

# Strong ARMing with MacOS: Adventures in Cross-Platform Emulation

[blogs.blackberry.com/en/2021/05/strong-arming-with-macos-adventures-in-cross-platform-emulation](https://blogs.blackberry.com/en/2021/05/strong-arming-with-macos-adventures-in-cross-platform-emulation)

The BlackBerry Research & Intelligence Team



## Summary:

In a world where adversaries are becoming more sophisticated by the day, it is important that threat hunters can keep a competitive advantage and remain one step ahead of threat actors. Recent developments in Apple® hardware have made it even more difficult for security researchers to keep up, and the demand for ARM-targeted testing environments is increasing.

BlackBerry recognizes the importance of supporting the cybersecurity community in the fight against cyberthreats, and is therefore following up its release of the [PE Tree Tool](#) in 2020 by sharing this methodology report to inform security researchers and pen-testers on how to successfully emulate a MacOS ARM64 kernel under QEMU.

Pen-testers and researchers can use the virtualized environment of a stripped-down MacOS kernel for debugging and vulnerability discovery, and this illustrates the extent to which one can use emulation to manipulate and control the kernel to their desired ends,

whether it be to find a critical bug or to patch an area of the kernel.

More importantly, this project was a successful experiment in cross-platform emulation that has the potential for future development.

## Introduction

---

Demand for ARM-targeted testing environments is increasing. The first Apple silicon processors are appearing in the market in conjunction with the growing extent of ARM64 support on the most popular operating systems. This project was inspired by a series of recent developments in emulation software and Apple hardware as well as a race to be the first to coalesce them. iOS® kernel emulation on a MacOS host had already been attempted, accomplished, and published. Cross-platform virtualization like this is nothing new: ARM-based systems have been virtualizable on Intel-based host systems as early as 2009.

QEMU, the versatile and dynamic emulator responsible for bringing this practice into practicality, is popular among developers and pen-testers for cross-platform emulation. Even the Android™ emulator is based on QEMU. It was only a matter of time before XNU, Apple's own Unix-derived kernel, joined the party.

## Background

---

When emulating a kernel image, the first phase of the kernel boot stage is typically referred to as the 'bootstrap' phase. This is normally when the earliest kernel output appears and is the first visible output during an emulation session of the MacOS® ARM64e kernel. The MacOS 11.1 ARM64e kernel bootstrap process is shown below:

```
Darwin Kernel Version 20.2.0: Wed Dec  2 20:40:22 PST 2020; root:xnu-7195.60.75~1/RELEASE_ARM64_T8020
pmap_startup() init/release time: 106420 microsec
pmap_startup() delayed init/release of 0 pages
vm_page_bootstrap: 383584 free pages, 115105 wired pages, (up to 0 of which are delayed free)
"vm_compressor_mode" is 4
oslog_init completed, 16 chunks, 8 io pages
standard timeslicing quantum is 10000 us
standard background quantum is 2500 us
WQ[wql_init]: init linktable with max:262144 elements (8388608 bytes)
WQ[wqp_init]: init prepost table with max:262144 elements (8388608 bytes)
mig_table_max_displ = 53 mach_kobj_count = 365
debug_log_init: Error!! gPanicBase is still not initialized
debug_log_init: Error!! gPanicBase is still not initialized
kdp_core zlib memory 0x8000
Serial requested, consistent debug disabled or debug boot arg not present,
configuring debugging over serial
iBoot version:
```

Fifty seconds, 5086 lines, and 113 kexts later:

```
bash-3.2# ls
.fseventsd      dev            mnt2           mnt5           mnt8           sbin
System          etc            mnt3           mnt6           mnt9           usr
bin             mnt1          mnt4           mnt7           private        var
bash-3.2# ps -ef
  UID   PID  PPID  C  STIME  TTY          TIME CMD
    0     1     0  0 12:02AM  ??          0:10.31 /sbin/launchd
    0     3     1  0 12:02AM  ??          0:08.71 /bin/bash
    0     5     3  0 12:05AM  ??          0:03.41 ps -ef
```

All of this is virtualized in a QEMU session, on a Linux® host, running an Intel® Core™ i5-7500 CPU @ 3.40GHz. You can see the full output on our GitHub page:

<https://github.com/cylance/macOS-arm64-emulation/blob/main/macOS-qemu.log>

## Getting the Files

---

In June 2020, Apple announced the first beta releases of macOS 11 (Big Sur) along with universal binary support for both x86-64 and ARM64. Does that mean we can expect to find both the x86-64 and ARM64 kernels in this release?

Yes!

The [OSX-KVM](#) project provides a script to download the Big Sur installer package. From there, it was simply a matter of extracting one nested archive after the other to find the kernel image. This script does not have a good track record when it comes to reading Apple's software update catalogs. Therefore, we've provided a link to the kernelcache, ramdisk, and device tree files below:

[https://mega.nz/file/GZwzGYKb#HscZIOg\\_K5JdUlvlLwwwW7\\_Ntc1z9c7QPOcEQRKwp8c](https://mega.nz/file/GZwzGYKb#HscZIOg_K5JdUlvlLwwwW7_Ntc1z9c7QPOcEQRKwp8c)

Note that the next few steps are only necessary if these files are extracted from the installer package referenced below, instead of from the link above. Skip ahead to the Modifying QEMU section, or continue below if you are extracting the files from the installer package:

```
# Download Big Sur installer and extract the HFS file system
$ wget http://swcdn.apple.com/content/downloads/00/55/001-86606-A_9SF1TL01U7/5duug9lar1gypwunjfl96dza0upa854qgg/InstallAssistant.pkg
$ xar -xf InstallAssistant.pkg SharedSupport.dmg
$ 7z e SharedSupport.dmg 5.hfs
```

An archive inside the SFR software update directory with a hash-style name contains the files we need. We must also extract the Mac® software update archive that contains the APFS file system. This file system contains many ARM64e binaries that are not present on the ramdisk, including bash and ls:

```
# Find and extract the SFR and Mac software update archives
$ 7z l -ba 5.hfs | grep ".zip"
2020-12-08 01:49:10 .....          1982210          1982464  Shared Support/UpdateBrain.zip
2020-12-08 01:52:41 .....          927135894         927137792  Shared
Support/SFR/com_apple_MobileAsset_SFRSoftwareUpdate/aabc1798a59cc185ea5a87bfd4dec012f

2020-12-08 01:52:27 .....          11256421743        2666487808  Shared
Support/com_apple_MobileAsset_MacSoftwareUpdate/6c799f422b6d995ccc7f3fb669fe3246fd9fc

$ 7z e -so 5.hfs "Shared
Support/SFR/com_apple_MobileAsset_SFRSoftwareUpdate/aabc1798a59cc185ea5a87bfd4dec012f
> sfr.zip
$ 7z e -so 5.hfs "Shared
Support/com_apple_MobileAsset_MacSoftwareUpdate/6c799f422b6d995ccc7f3fb669fe3246fd9fc
> mac.zip
```

It's important to note that the long, hash-style archive file names will vary from version to version. The ramdisk, device tree and kernel files can be easily extracted the SFR archive:

```
# Extract the ramdisk, device tree, and kernel from the SFR archive
$ 7z e sfr.zip AssetData/usr/standalone/update/ramdisk/arm64eSURamDisk.dmg
$ 7z e sfr.zip AssetData/boot/Firmware/all_flash/DeviceTree.j273aap.im4p
$ 7z e sfr.zip AssetData/boot/kernelcache.release.j273
```

The ramdisk file functions as the operating system. The device tree file identifies the devices for loading the relevant drivers. The kernel, begetter of all running processes, boots the system.

## Decoding and Decompressing

---

Now we've discovered the kernel image, ramdisk image, and device tree binary and can gather the requisite files into a single directory. Next, we move ahead to decode each of the three ASN1-encoded files with these scripts:

```

$ SCRIPTS=~/.source/xnu-qemu-arm64-tools/bootstrap_scripts
$ python $SCRIPTS/asn1kerneldecode.py kernelcache.release.j273
kernelcache.release.j273.asn1decoded
$ python $SCRIPTS/asn1rdskdecode.py arm64eSURamDisk.dmg
arm64eSURamDisk.dmg.asn1decoded
$ python $SCRIPTS/asn1dtredecode.py DeviceTree.j273aap.im4p
DeviceTree.j273aap.im4p.asn1decoded

```

The decoded device tree file and kernel image were LZFSE compressed, unlike the LZSS-compressed iOS kernel. LZFSE features a `-decode` option for such files:

```

$ lzfse -decode -i kernelcache.release.j273.asn1decoded -o
kernelcache.release.j273.out
$ lzfse -decode -i DeviceTree.j273aap.im4p.asn1decoded -o
DeviceTree.j273aap.im4p.out

```

## Getting Bash and Other Binaries

---

The root file system on the ramdisk was missing many common command line tools, including a shell client binary. Even the `ls` program was completely absent. This brings us to the `mac.zip` archive extracted earlier. Below are the contents of the `AssetData/Restore` directory in this archive:

```

$ 7z l -ba mac.zip | grep "AssetData/Restore"
2020-12-07 23:17:30 D....          0          0 AssetData/Restore
2020-12-07 23:17:30 .....      2871122      2841290
AssetData/Restore/AppleDiagnostics.dmg
2020-12-07 23:17:30 .....         328         328
AssetData/Restore/AppleDiagnostics.chunklist
2020-12-07 23:17:30 .....         2416         2325
AssetData/Restore/BaseSystem.chunklist
2020-12-07 23:16:50 .....     908228542     658466440 AssetData/Restore/022-10310-
098.dmg
2020-12-07 23:17:32 .....     610378184     605691452
AssetData/Restore/BaseSystem.dmg
2020-12-07 23:15:22 .....         3424         3261 AssetData/Restore/022-10310-
098.chunklist

```

`BaseSystem.dmg` is for `x86_64` installations only. `022-10310-098.dmg` is for ARM64e installations only. After extracting the ARM64e installer and examining the contents:

```

$ 7z e mac.zip AssetData/Restore/022-10310-098.dmg
$ 7z l -ba 022-10310-098.dmg
          .....          512          512 0 - MBR
          .....          512          512 1 - Primary GPT Header
          .....         16384         16384 2 - Primary GPT Table
          .....    926695424    908185600 3 - Apple_APFS
          .....         16384         16384 4 - Backup GPT Table
          .....          512          512 5 - Backup GPT Header
3 - Apple_APFS is the file system and contains the arm64e Mach-O binaries we need.
We extracted and mounted with apfs-fuse:
$ 7z e 022-10310-098.dmg "3 - Apple_APFS"
$ mkdir apfs
$ apfs-fuse -o allow_other "3 - Apple_APFS" apfs

```

Discovering a bash file within, we check the file type:

```

$ find apfs -type f -name bash
apfs/root/bin/bash
$ file apfs/root/bin/bash
apfs/root/bin/bash: Mach-O 64-bit arm64 executable, flags: <
NOUNDEFS|DYLDLINK|TWOLEVEL|PIE >

```

It turns out all of the Mach-O binaries in this directory were purely ARM64e executables, as well as those in the `/sbin`, `/usr/bin`, and `/usr/sbin` directories. To fit these binaries into the original ramdisk file, the ramdisk had to be resized. `Hdiutil` is the only tool for the job, but no port of `hdiutil` existed outside of MacOS. This means the ramdisk needed to be resized in a MacOS system:

```

# cp arm64eSURamDisk.dmg.asn1decoded arm64eSURamDisk.dmg.out
# hdiutil resize -size 1.5G -imagekey diskimage-class=CRawDiskImage
arm64eSURamDisk.dmg.out

```

This was the only time throughout the entire project that access to a MacOS system was required. Fortunately, this can be done in a MacOS virtual machine (VM) that can be created with `OSX-KVM`. We mounted the ramdisk, cleared out its `/System/Library/LaunchDaemons` directory and transferred the binaries into the ramdisk file system:

```
$ mkdir ramdisk
$ sudo mount -t hfsplus -o force,rw arm64eSURamDisk.dmg.out ramdisk
$ sudo rm -rf ramdisk/System/Library/LaunchDaemons/*
$ sudo cp apfs/root/bin/* ramdisk/bin/
$ sudo cp apfs/root/sbin/* ramdisk/sbin/
$ sudo cp apfs/root/usr/bin/* ramdisk/usr/bin/
$ sudo cp apfs/root/usr/sbin/* ramdisk/usr/sbin/
```

We then created a new file at  
ramdisk/System/Library/LaunchDaemons/com.apple.bash.plist:

```
$ sudo touch ramdisk/System/Library/LaunchDaemons/com.apple.bash.plist
```

Afterwards we copied the following code into it:

```
< ?xml version="1.0" encoding="UTF-8"? >
< !DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"https://www.apple.com/DTDs/PropertyList-1.0.dtd" >
< plist version="1.0" >
< dict >
  < key >Label< /key >
  < string >com.apple.bash< /string >
  < key >Umask< /key >
  < integer >0< /integer >
  < key >RunAtLoad< /key >
  < true/ >
  < key >ProgramArguments< /key >
  < array >
    < string >/bin/bash< /string >
  < /array >
  < key >StandardInPath< /key >
  < string >/dev/console< /string >
  < key >StandardOutPath< /key >
  < string >/dev/console< /string >
  < key >StandardErrorPath< /key >
  < string >/dev/console< /string >
  < key >POSIXSpawnType< /key >
  < string >Interactive< /string >
  < key >EnablePressuredExit< /key >
  < false/ >
  < key >UserName< /key >
  < string >root< /string >
< /dict >
< /plis t>
```

Finally, we unmounted the file system and ramdisk:

```
$ sudo umount apfs ramdisk
```

Then it was time to begin testing.

## Modifying QEMU

---

Since the kernelcache binary already contained all the necessary kexts, it was not necessary to create a kext collection. Thanks to the folks at [Aleph Research](#) for providing a modified version of QEMU that supports Apple's XNU kernel. With access to this source, we managed to add support for MacOS on top of the iOS support already implemented.

## Building QEMU

---

We skimmed through the source files in xnu-qemu-arm64 and found two files that specifically targeted the iOS kernel used by the iPhone® 6s Plus:

**include/hw/arm/n66\_iphone6plus.h** and **hw/arm/n66\_iphone6splus.c**. These files target a very specific iOS kernel: N66, build 16B92. The definitions and configurations in these files would likely be incompatible with the kernel we were using (J273, build 20C69), let alone any macOS kernel. To add additional support for the MacOS kernel, we:

- Copied these files;
- Renamed the variables, functions, and preprocessor directives to match the names of the MacOS kernel (J273), kernel version (20C69), and chipset (A21Z); and
- Updated the filenames in the QEMU command line:

```
cp hw/arm/n66_iphone6splus.c hw/arm/j273_macos11.c
cp include/hw/arm/n66_iphone6splus.h include/hw/arm/j273_macos11.h
sed -i 's/N66/J273/g' hw/arm/j273_macos11.c include/hw/arm/j273_macos11.h
sed -i 's/n66/j273/g' hw/arm/j273_macos11.c include/hw/arm/j273_macos11.h
sed -i 's/16B92/20C69/g' hw/arm/j273_macos11.c include/hw/arm/j273_macos11.h
sed -i 's/S8000/A12Z/g' hw/arm/j273_macos11.c include/hw/arm/j273_macos11.h
```

As in the original xnu-qemu-arm64, we included the generated object files in hw/arm/Makefile.objs:



```

sed -i "s/obj-y += boot.o/obj-y += boot.o \
xnu_fb_cfg.o \
xnu_trampoline_hook.o \
xnu_pagetable.o xnu_cpacr.o \
xnu_dtb.o \
xnu_file_mmio_dev.o \
xnu_mem.o \
xnu.o \
n66_iphone6splus.o \
j273_macos11.o \
guest-services.o \
guest-socket.o \
guest-fds.o \
guest-file.o/g" hw/arm/Makefile.objs

```

After this, we began the QEMU build. Unfortunately, and unsurprisingly, the compiler produced an error:

```

$ make -j6
... (truncated output)
scsi/qemu-pr-helper.c: In function 'multipath_pr_out':
scsi/qemu-pr-helper.c:523:32: error: array subscript is outside array bounds of
'struct transportid *[]' [-Werror=array-bounds]
    523 |             paramp.trnptid_list[paramp.num_transportid++] = id;
        |             ~~~~~^~~~~~
... (truncated output)

```

But what if the linker doesn't even need this qemu-pr-helper.c module or any other potentially unbuildable modules? Let's run make again with the -k flag and CFLAGS="-Wno-error":

```

$ make -j6 -k CFLAGS="-Wno-error"
... (snip)
$ ls ./aarch64-softmmu/qemu-system-aarch64
./aarch64-softmmu/qemu-system-aarch64

```

This appears to have succeeded. However, we are not out of the woods yet.

## Switch to QEMU 5.1.0

---

QEMU 5.1.0 supports the LDAPR instruction. QEMU 4.2.0 does not. The official ARM documentation states the following about this instruction:

*This instruction is supported in architectures ARMv8.3-A and later. It is optionally supported in ARMv8.2-A with the RCpc extension.*

Xnu-qemu-arm64 is based on 4.2.0. QEMU 4.2.0 has very limited support of ARMv8.3 and no support for the LDAPR instruction. The immediate task ahead was to move all the xnu-related source files over to a freshly downloaded source of QEMU 5.1.0. Below is a Git diff showing the files added to the official QEMU 5.1.0 source from xnu-qemu-arm64:

```
$ git diff --no-index --name-only --diff-filter=A qemu-5.1.0 xnu-qemu-arm64-5.1.0
xnu-qemu-arm64-5.1.0/hw/arm/guest-fds.c
xnu-qemu-arm64-5.1.0/hw/arm/guest-file.c
xnu-qemu-arm64-5.1.0/hw/arm/guest-services.c
xnu-qemu-arm64-5.1.0/hw/arm/guest-socket.c
xnu-qemu-arm64-5.1.0/hw/arm/j273_macos11.c
xnu-qemu-arm64-5.1.0/hw/arm/n66_iphone6splus.c
xnu-qemu-arm64-5.1.0/hw/arm/xnu.c
xnu-qemu-arm64-5.1.0/hw/arm/xnu_cpacr.c
xnu-qemu-arm64-5.1.0/hw/arm/xnu_dtb.c
xnu-qemu-arm64-5.1.0/hw/arm/xnu_fb_cfg.c
xnu-qemu-arm64-5.1.0/hw/arm/xnu_file_mmio_dev.c
xnu-qemu-arm64-5.1.0/hw/arm/xnu_mem.c
xnu-qemu-arm64-5.1.0/hw/arm/xnu_pagetable.c
xnu-qemu-arm64-5.1.0/hw/arm/xnu_trampoline_hook.c
xnu-qemu-arm64-5.1.0/hw/display/xnu_ramfb.c
xnu-qemu-arm64-5.1.0/include/hw/arm/guest-services/fds.h
xnu-qemu-arm64-5.1.0/include/hw/arm/guest-services/file.h
xnu-qemu-arm64-5.1.0/include/hw/arm/guest-services/general.h
xnu-qemu-arm64-5.1.0/include/hw/arm/guest-services/socket.h
xnu-qemu-arm64-5.1.0/include/hw/arm/j273_macos11.h
xnu-qemu-arm64-5.1.0/include/hw/arm/n66_iphone6splus.h
xnu-qemu-arm64-5.1.0/include/hw/arm/xnu.h
xnu-qemu-arm64-5.1.0/include/hw/arm/xnu_cpacr.h
xnu-qemu-arm64-5.1.0/include/hw/arm/xnu_dtb.h
xnu-qemu-arm64-5.1.0/include/hw/arm/xnu_fb_cfg.h
xnu-qemu-arm64-5.1.0/include/hw/arm/xnu_file_mmio_dev.h
xnu-qemu-arm64-5.1.0/include/hw/arm/xnu_mem.h
xnu-qemu-arm64-5.1.0/include/hw/arm/xnu_pagetable.h
xnu-qemu-arm64-5.1.0/include/hw/arm/xnu_trampoline_hook.h
xnu-qemu-arm64-5.1.0/include/hw/display/xnu_ramfb.h
```

In addition to the following modified files in the official source:

```
$ git diff --no-index --name-only --diff-filter=M qemu-5.1.0 xnu-qemu-arm64-5.1.0
xnu-qemu-arm64-5.1.0/hw/arm/Makefile.objs
xnu-qemu-arm64-5.1.0/target/arm/helper.c
```

One particular function proved problematic:

```
void allocate_ram(MemoryRegion *top, const char *name, hwaddr addr,
                 hwaddr size)
{
    MemoryRegion *sec = g_new(MemoryRegion, 1);
    memory_region_init_ram(sec, NULL, name, size, &error_fatal);
    memory_region_add_subregion(top, addr, sec);
}
```

Due to changes in the source from QEMU 4.2.0 to 5.1.0, **memory\_region\_allocate\_system\_memory** had to be changed to **memory\_region\_init\_ram**. It takes the same arguments in the same order plus one extra (&error\_fatal). The full Git diff file can be downloaded below:

<https://github.com/cylance/macos-arm64-emulation/blob/main/xnu-qemu-arm64-5.1.0.diff>

To apply the diff and build the modified QEMU source:

- Download the source for QEMU 5.1.0 to the same directory as the Git diff file
- Extract it
- Rename it to xnu-qemu-arm64-5.1.0
- Apply the Git diff:

```
$ wget https://download.qemu.org/qemu-5.1.0.tar.xz
$ tar xf qemu-5.1.0.tar.xz
$ mv qemu-5.1.0.tar.xz xnu-qemu-arm64-5.1.0
$ git apply xnu-qemu-arm64-5.1.0.diff
```

Configure the source and build it per the instructions provided by Aleph Research:

```
$ cd xnu-qemu-arm64-5.1.0
$ ./configure --target-list=aarch64-softmmu --disable-capstone --disable-pie --
disable-slirp
$ make -j6
```

No bypasses or build flags are necessary.

## QEMU Dry Run

---

The next thing to determine is: will it run? The run.sh script below has the updated QEMU command line, where we enabled remote kernel debugging with the `-S -s` option:

```
$ cat run.sh
~/source/xnu-qemu-arm64-5.1.0/aarch64-softmmu/qemu-system-aarch64 \
-M macos11-j273-a12z, \
kernel-filename=kernelcache.release.j273.out, \
dtb-filename=Firmware/all_flash/DeviceTree.j273aap.im4p.out, \
ramdisk-filename=arm64eSURamDisk.dmg.out, \
kern-cmd-args="kextlog=0xffff cpus=1 rd=md0 serial=2 -noprogess", \
xnu-ramfb=off \
-cpu max \
-m 6G \
-serial mon:stdio \
-nographic \
-S -s
$ ./run.sh
```

Attaching the remote debugger on the same host:

```
$ aarch64-linux-gnu-gdb -q -ex "target remote:1234"
Remote debugging using :1234
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
0x0000000047ac4580 in ?? ()
(gdb)
```

0x47ac4580 is the entry point to our MacOS 11 Big Sur ARM64e kernel image. On entry, addresses seen in QEMU will be physical addresses. Since this is an initial dry run, we let it loose:

```
$ ./run.sh
```

Or, in this case, we let it spin around endlessly on the same three instructions:

```

0x479f4388      mrs      x0, s3_4_c15_c0_4 ; APCTL_EL1
|
0x479f438c      and      x1, x0, #0x2
|
0x479f4390      cbz      x1, 0x479f4388 ; infinite loop

```

Bit 1 (#0x2) is never set in the system coprocessor register s3\_4\_c15\_c0\_4, so it never breaks the loop. This is an Apple-specific hardware register. Apple registers are unrecognized by the official QEMU branch, but xnu-qemu-arm64 added several Apple registers to boot the iOS kernel, including s3\_4\_c15\_c0\_4. This register is also known as APCTL\_EL1/MIGSTS.

## Patching the Kernel

---

That dry run barely got us on our feet. Not easily discouraged, we began skimming the QEMU source files for clues. After looking at the patching function we disabled, we could find nothing directly addressing the elusive “APCTL\_EL1” register. The infinite loop above does show up in the XNU source in xnu-6153.141.1/osfmk/arm64/start.s:

```

#ifdef HAS_APPLE_PAC
#ifdef __APSTS_SUPPORTED__
... (snip)
#else
mrs      x0, ARM64_REG_APCTL_EL1
and      x1, x0, #(APCTL_EL1_MKEYVld)
cbz      x1, 1b // Poll APCTL_EL1.MKEYVld
... (snip)

```

It is polling a flag by the name of MKEYVld.

## MKEYVld

---

What is the MKEYVld flag? Not many clues are in the XNU source. Perhaps a flag indicating some kind of validation status (MKEYVld = MAC key validated?). Most likely the kernel is waiting for it to be set by some other piece of hardware. We can force set this flag ourselves by adding to the following patch already provided by Aleph Research in our copied hw/arm/j273\_macos11.c:

```

static uint32_t g_set_cpacr_and_branch_inst[] = {
    // 91400c21      add x1, x1, 3, lsl 12    # x1 = x1 + 0x3000
    // d378dc21      lsl x1, x1, 8           # x1 = x1 * 0x100 (x1 = 0x300000)
    // d5181041      msr cpacr_el1, x1     # cpacr_el1 = x1 (enable FP)
    // d2800041      mov x1, #2             # MKEYVld
    // d51cf081      mov apctl_el1, x1
    // aa1f03e1      mov x1, xzr             # x1 = 0
    // 14000eb5      b 0x1fc0           # branch to regular start
    0x91400c21, 0xd378dc21, 0xd5181041,
    0xd2800041, 0xd51cf081, 0xaa1f03e1,
    0x14000eb5
};

```

We introduced two more instructions that set the MKEYVld flag (bit 1) in APCTL\_EL1:

```

// d2800041      mov x1, #2             # MKEYVld
// d51cf081      mov apctl_el1, x1

```

The presence of several fixed offsets in xnu-qemu-arm64/hw/arm/n66\_iphone6splus.c shows that there were 11 places in the iOS kernel that required patching:

```

#define INITIAL_BRANCH_VADDR_16B92 (0xffffffff0070a5098)
#define BZERO_COND_BRANCH_VADDR_16B92 (0xffffffff0070996d8)
#define SMC_INST_VADDR_16B92 (0xffffffff0070a7d3c)
#define SLIDE_SET_INST_VADDR_16B92 (0xffffffff00748ef30)
#define NOTIFY_KERNEL_TASK_PTR_16B92 (0xffffffff0070f4d90)
#define CORE_TRUST_CHECK_16B92 (0xffffffff0061e136c)
#define TFP0_TASK_FOR_PID_16B92 (0xffffffff0074a27bc)
#define TFP0_CNVRT_PORT_TO_TASK_16B92 (0xffffffff0070d7cb8)
#define TFP0_PORT_NAME_TO_TASK_16B92 (0xffffffff0070d82d8)
#define TFP0_KERNEL_TASK_CMP_1_16B92 (0xffffffff0070d7b04)
#define TFP0_KERNEL_TASK_CMP_2_16B92 (0xffffffff0070d810c)

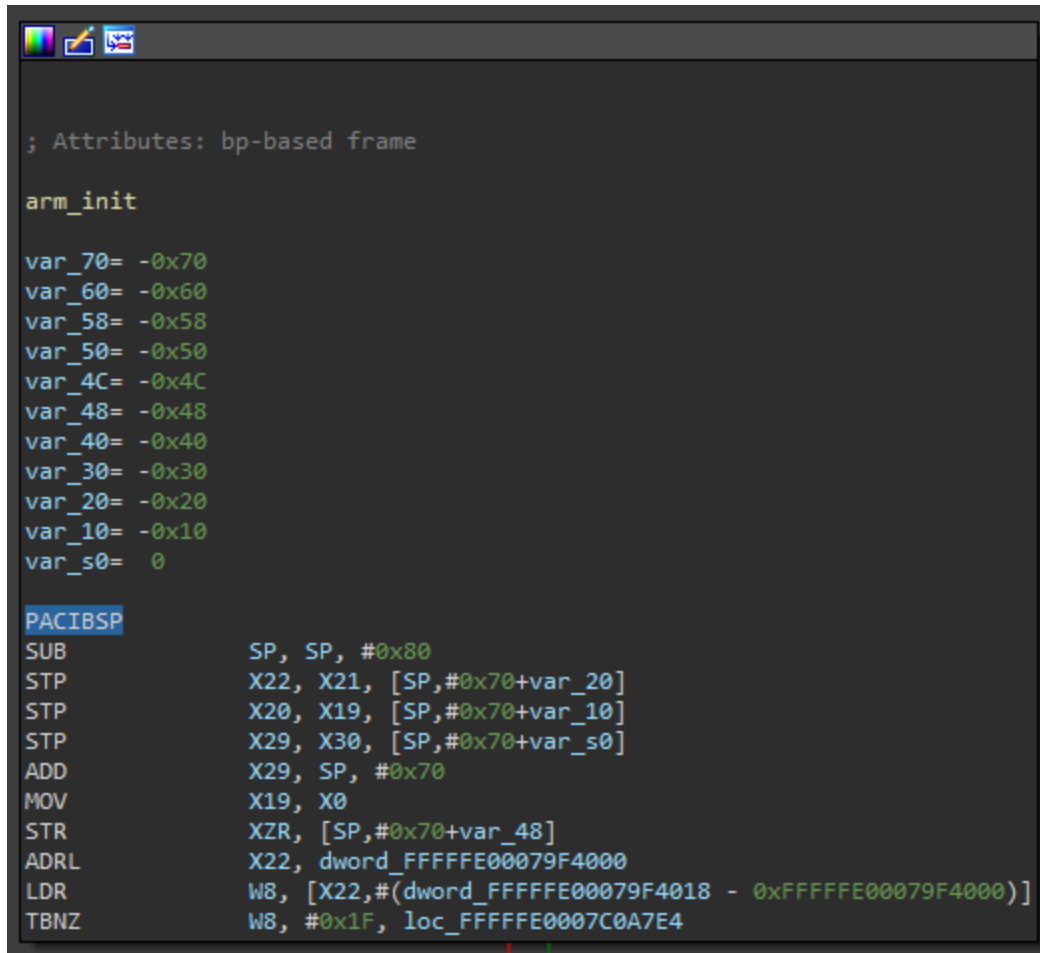
```

There was no way these hard-coded offsets would be compatible with the MacOS kernel image. This is when we realized we would need to tear the MacOS kernel apart in a disassembler to find the offsets.

## IDA

---

Completing this project would have been impossible without a disassembler. IDA 7.5 was the primary candidate, chiefly because of its support for ARM64e binaries and the latest A64 instruction set. For instance, earlier versions of IDA (namely 7.0) do not recognize the pointer authentication code for instruction key B (PACIBSP) instruction, which appears at the start of nearly every function in the MacOS ARM64e kernel:



```
; Attributes: bp-based frame

arm_init

var_70= -0x70
var_60= -0x60
var_58= -0x58
var_50= -0x50
var_4C= -0x4C
var_48= -0x48
var_40= -0x40
var_30= -0x30
var_20= -0x20
var_10= -0x10
var_s0= 0

PACIBSP
SUB     SP, SP, #0x80
STP     X22, X21, [SP,#0x70+var_20]
STP     X20, X19, [SP,#0x70+var_10]
STP     X29, X30, [SP,#0x70+var_s0]
ADD     X29, SP, #0x70
MOV     X19, X0
STR     XZR, [SP,#0x70+var_48]
ADRL   X22, dword_FFFFFFFE00079F4000
LDR     W8, [X22,#(dword_FFFFFFFE00079F4018 - 0xFFFFFFE00079F4000)]
TBNZ   W8, #0x1F, loc_FFFFFFFE0007C0A7E4
```

Figure 1.

Moreover, the kernel image used in this project contained no symbols. Functions had to be manually named, one by one, throughout the two-month testing and research period. ASCII strings offered the most reliable clues. The open-source XNU kernel was crucial in the struggle to identify the culprit of a crash or freeze. A total of 122645 functions have been defined in the IDA project so far. On initial analysis, however, IDA failed to define nearly every single function in the kernel binary. A script was needed to rectify this:

```

import idc
import struct
import idutils
pacibsp = "7F 23 03 D5" # PACIBSP
matches = []
ea = idc.find_binary(0, 1, pacibsp)
while ea != idc.BADADDR:
    matches.append(ea)
    ea = idc.find_binary(ea + 4, 1, pacibsp)
# Move backwards to avoid nesting functions
for matchea in reversed(matches):
    if not idc.get_func_name(matchea):
        idc.add_func(matchea)

```

The kernel image, kernelcache.release.j273.out, is ~83MB. While the script only took around 10 minutes to execute, the resulting mass of new functions and cross-references took over an hour to finish generating. After weeks of research and testing, all patches were written, and offsets defined:

```

#define INITIAL_BRANCH_VADDR_20C69 (0xfffffe0007ac4580)
#define BZERO_COND_BRANCH_VADDR_20C69 (0xfffffe0007ab8a3c)
#define SLIDE_SET_INST_VADDR_20C69 (0xfffffe000806b438)
#define CORE_TRUST_CHECK_20C69 (0xfffffe0008cb6538)
#define DISABLE_IMGPF_NOJOP_20C69 (0xfffffe000806b234)

```

## MSR Instructions

---

QEMU is not equipped to emulate ARM-based Apple systems. Over 110 of Apple’s model-specific hardware registers (MSR), in addition to hundreds of others, are currently unrecognized by QEMU. The Aleph Research team added support for 12 Apple-specific registers required for the iOS kernel to boot. To reach the goal of fully booting the MacOS ARM64 kernel, 24 more hardware registers needed support. But which registers would we need to add?

## Finding the Necessary Registers

---

Getting to that bash prompt after two grueling months of research and testing was anything but a straightforward process. Unsupported MSR registers tended to pop up intermittently as we diagnosed and fixed one crash after another. We typically followed a “panic, crash and patch” strategy, adding register support for individual MSR’s on an ad hoc basis. The list of definitions below from hw/arm/j273\_macos11.c are the result:



```

J273_CPREG_DEF(ARM64_REG_EHID1, 3, 0, 15, 3, 1, PL1_RW),
J273_CPREG_DEF(ARM64_REG_EHID10, 3, 0, 15, 10, 1, PL1_RW),
J273_CPREG_DEF(ARM64_REG_EHID4, 3, 0, 15, 4, 1, PL1_RW),
// EL2 registers
J273_CPREG_DEF(ARM64_REG_MIGSTS_EL1, 3, 4, 15, 0, 4, PL1_RW),
J273_CPREG_DEF(ARM64_REG_KERNELKEYLO_EL1, 3, 4, 15, 1, 0, PL1_RW),
J273_CPREG_DEF(ARM64_REG_KERNELKEYHI_EL1, 3, 4, 15, 1, 1, PL1_RW),
J273_CPREG_DEF(ARM64_REG_VMSA_LOCK_EL1, 3, 4, 15, 1, 2, PL1_RW),
J273_CPREG_DEF(APRR_EL0, 3, 4, 15, 2, 0, PL1_RW),
J273_CPREG_DEF(APRR_EL1, 3, 4, 15, 2, 1, PL1_RW),
J273_CPREG_DEF(CTRR_LOCK, 3, 4, 15, 2, 2, PL1_RW),
J273_CPREG_DEF(CTRR_A_LWR_EL1, 3, 4, 15, 2, 3, PL1_RW),
J273_CPREG_DEF(CTRR_A_UPR_EL1, 3, 4, 15, 2, 4, PL1_RW),
J273_CPREG_DEF(CTRR_CTL_EL1, 3, 4, 15, 2, 5, PL1_RW),
J273_CPREG_DEF(APRR_MASK_EN_EL1, 3, 4, 15, 2, 6, PL1_RW),
J273_CPREG_DEF(APRR_MASK_EL0, 3, 4, 15, 2, 7, PL1_RW),
J273_CPREG_DEF(ACC_CTRR_A_LWR_EL2, 3, 4, 15, 11, 0, PL1_RW),
J273_CPREG_DEF(ACC_CTRR_A_UPR_EL2, 3, 4, 15, 11, 1, PL1_RW),
J273_CPREG_DEF(ACC_CTRR_CTL_EL2, 3, 4, 15, 11, 4, PL1_RW),
J273_CPREG_DEF(ACC_CTRR_LOCK_EL2, 3, 4, 15, 11, 5, PL1_RW),
J273_CPREG_DEF(ARM64_REG_CYC_CFG, 3, 5, 15, 4, 0, PL1_RW),
J273_CPREG_DEF(ARM64_REG_CYC_OVRD, 3, 5, 15, 5, 0, PL1_RW),
J273_CPREG_DEF(IPI_SR, 3, 5, 15, 1, 1, PL1_RW),
J273_CPREG_DEF(UPMCR0, 3, 7, 15, 0, 4, PL1_RW),
J273_CPREG_DEF(UPMPCM, 3, 7, 15, 5, 4, PL1_RW),

```

Unrecognized system registers typically appear as `s#_#_c#_c#_#` in various debuggers, where `#` corresponds to the arguments in each of the definitions above. For example, `ARM64_REG_EHID1` or `APRR_EL0` are parsed as `s3_0_c15_c3_1` and `s3_4_c15_c2_0`, respectively. The last argument `PL1_RW` means “exception level 1 read/write”, which specifies the privilege level of the given registers. This indicates that the register is accessible in exception level 1 (EL1). Yet the level 2 (EL2) registers are marked with `PL1_RW`. This is because the following line in the function `j273_cpu_setup` prohibits EL2 registers:

```
object_property_set_bool(cpuobj, "has_el2", false, NULL);
```

Setting this to true proved problematic, as QEMU then became unable to switch from physical to virtual addressing early in the boot process. The fix involved forcing QEMU to treat these registers as EL1 registers in `define_one_arm_cp_reg_with_opaque` (`target/arm/helper.c`):

```
case 4:
case 5:
    /* min_EL EL2 */
    mask = PL1_RW; // changed from mask = PL2_RW
    break;
```

We were initially apprehensive of this fix, as modifying any official QEMU source files to bypass errors may prove to be a dangerous endeavor. Fortunately, no calamities arose, and eventually all MSR registers were accounted for.

## Device Tree

---

The device tree (DeviceTree.j273aap.im4p.out) was responsible for over half of the panics during testing. Several properties and devices were either absent from the tree entirely, usually causing a crash, or needed to be manually adjusted to prevent later issues. We have written a program that can apply the necessary changes to a device tree file using a diff-style file:

<https://github.com/cylance/macos-arm64-emulation/tree/main/dtetool>

To create a compatible device tree, back up the device tree file and apply the changes specified in dtediff\_20C69 with the dtetool program:

```
cp DeviceTree.j273aap.im4p.out DeviceTree.j273aap.im4p.out.backup
./dtetool DeviceTree.j273aap.im4p.out.backup -d dtediff_20C69 -o
DeviceTree.j273aap.im4p.out
```

The following section provides a more detailed description of each modification.

## Device Tree Modifications

---

The following property is changed to “running” to avoid an infinite loop in pe\_identify\_machine:

```
device-tree/cpus/cpu0/state                8  running
```

The first element in arm-io/ranges is changed to 0x100000000:

device-tree/arm-io/ranges

8 0x100000000

The following node is removed to ignore dockchannel-uart in order to use the default uart0 in serial\_init:

device-tree/dockchannel-uart

The following properties are added to the “chosen” node to avoid the panics at the end of arm\_init:

device-tree/chosen/dram-base

8 0

device-tree/chosen/dram-size

8 0

The following properties are added to the lock-regs node to avoid the panic in subroutine 0xffffe0007b2af00:

device-tree/chosen/lock-regs/amcc/aperture-count 4 0 d  
device-tree/chosen/lock-regs/amcc/aperture-size 4 0 d  
device-tree/chosen/lock-regs/amcc/plane-count 4 0 d  
device-tree/chosen/lock-regs/amcc/plane-stride 4 0 d  
device-tree/chosen/lock-regs/amcc/aperture-phys-addr 0  
device-tree/chosen/lock-regs/amcc/cache-status 4 0 d  
device-tree/chosen/lock-regs/amcc/cache-status-reg-offset 4 0 d  
device-tree/chosen/lock-regs/amcc/cache-status-reg-mask 4 0 d  
device-tree/chosen/lock-regs/amcc/cache-status-reg-value 4 0 d  
device-tree/chosen/lock-regs/amcc/amcc-ctrr-a/page-size-shift 4 0 d  
device-tree/chosen/lock-regs/amcc/amcc-ctrr-a/lower-limit 4 0 d  
device-tree/chosen/lock-regs/amcc/amcc-ctrr-a/lower-limit-reg-offset 4 0 d  
device-tree/chosen/lock-regs/amcc/amcc-ctrr-a/lower-limit-reg-mask 4 0 d  
device-tree/chosen/lock-regs/amcc/amcc-ctrr-a/upper-limit 4 0 d  
device-tree/chosen/lock-regs/amcc/amcc-ctrr-a/upper-limit-reg-offset 4 0 d  
device-tree/chosen/lock-regs/amcc/amcc-ctrr-a/upper-limit-reg-mask 4 0 d  
device-tree/chosen/lock-regs/amcc/amcc-ctrr-a/lock 4 0 d  
device-tree/chosen/lock-regs/amcc/amcc-ctrr-a/lock-reg-offset 4 0 d  
device-tree/chosen/lock-regs/amcc/amcc-ctrr-a/lock-reg-mask 4 0 d  
device-tree/chosen/lock-regs/amcc/amcc-ctrr-a/lock-reg-value 4 0 d  
device-tree/chosen/lock-regs/amcc/amcc-ctrr-a/enable 4 0 d  
device-tree/chosen/lock-regs/amcc/amcc-ctrr-a/write-disable 4 0 d

One final device tree property is modified: the nvram.

## **NVRAM**

---

No external NVRAM file is necessary, as the device tree file can house all NVRAM data in a property called nvram-proxy-data. By default, this property is completely empty, which led to several problems later (i.e. panics), which were related to null pointers. Crafting and configuring the NVRAM to the kernel's liking was a time-consuming task, albeit with a fairly simple outcome. Several issues cropped up in succession, and additional changes had to be made.

## **Null Pointers**

---

The first of the NVRAM-related issues happened in IODTNVRAM::init, where a null pointer reference to a lock variable caused a panic. This lock variable was supposed to have been initialized in IODTNVRAM::initNVRAMImage, but this function was never called. We discovered that the device-tree/chosen/nvram-total-size property in the device tree file was zero. Another panic occurred due to a null pointer reference in the IODTNVRAM::initOFVariables function. Apparently the nvram partition dictionary was not being set due to missing partition information in the device tree's nvram data.

## **Nvram-Proxy-Data**

---

The solution to this problem was to tailor the device-tree/chosen/nvram-proxy-data and device-tree/chosen/nvram-total-size properties to the kernel's needs. Clues as to what format this data must be in were given in IODTNVRAM::initNVRAMImage:

```

void
IODTNVRAM::initNVRAMImage(void)
{
// ... (snip)
// Look through the partitions to find the OF, MacOS partitions.
while (currentOffset < kIODTNVRAMImageSize) {
currentLength = ((UInt16 *)(_nvramImage + currentOffset))[1] * 16;
if (currentLength < 16) {
break;
}
partitionOffset = currentOffset + 16;
partitionLength = currentLength - 16;
if ((partitionOffset + partitionLength) > kIODTNVRAMImageSize) {
break;
}
if (strncmp((const char *)_nvramImage + currentOffset + 4,
kIODTNVRAMOFPartitionName, 12) == 0) {
_ofPartitionOffset = partitionOffset;
_ofPartitionSize = partitionLength;
}
// ... (snip)
initOFVariables();
}

```

When initialized, the NVRAM must be a valid, non-zero size no greater than 65536. This is specified in the nvram-total-size property. In addition to a valid size the nvram must have at least one valid partition with a size of at least 32 bytes. Valid partition names include “common” (defined as kIODTNVRAMOFPartitionName) or “system” (not in the most recent XNU source). Below is the updated device tree data for nvram-proxy-data:

```

...
00000f60: 73 79 73 63 66 67 2f 42 47 4d 74 00 6e 76 72 61  syscfg/BGMt.nvra
00000f70: 6d 2d 70 72 6f 78 79 2d 64 61 74 61 00 00 00 00  m-proxy-data....
00000f80: 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00  .....
00000f90: 00 00 02 00 63 6f 6d 6f 6e 00 00 00 00 00 00  ....common.....
00000fa0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
...

```

Actual data starts at offset 0xf90 in our modified device tree file. The partition’s size is calculated by multiplying the 16-bit integer at offset 0xf92 by 16. In this case, 2 \* 16 = 32 bytes. This includes the first 16 bytes containing the partition’s name and the remaining 16 null bytes. Now that the kernel is satisfied with our empty partition, the mystery of the missing NVRAM is solved.

## Forcing JOP

---

All the required MSR registers had been added to QEMU. The device tree was properly tailored to the kernel's requirements. The following launchd greeting in a GDB session gave a small boost of optimism:

```
Thu Jan 1 00:02:10 1970 localhost com.apple.xpc.launchd[1] <Notice>: hello
```

The holy grail of a shell prompt is still beyond our grasp at this point. Something was spinning in the kernel, impeding progress once again:

```
... (snip)
load_init_program: attempting to load /sbin/launchd
getExceptionList: failed to open /System/Library/Security/HardeningExceptions.plist
dyld: setting comm page to 0x800000000
000120.870479 wlan0.A[3] initWithProvider@120:amfm not matched
000120.913992 wlan0.A[4] deferredStart@1726: Lowered adjustBusy(-1), getBusyState()
-> 4
Thu Jan 1 00:02:10 1970 localhost com.apple.xpc.launchd[1] < Notice >: hello
Darwin Bootstrapper Version 7.0.0: Tue Aug 25 21:19:12 PDT 2020;
root:libxpc_executables-2038.40.23.161.1~1/launchd/RELEASE_arm64e
boot-args = debug=0x8 kextlog=0xffff cpus=1 rd=md0 serial=2
Thu Jan 1 00:02:10 1970 localhost com.apple.xpc.launchd[1] < Notice >: Restore
environment starting.
Thu Jan 1 00:02:10 1970 localhost com.apple.xpc.launchd[1] < Notice >: System
Integrity Protection is engaged.
Thu Jan 1 00:02:10 1970 localhost com.apple.xpc.launchd[1]
(com.apple.xpc.launchd.domain.system) < Notice >: entering ondemand mode
Thu Jan 1 00:02:10 1970 localhost com.apple.xpc.launchd[1] < Notice >: Skipping
boot-task: fsck
Thu Jan 1 00:02:10 1970 localhost com.apple.xpc.launchd[1] < Notice >: Skipping
boot-task: mount-phase-1
Thu Jan 1 00:02:10 1970 localhost com.apple.xpc.launchd[1] < Notice >: Skipping
boot-task: data-protection
Thu Jan 1 00:02:10 1970 localhost com.apple.xpc.launchd[1] < Notice >: Skipping
boot-task: check-migration-mode
Thu Jan 1 00:02:10 1970 localhost com.apple.xpc.launchd[1] < Notice >: Skipping
boot-task: unlock-data-volume
Thu Jan 1 00:02:10 1970 localhost com.apple.xpc.launchd[1] < Notice >: Doing boot
task: commit-boot-mode
Thu Jan 1 00:02:10 1970 localhost com.apple.xpc.launchd[1] < Notice >: boot-mode
committed: (null)
... (spinning)
```

In an infinite loop at 0xffffe00079ebcbc:

```

(gdb) c
Continuing.
^C
Program received signal SIGINT, Interrupt.
0xfffffe00079ebcbc in ?? ()
(gdb)

```

In Ldisable\_jop:

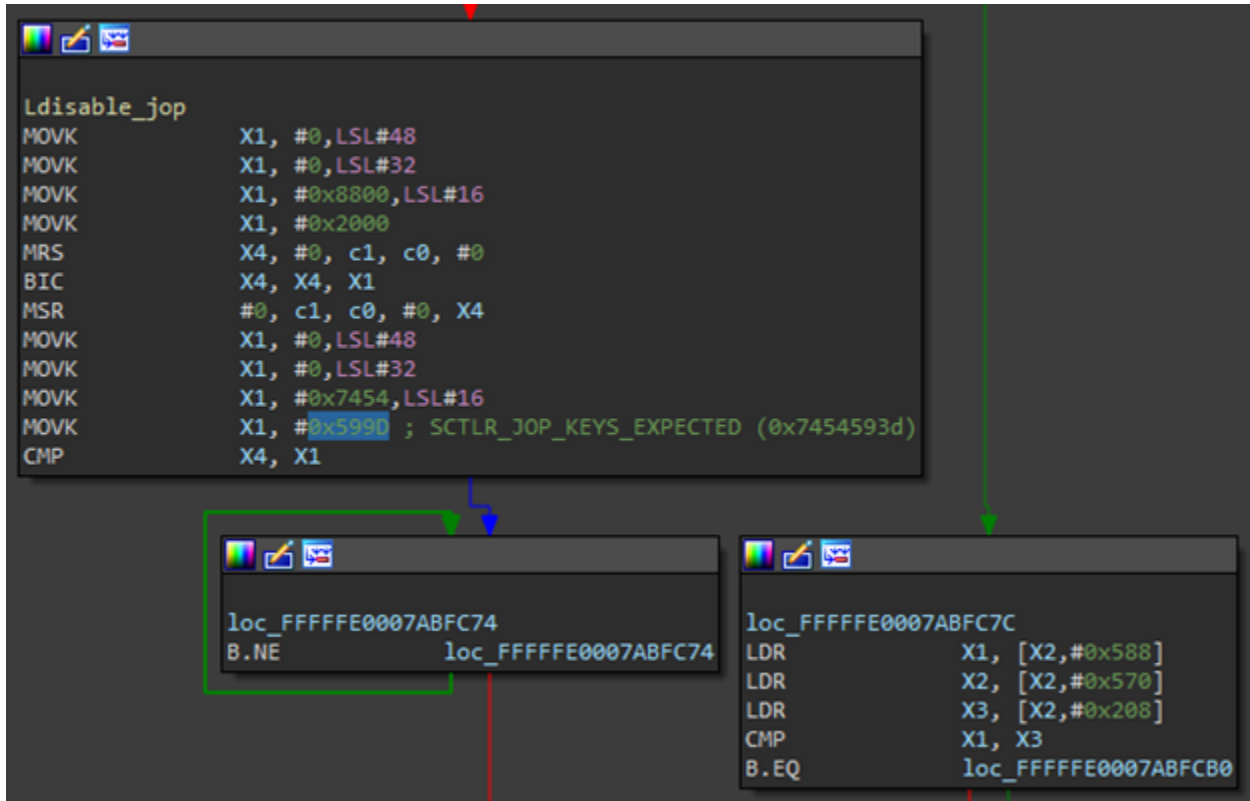


Figure 2.

Notice the infinitely looping B.NE instruction. How did it get here? We looked a bit further up in the disassembly:

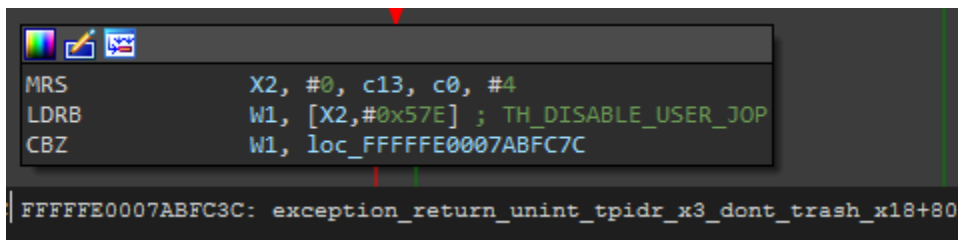
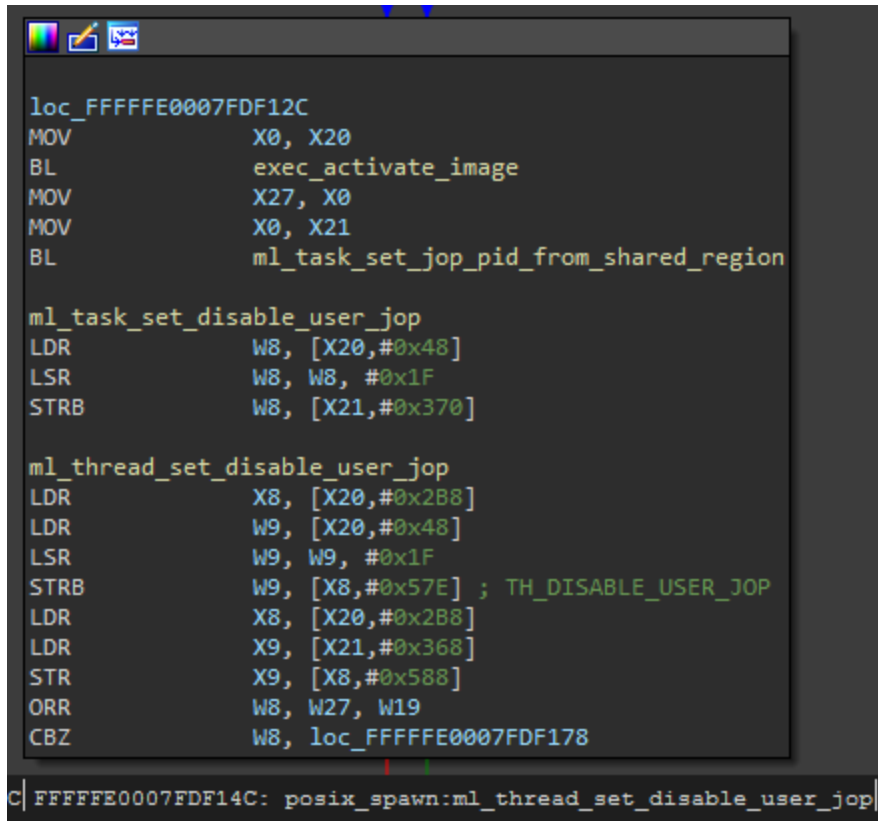


Figure 3.

XNU kernel threads have a member called TH\_DISABLE\_USER\_JOP. If set to a non-zero value, Ldisable\_jop is invoked. SCTLR\_EL1 (system control register) is then validated against a constant (0x7454599d) and freezes execution if the values do not match. Where is this thread property being set, and how can we prevent it in the most orthodox manner possible? Setting a write watchpoint in gdb for the address of the blocking thread's TH\_DISABLE\_USER\_JOP property leads to this location in posix\_spawn:



```
loc_FFFFFFFE0007FDF12C
MOV      X0, X20
BL       exec_activate_image
MOV      X27, X0
MOV      X0, X21
BL       ml_task_set_jop_pid_from_shared_region

ml_task_set_disable_user_jop
LDR      W8, [X20,#0x48]
LSR      W8, W8, #0x1F
STRB     W8, [X21,#0x370]

ml_thread_set_disable_user_jop
LDR      X8, [X20,#0x2B8]
LDR      W9, [X20,#0x48]
LSR      W9, W9, #0x1F
STRB     W9, [X8,#0x57E] ; TH_DISABLE_USER_JOP
LDR      X8, [X20,#0x2B8]
LDR      X9, [X21,#0x368]
STR      X9, [X8,#0x588]
ORR      W8, W27, W19
CBZ     W8, loc_FFFFFFFE0007FDF178

c| FFFFFFFE0007FDF14C: posix_spawn:ml_thread_set_disable_user_jop
```

Figure 4.

In posix\_spawn (xnu-6153.141.1/bsd/kern/kern\_exec.c):

```
error = exec_activate_image(imgp);
#if defined(HAS_APPLE_PAC)
ml_task_set_disable_user_jop(new_task, imgp->ip_flags & IMGPF_NOJOP ? TRUE : FALSE);
ml_thread_set_disable_user_jop(imgp->ip_new_thread, imgp->ip_flags & IMGPF_NOJOP ?
TRUE : FALSE);
#endif
```

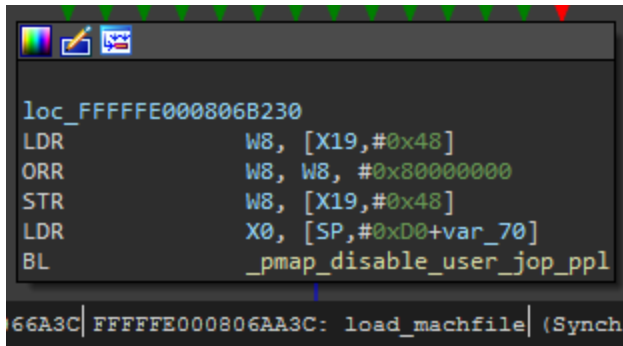
In the case of the blocking thread, imgp->ip\_flags is set to 0x80000000, which indicates the IMGPF\_NOJOP flag is enabled. The enabled status of this flag is written to the thread's TH\_DISABLE\_USER\_JOP property. So where is imgp->ip\_flags set?



## Load\_machfile

---

IMGPF\_NOJOP is enabled in load\_machfile near the end of the function:



```
loc_FFFFFFFF000806B230
LDR      W8, [X19,#0x48]
ORR      W8, W8, #0x80000000
STR      W8, [X19,#0x48]
LDR      X0, [SP,#0xD0+var_70]
BL       _pmap_disable_user_jop_ppl

66A3C| FFFFFFFF000806AA3C: load_machfile| (Synch
```

Figure 5.

Load\_machfile compares the Mach-O executable's identifier against several strings and enables IMGPF\_NOJOP if any are a match:

```
com.apple.security.cs.disable-library-validation
com.apple.private.cs.automator-plugins
com.apple.private.security.clear-library-validation
com.apple.perl5
com.apple.perl
org.python.python
com.apple.expect
com.tcltk.wish
com.tcltk.tclsh
com.apple.ruby
com.apple.bash
com.apple.zsh
com.apple.ksh
```

The bash identifier is among them. This flag may be a security mechanism to prevent certain executables from loading at boot time, as they may be used to compromise the system. Among them are several shell clients and script interpreters. The kernel blocks the executing thread if the Mach-O file's identifier matches any of the above strings.

## Solution

---

NOP over the ORR W8, W8 #0x80000000 instruction to keep IMGPF\_NOJOP disabled. This gave the go-ahead to the kernel to allow launchd to execute bash (via xpcproxy):

```
Thu Jan 1 00:01:58 1970 localhost com.apple.xpc.launchd[1] < Notice >: Skipping
boot-task: auto-pivot-root
Thu Jan 1 00:01:58 1970 localhost com.apple.xpc.launchd[1] < Notice >: Skipping
boot-task: allow-non-platform-code
Thu Jan 1 00:01:58 1970 localhost com.apple.xpc.launchd[1] < Notice >: Doing boot
task: restore-datapartition
Thu Jan 1 00:01:58 1970 localhost com.apple.xpc.launchd[1] < Notice >: restore-
datapartition: optional boot task not present
Thu Jan 1 00:01:58 1970 localhost com.apple.xpc.launchd[1] < Notice >: Skipping
boot-task: mount-phase-2
Thu Jan 1 00:01:58 1970 localhost com.apple.xpc.launchd[1] < Notice >: Skipping
boot-task: enable-swap
Thu Jan 1 00:01:58 1970 localhost com.apple.xpc.launchd[1] < Notice >: Skipping
boot-task: sandbox-enable-root-translation
Thu Jan 1 00:01:58 1970 localhost com.apple.xpc.launchd[1] < Notice >: Doing boot
task: init-with-data-volume
Thu Jan 1 00:01:58 1970 localhost com.apple.xpc.launchd[1] < Notice >: Skipping
boot-task: deferred_install
Thu Jan 1 00:01:58 1970 localhost com.apple.xpc.launchd[1] < Notice >: Skipping
boot-task: fips
Thu Jan 1 00:01:58 1970 localhost com.apple.xpc.launchd[1] < Notice >: Skipping
boot-task: cache-start
Thu Jan 1 00:01:58 1970 localhost com.apple.xpc.launchd[1] < Notice >: Skipping
boot-task: bootroot
Thu Jan 1 00:01:58 1970 localhost com.apple.xpc.launchd[1] < Notice >: Skipping
boot-task: init_featureflags
Thu Jan 1 00:01:58 1970 localhost com.apple.xpc.launchd[1] < Notice >: Skipping
boot-task: EndpointSecurity
Thu Jan 1 00:01:58 1970 localhost com.apple.xpc.launchd[1] < Notice >: Skipping
boot-task: rc.server
Thu Jan 1 00:01:58 1970 localhost com.apple.xpc.launchd[1] < Notice >: Skipping
boot-task: tzinit
Thu Jan 1 00:01:58 1970 localhost com.apple.xpc.launchd[1] < Notice >: Skipping
boot-task: dirhelper
Thu Jan 1 00:01:58 1970 localhost com.apple.xpc.launchd[1] < Notice >: Skipping
boot-task: rootless-init
Thu Jan 1 00:01:58 1970 localhost com.apple.xpc.launchd[1] < Notice >: Skipping
boot-task: finish-demo-restore
Thu Jan 1 00:01:58 1970 localhost com.apple.xpc.launchd[1] < Notice >: Skipping
boot-task: systemstats
Thu Jan 1 00:01:58 1970 localhost com.apple.xpc.launchd[1] < Notice >: Skipping
boot-task: prng_seedctl
Thu Jan 1 00:01:58 1970 localhost com.apple.xpc.launchd[1] < Warning >: Unable to
load cache
Thu Jan 1 00:01:58 1970 localhost com.apple.xpc.launchd[1] < Notice >: launchd
UUID: 438960C9-7E3C-3D4A-9EA8-643FF64ACDF2
Thu Jan 1 00:01:58 1970 localhost com.apple.xpc.launchd[1] < Notice >: Early boot
complete. Continuing system boot.
Thu Jan 1 00:01:58 1970 localhost com.apple.xpc.launchd[1]
(com.apple.xpc.launchd.domain.system) < Notice >: entering bootstrap mode
Thu Jan 1 00:01:58 1970 localhost com.apple.xpc.launchd[1]
(com.apple.xpc.launchd.domain.system) < Error >: Failed to bootstrap path: path =
/Library/Apple/System/Library/LaunchDaemons, error = 2: No such file or directory
Thu Jan 1 00:01:58 1970 localhost com.apple.xpc.launchd[1]
```

```
(com.apple.xpc.launchd.domain.system) < Critical >: No task-access server
configured! The system will not get very far.
Thu Jan  1 00:01:58 1970 localhost com.apple.xpc.launchd[1]
(com.apple.xpc.launchd.domain.system) < Notice >: exiting bootstrap mode
Thu Jan  1 00:01:58 1970 localhost com.apple.xpc.launchd[1] < Notice >: Skipping
boot-task: cache-tag
Thu Jan  1 00:01:58 1970 localhost com.apple.xpc.launchd[1]
(com.apple.xpc.launchd.domain.system) < Notice >: exiting ondemand mode
The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
bash-3.2#
```

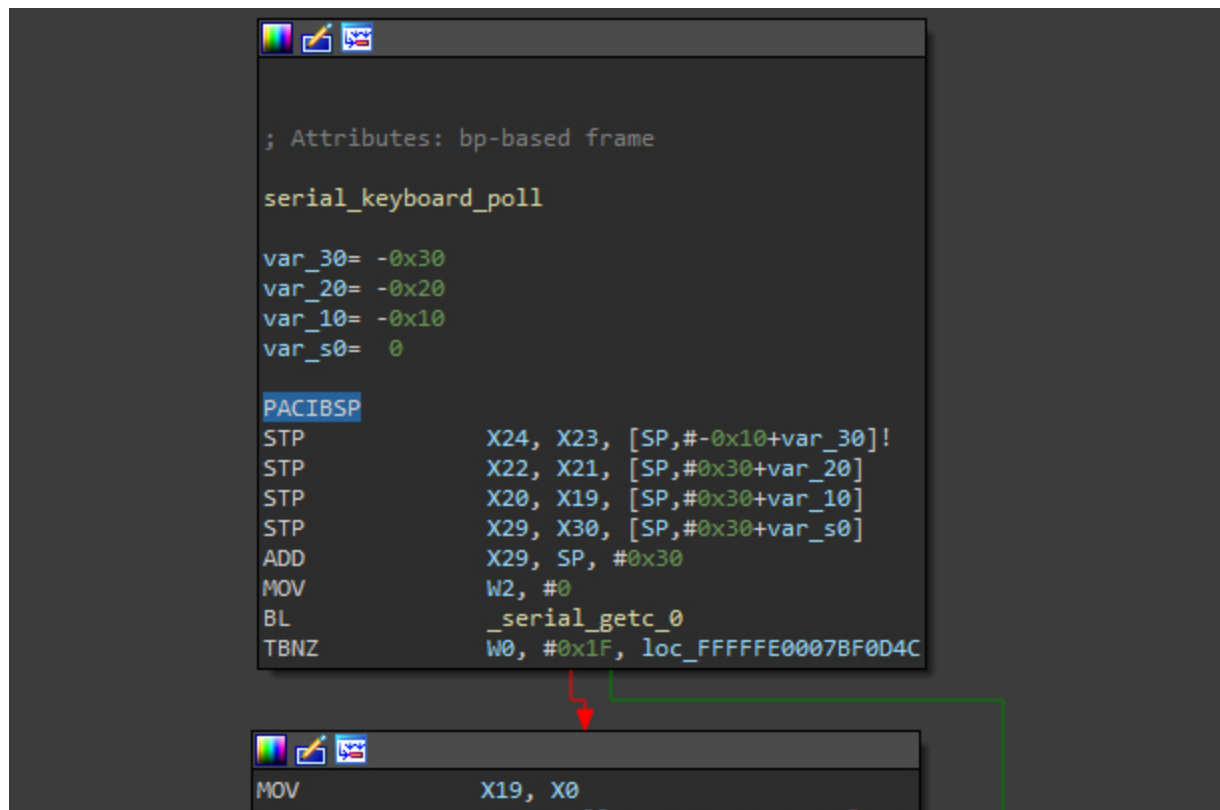
A prompt appeared at last. Yet stdin is not working and there is no keyboard input.

## Serial Keyboard and FIQ

A fully functioning bash prompt is useless without keyboard input. Keyboard input is read in a separate thread in a function called `serial_keyboard_poll`. The `serial_keyboard_poll`:

- Reads all pending characters from the stdin buffer
- Sets the 16-millisecond deadline by calling `assert_wait_deadline`
- Blocks execution in `thread_block` until the deadline has passed

`serial_keyboard_poll` is then re-invoked by `thread_invoke`, and this sequence of events repeats indefinitely for as long as bash is open.



```
ADRP      X21, #011_FFFFFFF000AF3C9B8@PAGE
ADRL      X22, unk_FFFFFFF000AEE29E0
MOV       W23, #0x90E1
```

```
loc_FFFFFFF0007BF0D10
LDR       X20, [X21,#off_FFFFFFF000AF3C9B8@PAGEOFF]
MOV       X0, X20
BL        lck_mtx_lock
LDRSW    X8, [X20,#0xB8]
ADD       X8, X22, X8,LSL#6
LDR       X8, [X8,#0x28]
SXTB     W0, W19
MOV       X1, X20
BLRAA    X8, X23
MOV       X0, X20
BL        lck_mtx_unlock
MOV       W2, #0
BL        _serial_getc_0
MOV       X19, X0
TBZ      W0, #0x1F, loc_FFFFFFF0007BF0D10
```

```
loc_FFFFFFF0007BF0D4C
ADRL      X8, off_FFFFFFF000AEAC000
LDR       W8, [X8]
MOV       W9, #0xF42400
MUL      X8, X8, X9
LSR      X8, X8, #9
MOV       X9, #0x44B82FA09B5A53
UMULH    X8, X8, X9
ISB
MOV       X0, #0
MRS      X9, #3, c14, c0, #2
MRS      X10, #0, c13, c0, #4
LDR      X10, [X10,#0x570]
LDR      X10, [X10,#0x58]
ADD      X9, X10, X9
ADDS     X8, X9, X8,LSR#11
CSINV    X2, X8, XZR, CC
ADRL     X16, serial_keyboard_poll
MOV      X17, #0x4A27
PACIA    X16, X17
CBZ      X16, loc_FFFFFFF0007BF0DC4
```

```
ADRL      X16, serial_keyboard_poll
PACIZA    X16
MOV       X0, X16
```

```
loc_FFFFFFF0007BF0DC4
MOV       W1, #0
BL        assert_wait_deadline
```

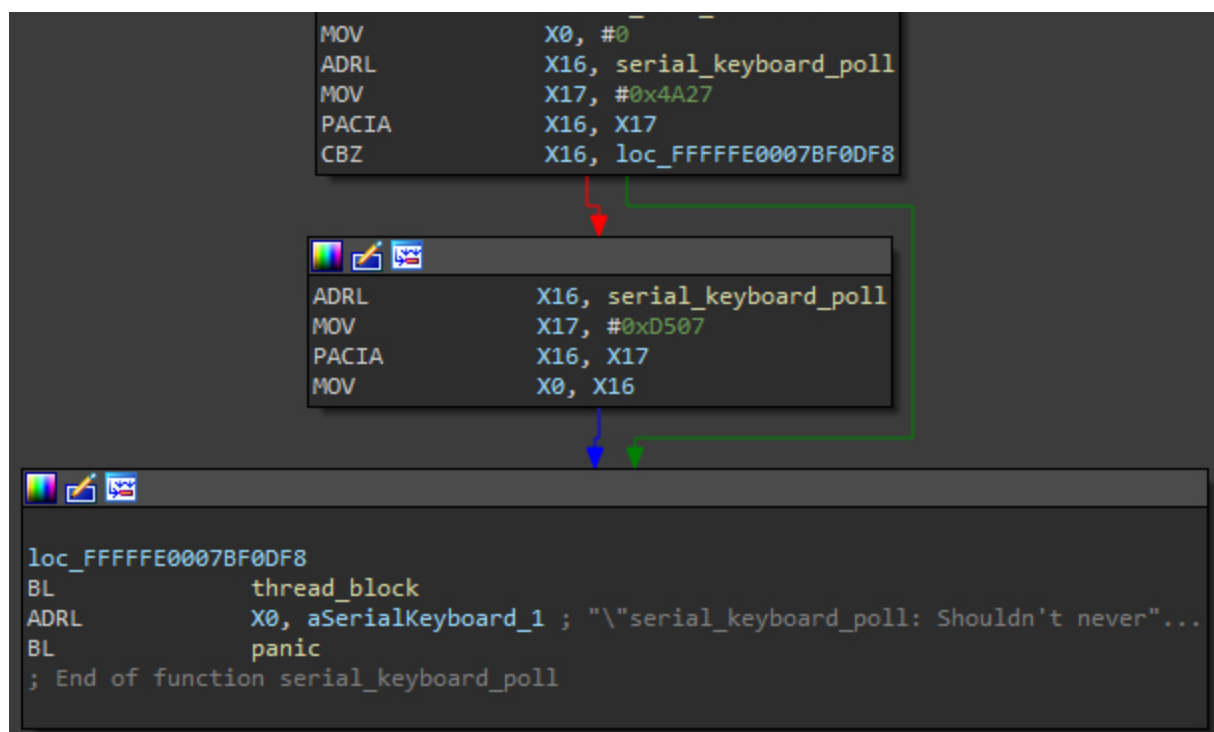


Figure 6.

Yet this wasn't happening. Thread\_block was called only once, the thread stalled, and serial\_keyboard\_poll was never invoked again.

We thought: maybe the deadline wasn't being reached? Wait deadlines are specified in the assert\_wait\_deadline function. This function creates a waitq object for the current keyboard-polling thread with an optional deadline, in nanoseconds. When thread\_block is called shortly afterwards, the thread hangs until the thread\_clear\_waitq\_state kernel function clears the thread's waitq object:

```
static inline void
thread_clear_waitq_state(thread_t thread)
{
    thread->waitq = NULL;
    thread->wait_event = NO_EVENT64;
    thread->at_safe_point = FALSE;
}
```

But this function never got called, and the deadline was never reached. What does the thread do? It switches context to the system idle thread and blocks indefinitely, never invoking a continuation of serial\_keyboard\_poll to continue polling for stdin. Surely something somewhere is called that may lead to thread\_clear\_waitq\_state? By back tracing the sequence of function calls in the XNU source, we discovered the origin of interrupt timers:

```
thread_clear_waitq_state
waitq_pull_thread_locked
clear_wait_internal
thread_timer_expire
timer_queue_expire_with_options
timer_intr
rtclock_intr
sleh_fiq
_fleh_fiq
```

After analyzing this call flow, we were inching ever closer to the source of the problem. Analyzing the interoperability of threads, timers, clocks, and interrupts in the XNU kernel was slowly paying dividends.

## Fast Interrupt Requests (FIQ)

---

Further investigation revealed that not even fleh\_fiq (first-level exception handler for Fast Interrupt Request) was being called. Something was seriously wrong, as fleh\_fiq is integral to a working interrupt timer. Peripheral devices like keyboards and mice typically communicate with the kernel using Fast Interrupt Requests (FIQ) in the ARM architecture. We confirmed that FIQs were not firing in the emulator after looking at QEMU's source. We discovered that arm\_cpu\_exec\_interrupt, which is called for a fast interrupt request, never got called. No amount of keyboard-spamming would trigger this or any of the kernel functions listed above.

## Hardware Timers

---

Perhaps another possibility for a non-operational FIQ handler was timer-related? Threads often rely on hardware timers to notify the thread of a passed deadline. If no timer exists, no countdowns can be performed and idle threads waiting for a deadline will stall the system. Without a hardware timer to poll for FIQs, MacOS will ignore them. This is significant, as it would have affected not only standard keyboard input, but other areas of the kernel as well. Wait deadlines would never be reached, threads would hang, and services would never start simply because no hardware timer was present.

## Enable\_timebase\_event\_stream

---

We did a comparison between the iOS and MacOS kernel for the enable\_timebase\_event\_stream function. Why this function? Because this function modifies three timer-related system registers and could provide the answer to our timer dilemma. Below is from the MacOS kernel:

```

_enable_virtual_timer
MOV      W8, #1
MSR      #3, c14, c3, #1, X8 ; CNTV_CTL_EL0 (Counter-timer Virtual Timer Control Register)
MOV      W8, #2
MSR      #3, c14, c2, #1, X8 ; CNTP_CTL_EL0 (Counter-timer Physical Timer Control register)
ADRL     X0, aAprrJit ; "aprr_jit"
ADRL     X19, dword_FFFFFFF000A372568
MOV      X1, X19
MOV      W2, #4
MOV      W3, #0
BL       PE_parse_boot_argn_internal
LDR      W8, [X19]
CBNZ    W8, loc_FFFFFFF0007C0B1A8

```

Figure 7.

Contrast the above with the corresponding iOS kernel disassembly:

```

loc_FFFFFFF0071BB05C ; CNTKCTL_EL1 (Counter-timer Kernel Control register)
MRS      X8, #0, c14, c1, #0
MOV      W9, #0xD
BFI      W9, W19, #4, #0x1C
ORR      X8, X8, X9
MSR      #0, c14, c1, #0, X8 ; CNTKCTL_EL1 (Counter-timer Kernel Control register)

_enable_virtual_timer
MOV      W8, #1
MSR      #3, c14, c2, #1, X8 ; CNTP_CTL_EL0 (Counter-timer Physical Timer Control register)
BL       early_random
AND      X8, X0, #0xFFFFFFFFFFFFFFFF
ADRP     X9, #__stack_chk_guard@PAGE
NOP
STR      X8, [X9, #__stack_chk_guard@PAGEOFF]
BL       sub_FFFFFFF0071C1FC8

```

Figure 8.

We immediately noticed the presence of an extra register in the MacOS kernel: CNTV\_CTL\_EL0. With the help of a comprehensive list of iOS ARM64 system registers, we began to make the connection between these registers and the timer issues. The iOS kernel appeared to be enabling the physical timer by writing 1 to it, while the MacOS kernel was enabling the virtual timer by writing 1 to it. Then we noticed the line in the xnu-gemu-ARM64 source:

```
qdev_connect_gpio_out(cpudev, GTIMER_PHYS,  
                      qdev_get_gpio_in(cpudev, ARM_CPU_FIQ));
```

The GTIMER\_VIRT constant gave it away almost instantly. QEMU's official source lists five different global timer types in target/arm/cpu.h:

```
#define GTIMER_PHYS      0  
#define GTIMER_VIRT     1  
#define GTIMER_HYP      2  
#define GTIMER_SEC      3  
#define GTIMER_HYPVIRT  4
```

The solution was incredibly simple: MacOS uses a virtual timer. iOS uses a physical timer. Therefore, QEMU must specify a virtual timer with GTIMER\_VIRT instead of GTIMER\_PHYS when linking the timer to FIQ. Switching to GTIMER\_VIRT:

```
qdev_connect_gpio_out(cpudev, GTIMER_VIRT,  
                      qdev_get_gpio_in(cpudev, ARM_CPU_FIQ));
```

Followed by booting up the MacOS kernel, waiting for the bash prompt, then:

```
bash-3.2# asdfasdfasdf  
bash-3.2# ls  
.fseventsd      dev             mnt3            mnt7            sbin  
Library         etc             mnt4            mnt8            usr  
System          mnt1           mnt5            mnt9            var  
bin             mnt2           mnt6            private  
bash-3.2#
```

Such a simple answer to a cryptic problem. This final fix marks the end of the final phase of a fully bootable MacOS 11 ARM64e kernel.

## Achieving a Functioning Emulator

---

This chronicling of discoveries, fixes and accomplishments would not be complete without long-term failures and ineffective bypasses. Below are several examples that involved multiple days of research and testing and created quite a bit of frustration throughout.



## IORTC

---

Before we discovered, diagnosed, and mitigated the timer dilemma, something related to the initialization of the RTC (real-time clock) had been blocking the `bsd_init` thread. `IOKitInitializeTime` was waiting for a matching IORTC service. The wait was initiated by `IOService::waitForMatchingService`, which relies on `assert_wait` or `assert_wait_deadline` to begin blocking the thread until a condition is met. `assert_wait_deadline` is non-functional without a working hardware timer. The kernel had no working hardware timer. We tried adding the `no-rtc` property to the device tree root and a child node (with the name `rtc`) to the `arm-io` node. This would invoke the bootstrap to immediately publish the IORTC service and skip the wait. After the real source of the problem was determined and fixed, these device tree nodes were removed.

## Task-access Server and SIP

---

The following message from the `launchd` output was a bit disconcerting:

```
< Critical >: No task-access server configured! The system will not get very far.
```

It was originally attributed to code-signing, then attributed to SIP, and finally ignored once a functioning bash prompt was operational. This message caused several unneeded headaches. A task-access server is related to communication between tasks over the task-access port (defined as constant `TASK_ACCESS_PORT` with a value of 9). Assuming this only affected TCP connections, which this emulation project currently does not support, we ignored the error.

## iOS Binary Incompatibility

---

RootlessJB provides common Mach-O ARM64e binaries, including `bash`, that are runnable in the iOS kernel. These are iOS binaries for an iOS kernel. This explains why one is likely to see the following message when attempting to run said binaries on a MacOS system and never see a bash prompt:

```
Using iOS Platform policy
port is not ready for callouts
```

The necessary command line tools are part of the base ARM64 system, archived deep within the MacOS Big Sur installer package. This is simply a warning to those attempting execution of the aforementioned iOS binaries in a MacOS environment: it probably won't work.

## **Conclusion**

---

This project was a successful experiment in cross-platform emulation that has potential for future development. Hard disk and TCP tunneling (which xnu-qemu-arm64 already supports for iOS) still await implementation. Multi-core and KVM support would dramatically reduce the boot time, perhaps to mere seconds, and eliminate massive overhead. Full graphical support is a mere prospect (even less so in a cross-platform environment). But graphical support is low priority, so long as a functioning shell client is present. If it works, that is enough of a motivation to make it work well.

## **Example Commands**

---

To complement the article, we have decided to provide examples of command output from an emulated MacOS 11 ARM64e guest. For example, the following shows output from the lsof program:

```
bash-3.2# lsof -c launchd
```

COMMAND	PID	USER	FD	TYPE	DEVICE	SIZE/OFF	NODE	NAME
launchd	1	root	cwd	DIR	2,0	748	2	/
launchd	1	root	txt	REG	2,0	418976	940	/sbin/launchd
launchd	1	root	txt	REG	2,0	821120	958	/usr/lib/dyld
launchd	1	root	txt	REG	2,0	60880	977	/usr/lib/libSystem.B.dylib
launchd	1	root	txt	REG	2,0	443120	1024	/usr/lib/libobjc.A.dylib
launchd	1	root	txt	REG	2,0	79072	985	/usr/lib/libauditd.0.dylib
launchd	1	root	txt	REG	2,0	161504	989	/usr/lib/libbsm.0.dylib
launchd	1	root	txt	REG	2,0	77424	1044	/usr/lib/system/libcache.dylib
launchd	1	root	txt	REG	2,0	163856	1045	/usr/lib/system/libcommonCrypto.dylib
launchd	1	root	txt	REG	2,0	64304	1046	/usr/lib/system/libcompiler_rt.dylib
launchd	1	root	txt	REG	2,0	101696	1047	/usr/lib/system/libcopyfile.dylib
launchd	1	root	txt	REG	2,0	656832	1048	/usr/lib/system/libcorecrypto.dylib
launchd	1	root	txt	REG	2,0	557520	1051	/usr/lib/system/libdispatch.dylib
launchd	1	root	txt	REG	2,0	501456	1052	/usr/lib/system/libdyld.dylib
launchd	1	root	txt	REG	2,0	57616	1053	/usr/lib/system/libkeymgr.dylib
launchd	1	root	txt	REG	2,0	35296	1055	/usr/lib/system/liblaunch.dylib
launchd	1	root	txt	REG	2,0	80192	1056	/usr/lib/system/libmacho.dylib
launchd	1	root	txt	REG	2,0	64432	1059	/usr/lib/system/libquarantine.dylib
launchd	1	root	txt	REG	2,0	59168	1060	/usr/lib/system/libremovefile.dylib
launchd	1	root	txt	REG	2,0	197136	1061	/usr/lib/system/libsystem_asl.dylib
launchd	1	root	txt	REG	2,0	110560	1062	/usr/lib/system/libsystem_blocks.dylib
launchd	1	root	txt	REG	2,0	758912	1063	/usr/lib/system/libsystem_c.dylib
launchd	1	root	txt	REG	2,0	53456	1064	/usr/lib/system/libsystem_collections.dylib
launchd	1	root	txt	REG	2,0	100864	1065	/usr/lib/system/libsystem_configuration.dylib
launchd	1	root	txt	REG	2,0	176752	1066	/usr/lib/system/libsystem_containermanager.dylib
launchd	1	root	txt	REG	2,0	99584	1067	/usr/lib/system/libsystem_coreservices.dylib
launchd	1	root	txt	REG	2,0	144048	1068	/usr/lib/system/libsystem_darwin.dylib
launchd	1	root	txt	REG	2,0	132912	1069	/usr/lib/system/libsystem_dnssd.dylib
launchd	1	root	txt	REG	2,0	75376	1070	/usr/lib/system/libsystem_featureflags.dylib
launchd	1	root	txt	REG	2,0	451008	1071	/usr/lib/system/libsystem_info.dylib
launchd	1	root	txt	REG	2,0	244320	1073	/usr/lib/system/libsystem_m.dylib
launchd	1	root	txt	REG	2,0	317408	1074	/usr/lib/system/libsystem_malloc.dylib
launchd	1	root	txt	REG	2,0	166784	1075	/usr/lib/system/libsystem_networkextension.dylib
launchd	1	root	txt	REG	2,0	115616	1076	/usr/lib/system/libsystem_notify.dylib

```

launchd  1 root  txt    REG    2,0    21952 1078
/usr/lib/system/libsystem_product_info_filter.dylib
launchd  1 root  txt    REG    2,0    103232 1080
/usr/lib/system/libsystem_sandbox.dylib
launchd  1 root  txt    REG    2,0    80112 1081
/usr/lib/system/libsystem_secinit.dylib
launchd  1 root  txt    REG    2,0    373952 1072
/usr/lib/system/libsystem_kernel.dylib
launchd  1 root  txt    REG    2,0    105232 1077
/usr/lib/system/libsystem_platform.dylib
launchd  1 root  txt    REG    2,0    149616 1079
/usr/lib/system/libsystem_pthread.dylib
launchd  1 root  txt    REG    2,0    101552 1082
/usr/lib/system/libsystem_symptoms.dylib
launchd  1 root  txt    REG    2,0    242032 1083
/usr/lib/system/libsystem_trace.dylib
launchd  1 root  txt    REG    2,0    114304 1085 /usr/lib/system/libunwind.dylib
launchd  1 root  txt    REG    2,0    467728 1086 /usr/lib/system/libxpc.dylib
launchd  1 root  txt    REG    2,0    268336 993 /usr/lib/libc++abi.dylib
launchd  1 root  txt    REG    2,0    80128 1022 /usr/lib/liboah.dylib
launchd  1 root  txt    REG    2,0    787520 991 /usr/lib/libc++.1.dylib
launchd  1 root  0w    CHR    0,0    0t4982 291 /dev/console
launchd  1 root  1w    CHR    0,0    0t4982 291 /dev/console

```

The df command lists active file systems, such as the root device where launchd, bash, and all other userland programs are located:

```

bash-3.2# df -h
Filesystem      Size  Used Avail Capacity iused      ifree %iused  Mounted on
root_device    1.5Gi 130Mi 1.4Gi    9%    1668 4294965611    0%  /
devfs          168Ki   0Bi  100%    580          0 100%  /dev

```

View network interfaces with ifconfig:

```

bash-3.2# ifconfig
ALF, old data swfs_pid_entry , updaterules_msg < ptr >, updaterules_state < ptr >
>lo0: flags=8049< UP,LOOPBACK,RUNNING,MULTICAST > mtu 16384
    options=1203< RXCSUM,TXCSUM,TXSTATUS,SW_TIMESTAMP >
    inet 127.0.0.1 netmask 0xff000000
    inet6 ::1 prefixlen 128
    inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
    nd6 options=201< PERFORMNUD,DAD >
gif0: flags=8010< POINTOPOINT,MULTICAST > mtu 1280
stf0: flags=0<> mtu 1280

```

Finally, the full output of a shutdown command:

```
bash-3.2# shutdown -h now
Shutdown NOW!
System shutdown time has arrived
Thu Jan  1 00:06:11 1970 localhost com.apple.xpc.launchd[1]
(com.apple.xpc.launchd.domain.system) : System shutdown initiated by: shutdown.10<-
bash.3<-launchd.1
Thu Jan  1 00:06:11 1970 localhost com.apple.xpc.launchd[1]
(com.apple.xpc.launchd.domain.system) < Notice >: committing to system shutdown
Kext loading now disabled.
Kext unloading now disabled.
Kext autounloading now disabled.
Kernel requests now disabled.
System shutdown; requesting immediate kernelmanagerd exit.
ASP: System is shutting down, preventing further ASP upcalls
ASP: ASP: shutting down, drained
bash-3.2# Thu Jan  1 00:06:41 1970 localhost com.apple.xpc.launchd[1]
(com.apple.xpc.launchd.domain.system) : shutdown UNINITIALIZED -> COMMITTED
Thu Jan  1 00:06:41 1970 localhost com.apple.xpc.launchd[1]
(com.apple.xpc.launchd.domain.system) < Notice >: shutdown already committed
Thu Jan  1 00:06:41 1970 localhost com.apple.xpc.launchd[1]
(com.apple.xpc.launchd.domain.system) < Notice >: shutdown COMMITTED -> COMMITTED
Thu Jan  1 00:06:41 1970 localhost com.apple.xpc.launchd[1]
(com.apple.xpc.launchd.domain.system) < Notice >: shutdown COMMITTED ->
WAITING_ON_SERVICES
ASP: System is shutting down, (SIP is ENABLED) allowing process at path:
/usr/sbin/spindump
Thu Jan  1 00:06:43 1970 localhost com.apple.xpc.launchd[1] < Notice >: Generating
report...
dyld: dyld cache load error: shared cache file open() failed
dyld: Library not loaded:
/System/Library/PrivateFrameworks/CoreSymbolication.framework/Versions/A/CoreSymbolic
    Referenced from: /usr/sbin/spindump
    Reason: image not found
AMFI: Denying core dump for pid 11 (spindump)Thu Jan  1 00:06:44 1970 localhost
com.apple.xpc.launchd[1] < Warning >: shutdown-stall: non-boot task exited with
status 6
Thu Jan  1 00:06:44 1970 localhost com.apple.xpc.launchd[1] < Notice >: Report
generated in 0 seconds.
Thu Jan  1 00:06:44 1970 localhost com.apple.xpc.launchd[1] < Error >: Host-level
exception raised: pid = 11, thread = 0x50f, exception type = 0xd, codes = {
25769803777 }, states = { 0 }
syncing disks... Killing all processes
continuing
hfs: unmount initiated on GoldenGateC20C69.arm64eSURamDisk on device b(2, 0)
done
CPU halted
ASP: ASP: shutting down
panic(cpu 0 caller 0xfffffe0008289f38): "Halt/Restart Timed Out"
```

Yes, that is a panic at the end. Add “fatal shutdown” to the list of issues awaiting a fix.

## Resources

---

### Setup Guide

<https://github.com/cylance/macos-arm64-emulation>

### Aleph Research

[Running iOS in QEMU to an interactive bash shell \(1\): tutorial](#)

<https://github.com/alephsecurity/xnu-qemu-arm64>

<https://github.com/alephsecurity/xnu-qemu-arm64-tools>

### XNU

XNU source tarballs - <https://opensource.apple.com/tarballs/xnu/>

XNU source - <https://github.com/apple-opensource/xnu>

### Tools

QEMU 5.1.0 - <https://www.qemu.org/download/#source>

OSX-KVM - <https://github.com/kholia/OSX-KVM>

xar - <https://github.com/mackyle/xar>

apfs-fuse - <https://github.com/sgan81/apfs-fuse>

### Links

<https://landley.net/aboriginal/presentation.html>

<https://developer.android.com/studio/run/emulator-commandline>

<https://developer.arm.com/documentation/dui0801/g/A64-Data-Transfer-Instructions/LDAPR>

<https://developer.arm.com/architectures/instruction-sets/base-isas/a64>

<https://worthdoingbadly.com/xnuqemu3/>



## About The BlackBerry Research & Intelligence Team

---

The BlackBerry Research & Intelligence team examines emerging and persistent threats, providing intelligence analysis for the benefit of defenders and the organizations they serve.

---

[Back](#)