskip_ktrr
ADRL X19, _rorgn_end
LDR X19, [X19]
CBZ X19, Lskip_ktrr
MSR ARM64_REG_KTRR_LOWER_EL1, X17
SUB X19, X19, #4,LSL#12
MSR ARM64_REG_KTRR_UPPER_EL1, X19
MOV X17, #1
MSR ARM64_REG_KTRR_LOCK_EL1, X17

Lskip_ktrr ; CODE XREF: LowResetVectorBase+8C↑j

Take this branch
LowResetVectorBase

MSR #0, c1, c0, #4
MSR #6, #0xF

ADRL X17, _rorgn_begin
LDR X17, [X17]
CBZ X17, Lskip_ktrr
ADRL X19, _rorgn_end
LDR X19, [X19]
CBZ X19, Lskip_ktrr

MSR ARM64_REG_KTRR_LOWER_EL1, X17
SUB X19, X19, #4, LSL#12
MSR ARM64_REG_KTRR_UPPER_EL1, X19
MOV X17, #1
MSR ARM64_REG_KTRR_LOCK_EL1, X17

Lskip_ktrr ; CODE XREF: LowResetVectorBase+8C↑j
Kernel shellcode mode activated!
Building KTRW
Steps to building a kernel debugger

- Remapping the kernel
- Loading a kernel extension
- Interrupt handling
- Communication channel
- GDB stub
CPU cores

L2 cache

MMUs

DRAM

AMCC
RoRgn pages are still readonly
Remapping the kernel

We need to modify page table permissions to make the kext's memory executable.

Root page tables are in the KTRR region (still readonly).

Solution: Remap the kernel onto fresh, writable pages and set TTBR1_EL1 to point to the new page tables.
CPU core

TTBR1_EL1

L2 cache

DRAM

AMCC

MMUs
Loading kernel extensions

Allocate kernel memory for the kext

Copy in the kext binary

Dynamically link against kernel symbols

Modify page tables to make it executable

Call the kext_start() function
KTRW: High-level design

One (monitor) core reserved for KTRW

Other (debugged) cores run XNU normally

Breakpoints/watchpoints cause the debugged core to halt and enter Debug state

Monitor core polls for entry to Debug state and notifies LLDB over some communication channel
Need some channel for LLDB to communicate with the KTRW kext
# Communication channels

<table>
<thead>
<tr>
<th>Serial</th>
<th>USB</th>
<th>WiFi</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Communication channels

<table>
<thead>
<tr>
<th>Serial</th>
<th>USB</th>
<th>WiFi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easy</td>
<td>Fast</td>
<td>An option</td>
</tr>
</tbody>
</table>
## Communication channels

<table>
<thead>
<tr>
<th></th>
<th>Serial</th>
<th>USB</th>
<th>WiFi</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Easy</td>
<td>Fast</td>
<td>An option</td>
</tr>
<tr>
<td>Requires</td>
<td>special hardware</td>
<td>Write a custom</td>
<td>Write a custom</td>
</tr>
<tr>
<td></td>
<td>hardware</td>
<td>USB stack</td>
<td>WiFi driver</td>
</tr>
</tbody>
</table>
Communication channels

<table>
<thead>
<tr>
<th>Serial</th>
<th>USB</th>
<th>WiFi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easy</td>
<td>Fast</td>
<td>An option</td>
</tr>
<tr>
<td>Requires <em>special</em> hardware</td>
<td>Write a custom USB stack</td>
<td>Write a custom WiFi driver</td>
</tr>
</tbody>
</table>
DesignWare Hi-Speed USB 2.0 On-the-Go Controller

The DesignWare® Hi-Speed USB 2.0 On-The-Go (HS OTG) Controller provides designers with high-quality USB IP for the most demanding USB 2.0 peripherals. The controller performs as a standard Hi-Speed Dual-Role Device (DRD), operating as either a USB 2.0 Hi-Speed peripheral, or Hi-Speed USB 2.0 Host. Based on Synopsys' success in building and deploying Hi-Speed USB 2.0 Host, Device and PHY designs, the DesignWare USB 2.0 HS OTG Controller incorporates Synopsys expertise in Reuse Methodology, Constrained Random Verification, and USB PHY interoperability to deliver flexible, quality IP in Verilog source. The controller is optimized for area- and power-sensitive markets such as Internet of Things (IoT).

DesignWare IP Prototyping Kit for USB 2.0 HS OTG
DesignWare IP Prototyping Kits
typedef union dct1_data {
    /* raw register data */
    uint32_t d32;
    /* register bits */
    struct {
        /* Remote Wakeup */
        unsigned rmtwupsig:1;
        /* Soft Disconnect */
        unsigned sftdiscon:1;
        /* Global Non-Periodic IN NAK Status */
        unsigned gnppinnaks:1;
        /* Global OUT NAK Status */
        unsigned goutnaks:1;
    } r;
} struct { /* Remote Wakeup */
out_daint = (unsigned int)daint >> 16;
in_daint = (unsigned __int16)daint;

for ( diepint_reg_off = 0x908LL; ; diepint_reg_off += 0x20LL )
{
    if ( !(in_daint | out_daint) || ep_num > 5 )
        goto DONE;
    if ( in_daint & 1 )
    {
        diepint = *(DWORD *)(diepint_reg_off | 0x2301000000LL); // DIEPINT
        syn_otg.endpoints[ep_idx_in].depint = diepint;
        *(DWORD *)(diepint_reg_off | 0x2301000000LL) = diepint;
        if ( syn_otg.endpoints[ep_idx_in].is_active )
        {
            if ( diepint & 1 ) // xfercompl -- when is this signalled? When all packets have been sent
            { //
            // Well I'm dealing with control transfers, so it's on each transaction
            
            endpoint_off = (int *)__shifted(syn_otg_endpoint,-0x10)&syn_otg.pending_setup_pkt + ep_idx_in * 80;
            cur_ioreq = syn_otg.endpoints[ep_idx_in].current_io_req;
            if ( cur_ioreq )
            {
                cur_ioreq->transferred_size += ADJ(endpoint_off)[1].current_transfer_size;
                *(DWORD *)&ADJ(endpoint_off)[1].current_transfer_size = 0LL;
                cur_ioreq = ADJ(endpoint_off)[1].current_io_req;
                if ( cur_ioreq->transferred_size >= cur_ioreq->size )
                {
                    cur_ioreq->state = 0;
                    cur_ioreq = ADJ(endpoint_off)[1].current_io_req;
                    endpoint_off = (int *)__shifted(syn_otg_endpoint,-0x10)&syn_otg.pending_setup_pkt + ep_idx_in * 80;
                }
            }
        }
    }

    DONE:
}
### KTRW:

<table>
<thead>
<tr>
<th>Product ID</th>
<th>0x1337</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vendor ID</td>
<td>0x05ac (Apple Inc.)</td>
</tr>
<tr>
<td>Version</td>
<td>0.00</td>
</tr>
<tr>
<td>Serial Number</td>
<td>Google Project Zero</td>
</tr>
<tr>
<td>Speed</td>
<td>Up to 480 Mb/s</td>
</tr>
<tr>
<td>Manufacturer</td>
<td>Brandon Azad</td>
</tr>
<tr>
<td>Location ID</td>
<td>0x14200000 / 3</td>
</tr>
<tr>
<td>Current Available (mA)</td>
<td>500</td>
</tr>
<tr>
<td>Current Required (mA)</td>
<td>500</td>
</tr>
<tr>
<td>Extra Operating Current (mA)</td>
<td>0</td>
</tr>
</tbody>
</table>
static sends_a_packet

```
gdb_pkt__m(struct packet *pkt) {
    uint64_t address;
    uint64_t length;
    bool ok = pkt_read_hex_u64(pkt, &address)
        && pkt_read_match(pkt, ",," )
        && pkt_read_hex_u64(pkt, &length)
        && pkt_empty(pkt);  
    if (!ok) {
        return send_error_bad_packet("m");
    }
    int8_t data[GDB_RSP_MAX_PACKET_SIZE / 2];
    if (length > sizeof(data)) {
        return send_error_invalid_length("m");
    }
    size_t read = gdb_stub_read_memory(gdb.current_cpu, ad
    if (read == 0 && length > 0) {
```

oob_timestamp
DEMO