Debugging the XNU Kernel with IDA Pro

Table of Contents

1. Purpose . . . 1
2. Debugging OSX with VMware . . . 1
   2.1. Quick Start . . . 1
   2.2. Using the KDK . . . 3
   2.3. Debugging a Development Kernel . . . 4
   2.4. Assembly-Level Debugging + DWARF . . . 5
   2.5. KEXT Debugging . . . 6
   2.6. Debugging a Prelinked Kernelcache . . . 7
   2.7. Debugging the OSX Kernel Entry Point . . . 10
3. macOS11 Kernel Debugging . . . 12
   3.1. KernelCollections . . . 12
   3.2. Creating a macOS11 VM . . . 12
   3.3. Debugging: Quick Start . . . 13
   3.4. Symbolicating KernelCollections . . . 14
   3.5. Debugging KernelCollections . . . 16
   3.6. macOS11 + DWARF . . . 17
4. UEFI Debugging . . . 17
   4.1. Debugging the OSX Bootloader . . . 18
   4.2. GetMemoryMap . . . 19
   4.3. UEFI Debugging + DWARF . . . 19
5. Debugging iOS with Corellium . . . 21
   5.1. Quick Start . . . 21
   5.2. Creating a KDK for iOS . . . 23
   5.3. Debugging the iOS Kernel Entry Point . . . 24
6. Known Issues and Limitations . . . 25
   6.1. iBoot debugging . . . 25
   6.2. 32-bit XNU . . . 26
   6.3. KDP . . . 26

Last updated on November 1, 2020 — v1.1

1. Purpose

IDA 7.3 introduces the Remote XNU Debugger. It is designed to communicate with the GDB stub included with popular virtualization tools, namely VMware Fusion (for OSX) and Corellium (for iOS). The debugger allows you to observe the Darwin kernel as it is running, while at the same time utilising the full power of IDA’s analysis capabilities. It works equally well on Mac, Windows, and Linux.

This writeup is intended to quickly get you familiar with debugger, as well as offer some hints to make the experience as smooth as possible.

2. Debugging OSX with VMware

2.1. Quick Start

To get started with debugging OSX, we will perform a simple experiment. This is the same experiment outlined in this great writeup by GeoSn0w, but we will be performing the equivalent in IDA - which we hope you’ll find is much simpler.

Begin with the following setup:

1. create an OSX virtual machine with VMware Fusion. in this example the VM is OSX 10.13.6, but the experiment should work with any recent OSX version.
2. open Terminal in the VM and enable some basic XNU debugging options:

```
$ sudo nvram boot-args="slide=0 debug=0x100 keepsyms=1"
```

3. shut down the VM and add the following line to the .vmx file:

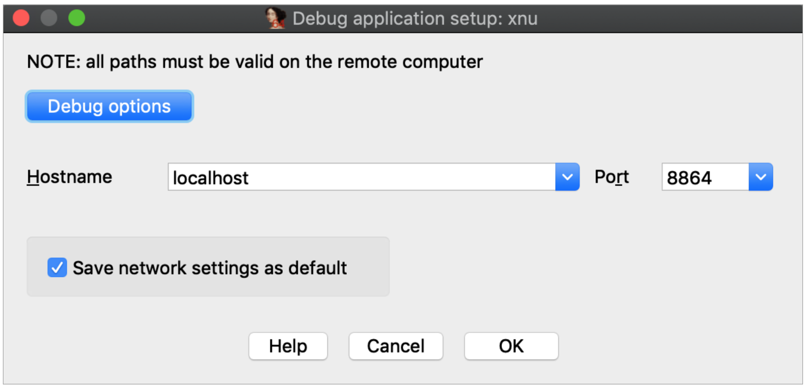
```
debugStub.listen.guest64 = "TRUE"
```

4. power on the virtual machine, open Terminal, and run this command:

```
$ uname -v
Darwin Kernel Version 17.7.0 ... root:xnu-4570.71.17~1/RELEASE_X86_64
```

Let's use IDA to modify this version string.

Launch IDA, and when prompted with the window **IDA: Quick start**, choose **Go** to start with an empty database. Then go to menu **Debugger>Attach>Remote XNU Debugger** and set the following options:

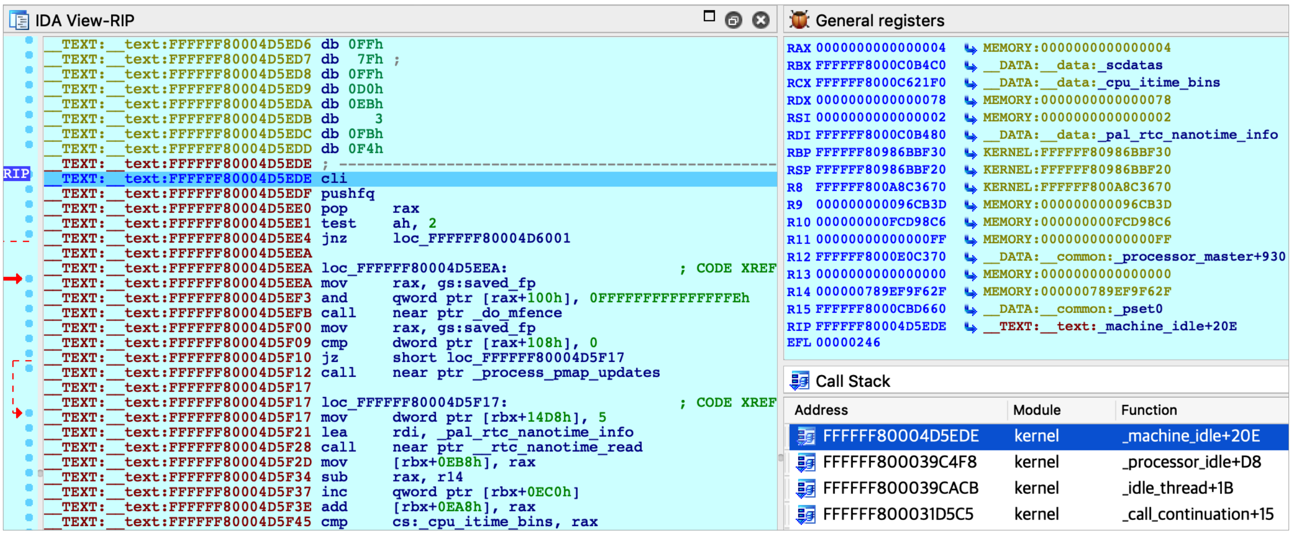


NOTE: all paths must be valid on the remote computer

Hostname: localhost
Port: 8864

☑ Save network settings as default

Click OK, then select **<attach to the process started on target>**, and wait for IDA to attach. This step might take a few seconds (later we'll discuss how to speed things up). Once attached, the target is usually suspended in machine_idle:



```
__TEXT:__text:FFFFFF80004D5ED6 db 0FFh
__TEXT:__text:FFFFFF80004D5ED7 db  7Fh ;
__TEXT:__text:FFFFFF80004D5ED8 db 0FFh
__TEXT:__text:FFFFFF80004D5ED9 db 0D0h
__TEXT:__text:FFFFFF80004D5EDA db 0EBh
__TEXT:__text:FFFFFF80004D5EDB db    3
__TEXT:__text:FFFFFF80004D5EDC db 0FBh
__TEXT:__text:FFFFFF80004D5EDD db 0F4h
__TEXT:__text:FFFFFF80004D5EDE ; ---------------------------------------------
__TEXT:__text:FFFFFF80004D5EDE cli
__TEXT:__text:FFFFFF80004D5EDF pushfq
__TEXT:__text:FFFFFF80004D5EE0 pop     rax
__TEXT:__text:FFFFFF80004D5EE1 test    ah, 2
__TEXT:__text:FFFFFF80004D5EE4 jnz     loc_FFFFFF80004D6001
__TEXT:__text:FFFFFF80004D5EEA
__TEXT:__text:FFFFFF80004D5EEA loc_FFFFFF80004D5EEA:                  ; CODE XREF
__TEXT:__text:FFFFFF80004D5EEA mov     rax, gs:saved_fp
__TEXT:__text:FFFFFF80004D5EF3 and     qword ptr [rax+100h], 0FFFFFFFFFFFFFFFEh
__TEXT:__text:FFFFFF80004D5EFB call    near ptr _do_mfence
__TEXT:__text:FFFFFF80004D5F00 mov     rax, gs:saved_fp
__TEXT:__text:FFFFFF80004D5F09 cmp     dword ptr [rax+108h], 0
__TEXT:__text:FFFFFF80004D5F10 jz      short loc_FFFFFF80004D5F17
__TEXT:__text:FFFFFF80004D5F12 call    near ptr _process_pmap_updates
__TEXT:__text:FFFFFF80004D5F17
__TEXT:__text:FFFFFF80004D5F17 loc_FFFFFF80004D5F17:                  ; CODE XREF
__TEXT:__text:FFFFFF80004D5F17 mov     dword ptr [rbx+14D8h], 5
__TEXT:__text:FFFFFF80004D5F21 lea     rdi, _pal_rtc_nanotime_info
__TEXT:__text:FFFFFF80004D5F28 call    near ptr __rtc_nanotime_read
__TEXT:__text:FFFFFF80004D5F2D mov     [rbx+0EB8h], rax
__TEXT:__text:FFFFFF80004D5F34 sub     rax, r14
__TEXT:__text:FFFFFF80004D5F37 inc     qword ptr [rbx+0EC0h]
__TEXT:__text:FFFFFF80004D5F3E add     [rbx+0EA8h], rax
__TEXT:__text:FFFFFF80004D5F45 cmp     cs:_cpu_itime_bins, rax
```

```
RAX 0000000000000004   MEMORY:0000000000000004
RBX FFFFFF8000C0B4C0   __DATA:__data:_scdatas
RCX FFFFFF8000C621F0   __DATA:__data:_cpu_itime_bins
RDX 0000000000000078   MEMORY:0000000000000078
RSI 0000000000000002   MEMORY:0000000000000002
RDI FFFFFF8000C0B480   __DATA:__data:_pal_rtc_nanotime_info
RBP FFFFFF80986BBF30   KERNEL:FFFFFF80986BBF30
RSP FFFFFF80986BBF20   KERNEL:FFFFFF80986BBF20
R8  FFFFFF800A8C3670   KERNEL:FFFFFF800A8C3670
R9  000000000096CB3D   MEMORY:000000000096CB3D
R10 00000000FCD98C6    MEMORY:00000000FCD98C6
R11 00000000000000FF   MEMORY:00000000000000FF
R12 FFFFFF8000E0C370   __DATA:__common:_processor_master+930
R13 0000000000000000   MEMORY:0000000000000000
R14 000000789EF9F62F   MEMORY:000000789EF9F62F
R15 FFFFFF8000CBD660   __DATA:__common:_pset0
RIP FFFFFF80004D5EDE   __TEXT:__text:_machine_idle+20E
EFL 00000246
```

Address	Module	Function
FFFFFF80004D5EDE	kernel	_machine_idle+20E
FFFFFF800039C4F8	kernel	_processor_idle+D8
FFFFFF800039CACB	kernel	_idle_thread+1B
FFFFFF800031D5C5	kernel	_call_continuation+15

IDA should have printed the message **FFFFFF8000200000: process kernel has started**, meaning it successfully detected the kernel image in memory. Now let's find the version string. Conveniently, the string appears in the kernel's symbol table, so we can simply use shortcut **G** and enter the name **_version** to jump right to it:

```
__TEXT:__const:FFFFFF8000AF6A00 _version db  44h ; D
__TEXT:__const:FFFFFF8000AF6A01 db  61h ; a
__TEXT:__const:FFFFFF8000AF6A02 db  72h ; r
__TEXT:__const:FFFFFF8000AF6A03 db  77h ; w
__TEXT:__const:FFFFFF8000AF6A04 db  69h ; i
__TEXT:__const:FFFFFF8000AF6A05 db  6Eh ; n
__TEXT:__const:FFFFFF8000AF6A06 db  20h
__TEXT:__const:FFFFFF8000AF6A07 db  4Bh ; K
__TEXT:__const:FFFFFF8000AF6A08 db  65h ; e
__TEXT:__const:FFFFFF8000AF6A09 db  72h ; r
__TEXT:__const:FFFFFF8000AF6A0A db  6Eh ; n
__TEXT:__const:FFFFFF8000AF6A0B db  65h ; e
__TEXT:__const:FFFFFF8000AF6A0C db  6Ch ; l
```

Use IDAPython to overwrite the bytes at this address:

```
idaapi.dbg_write_memory(0xFFFFFF8000AF6A00, "IDAPRO".encode('utf-8'))
```

Resume the process and allow the VM to run freely. Go back to Terminal in the VM and run the same command as before:

```
$ uname -v
IDAPRO Kernel Version 17.7.0 ... root:xnu-4570.71.17~1/RELEASE_X86_64
```

The output should look almost the same, except **Darwin** has been replaced with **IDAPRO**. So, we have modified kernel memory without breaking anything! You can continue to explore memory, set breakpoints, pause and resume the OS as you desire.

2.2. Using the KDK

If you have installed a Kernel Development Kit from Apple, you can set **KDK_PATH** in dbg_xnu.cfg to enable DWARF debugging:

```
KDK_PATH = "/Library/Developer/KDKs/KDK_10.13.6_17G4015.kdk";
```

Even if there is no KDK available for your OSX version, you can still utilise the KDK_PATH option in IDA to speed up debugging. For example, in the experiment above we could have done the following:

1. make your own KDK directory:

   ```
   $ mkdir ~/MyKDK
   ```

2. copy the kernelcache from your VM:

   ```
   $ scp user@vm:/System/Library/PrelinkedKernels/prelinkedkernel ~/MyKDK
   ```

3. decompress the kernelcache:

   ```
   $ kextcache -c ~/MyKDK/prelinkedkernel -uncompressed
   ```

4. set KDK_PATH in dbg_xnu.cfg:

   ```
   KDK_PATH = "~/MyKDK";
   ```

Now whenever IDA needs to extract information from the kernel or kexts, it will parse the kernelcache file on disk instead of parsing the images in memory. This should be noticeably faster.

2.3. Debugging a Development Kernel

Our next goal is to use the KDK to create a rich database that can be used to debug XNU in greater detail. In this example we will debug the development kernel included in the Apple KDK. Let's open this file in IDA:

```
$ export KDK=/Library/Developer/KDKs/KDK_10.13.6_17G4015.kdk
$ export KERNELS=$KDK/System/Library/Kernels
$ ida64 -okernel.i64 $KERNELS/kernel.development
```

Wait for IDA to load the DWARF info and complete the autoanalysis. This may take a few minutes, but we only need to do it once.

While we wait, we can prepare the virtual machine to use the development kernel instead of the release kernel that is shipped with OSX (Note: System Integrity Protection must now be disabled in the VM). Open Terminal in the VM and run the following commands:

1. copy the development kernel from the KDK:

```
$ sudo scp user@host:"\$KERNELS/kernel.development" /System/Library/Kernels/
```

2. reconstruct the kernelcache:

```
$ sudo kextcache -i /
```

3. reboot:

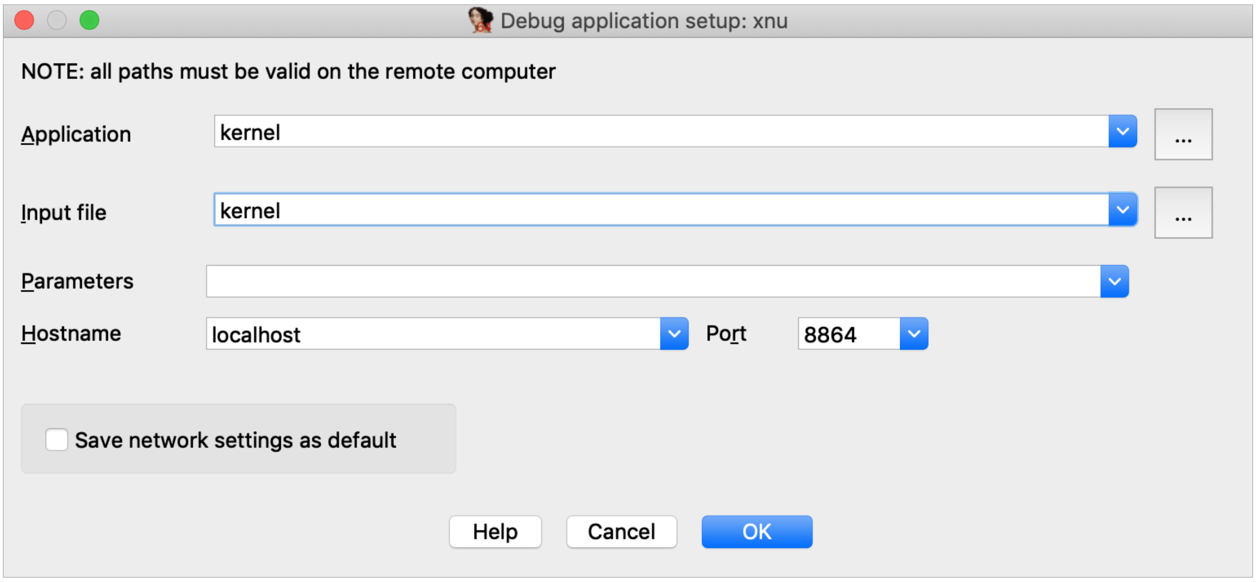
```
$ sudo shutdown -r now
```

4. after rebooting, check that the development kernel was properly installed:

```
$ uname -v
... root:xnu-4570.71.17~1/DEVELOPMENT_X86_64
```

The VM is now ready for debugging.

Return to IDA and use **Debugger>Select debugger** to select **Remote XNU Debugger**. Then open **Debugger>Process options** and set the following fields:



NOTE: all paths must be valid on the remote computer

Application: kernel

Input file: kernel

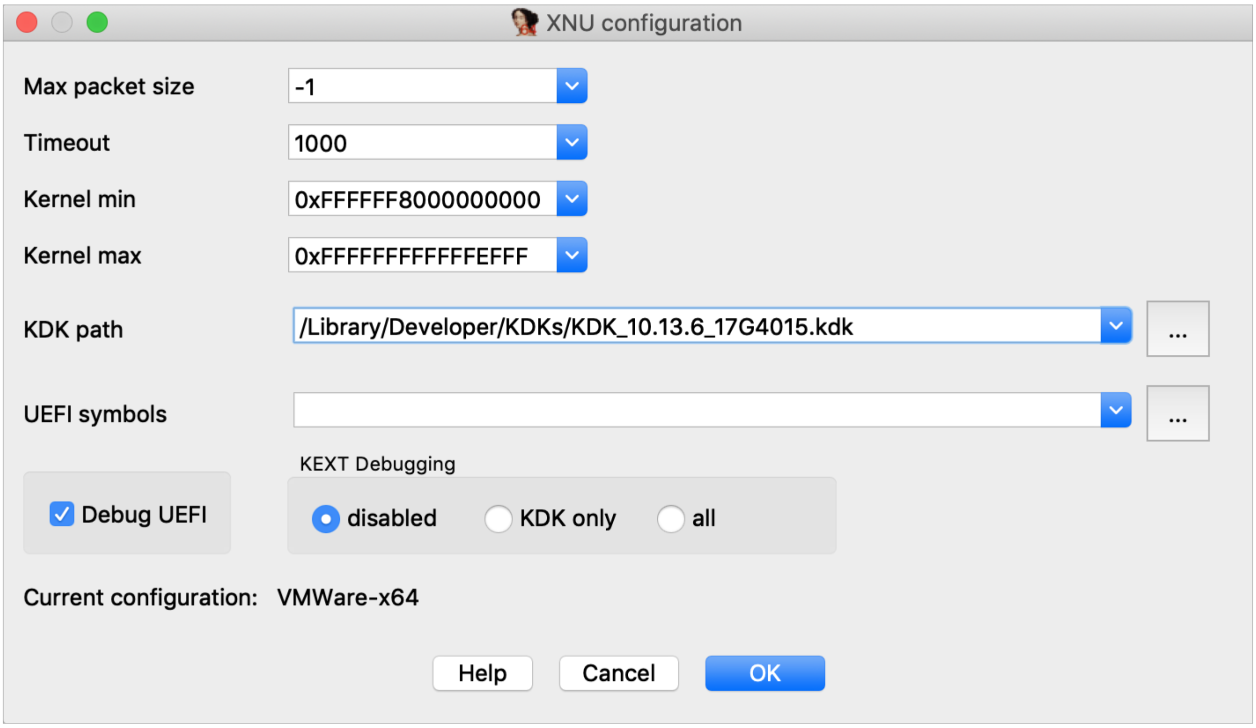
Parameters:

Hostname: localhost

Port: 8864

☐ Save network settings as default

Now go to **Debugger>Debugger options>Set specific options** and make sure the **KDK path** field is set:



Max packet size: -1

Timeout: 1000

Kernel min: 0xFFFFFF8000000000

Kernel max: 0xFFFFFFFFFFFFEFFF

KDK path: /Library/Developer/KDKs/KDK_10.13.6_17G4015.kdk

UEFI symbols:

☑ Debug UEFI

KEXT Debugging: ● disabled ○ KDK only ○ all

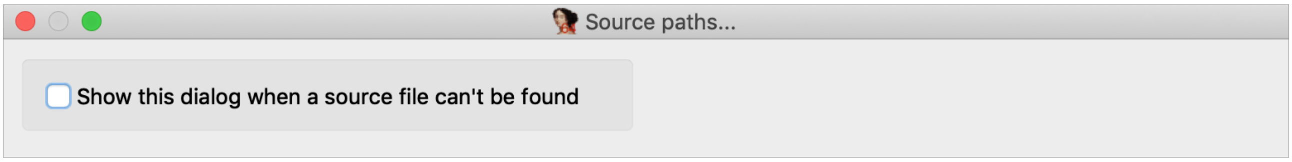
Current configuration: VMWare-x64

You can ignore the other options for now, and press OK.

2.4. Assembly-Level Debugging + DWARF

IDA supports source-level debugging for the XNU Kernel. However for demonstration purposes we will focus on assembly-level debugging, while taking advantage of source-level DWARF information like local variables. This is a bit more stable, and is still quite useful.

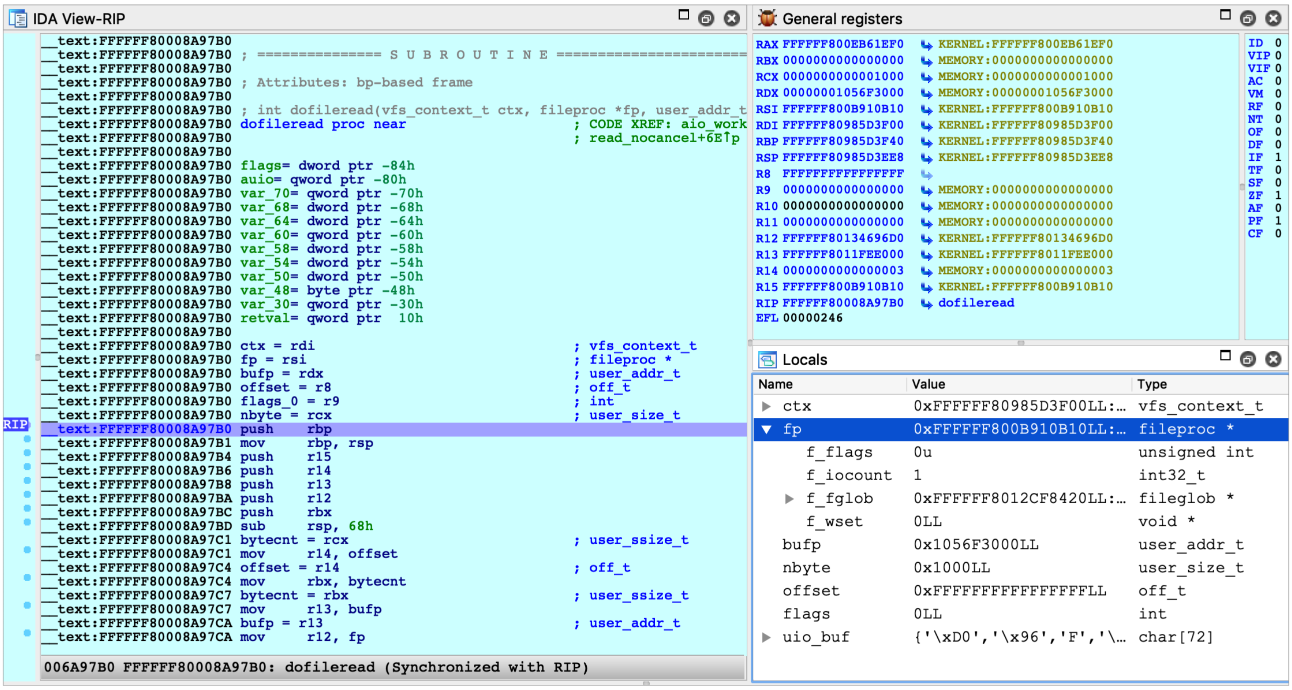
Before attaching the debugger, open **Options>Source paths...** and un-check the checkbox:



☐ Show this dialog when a source file can't be found

Then click **Apply**. This will prevent IDA from complaining when it can't find a source file.

Finally, select **Debugger>Attach to process>attach to the process started on target**. After attaching, jump to function **dofileread**, and use **F2** to set a breakpoint. Resume the debugger and and wait for the breakpoint to be hit (typically it will be hit right away, if not try simply running a terminal command in the guest). Once XNU hits our breakpoint, open **Debugger>Debugger windows>Locals:**



IDA View-RIP

```
__text:FFFFFF80008A97B0
__text:FFFFFF80008A97B0 ; =============== S U B R O U T I N E =======================
__text:FFFFFF80008A97B0
__text:FFFFFF80008A97B0 ; Attributes: bp-based frame
__text:FFFFFF80008A97B0
__text:FFFFFF80008A97B0 ; int dofileread(vfs_context_t ctx, fileproc *fp, user_addr_t
__text:FFFFFF80008A97B0 dofileread proc near                    ; CODE XREF: aio_work
__text:FFFFFF80008A97B0                                         ; read_nocancel+6E↑p
__text:FFFFFF80008A97B0
__text:FFFFFF80008A97B0 flags= dword ptr -84h
__text:FFFFFF80008A97B0 auio= qword ptr -80h
__text:FFFFFF80008A97B0 var_70= qword ptr -70h
__text:FFFFFF80008A97B0 var_68= dword ptr -68h
__text:FFFFFF80008A97B0 var_64= dword ptr -64h
__text:FFFFFF80008A97B0 var_60= qword ptr -60h
__text:FFFFFF80008A97B0 var_58= dword ptr -58h
__text:FFFFFF80008A97B0 var_54= dword ptr -54h
__text:FFFFFF80008A97B0 var_50= dword ptr -50h
__text:FFFFFF80008A97B0 var_48= byte ptr -48h
__text:FFFFFF80008A97B0 var_30= qword ptr -30h
__text:FFFFFF80008A97B0 retval= qword ptr  10h
__text:FFFFFF80008A97B0
__text:FFFFFF80008A97B0 ctx = rdi                               ; vfs_context_t
__text:FFFFFF80008A97B0 fp = rsi                                ; fileproc *
__text:FFFFFF80008A97B0 bufp = rdx                              ; user_addr_t
__text:FFFFFF80008A97B0 offset = r8                             ; off_t
__text:FFFFFF80008A97B0 flags_0 = r9                            ; int
__text:FFFFFF80008A97B0 nbyte = rcx                             ; user_size_t
__text:FFFFFF80008A97B0 push    rbp
__text:FFFFFF80008A97B1 mov     rbp, rsp
__text:FFFFFF80008A97B4 push    r15
__text:FFFFFF80008A97B6 push    r14
__text:FFFFFF80008A97B8 push    r13
__text:FFFFFF80008A97BA push    r12
__text:FFFFFF80008A97BC push    rbx
__text:FFFFFF80008A97BD sub     rsp, 68h
__text:FFFFFF80008A97C1 bytecnt = rcx                           ; user_ssize_t
__text:FFFFFF80008A97C1 mov     r14, offset
__text:FFFFFF80008A97C4 offset = r14                            ; off_t
__text:FFFFFF80008A97C4 mov     rbx, bytecnt
__text:FFFFFF80008A97C7 bytecnt = rbx                           ; user_ssize_t
__text:FFFFFF80008A97C7 mov     r13, bufp
__text:FFFFFF80008A97CA bufp = r13                              ; user_addr_t
__text:FFFFFF80008A97CA mov     r12, fp
006A97B0 FFFFFF80008A97B0: dofileread (Synchronized with RIP)
```

General registers

```
RAX FFFFFF800EB61EF0  ↳ KERNEL:FFFFFF800EB61EF0     ID  0
RBX 0000000000000000  ↳ MEMORY:0000000000000000     VIP 0
RCX 0000000000001000  ↳ MEMORY:0000000000001000     VIF 0
RDX 00000001056F3000  ↳ MEMORY:00000001056F3000     AC  0
RSI FFFFFF800B910B10  ↳ KERNEL:FFFFFF800B910B10     VM  0
RDI FFFFFF80985D3F00  ↳ KERNEL:FFFFFF80985D3F00     RF  0
RBP FFFFFF80985D3F40  ↳ KERNEL:FFFFFF80985D3F40     NT  0
RSP FFFFFF80985D3EE8  ↳ KERNEL:FFFFFF80985D3EE8     OF  0
R8  FFFFFFFFFFFFFFFF  ↳                             DF  0
R9  0000000000000000  ↳ MEMORY:0000000000000000     IF  1
R10 0000000000000000  ↳ MEMORY:0000000000000000     TF  0
R11 0000000000000000  ↳ MEMORY:0000000000000000     SF  0
R12 FFFFFF80134696D0  ↳ KERNEL:FFFFFF80134696D0     ZF  1
R13 FFFFFF8011FEE000  ↳ KERNEL:FFFFFF8011FEE000     AF  0
R14 0000000000000003  ↳ MEMORY:0000000000000003     PF  1
R15 FFFFFF800B910B10  ↳ KERNEL:FFFFFF800B910B10     CF  0
RIP FFFFFF80008A97B0  ↳ dofileread
EFL 00000246
```

Locals

```
Name            Value                       Type
▶ ctx           0xFFFFFF80985D3F00LL:…      vfs_context_t
▼ fp            0xFFFFFF800B910B10LL:…      fileproc *
    f_flags     0u                          unsigned int
    f_iocount   1                           int32_t
  ▶ f_fglob     0xFFFFFF8012CF8420LL:…      fileglob *
    f_wset      0LL                         void *
  bufp          0x1056F3000LL               user_addr_t
  nbyte         0x1000LL                    user_size_t
  offset        0xFFFFFFFFFFFFFFFFLL        off_t
  flags         0LL                         int
▶ uio_buf       {'\xD0','\x96','F','\…      char[72]
```

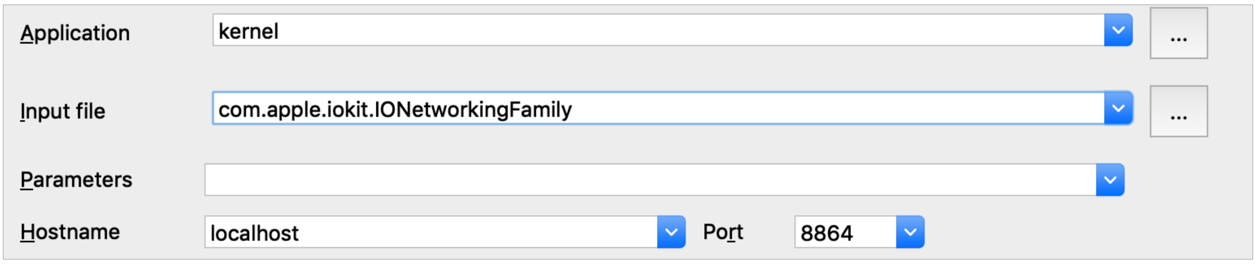
We can now perform detailed instruction-level debugging with the assistance of DWARF. You can continue to single step, set breakpoints, and inspect or modify local variables just like any other IDA debugger.

2.5. KEXT Debugging

IDA also supports debugging kext binaries. To demonstrate this, we will debug **IONetworkingFamily**, a submodule of IOKit that is typically shipped with the KDK. Begin by opening the binary in IDA:

```
$ export KEXTS=$KDK/System/Library/Extensions
$ ida64 -onet.i64 $KEXTS/IONetworkingFamily.kext/Contents/MacOS/IONetworkingFamily
```

Select **Remote XNU Debugger** from the debugger menu. Then in **Debugger>Process options**, set:



Application: kernel

Input file: com.apple.iokit.IONetworkingFamily

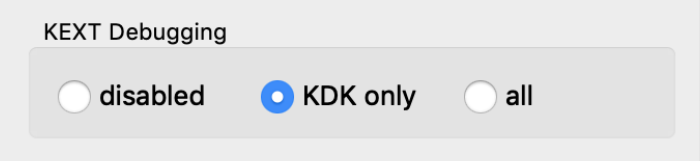
Parameters:

Hostname: localhost

Port: 8864

Note that we provide the bundle ID of the kext (com.apple.iokit.IONetworkingFamily) as the **Input file** field. This allows the debugger to easily identify the target kext at runtime.

Also note that loading all kexts in kernel memory can be a slow operation, which is why it is disabled by default. Open **Debugger>Debugger options>Set specific options** and ensure the **KDK path** field is set, then set the **KEXT Debugging** option to **KDK only**:

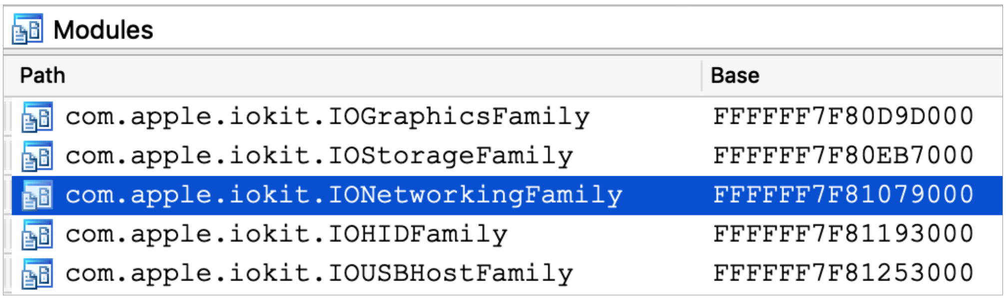


KEXT Debugging

○ disabled ● KDK only ○ all

This tells the debugger to only load kexts that are present in the KDK. Since the KDK binaries are on the local filesystem, IDA can parse the kexts in a negligible amount of time - which is ideal since we're really only interested in IONetworkingFamily.

Now power on your VM and allow it to boot up. Once it is running idle, attach the debugger. Immediately IDA should detect the kernel and all relevant kexts in memory, including IONetworkingFamily:



Modules

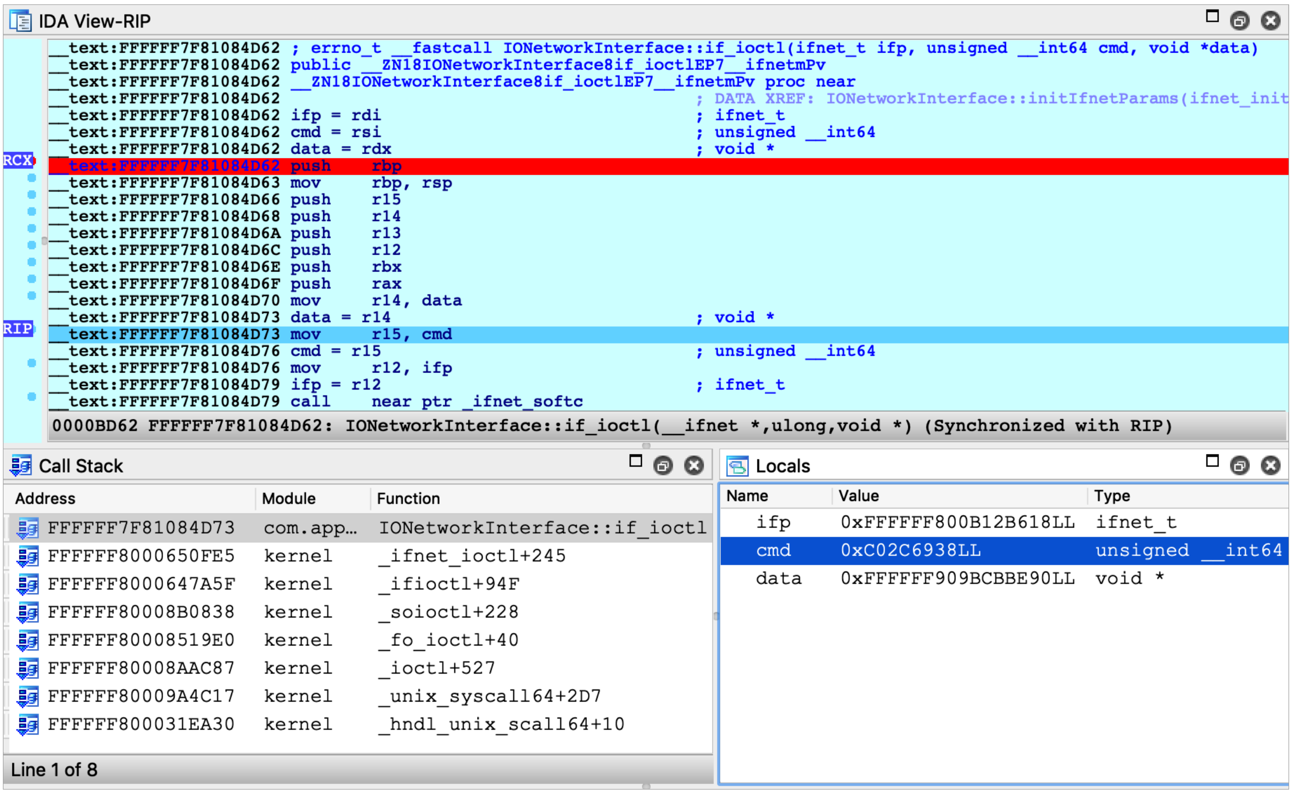
Path	Base
com.apple.iokit.IOGraphicsFamily	FFFFFF7F80D9D000
com.apple.iokit.IOStorageFamily	FFFFFF7F80EB7000
com.apple.iokit.IONetworkingFamily	FFFFFF7F81079000
com.apple.iokit.IOHIDFamily	FFFFFF7F81193000
com.apple.iokit.IOUSBHostFamily	FFFFFF7F81253000

Double-click to bring up the debug names for this module, and search for **IONetworkInterface::if_ioctl**:

Name	Address
IONetworkInterface::free(void)	FFFFFF7F81084900
IONetworkInterface::isPrimaryInterface(void)	FFFFFF7F81084AEC
IONetworkInterface::getController(void)	FFFFFF7F81084B44
IONetworkInterface::initIfnet(ifnet *)	FFFFFF7F81084B58
IONetworkInterface::initIfnetParams(ifnet_init_params *)	FFFFFF7F81084B60
IONetworkInterface::if_output(__ifnet *,__mbuf *)	FFFFFF7F81084BC4
IONetworkInterface::if_ioctl(__ifnet *,ulong,void *)	FFFFFF7F81084D62
IONetworkInterface::if_set_bpf_tap(__ifnet *,uint,int (*)(…	FFFFFF7F81084DE2
IONetworkInterface::if_detach(__ifnet *)	FFFFFF7F81084F50
IONetworkInterface::configureOutputStartDelay(ushort,ushor…	FFFFFF7F81084FA4
IONetworkInterface::lock(void)	FFFFFF7F81085006
IONetworkInterface::unlock(void)	FFFFFF7F8108501C

Line 514 of 1017

Now set a breakpoint at this function and resume the OS. Typically the breakpoint will be hit right away, but if it isn't try performing an action that requires a network interface (for instance, performing a google search). Once execution breaks in the kext we can use the database to debug it in detail:



```
__text:FFFFFF7F81084D62 ; errno_t __fastcall IONetworkInterface::if_ioctl(ifnet_t ifp, unsigned __int64 cmd, void *data)
__text:FFFFFF7F81084D62 public __ZN18IONetworkInterface8if_ioctlEP7__ifnetmPv
__text:FFFFFF7F81084D62 __ZN18IONetworkInterface8if_ioctlEP7__ifnetmPv proc near
__text:FFFFFF7F81084D62                                         ; DATA XREF: IONetworkInterface::initIfnetParams(ifnet_init
__text:FFFFFF7F81084D62 ifp = rdi                               ; ifnet_t
__text:FFFFFF7F81084D62 cmd = rsi                               ; unsigned __int64
__text:FFFFFF7F81084D62 data = rdx                              ; void *
__text:FFFFFF7F81084D62 push    rbp
__text:FFFFFF7F81084D63 mov     rbp, rsp
__text:FFFFFF7F81084D66 push    r15
__text:FFFFFF7F81084D68 push    r14
__text:FFFFFF7F81084D6A push    r13
__text:FFFFFF7F81084D6C push    r12
__text:FFFFFF7F81084D6E push    rbx
__text:FFFFFF7F81084D6F push    rax
__text:FFFFFF7F81084D70 mov     r14, data
__text:FFFFFF7F81084D73 data = r14                              ; void *
__text:FFFFFF7F81084D73 mov     r15, cmd
__text:FFFFFF7F81084D76 cmd = r15                               ; unsigned __int64
__text:FFFFFF7F81084D76 mov     r12, ifp
__text:FFFFFF7F81084D79 ifp = r12                               ; ifnet_t
__text:FFFFFF7F81084D79 call    near ptr _ifnet_softc
0000BD62 FFFFFF7F81084D62: IONetworkInterface::if_ioctl(__ifnet *,ulong,void *) (Synchronized with RIP)
```

Call Stack

Address	Module	Function
FFFFFF7F81084D73	com.app…	IONetworkInterface::if_ioctl
FFFFFF8000650FE5	kernel	_ifnet_ioctl+245
FFFFFF8000647A5F	kernel	_ifioctl+94F
FFFFFF80008B0838	kernel	_soioctl+228
FFFFFF80008519E0	kernel	_fo_ioctl+40
FFFFFF80008AAC87	kernel	_ioctl+527
FFFFFF80009A4C17	kernel	_unix_syscall64+2D7
FFFFFF800031EA30	kernel	_hndl_unix_scall64+10

Line 1 of 8

Locals

Name	Value	Type
ifp	0xFFFFFF800B12B618LL	ifnet_t
cmd	0xC02C6938LL	unsigned __int64
data	0xFFFFFF909BCBBE90LL	void *

2.6. Debugging a Prelinked Kernelcache

For simplicity, all of the examples up until now have dealt with a subset of the kernel, but it is also possible to load a complete prelinked kernelcache in IDA and debug it. Naturally, we have some suggestions for this.

2.6.1. Extending the KDK

If you're interested in debugging the entire prelinked kernel, the biggest concern is speed. IDA must create a detailed and accurate depiction of kernel memory, which could contain hundreds of kext modules. If we're not careful, this can be slow.

Fortunately there is an easy solution. Try the following:

1. create a writable copy of Apple's KDK:

```
$ cp -r /Library/Developer/KDKs/KDK_10.13.6_17G4015.kdk ~/MyKDK
```

2. copy the kernelcache from your VM to the new KDK:

```
$ scp user@vm:/System/Library/PrelinkedKernels/prelinkedkernel ~/MyKDK
```

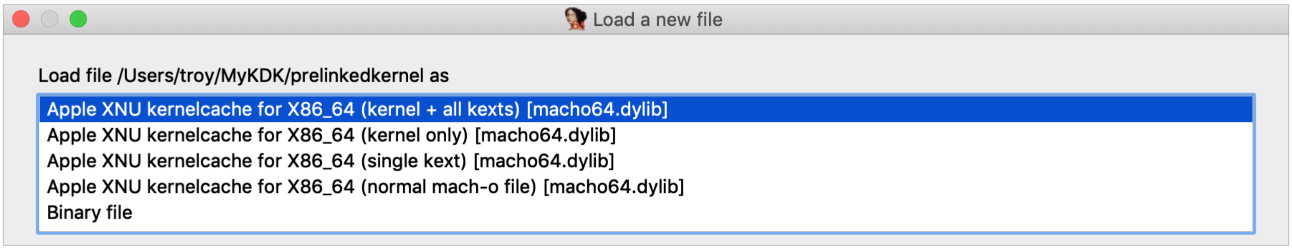
3. decompress the kernelcache:

```
$ kextcache -c ~/MyKDK/prelinkedkernel -uncompressed
```

Now IDA can use both the KDK and the kernelcache to extract debugging information for almost any kext at runtime. This should be fast.

2.6.2. Loading the Kernelcache

When loading a kernelcache, IDA now offers more load options:



Load file /Users/troy/MyKDK/prelinkedkernel as

Apple XNU kernelcache for X86_64 (kernel + all kexts) [macho64.dylib]
Apple XNU kernelcache for X86_64 (kernel only) [macho64.dylib]
Apple XNU kernelcache for X86_64 (single kext) [macho64.dylib]
Apple XNU kernelcache for X86_64 (normal mach-o file) [macho64.dylib]
Binary file

In this example we want to load everything, so choose the **kernel + all kexts** option and wait for IDA to load all the subfiles and finish the autoanalysis. This will take a while but there's no way around it, it's a lot of code.

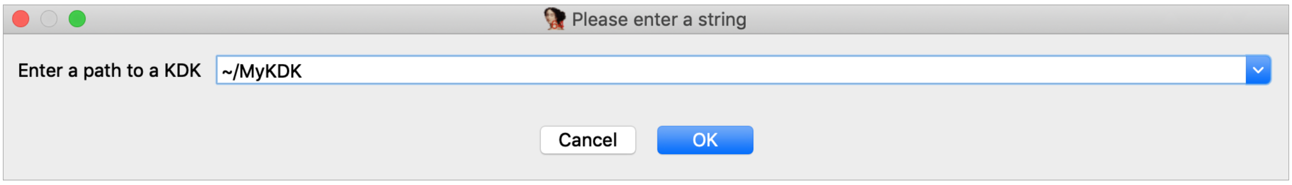
IMPORTANT NOTE: Try to avoid saving the IDA database file in the KDK directory. It is important to keep irrelevant files out of the KDK since they might slow down IDA's KDK parsing algorithm.

Now we might want to improve the static analysis by loading DWARF info from the KDK. In IDA 7.3 the dwarf plugin supports batch-loading all DWARF info from a KDK into a kernelcache database. Currently this feature must be invoked manually, so we have provided this script to make it easier.

Copy kdk_utils.py to the plugins directory of your IDA installation. This plugin will create a new menu **Edit>Other>KDK utils**, with two new menu actions:

- **Load KDK:** This action will automatically detect all matching DWARF files in a given KDK, then apply the DWARF info to the subfiles in the database (including the kernel itself).
- **Load DWARF for a prelinked KEXT:** This action is useful if you have DWARF info for a prelinked kext that is not included in Apple's KDK. For a given DWARF file, the action will find a matching kext in the database and apply the DWARF info to this subfile.

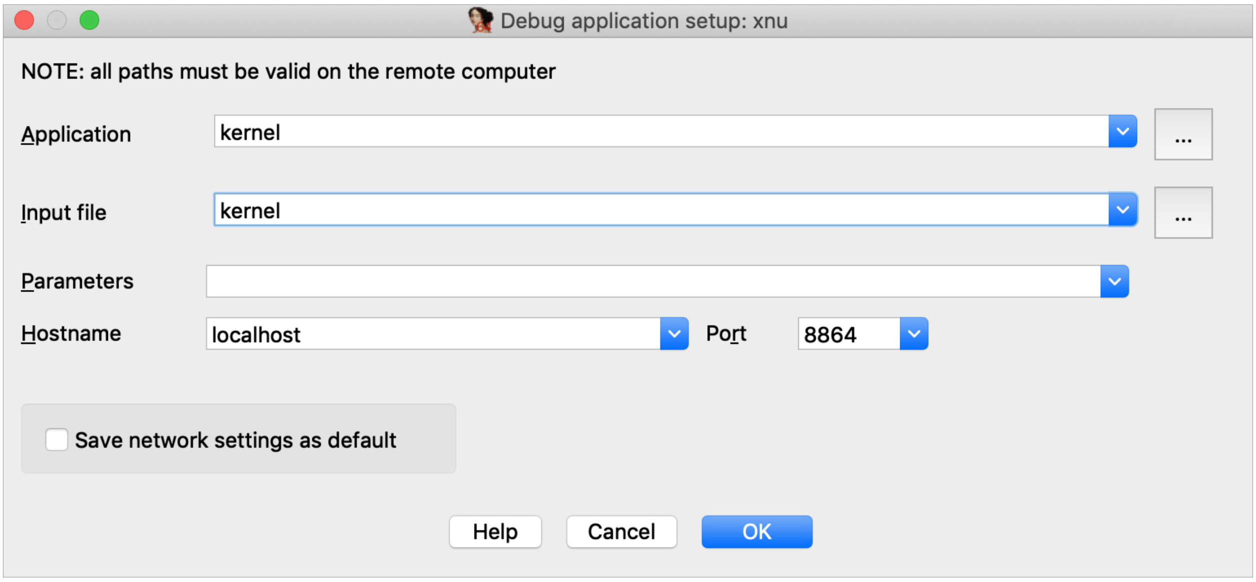
Try opening **Edit>Other>KDK utils>Load KDK** and provide the KDK path:


Please enter a string

Enter a path to a KDK: ~/MyKDK

Wait for IDA to scan the KDK for matching DWARF files and load them. This operation can also take a while, but it's worth it for all the extra structures, prototypes, and names that are added to the database. In the end we have a very detailed database that we are ready to use for debugging.

Now open **Debugger>Process options** and set the following options:


Debug application setup: xnu

NOTE: all paths must be valid on the remote computer

Application: kernel

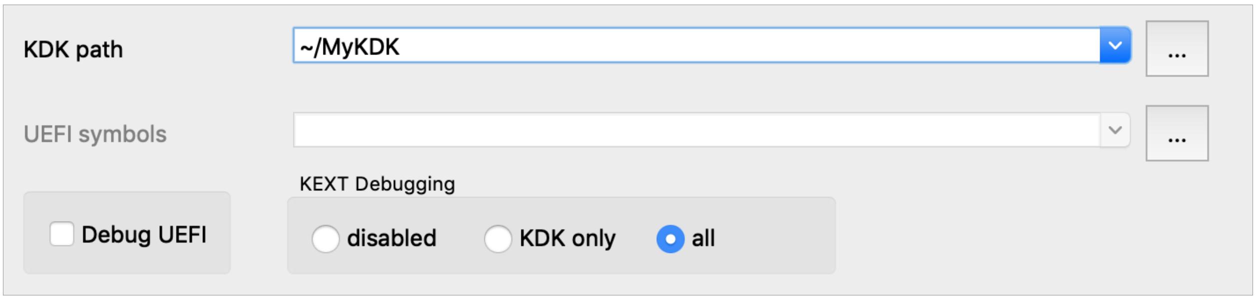
Input file: kernel

Parameters:

Hostname: localhost Port: 8864

☐ Save network settings as default

Then open **Debugger>Debugger options>Set specific options** and set the following fields:


KDK path: ~/MyKDK

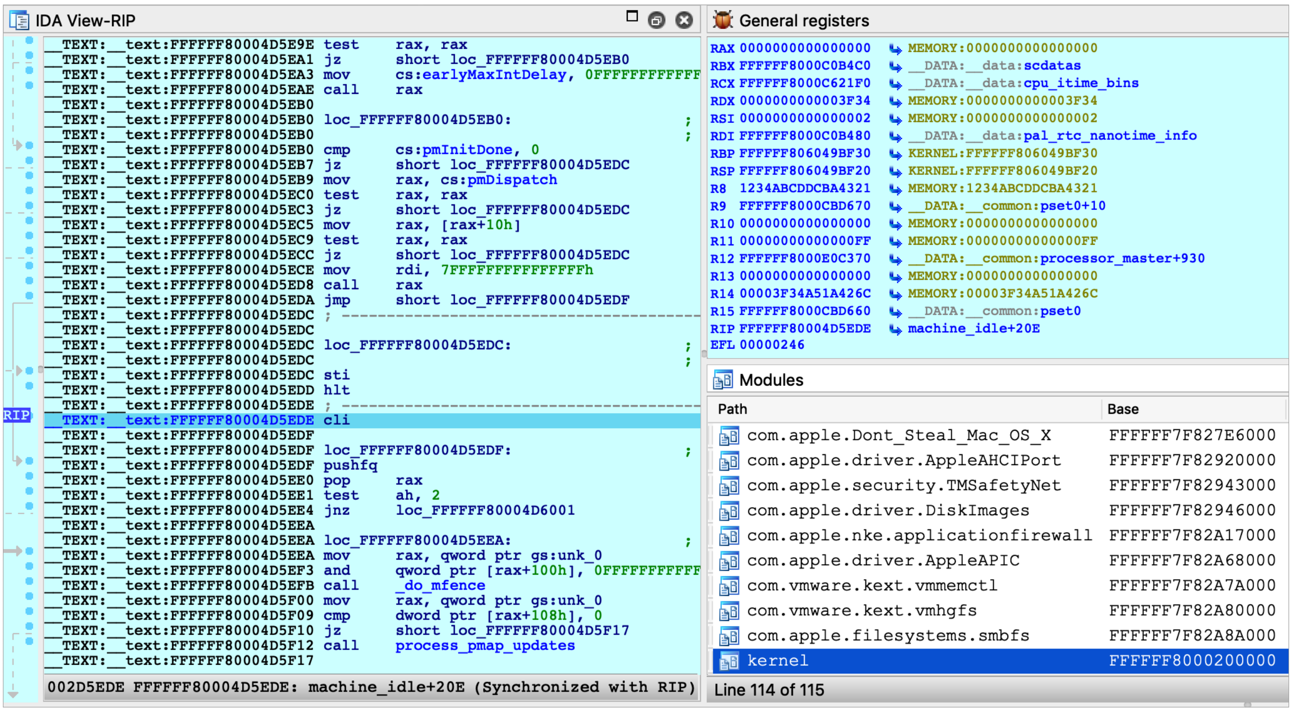
UEFI symbols:

☐ Debug UEFI

KEXT Debugging: ○ disabled ○ KDK only ● all

Note that we set the **KEXT Debugging** option to **all**. This tells the debugger to detect every kext that has been loaded into memory and add it to the Modules list, including any non-prelinked kexts (there are likely only a handful of them, so it doesn't hurt).

Finally, power on the VM and attach to it with **Debugger>Attach to process>attach to the process started on target**. IDA should be able to quickly generate modules for the kernel and all loaded kexts:



```
__TEXT:__text:FFFFFF80004D5E9E test    rax, rax
__TEXT:__text:FFFFFF80004D5EA1 jz      short loc_FFFFFF80004D5EB0
__TEXT:__text:FFFFFF80004D5EA3 mov     cs:earlyMaxIntDelay, 0FFFFFFFFFFFF
__TEXT:__text:FFFFFF80004D5EAE call    rax
__TEXT:__text:FFFFFF80004D5EB0
__TEXT:__text:FFFFFF80004D5EB0 loc_FFFFFF80004D5EB0:                      ;
__TEXT:__text:FFFFFF80004D5EB0                                            ;
__TEXT:__text:FFFFFF80004D5EB0 cmp     cs:pmInitDone, 0
__TEXT:__text:FFFFFF80004D5EB7 jz      short loc_FFFFFF80004D5EDC
__TEXT:__text:FFFFFF80004D5EB9 mov     rax, cs:pmDispatch
__TEXT:__text:FFFFFF80004D5EC0 test    rax, rax
__TEXT:__text:FFFFFF80004D5EC3 jz      short loc_FFFFFF80004D5EDC
__TEXT:__text:FFFFFF80004D5EC5 mov     rax, [rax+10h]
__TEXT:__text:FFFFFF80004D5EC9 test    rax, rax
__TEXT:__text:FFFFFF80004D5ECC jz      short loc_FFFFFF80004D5EDC
__TEXT:__text:FFFFFF80004D5ECE mov     rdi, 7FFFFFFFFFFFFFFFh
__TEXT:__text:FFFFFF80004D5ED8 call    rax
__TEXT:__text:FFFFFF80004D5EDA jmp     short loc_FFFFFF80004D5EDF
__TEXT:__text:FFFFFF80004D5EDC ; ------------------------------------------
__TEXT:__text:FFFFFF80004D5EDC
__TEXT:__text:FFFFFF80004D5EDC loc_FFFFFF80004D5EDC:                      ;
__TEXT:__text:FFFFFF80004D5EDC                                            ;
__TEXT:__text:FFFFFF80004D5EDC sti
__TEXT:__text:FFFFFF80004D5EDD hlt
__TEXT:__text:FFFFFF80004D5EDE ; ------------------------------------------
__TEXT:__text:FFFFFF80004D5EDE cli
__TEXT:__text:FFFFFF80004D5EDF
__TEXT:__text:FFFFFF80004D5EDF loc_FFFFFF80004D5EDF:                      ;
__TEXT:__text:FFFFFF80004D5EDF pushfq
__TEXT:__text:FFFFFF80004D5EE0 pop     rax
__TEXT:__text:FFFFFF80004D5EE1 test    ah, 2
__TEXT:__text:FFFFFF80004D5EE4 jnz     loc_FFFFFF80004D6001
__TEXT:__text:FFFFFF80004D5EEA
__TEXT:__text:FFFFFF80004D5EEA loc_FFFFFF80004D5EEA:                      ;
__TEXT:__text:FFFFFF80004D5EEA mov     rax, qword ptr gs:unk_0
__TEXT:__text:FFFFFF80004D5EF3 and     qword ptr [rax+100h], 0FFFFFFFFFFF
__TEXT:__text:FFFFFF80004D5EFB call    _do_mfence
__TEXT:__text:FFFFFF80004D5F00 mov     rax, qword ptr gs:unk_0
__TEXT:__text:FFFFFF80004D5F09 cmp     dword ptr [rax+108h], 0
__TEXT:__text:FFFFFF80004D5F10 jz      short loc_FFFFFF80004D5F17
__TEXT:__text:FFFFFF80004D5F12 call    process_pmap_updates
__TEXT:__text:FFFFFF80004D5F17
002D5EDE FFFFFF80004D5EDE: machine_idle+20E (Synchronized with RIP)
```

```
RAX 0000000000000000  MEMORY:0000000000000000
RBX FFFFFF8000C0B4C0  __DATA:__data:scdatas
RCX FFFFFF8000C621F0  __DATA:__data:cpu_itime_bins
RDX 0000000000003F34  MEMORY:0000000000003F34
RSI 0000000000000002  MEMORY:0000000000000002
RDI FFFFFF8000C0B480  __DATA:__data:pal_rtc_nanotime_info
RBP FFFFFF806049BF30  KERNEL:FFFFFF806049BF30
RSP FFFFFF806049BF20  KERNEL:FFFFFF806049BF20
R8  1234ABCDDCBA4321  MEMORY:1234ABCDDCBA4321
R9  FFFFFF8000CBD670  __DATA:__common:pset0+10
R10 0000000000000000  MEMORY:0000000000000000
R11 00000000000000FF  MEMORY:00000000000000FF
R12 FFFFFF8000E0C370  __DATA:__common:processor_master+930
R13 0000000000000000  MEMORY:0000000000000000
R14 00003F34A51A426C  MEMORY:00003F34A51A426C
R15 FFFFFF8000CBD660  __DATA:__common:pset0
RIP FFFFFF80004D5EDE  machine_idle+20E
EFL 00000246
```

Path	Base
com.apple.Dont_Steal_Mac_OS_X	FFFFFF7F827E6000
com.apple.driver.AppleAHCIPort	FFFFFF7F82920000
com.apple.security.TMSafetyNet	FFFFFF7F82943000
com.apple.driver.DiskImages	FFFFFF7F82946000
com.apple.nke.applicationfirewall	FFFFFF7F82A17000
com.apple.driver.AppleAPIC	FFFFFF7F82A68000
com.vmware.kext.vmmemctl	FFFFFF7F82A7A000
com.vmware.kext.vmhgfs	FFFFFF7F82A80000
com.apple.filesystems.smbfs	FFFFFF7F82A8A000
kernel	FFFFFF8000200000

Line 114 of 115

You are now free to explore the entire running kernel! Try performing any of the previous demos in this writeup. They should work about the same, but now they are all possible with one single database.

2.6.3. Kernel ASLR + Rebasing

It is worth noting that rebasing has been heavily improved in IDA 7.3. Even large databases like the one we just created can now be rebased in just a few seconds. Previous IDA versions would take quite a bit longer. Thus, IDA should be able to quickly handle kernel ASLR, even when working with prelinked kernelcaches.

2.7. Debugging the OSX Kernel Entry Point

In this example we demonstrate how to gain control of the OS as early as possible. This task requires very specific steps, and we document them here. Before we begin, we must make an important note about a limitation in VMware's GDB stub.

2.7.1. Limitation in VMware

Currently VMware's 64-bit GDB stub does not allow us to debug the kernel entry point in physical memory. According to VMware's support team, the correct approach is to use the 32-bit stub to debug the first few instructions of the kernel, then switch to a separate debugger connected to the 64-bit stub once the kernel switches to 64-bit addressing.

Since IDA's XNU debugger does not support 32-bit debugging, this approach is not really feasible (and it's not very practical anyway).

2.7.2. Workaround

Rather than add support for the 32-bit stub just to handle a few instructions, the official approach in IDA is to break at the first function executed in virtual memory (**i386_init**). This allows us to gain control of the OS while it is still in the early stages of initialization, which should be enough for most use cases.

Here's how you can do it:

1. Disable ALSR for the kernel. Open Terminal in the VM and run the following command:

   ```
   sudo nvram boot-args="slide=0"
   ```

   Then power off the VM.

2. Add this line to the .vmx file:

```
debugStub.hideBreakpoints = "TRUE"
```

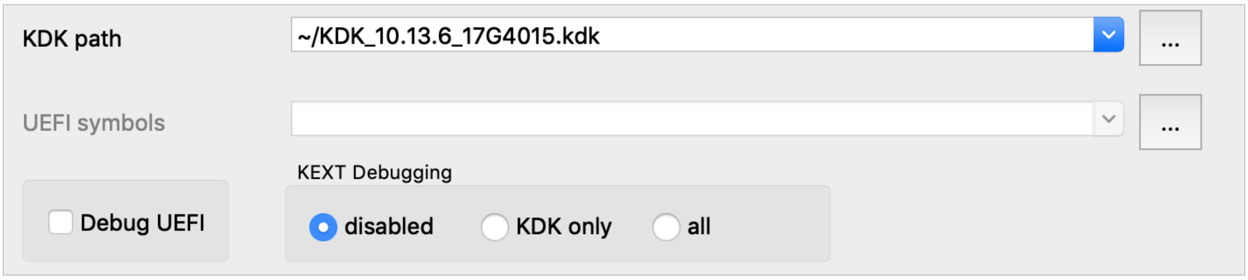
This ensures that hardware breakpoints are enabled in the GDB stub. For most versions of VMware, TRUE is the default value, but it's better to be safe.

3. Also add this line to the .vmx file:

```
monitor.debugOnStartGuest64 = "TRUE"
```

This will tell VMware to suspend the OS before it boots.

4. Power on the VM. It will remain suspended until we attach the debugger.
5. Load a kernel binary in IDA, and set the following XNU debugger options:



KDK path: ~/KDK_10.13.6_17G4015.kdk

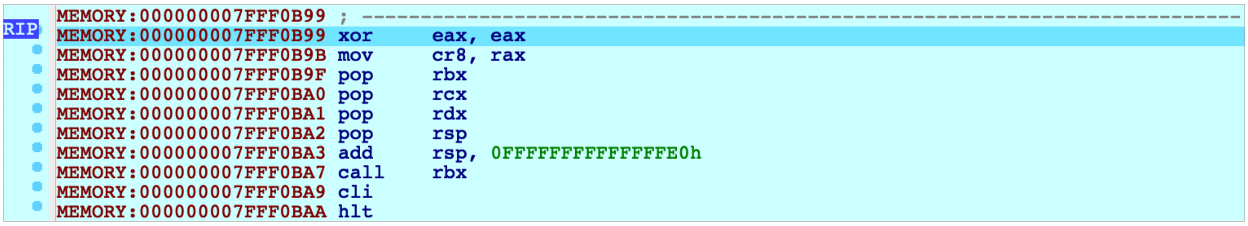
UEFI symbols:

☐ Debug UEFI

KEXT Debugging: ● disabled ○ KDK only ○ all

Note that we un-checked the **Debug UEFI** option. This option is explained in detail in the UEFI Debugging section, but for now just ensure it is disabled. This will prevent IDA from doing any unnecessary work.

6. Attach the debugger. The VM will be suspended in the firmware before the boot sequence has begun:



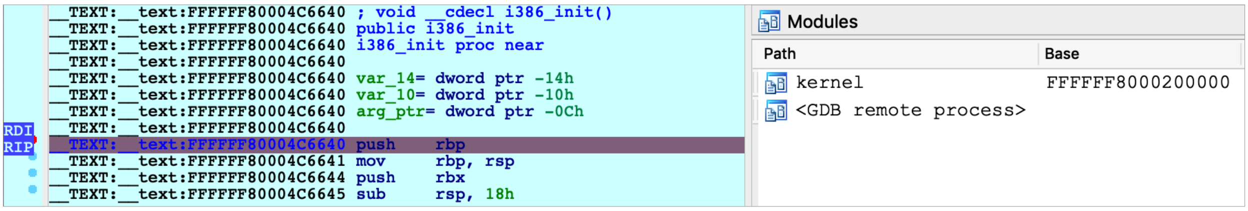
```
MEMORY:000000007FFF0B99 ; ---------------------------------------------------------------------------
MEMORY:000000007FFF0B99 xor     eax, eax
MEMORY:000000007FFF0B9B mov     cr8, rax
MEMORY:000000007FFF0B9F pop     rbx
MEMORY:000000007FFF0BA0 pop     rcx
MEMORY:000000007FFF0BA1 pop     rdx
MEMORY:000000007FFF0BA2 pop     rsp
MEMORY:000000007FFF0BA3 add     rsp, 0FFFFFFFFFFFFFFE0h
MEMORY:000000007FFF0BA7 call    rbx
MEMORY:000000007FFF0BA9 cli
MEMORY:000000007FFF0BAA hlt
```

7. Now jump to the function **_i386_init** and set a hardware breakpoint at this location:

```
idaapi.add_bpt(here(), 1, BPT_EXEC)
```

We must use a hardware breakpoint because the kernel has not been loaded and the address is not yet valid. This is why steps 1 and 2 were important. It ensures the stub can set a breakpoint at a deterministic address, without trying to write to memory.

8. Resume the OS, and wait for our breakpoint to be hit:



```
__TEXT:__text:FFFFFF80004C6640 ; void __cdecl i386_init()
__TEXT:__text:FFFFFF80004C6640 public i386_init
__TEXT:__text:FFFFFF80004C6640 i386_init proc near
__TEXT:__text:FFFFFF80004C6640
__TEXT:__text:FFFFFF80004C6640 var_14= dword ptr -14h
__TEXT:__text:FFFFFF80004C6640 var_10= dword ptr -10h
__TEXT:__text:FFFFFF80004C6640 arg_ptr= dword ptr -0Ch
__TEXT:__text:FFFFFF80004C6640
__TEXT:__text:FFFFFF80004C6640 push    rbp
__TEXT:__text:FFFFFF80004C6641 mov     rbp, rsp
__TEXT:__text:FFFFFF80004C6644 push    rbx
__TEXT:__text:FFFFFF80004C6645 sub     rsp, 18h
```

Modules

Path	Base
kernel	FFFFFF8000200000
<GDB remote process>	

IDA should detect that execution has reached the kernel and load the kernel module on-the-fly. You can now continue to debug the kernel normally.

3. macOS11 Kernel Debugging

VMware Fusion 12 now supports macOS11 Big Sur, including support for the built-in gdb server.

There have been a lot of changes to the XNU kernel architecture in macOS11, so it might be interesting to use IDA to discover these changes and discuss how they affect our analysis.

`NOTE: full support for macOS11 kernel debugging requires IDA 7.5 SP3`

3.1. KernelCollections

The most notable change in macOS11 is the kernelcache format. In previous macOS versions a kernelcache was just the mach kernel binary, prelinked with some extra segments containing embedded KEXTs.

In macOS11, Apple introduced an entirely new file format (MH_FILESET) specifically designed to be a generic container for the mach kernel and KEXTs. So a macOS11 kernelcache is a non-executable file that simply advertises a list of subfiles contained within it, via the LC_FILESET_ENTRY load commands.

Such files are now called "KernelCollections", and they are found in the /System/Library/KernelCollections/ directory of your macOS11 installation.

Let's try loading one of them in IDA:

```
$ ida64 -o/tmp/boot.i64 /System/Library/KernelCollections/BootKernelExtensions.kc
```

This cache contains the essential modules that macOS11 needs to boot, including the mach kernel itself (com.apple.kernel):

```
__TEXT:HEADER:FFFFFF80002085C8 ; LC_FILESET_ENTRY
__TEXT:HEADER:FFFFFF80002085C8                 fileset_entry_command <80000035h, 38h, 0FFFFFF8000210000h, 10000h, <\
__TEXT:HEADER:FFFFFF80002085C8                                        20h>, 0>
__TEXT:HEADER:FFFFFF80002085E8 aComAppleKernel db 'com.apple.kernel',0 ; entry id
__TEXT:HEADER:FFFFFF80002085F9                 align 20h
__TEXT:HEADER:FFFFFF8000208600 ; LC_FILESET_ENTRY
__TEXT:HEADER:FFFFFF8000208600                 fileset_entry_command <80000035h, 48h, 0FFFFFF8001268000h, 1108000h, <\
__TEXT:HEADER:FFFFFF8000208600                                        20h>, 0>
__TEXT:HEADER:FFFFFF8000208620 aComAppleNkeApp db 'com.apple.nke.applicationfirewall',0 ; entry id
__TEXT:HEADER:FFFFFF8000208642                 align 8
__TEXT:HEADER:FFFFFF8000208648 ; LC_FILESET_ENTRY
__TEXT:HEADER:FFFFFF8000208648                 fileset_entry_command <80000035h, 48h, 0FFFFFF8001272000h, 1112000h, <\
__TEXT:HEADER:FFFFFF8000208648                                        20h>, 0>
__TEXT:HEADER:FFFFFF8000208668 aComAppleDriver db 'com.apple.driver.AppleACPIPlatform',0 ; entry id
__TEXT:HEADER:FFFFFF800020868B                 align 10h
```

Also present is another KernelCollection that contains many important system KEXTs:

```
/System/Library/KernelCollections/SystemKernelExtensions.kc
```

And yet another one is used to manage third-party KEXTs, located at:

```
/Library/KernelCollections/AuxiliaryKernelExtensions.kc
```

For a detailed explanation of the various "flavors" of KernelCollections, see the doc for the **kmutil** command:

```
$ man kmutil
```

It seems that all of these KernelCollections will somehow be utilized by a running instance of macOS11. Is this obvious when debugging the kernel in IDA? Let's try it out.

3.2. Creating a macOS11 VM

To start debugging macOS11, we'll need to set up a macOS11 virtual machine with System Integrity Protection disabled and kernel debugging options enabled:

```
$ sudo nvram boot-args="slide=0 debug=0x100 keepsyms=1"
```

To do this you will need to boot the VM in recovery mode. As of this writing, macOS11 is still in beta and booting a macOS11 VM in recovery mode is very unstable. According to this thread VMware is working on fixing it, but for now you will likely have to research the latest workaround (it changes almost every beta version).

macOS11 has also broken the debug registers, which VMware Fusion uses to set breakpoints in the guest OS. To work around this you must enable software breakpoints in the vmx file:

```
debugStub.hideBreakpoints = "FALSE"
```

According to this, VMware is aware of the issue and they are working resolve it. The workaround will do for now.

Also don't forget to enable the gdb stub in the vmx file:

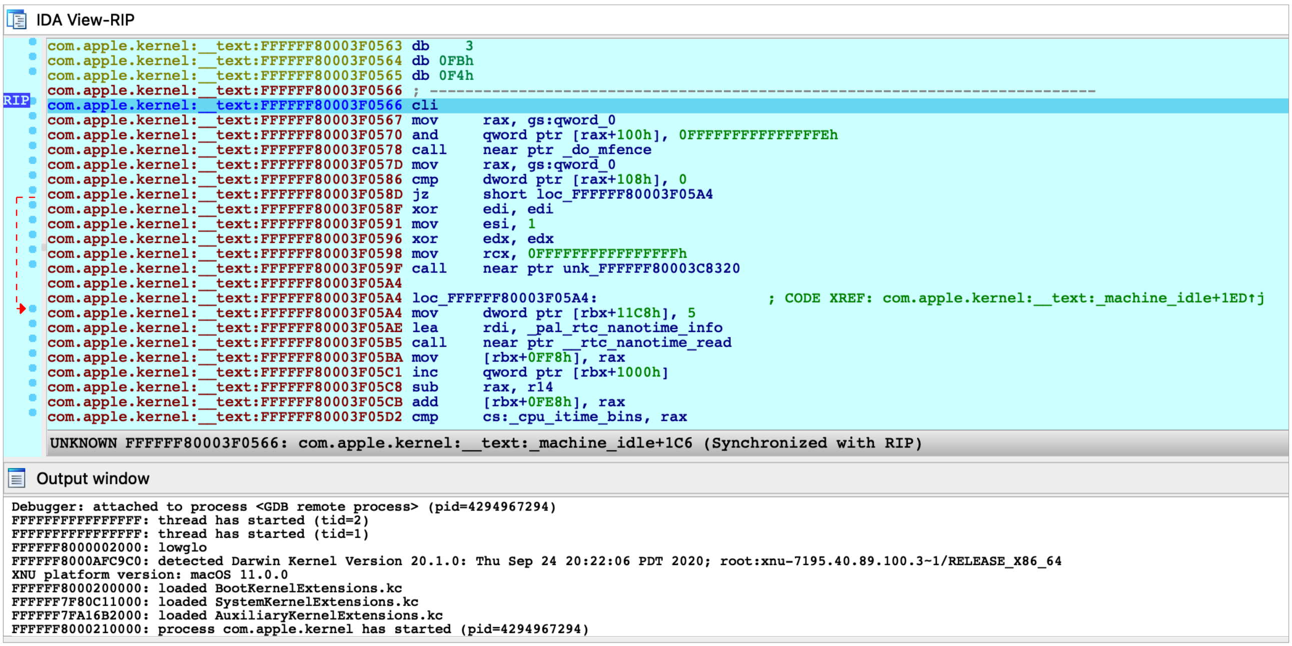
```
debugStub.listen.guest64 = "TRUE"
debugStub.port.guest64 = "8864"
```

3.3. Debugging: Quick Start

Power on the macOS11 VM and launch IDA with an empty database:

```
$ ida64 -t
```

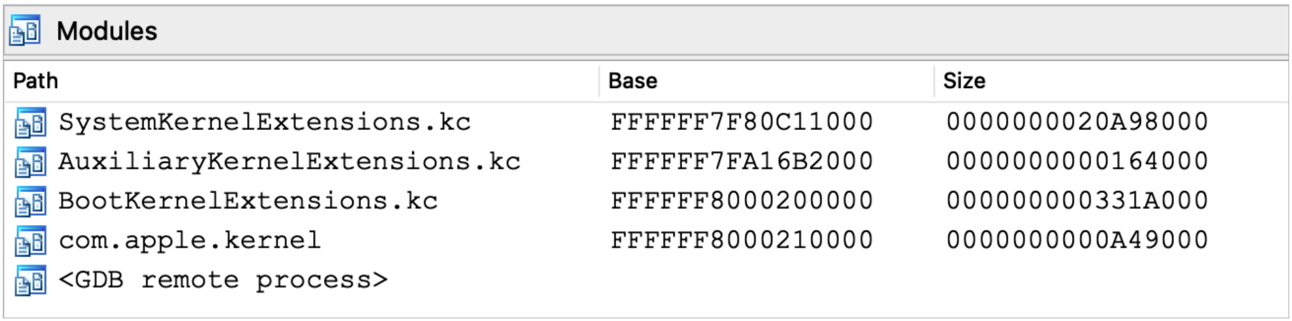
Use menu **Debugger>Select debugger** to select the **Remote XNU Debugger**, then use **Debugger>Process options** to set the **Hostname** and **Port** fields to localhost:8864, and finally **Debugger>Attach to process** to attach to the running VM:



```
com.apple.kernel:__text:FFFFFF80003F0563 db    3
com.apple.kernel:__text:FFFFFF80003F0564 db 0FBh
com.apple.kernel:__text:FFFFFF80003F0565 db 0F4h
com.apple.kernel:__text:FFFFFF80003F0566 ; ---------------------------------------------------------------------------
com.apple.kernel:__text:FFFFFF80003F0566 cli
com.apple.kernel:__text:FFFFFF80003F0567 mov     rax, gs:qword_0
com.apple.kernel:__text:FFFFFF80003F0570 and     qword ptr [rax+100h], 0FFFFFFFFFFFFFFEh
com.apple.kernel:__text:FFFFFF80003F0578 call    near ptr _do_mfence
com.apple.kernel:__text:FFFFFF80003F057D mov     rax, gs:qword_0
com.apple.kernel:__text:FFFFFF80003F0586 cmp     dword ptr [rax+108h], 0
com.apple.kernel:__text:FFFFFF80003F058D jz      short loc_FFFFFF80003F05A4
com.apple.kernel:__text:FFFFFF80003F058F xor     edi, edi
com.apple.kernel:__text:FFFFFF80003F0591 mov     esi, 1
com.apple.kernel:__text:FFFFFF80003F0596 xor     edx, edx
com.apple.kernel:__text:FFFFFF80003F0598 mov     rcx, 0FFFFFFFFFFFFFFFFh
com.apple.kernel:__text:FFFFFF80003F059F call    near ptr unk_FFFFFF80003C8320
com.apple.kernel:__text:FFFFFF80003F05A4
com.apple.kernel:__text:FFFFFF80003F05A4 loc_FFFFFF80003F05A4:                   ; CODE XREF: com.apple.kernel:__text:_machine_idle+1ED↑j
com.apple.kernel:__text:FFFFFF80003F05A4 mov     dword ptr [rbx+11C8h], 5
com.apple.kernel:__text:FFFFFF80003F05AE lea     rdi, _pal_rtc_nanotime_info
com.apple.kernel:__text:FFFFFF80003F05B5 call    near ptr __rtc_nanotime_read
com.apple.kernel:__text:FFFFFF80003F05BA mov     [rbx+0FF8h], rax
com.apple.kernel:__text:FFFFFF80003F05C1 inc     qword ptr [rbx+1000h]
com.apple.kernel:__text:FFFFFF80003F05C8 sub     rax, r14
com.apple.kernel:__text:FFFFFF80003F05CB add     [rbx+0FE8h], rax
com.apple.kernel:__text:FFFFFF80003F05D2 cmp     cs:_cpu_itime_bins, rax
UNKNOWN FFFFFF80003F0566: com.apple.kernel:__text:_machine_idle+1C6 (Synchronized with RIP)
```

```
Debugger: attached to process <GDB remote process> (pid=4294967294)
FFFFFFFFFFFFFFFF: thread has started (tid=2)
FFFFFFFFFFFFFFFF: thread has started (tid=1)
FFFFFF8000002000: lowglo
FFFFFF8000AFC9C0: detected Darwin Kernel Version 20.1.0: Thu Sep 24 20:22:06 PDT 2020; root:xnu-7195.40.89.100.3~1/RELEASE_X86_64
XNU platform version: macOS 11.0.0
FFFFFF8000200000: loaded BootKernelExtensions.kc
FFFFFF7F80C11000: loaded SystemKernelExtensions.kc
FFFFFF7FA16B2000: loaded AuxiliaryKernelExtensions.kc
FFFFFF8000210000: process com.apple.kernel has started (pid=4294967294)
```

From the Modules list we see that IDA was able to detect the kernel image, as well as all KernelCollections that have been loaded:

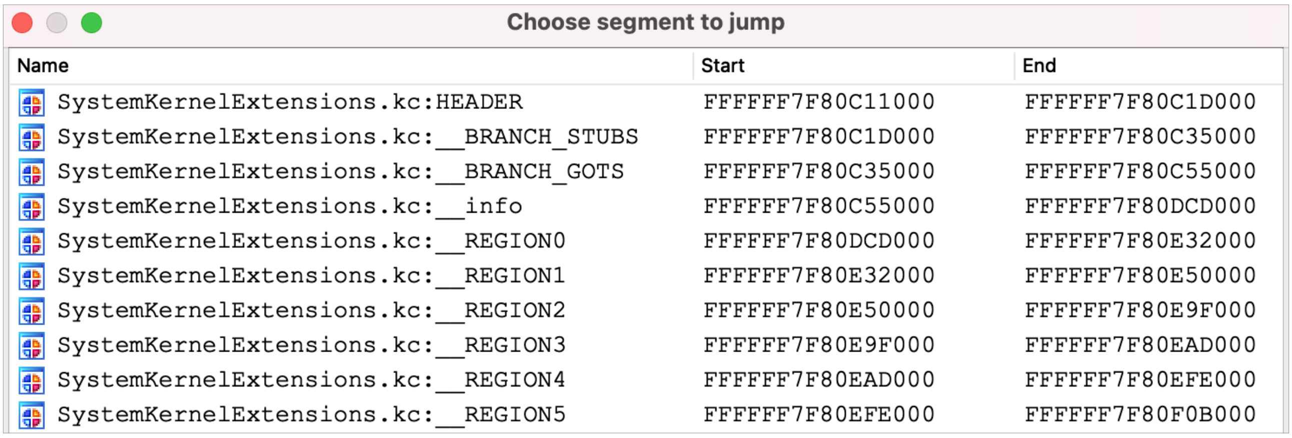


Path	Base	Size
SystemKernelExtensions.kc	FFFFFF7F80C11000	0000000020A98000
AuxiliaryKernelExtensions.kc	FFFFFF7FA16B2000	0000000000164000
BootKernelExtensions.kc	FFFFFF8000200000	000000000331A000
com.apple.kernel	FFFFFF8000210000	0000000000A49000
<GDB remote process>		

Note that although **com.apple.kernel** is a subset of **BootKernelExtensions.kc**, IDA still created a separate module for it.

Internally IDA expects the debugger engine to identify the executable module for any running process. The **com.apple.kernel** subfile is a logical choice, since it is the only Mach-O image in memory with file type MH_EXECUTE.

Now let's consider the **SystemKernelExtensions.kc** module. Use **Ctrl+S** to examine the debugger segments:



Choose segment to jump

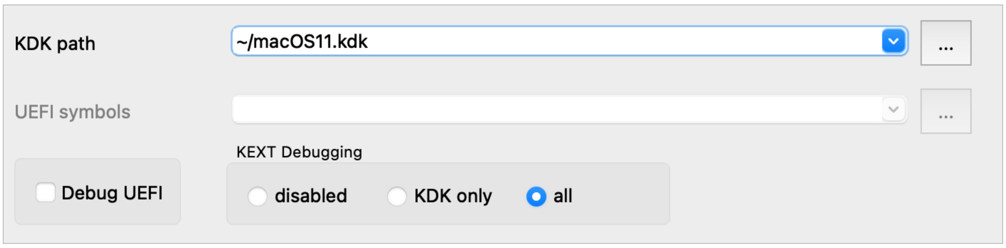
Name	Start	End
SystemKernelExtensions.kc:HEADER	FFFFFF7F80C11000	FFFFFF7F80C1D000
SystemKernelExtensions.kc:__BRANCH_STUBS	FFFFFF7F80C1D000	FFFFFF7F80C35000
SystemKernelExtensions.kc:__BRANCH_GOTS	FFFFFF7F80C35000	FFFFFF7F80C55000
SystemKernelExtensions.kc:__info	FFFFFF7F80C55000	FFFFFF7F80DCD000
SystemKernelExtensions.kc:__REGION0	FFFFFF7F80DCD000	FFFFFF7F80E32000
SystemKernelExtensions.kc:__REGION1	FFFFFF7F80E32000	FFFFFF7F80E50000
SystemKernelExtensions.kc:__REGION2	FFFFFF7F80E50000	FFFFFF7F80E9F000
SystemKernelExtensions.kc:__REGION3	FFFFFF7F80E9F000	FFFFFF7F80EAD000
SystemKernelExtensions.kc:__REGION4	FFFFFF7F80EAD000	FFFFFF7F80EFE000
SystemKernelExtensions.kc:__REGION5	FFFFFF7F80EFE000	FFFFFF7F80F0B000

The __REGION* segments contain the KEXT subfiles. By default the debugger does not recurse into KernelCollections to detect the loaded KEXTs, because it has the potential to be very slow. All KEXTs in the KernelCollection will share a common symbol table, which can be massive and takes long time to read from debugger memory.

To avoid slowdowns, create a rudimentary KDK for macOS11 by copying the KernelCollection files from the VM:

```
$ mkdir ~/macOS11.kdk
$ cd ~/macOS11.kdk
$ scp user@vm:/System/Library/KernelCollections/BootKernelExtensions.kc .
$ scp user@vm:/System/Library/KernelCollections/SystemKernelExtensions.kc .
$ scp user@vm:/Library/KernelCollections/AuxiliaryKernelExtensions.kc .
```

Detach from the VM with **Debugger>Detach from process**, then use **Debugger>Debugger options>Set specific options** to set the following fields:



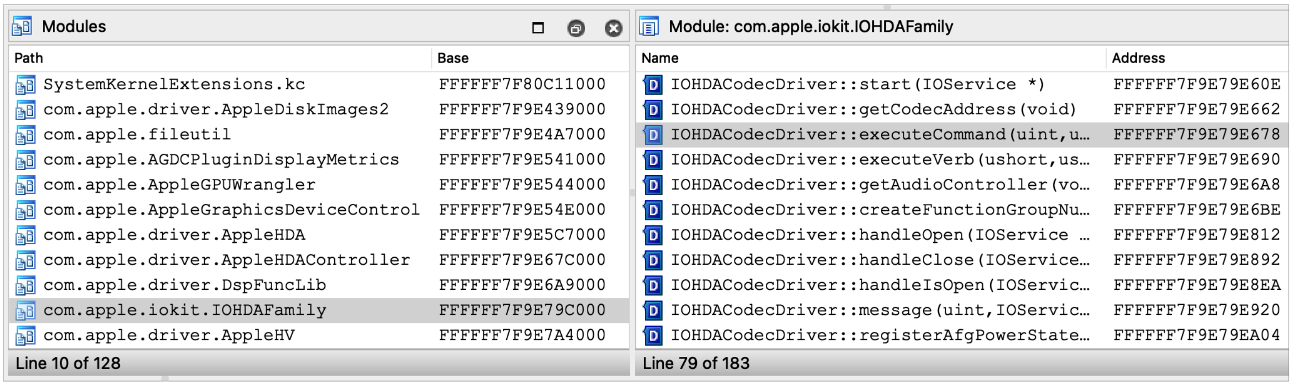
KDK path: ~/macOS11.kdk

UEFI symbols:

☐ Debug UEFI

KEXT Debugging: ○ disabled ○ KDK only ● all

Now reattach to the VM. Since we enabled KEXT debugging and specified a KDK, the debugger will quickly create a debugging environment that can be explored in greater detail:



Modules

Path	Base
SystemKernelExtensions.kc	FFFFFF7F80C11000
com.apple.driver.AppleDiskImages2	FFFFFF7F9E439000
com.apple.fileutil	FFFFFF7F9E4A7000
com.apple.AGDCPluginDisplayMetrics	FFFFFF7F9E541000
com.apple.AppleGPUWrangler	FFFFFF7F9E544000
com.apple.AppleGraphicsDeviceControl	FFFFFF7F9E54E000
com.apple.driver.AppleHDA	FFFFFF7F9E5C7000
com.apple.driver.AppleHDAController	FFFFFF7F9E67C000
com.apple.driver.DspFuncLib	FFFFFF7F9E6A9000
com.apple.iokit.IOHDAFamily	FFFFFF7F9E79C000
com.apple.driver.AppleHV	FFFFFF7F9E7A4000

Line 10 of 128

Module: com.apple.iokit.IOHDAFamily

Name	Address
IOHDACodecDriver::start(IOService *)	FFFFFF7F9E79E60E
IOHDACodecDriver::getCodecAddress(void)	FFFFFF7F9E79E662
IOHDACodecDriver::executeCommand(uint,u…	FFFFFF7F9E79E678
IOHDACodecDriver::executeVerb(ushort,us…	FFFFFF7F9E79E690
IOHDACodecDriver::getAudioController(vo…	FFFFFF7F9E79E6A8
IOHDACodecDriver::createFunctionGroupNu…	FFFFFF7F9E79E6BE
IOHDACodecDriver::handleOpen(IOService …	FFFFFF7F9E79E812
IOHDACodecDriver::handleClose(IOService…	FFFFFF7F9E79E892
IOHDACodecDriver::handleIsOpen(IOServic…	FFFFFF7F9E79E8EA
IOHDACodecDriver::message(uint,IOServic…	FFFFFF7F9E79E920
IOHDACodecDriver::registerAfgPowerState…	FFFFFF7F9E79EA04

Line 79 of 183

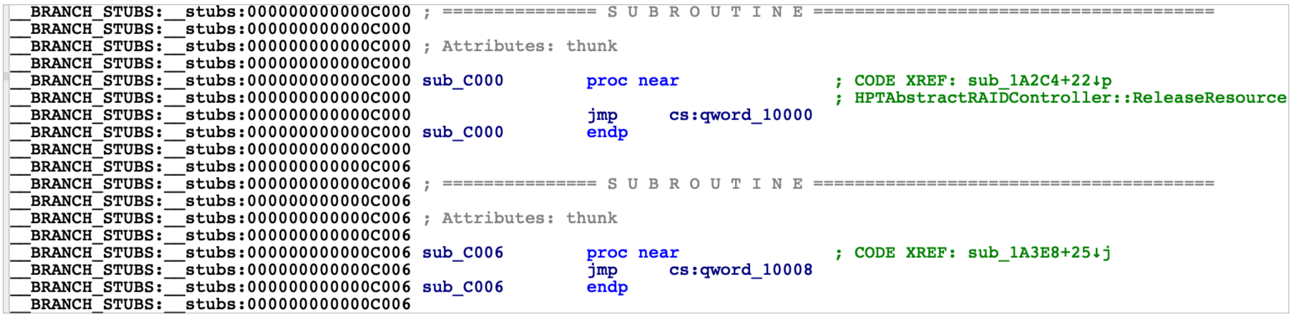
3.4. Symbolicating KernelCollections

Just like with previous macOS versions, IDA allows you to load a complete macOS11 kernelcache and debug it as a single input file. This can be done for any of the various KernelCollection types.

For example, let’s try loading the "Auxiliary" KernelCollection in IDA:

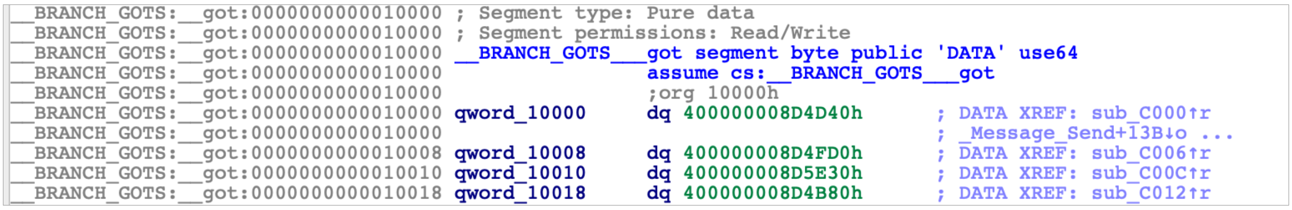
```
$ ida64 -o/tmp/aux.i64 ~/macOS11.kdk/AuxiliaryKernelExtensions.kc
```

This cache contains various third-party KEXTs, many of which depend on functionality in the kernel. How do these KEXTs manage to invoke kernel functions? You may have noticed in previous versions of IDA that AuxiliaryKernelExtensions.kc contains stub functions in the __BRANCH_STUBS segment:



```
__BRANCH_STUBS:__stubs:000000000000C000 ; =============== S U B R O U T I N E =======================================
__BRANCH_STUBS:__stubs:000000000000C000
__BRANCH_STUBS:__stubs:000000000000C000 ; Attributes: thunk
__BRANCH_STUBS:__stubs:000000000000C000
__BRANCH_STUBS:__stubs:000000000000C000 sub_C000        proc near               ; CODE XREF: sub_1A2C4+22↓p
__BRANCH_STUBS:__stubs:000000000000C000                                         ; HPTAbstractRAIDController::ReleaseResource
__BRANCH_STUBS:__stubs:000000000000C000                 jmp     cs:qword_10000
__BRANCH_STUBS:__stubs:000000000000C000 sub_C000        endp
__BRANCH_STUBS:__stubs:000000000000C000
__BRANCH_STUBS:__stubs:000000000000C006
__BRANCH_STUBS:__stubs:000000000000C006 ; =============== S U B R O U T I N E =======================================
__BRANCH_STUBS:__stubs:000000000000C006
__BRANCH_STUBS:__stubs:000000000000C006 ; Attributes: thunk
__BRANCH_STUBS:__stubs:000000000000C006
__BRANCH_STUBS:__stubs:000000000000C006 sub_C006        proc near               ; CODE XREF: sub_1A3E8+25↓j
__BRANCH_STUBS:__stubs:000000000000C006                 jmp     cs:qword_10008
__BRANCH_STUBS:__stubs:000000000000C006 sub_C006        endp
__BRANCH_STUBS:__stubs:000000000000C006
```

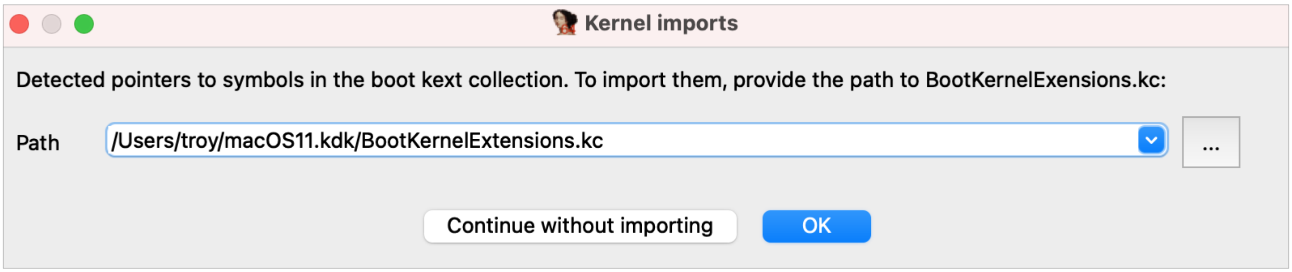
The stubs read function pointers from a global offset table:



```
__BRANCH_GOTS:__got:0000000000010000 ; Segment type: Pure data
__BRANCH_GOTS:__got:0000000000010000 ; Segment permissions: Read/Write
__BRANCH_GOTS:__got:0000000000010000 __BRANCH_GOTS___got segment byte public 'DATA' use64
__BRANCH_GOTS:__got:0000000000010000                 assume cs:__BRANCH_GOTS___got
__BRANCH_GOTS:__got:0000000000010000                 ;org 10000h
__BRANCH_GOTS:__got:0000000000010000 qword_10000     dq 400000008D4D40h      ; DATA XREF: sub_C000↑r
__BRANCH_GOTS:__got:0000000000010000                                         ; _Message_Send+13B↓o ...
__BRANCH_GOTS:__got:0000000000010008 qword_10008     dq 400000008D4FD0h      ; DATA XREF: sub_C006↑r
__BRANCH_GOTS:__got:0000000000010010 qword_10010     dq 400000008D5E30h      ; DATA XREF: sub_C00C↑r
__BRANCH_GOTS:__got:0000000000010018 qword_10018     dq 400000008D4B80h      ; DATA XREF: sub_C012↑r
```

The values in the table (e.g. 400000008D4D40) are actually tagged offsets into BootKernelExtensions.kc. These are effectively "imported" symbols from the kernel that will be resolved once AuxiliaryKernelExtensions.kc is loaded in memory. Unfortunately these imports don’t behave like imports in a normal Mach-O file, so IDA can’t properly resolve the symbol names without doing some extra work.

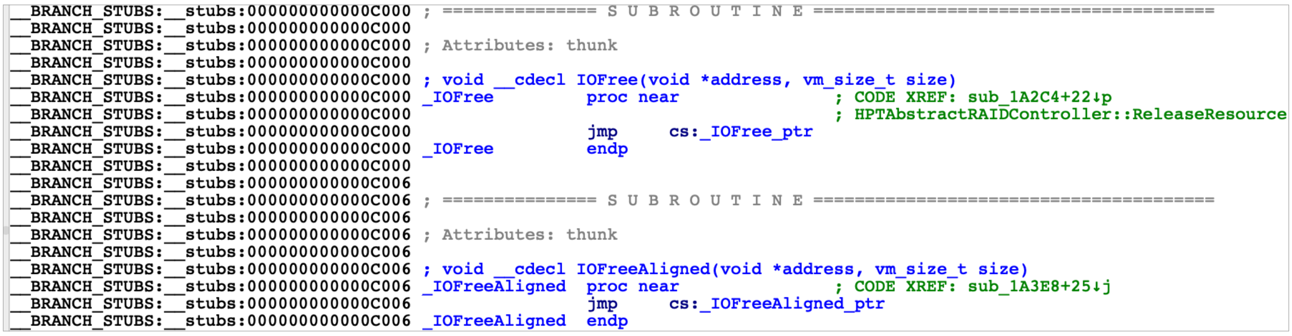
IDA 7.5 SP3 provides a workaround. When loading AuxiliaryKernelExtensions.kc, SP3 will detect that the file contains pointers into the kernel and will offer to resolve them:



Detected pointers to symbols in the boot kext collection. To import them, provide the path to BootKernelExensions.kc:

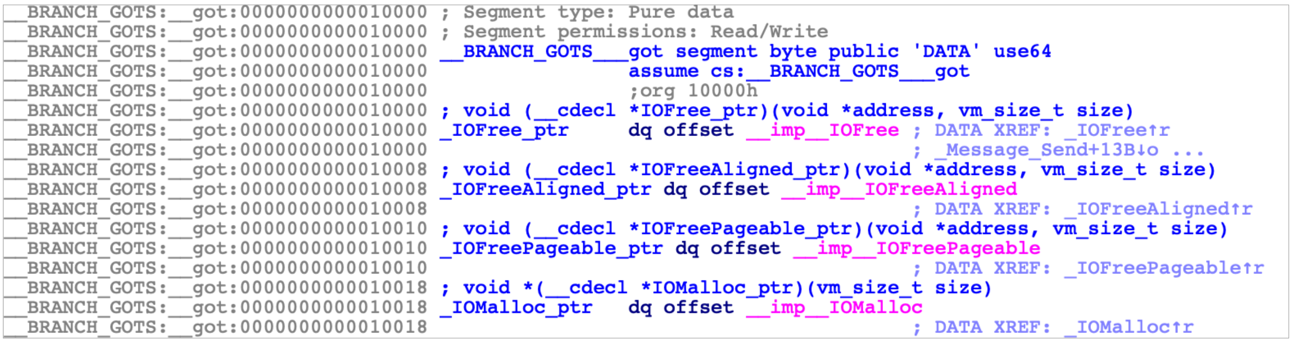
Path /Users/troy/macOS11.kdk/BootKernelExtensions.kc

This leads to much cleaner analysis of the kernel stubs:



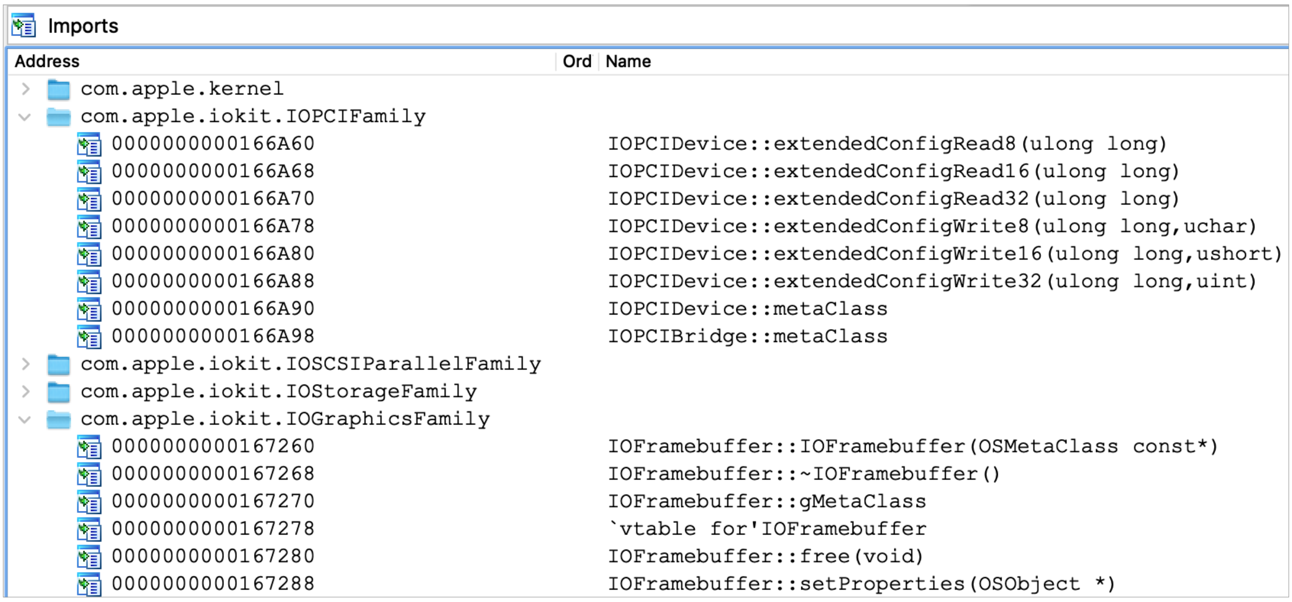
```
__BRANCH_STUBS:__stubs:000000000000C000 ; =============== S U B R O U T I N E =======================================
__BRANCH_STUBS:__stubs:000000000000C000
__BRANCH_STUBS:__stubs:000000000000C000 ; Attributes: thunk
__BRANCH_STUBS:__stubs:000000000000C000
__BRANCH_STUBS:__stubs:000000000000C000 ; void __cdecl IOFree(void *address, vm_size_t size)
__BRANCH_STUBS:__stubs:000000000000C000 _IOFree         proc near               ; CODE XREF: sub_1A2C4+22↓p
__BRANCH_STUBS:__stubs:000000000000C000                                         ; HPTAbstractRAIDController::ReleaseResource
__BRANCH_STUBS:__stubs:000000000000C000                 jmp     cs:_IOFree_ptr
__BRANCH_STUBS:__stubs:000000000000C000 _IOFree         endp
__BRANCH_STUBS:__stubs:000000000000C000
__BRANCH_STUBS:__stubs:000000000000C006
__BRANCH_STUBS:__stubs:000000000000C006 ; =============== S U B R O U T I N E =======================================
__BRANCH_STUBS:__stubs:000000000000C006
__BRANCH_STUBS:__stubs:000000000000C006 ; Attributes: thunk
__BRANCH_STUBS:__stubs:000000000000C006
__BRANCH_STUBS:__stubs:000000000000C006 ; void __cdecl IOFreeAligned(void *address, vm_size_t size)
__BRANCH_STUBS:__stubs:000000000000C006 _IOFreeAligned  proc near               ; CODE XREF: sub_1A3E8+25↓j
__BRANCH_STUBS:__stubs:000000000000C006                 jmp     cs:_IOFreeAligned_ptr
__BRANCH_STUBS:__stubs:000000000000C006 _IOFreeAligned  endp
```

The values in __got have been replaced by imported items, similar to what IDA does for regular Mach-O binaries:



```
__BRANCH_GOTS:__got:0000000000010000 ; Segment type: Pure data
__BRANCH_GOTS:__got:0000000000010000 ; Segment permissions: Read/Write
__BRANCH_GOTS:__got:0000000000010000 __BRANCH_GOTS___got segment byte public 'DATA' use64
__BRANCH_GOTS:__got:0000000000010000                 assume cs:__BRANCH_GOTS___got
__BRANCH_GOTS:__got:0000000000010000                 ;org 10000h
__BRANCH_GOTS:__got:0000000000010000 ; void (__cdecl *IOFree_ptr)(void *address, vm_size_t size)
__BRANCH_GOTS:__got:0000000000010000 _IOFree_ptr     dq offset __imp__IOFree ; DATA XREF: _IOFree↑r
__BRANCH_GOTS:__got:0000000000010000                                         ; _Message_Send+13B↓o ...
__BRANCH_GOTS:__got:0000000000010008 ; void (__cdecl *IOFreeAligned_ptr)(void *address, vm_size_t size)
__BRANCH_GOTS:__got:0000000000010008 _IOFreeAligned_ptr dq offset __imp__IOFreeAligned
__BRANCH_GOTS:__got:0000000000010008                                         ; DATA XREF: _IOFreeAligned↑r
__BRANCH_GOTS:__got:0000000000010010 ; void (__cdecl *IOFreePageable_ptr)(void *address, vm_size_t size)
__BRANCH_GOTS:__got:0000000000010010 _IOFreePageable_ptr dq offset __imp__IOFreePageable
__BRANCH_GOTS:__got:0000000000010010                                         ; DATA XREF: _IOFreePageable↑r
__BRANCH_GOTS:__got:0000000000010018 ; void *(__cdecl *IOMalloc_ptr)(vm_size_t size)
__BRANCH_GOTS:__got:0000000000010018 _IOMalloc_ptr   dq offset __imp__IOMalloc
__BRANCH_GOTS:__got:0000000000010018                                         ; DATA XREF: _IOMalloc↑r
```

You can also consult the Imports view for a summary of the imported symbols:



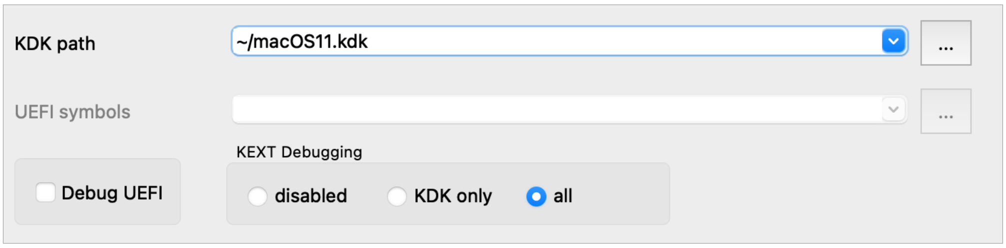
Address	Ord	Name
com.apple.kernel		
com.apple.iokit.IOPCIFamily		
0000000000166A60		IOPCIDevice::extendedConfigRead8(ulong long)
0000000000166A68		IOPCIDevice::extendedConfigRead16(ulong long)
0000000000166A70		IOPCIDevice::extendedConfigRead32(ulong long)
0000000000166A78		IOPCIDevice::extendedConfigWrite8(ulong long,uchar)
0000000000166A80		IOPCIDevice::extendedConfigWrite16(ulong long,ushort)
0000000000166A88		IOPCIDevice::extendedConfigWrite32(ulong long,uint)
0000000000166A90		IOPCIDevice::metaClass
0000000000166A98		IOPCIBridge::metaClass
com.apple.iokit.IOSCSIParallelFamily		
com.apple.iokit.IOStorageFamily		
com.apple.iokit.IOGraphicsFamily		
0000000000167260		IOFramebuffer::IOFramebuffer(OSMetaClass const*)
0000000000167268		IOFramebuffer::~IOFramebuffer()
0000000000167270		IOFramebuffer::gMetaClass
0000000000167278		`vtable for'IOFramebuffer
0000000000167280		IOFramebuffer::free(void)
0000000000167288		IOFramebuffer::setProperties(OSObject *)

Also note that kernelcache symbolication can be done automatically using the BOOT_KC_PATH option in macho.cfg, which can be useful when doing analysis in batch mode.

Now that the static analysis is sufficiently robust, let's try debugging this database.

3.5. Debugging KernelCollections

To debug our kernelcache database, use the same xnu debugging options as before:



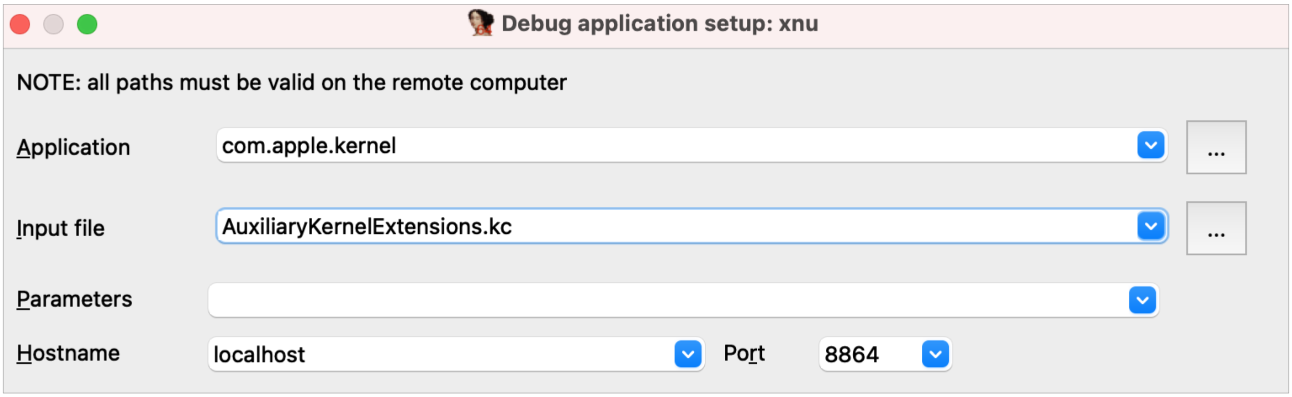
KDK path: ~/macOS11.kdk

UEFI symbols:

☐ Debug UEFI

KEXT Debugging: ○ disabled ○ KDK only ● all

Then use **Debugger>Process options** to set the following fields:



NOTE: all paths must be valid on the remote computer

Application: com.apple.kernel

Input file: AuxiliaryKernelExtensions.kc

Parameters:

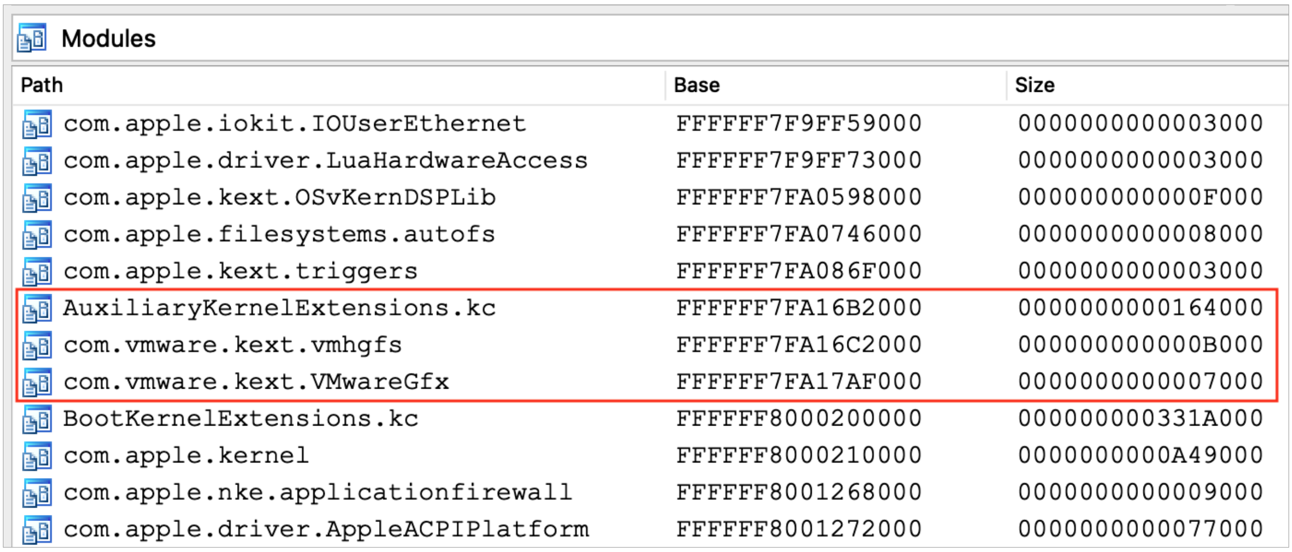
Hostname: localhost

Port: 8864

IDA will identify the load address of the input file after attaching, and rebase the database accordingly:

```
FFFFFF8000AFC9C0: detected Darwin Kernel Version 20.1.0: Thu Sep 24 20:22:06 PDT 2020; root:xnu-7195.40.89.100.3~1/RELEASE_X86_64
XNU platform version: macOS 11.0.0
found kernelcache: /Users/troy/macOS11.kdk/BootKernelExtensions.kc
found kernelcache: /Users/troy/macOS11.kdk/SystemKernelExtensions.kc
found kernelcache: /Users/troy/macOS11.kdk/AuxiliaryKernelExtensions.kc
FFFFFF80011ED840: _gLoadedKextSummaries
FFFFFF80009AB6C0: bpt hook for _OSKextLoadedKextSummariesUpdated: 1
FFFFFF8000200000: loaded BootKernelExtensions.kc
FFFFFF7F80C11000: loaded SystemKernelExtensions.kc
Rebasing program to 0xFFFFFF7FA16AA000...
FFFFFF7FA16B2000: loaded AuxiliaryKernelExtensions.kc
```

From the modules window we can see that although the entire AuxiliaryKernelExtensions.kc file was loaded into memory, only two of the KEXT subfiles are currently active:



Path	Base	Size
com.apple.iokit.IOUserEthernet	FFFFFF7F9FF59000	0000000000003000
com.apple.driver.LuaHardwareAccess	FFFFFF7F9FF73000	0000000000003000
com.apple.kext.OSvKernDSPLib	FFFFFF7FA0598000	000000000000F000
com.apple.filesystems.autofs	FFFFFF7FA0746000	0000000000008000
com.apple.kext.triggers	FFFFFF7FA086F000	0000000000003000
AuxiliaryKernelExtensions.kc	FFFFFF7FA16B2000	0000000000164000
com.vmware.kext.vmhgfs	FFFFFF7FA16C2000	000000000000B000
com.vmware.kext.VMwareGfx	FFFFFF7FA17AF000	0000000000007000
BootKernelExtensions.kc	FFFFFF8000200000	000000000331A000
com.apple.kernel	FFFFFF8000210000	0000000000A49000
com.apple.nke.applicationfirewall	FFFFFF8001268000	0000000000009000
com.apple.driver.AppleACPIPlatform	FFFFFF8001272000	0000000000077000

Hopefully we've shown that IDA allows you to debug a macOS11 KernelCollection almost as easy as any other Mach-O file. Also note that this same approach can be used to debug the "Boot" and "System" KernelCollections.

3.6. macOS11 + DWARF

So far Apple has published Kernel Debug Kits for the macOS11 beta versions, but they are of limited use.

Apparently when the mach kernel binary is inserted into BootKernelExtensions.kc, its segment mappings are modified. This means that the running kernel image in memory will differ from the kernel/DWARF binaries shipped with Apple's KDK. This will inevitably cause problems during debugging/analysis.

Apple seems to be doing something nonsensical here, so DWARF-enabled debugging with macOS11 is currently a TODO. Hopefully the situation will be resolved once macOS11 is officially released, and we will update this writeup.

4. UEFI Debugging

It is possible to debug the EFI firmware of a VMware Fusion guest. This section discusses some interesting examples.

4.1. Debugging the OSX Bootloader

Firmware debugging gives us the unique opportunity to debug the OSX bootloader. Here's how it can be easily done in IDA:

1. First copy the bootloader executable from your VM:

```
$ scp user@vm:/System/Library/CoreServices/boot.efi .
```

2. Now shut down the VM and add this line to the .vmx file:

```
monitor.debugOnStartGuest64 = "TRUE"
```

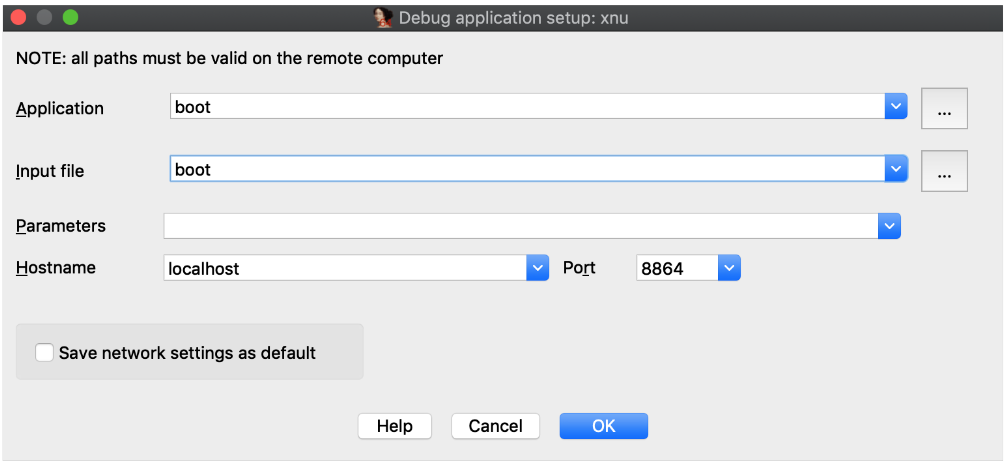
3. Load the boot.efi binary in IDA, open **Debugger>Debugger options**, check **Suspend on library load/unload**, and set **Event condition** to:

```
get_event_id() == LIB_LOADED && get_event_module_name() == "boot"
```

This will suspend the OS just before the bootloader entry point is invoked.

Note: For some older versions of OSX, the bootloader will be named "boot.sys". You can check the name under the **.debug** section of the executable.

4. Now select **Remote XNU Debugger** from the Debugger menu, and set the following fields in **Debugger>Process options**:



Debug application setup: xnu

NOTE: all paths must be valid on the remote computer

Application: boot

Input file: boot

Parameters:

Hostname: localhost

Port: 8864

☐ Save network settings as default

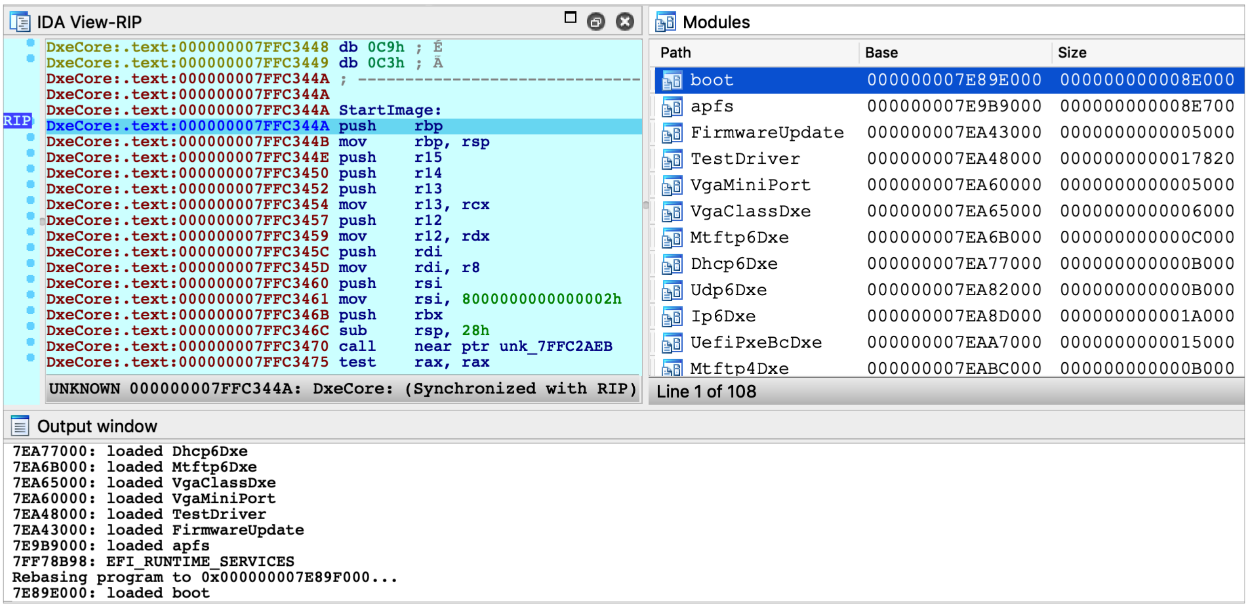
5. We're now ready to start debugging the bootloader.

Power on the VM (note that the VM is unresponsive since it is suspended), and attach to it with **Debugger>Attach to process**. After attaching IDA will try to detect the **EFI_BOOT_SERVICES** table. You should see the debugger print something like this to the console:

```
7FFD7430: EFI_BOOT_SERVICES
```

6. Now resume the process.

You should see many UEFI drivers being loaded, until eventually boot.efi is loaded and IDA suspends the process:



```
DxeCore:.text:000000007FFC3448 db 0C9h ; É
DxeCore:.text:000000007FFC3449 db 0C3h ; Ã
DxeCore:.text:000000007FFC344A ; ---------------------------------
DxeCore:.text:000000007FFC344A
DxeCore:.text:000000007FFC344A StartImage:
DxeCore:.text:000000007FFC344A push    rbp
DxeCore:.text:000000007FFC344B mov     rbp, rsp
DxeCore:.text:000000007FFC344E push    r15
DxeCore:.text:000000007FFC3450 push    r14
DxeCore:.text:000000007FFC3452 push    r13
DxeCore:.text:000000007FFC3454 mov     r13, rcx
DxeCore:.text:000000007FFC3457 push    r12
DxeCore:.text:000000007FFC3459 mov     r12, rdx
DxeCore:.text:000000007FFC345C push    rdi
DxeCore:.text:000000007FFC345D mov     rdi, r8
DxeCore:.text:000000007FFC3460 push    rsi
DxeCore:.text:000000007FFC3461 mov     rsi, 8000000000000002h
DxeCore:.text:000000007FFC346B push    rbx
DxeCore:.text:000000007FFC346C sub     rsp, 28h
DxeCore:.text:000000007FFC3470 call    near ptr unk_7FFC2AEB
DxeCore:.text:000000007FFC3475 test    rax, rax
UNKNOWN 000000007FFC344A: DxeCore: (Synchronized with RIP)
```

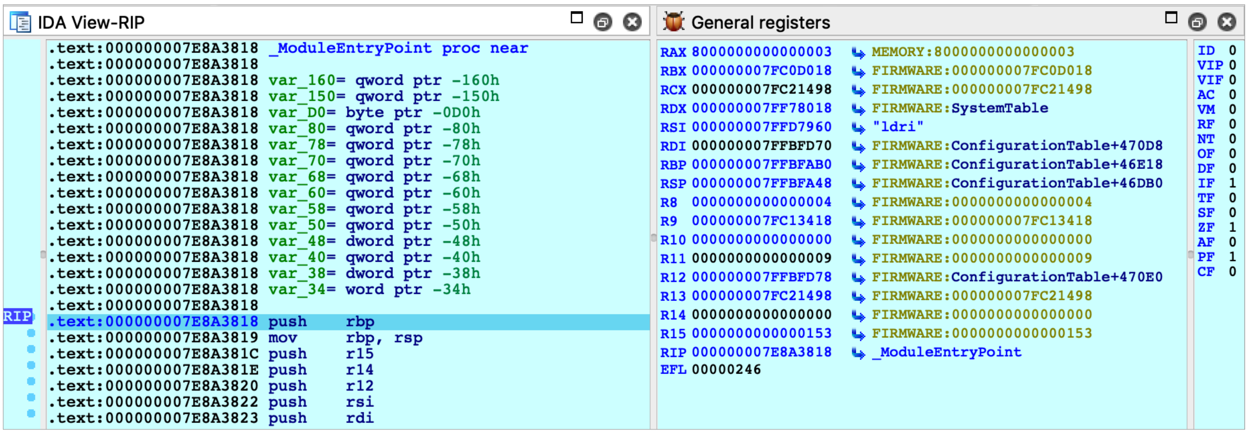
Path	Base	Size
boot	000000007E89E000	000000000008E000
apfs	000000007E9B9000	000000000008E700
FirmwareUpdate	000000007EA43000	0000000000005000
TestDriver	000000007EA48000	0000000000017820
VgaMiniPort	000000007EA60000	0000000000005000
VgaClassDxe	000000007EA65000	0000000000006000
Mtftp6Dxe	000000007EA6B000	000000000000C000
Dhcp6Dxe	000000007EA77000	000000000000B000
Udp6Dxe	000000007EA82000	000000000000B000
Ip6Dxe	000000007EA8D000	000000000001A000
UefiPxeBcDxe	000000007EAA7000	0000000000015000
Mtftp4Dxe	000000007EABC000	000000000000B000

Line 1 of 108

```
7EA77000: loaded Dhcp6Dxe
7EA6B000: loaded Mtftp6Dxe
7EA65000: loaded VgaClassDxe
7EA60000: loaded VgaMiniPort
7EA48000: loaded TestDriver
7EA43000: loaded FirmwareUpdate
7E9B9000: loaded apfs
7FF78B98: EFI_RUNTIME_SERVICES
Rebasing program to 0x000000007E89F000...
7E89E000: loaded boot
```

7. At this point, the bootloader entry point function is about to be invoked.

   Jump to **_ModuleEntryPoint** in boot.efi and press F4. We can now step through boot.efi:



```
.text:000000007E8A3818 _ModuleEntryPoint proc near
.text:000000007E8A3818
.text:000000007E8A3818 var_160= qword ptr -160h
.text:000000007E8A3818 var_150= qword ptr -150h
.text:000000007E8A3818 var_D0= byte ptr -0D0h
.text:000000007E8A3818 var_80= qword ptr -80h
.text:000000007E8A3818 var_78= qword ptr -78h
.text:000000007E8A3818 var_70= qword ptr -70h
.text:000000007E8A3818 var_68= qword ptr -68h
.text:000000007E8A3818 var_60= qword ptr -60h
.text:000000007E8A3818 var_58= qword ptr -58h
.text:000000007E8A3818 var_50= qword ptr -50h
.text:000000007E8A3818 var_48= dword ptr -48h
.text:000000007E8A3818 var_40= qword ptr -40h
.text:000000007E8A3818 var_38= dword ptr -38h
.text:000000007E8A3818 var_34= word ptr -34h
.text:000000007E8A3818
.text:000000007E8A3818 push    rbp
.text:000000007E8A3819 mov     rbp, rsp
.text:000000007E8A381C push    r15
.text:000000007E8A381E push    r14
.text:000000007E8A3820 push    r12
.text:000000007E8A3822 push    rsi
.text:000000007E8A3823 push    rdi
```

```
RAX 8000000000000003  ↳ MEMORY:8000000000000003               ID  0
RBX 000000007FC0D018  ↳ FIRMWARE:000000007FC0D018             VIP 0
RCX 000000007FC21498  ↳ FIRMWARE:000000007FC21498             VIF 0
RDX 000000007FF78018  ↳ FIRMWARE:SystemTable                  AC  0
RSI 000000007FFD7960  ↳ "ldri"                                VM  0
RDI 000000007FFBFD70  ↳ FIRMWARE:ConfigurationTable+470D8     RF  0
RBP 000000007FFBFAB0  ↳ FIRMWARE:ConfigurationTable+46E18     NT  0
RSP 000000007FFBFA48  ↳ FIRMWARE:ConfigurationTable+46DB0     OF  0
R8  0000000000000004  ↳ FIRMWARE:0000000000000004             DF  0
R9  000000007FC13418  ↳ FIRMWARE:000000007FC13418             IF  1
R10 0000000000000000  ↳ FIRMWARE:0000000000000000             TF  0
R11 0000000000000009  ↳ FIRMWARE:0000000000000009             SF  0
R12 000000007FFBFD78  ↳ FIRMWARE:ConfigurationTable+470E0     ZF  1
R13 000000007FC21498  ↳ FIRMWARE:000000007FC21498             AF  0
R14 0000000000000000  ↳ FIRMWARE:0000000000000000             PF  1
R15 0000000000000153  ↳ FIRMWARE:0000000000000153             CF  0
RIP 000000007E8A3818  ↳ _ModuleEntryPoint
EFL 00000246
```

4.2. GetMemoryMap

To facilitate UEFI debugging, IDA provides an IDC helper function: **xnu_get_efi_memory_map**.

This function will invoke the **GetMemoryMap** function in the **EFI_BOOT_SERVICES** table and return an array of **EFI_MEMORY_DESCRIPTOR** objects:

```
IDC>extern map;
IDC>map = xnu_get_efi_memory_map();
IDC>map.size
     35.       23h          43o
IDC>map[27]
object
    Attribute: 0x800000000000000Fi64
    NumberOfPages: 0x20i64
    PhysicalStart: 0x7FF09000i64
    Type: "EfiRuntimeServicesCode"
    VirtualStart: 0x0i64
```

This function can be invoked at any point during firmware debugging.

4.3. UEFI Debugging + DWARF

If you build your own EFI apps or drivers on OSX, you can use IDA to debug the source code.

In this example we will debug a sample EFI application. On OSX the convention is to build EFI apps in the Mach-O format, then convert the file to PE .efi with the **mtoc** utility. In this example, assume we have an EFI build on our OSX virtual machine that contains the following files in the ~/TestApp directory:

- **TestApp.efi** - the EFI application that will be run
- **TestApp.dll** - the original Mach-O binary
- **TestApp.dll.dSYM** - DWARF info for the app
- **TestApp.c** - source code for the app

Here's how we can debug this application in IDA:

1. On your host machine, create a directory that will mirror the directory on the VM:

```
mkdir ~/TestApp
```

2. Copy the efi, macho, dSYM, and c files from your VM:

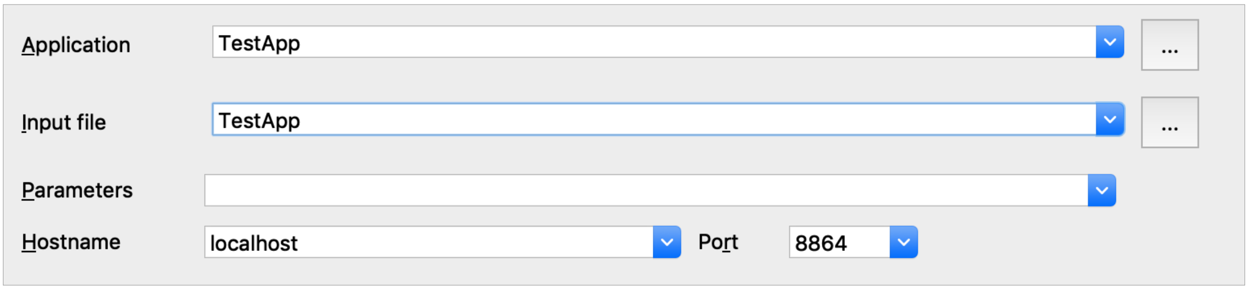
```
scp -r vmuser@vm:TestApp/TestApp.* ~/TestApp
```

3. Open the TestApp.efi binary in IDA, and wait for IDA to analyze it.

   Note that you can improve the disassembly by loading the DWARF file from TestApp.dll.dSYM. You can do this with **Edit>Plugins>Load DWARF file**, or you can load it programatically from IDAPython:

```
path = "~/TestApp/TestApp.dll.dSYM/Contents/Resources/DWARF/TestApp.dll"
node = idaapi.netnode()
node.create("$ dwarf_params")
node.supset(1, os.path.expanduser(path))
idaapi.load_and_run_plugin("dwarf", 3)
```

4. Select **Remote XNU Debugger** from the debugger menu, and set the following fields in **Debugger>Process options**:

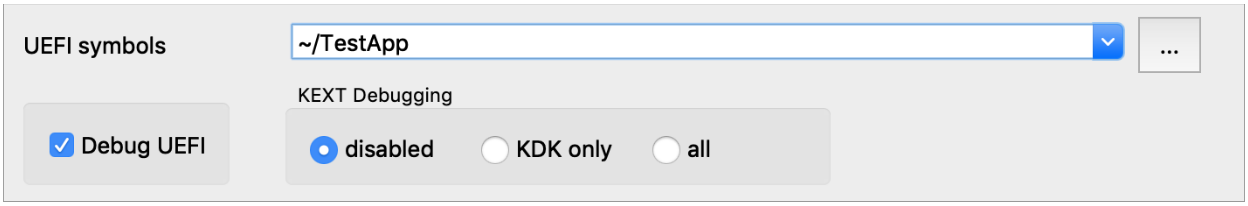


Application: TestApp
Input file: TestApp
Parameters:
Hostname: localhost
Port: 8864

5. In **Debugger>Debugger options**, enable **Suspend on library load/unload** and set the **Event condition** field to:

```
get_event_id() == LIB_LOADED && get_event_module_name() == "TestApp"
```

6. In **Debugger>Debugger options>Set specific options**, set the following fields:



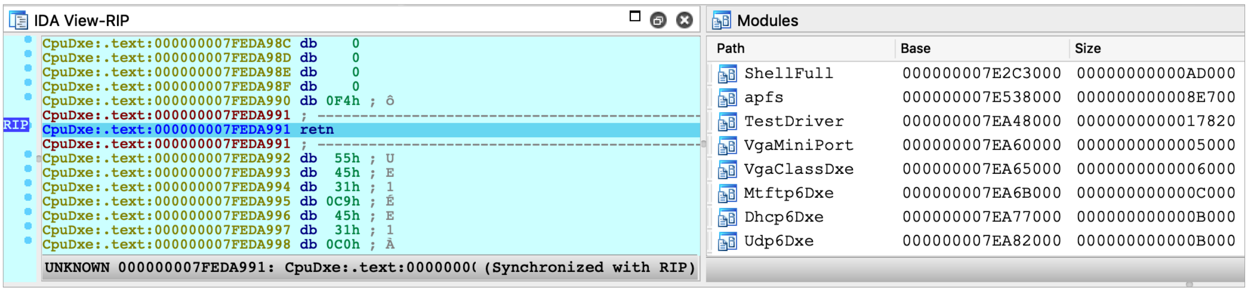
UEFI symbols: ~/TestApp
☑ Debug UEFI
KEXT Debugging: ● disabled ○ KDK only ○ all

   Note that we must enable the **Debug UEFI** option, and set the **UEFI symbols** option so the debugger can find DWARF info for the EFI app at runtime.

7. If the usernames on the host and VM are different, we will need a source path mapping:

```
idaapi.add_path_mapping("/Users/vmuser/TestApp", "/Users/hostuser/TestApp")
```

8. Reboot the VM and enter the EFI Shell
9. Attach the debugger. After attaching IDA will detect the firmware images that have already been loaded:



IDA View-RIP

```
CpuDxe:.text:000000007FEDA98C db    0
CpuDxe:.text:000000007FEDA98D db    0
CpuDxe:.text:000000007FEDA98E db    0
CpuDxe:.text:000000007FEDA98F db    0
CpuDxe:.text:000000007FEDA990 db 0F4h ; ô
CpuDxe:.text:000000007FEDA991 ; ---------------------------------------------
CpuDxe:.text:000000007FEDA991 retn
CpuDxe:.text:000000007FEDA991 ; ---------------------------------------------
CpuDxe:.text:000000007FEDA992 db  55h ; U
CpuDxe:.text:000000007FEDA993 db  45h ; E
CpuDxe:.text:000000007FEDA994 db  31h ; 1
CpuDxe:.text:000000007FEDA995 db 0C9h ; É
CpuDxe:.text:000000007FEDA996 db  45h ; E
CpuDxe:.text:000000007FEDA997 db  31h ; 1
CpuDxe:.text:000000007FEDA998 db 0C0h ; À
UNKNOWN 000000007FEDA991: CpuDxe:.text:0000000( (Synchronized with RIP)
```

Modules

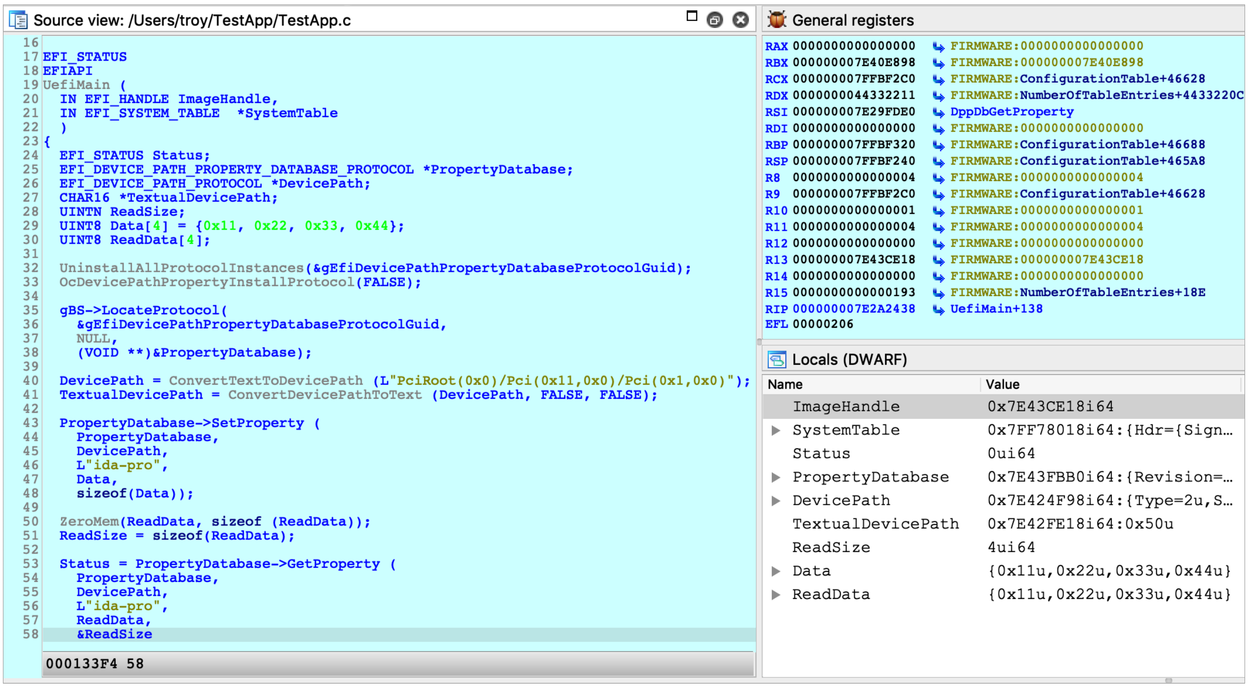
Path	Base	Size
ShellFull	000000007E2C3000	00000000000AD000
apfs	000000007E538000	000000000008E700
TestDriver	000000007EA48000	0000000000017820
VgaMiniPort	000000007EA60000	0000000000005000
VgaClassDxe	000000007EA65000	0000000000006000
Mtftp6Dxe	000000007EA6B000	000000000000C000
Dhcp6Dxe	000000007EA77000	000000000000B000
Udp6Dxe	000000007EA82000	000000000000B000

10. Resume the OS and launch TestApp from the EFI Shell prompt:

```
Shell>fs1:\Users\vmuser\TestApp\TestApp.efi
```

At this point IDA will detect that the target app has been loaded, and suspend the process just before the entry point of TestApp.efi (because of step 5).

11. Now we can set a breakpoint somewhere in TestApp.efi and resume the OS. The debugger will be able to load source file and local variable information from TestApp.dll.dSYM:



Source view: /Users/troy/TestApp/TestApp.c

```

EFI_STATUS
EFIAPI
UefiMain (
  IN EFI_HANDLE ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
{
  EFI_STATUS Status;
  EFI_DEVICE_PATH_PROPERTY_DATABASE_PROTOCOL *PropertyDatabase;
  EFI_DEVICE_PATH_PROTOCOL *DevicePath;
  CHAR16 *TextualDevicePath;
  UINTN ReadSize;
  UINT8 Data[4] = {0x11, 0x22, 0x33, 0x44};
  UINT8 ReadData[4];

  UninstallAllProtocolInstances(&gEfiDevicePathPropertyDatabaseProtocolGuid);
  OcDevicePathPropertyInstallProtocol(FALSE);

  gBS->LocateProtocol(
    &gEfiDevicePathPropertyDatabaseProtocolGuid,
    NULL,
    (VOID **)&PropertyDatabase);

  DevicePath = ConvertTextToDevicePath (L"PciRoot(0x0)/Pci(0x11,0x0)/Pci(0x1,0x0)");
  TextualDevicePath = ConvertDevicePathToText (DevicePath, FALSE, FALSE);

  PropertyDatabase->SetProperty (
    PropertyDatabase,
    DevicePath,
    L"ida-pro",
    Data,
    sizeof(Data));

  ZeroMem(ReadData, sizeof (ReadData));
  ReadSize = sizeof(ReadData);

  Status = PropertyDatabase->GetProperty (
    PropertyDatabase,
    DevicePath,
    L"ida-pro",
    ReadData,
    &ReadSize
000133F4 58
```

General registers

```
RAX 0000000000000000  FIRMWARE:0000000000000000
RBX 000000007E40E898  FIRMWARE:000000007E40E898
RCX 000000007FFBF2C0  FIRMWARE:ConfigurationTable+46628
RDX 0000000044332211  FIRMWARE:NumberOfTableEntries+4433220C
RSI 000000007E29FDE0  DppDbGetProperty
RDI 0000000000000000  FIRMWARE:0000000000000000
RBP 000000007FFBF320  FIRMWARE:ConfigurationTable+46688
RSP 000000007FFBF240  FIRMWARE:ConfigurationTable+465A8
R8  0000000000000004  FIRMWARE:0000000000000004
R9  000000007FFBF2C0  FIRMWARE:ConfigurationTable+46628
R10 0000000000000001  FIRMWARE:0000000000000001
R11 0000000000000004  FIRMWARE:0000000000000004
R12 0000000000000000  FIRMWARE:0000000000000000
R13 000000007E43CE18  FIRMWARE:000000007E43CE18
R14 0000000000000000  FIRMWARE:0000000000000000
R15 0000000000000193  FIRMWARE:NumberOfTableEntries+18E
RIP 000000007E2A2438  UefiMain+138
EFL 00000206
```

Locals (DWARF)

Name	Value
ImageHandle	0x7E43CE18i64
▶ SystemTable	0x7FF78018i64:{Hdr={Sign…
Status	0ui64
▶ PropertyDatabase	0x7E43FBB0i64:{Revision=…
▶ DevicePath	0x7E424F98i64:{Type=2u,S…
TextualDevicePath	0x7E42FE18i64:0x50u
ReadSize	4ui64
▶ Data	{0x11u,0x22u,0x33u,0x44u}
▶ ReadData	{0x11u,0x22u,0x33u,0x44u}

IMPORTANT NOTE: You must wait until TestApp has been loaded into memory before setting any breakpoints. If you add a breakpoint in the database before attaching the debugger, IDA might not set the breakpoint at the correct address. This is a limitation in IDA that we must work around for now.

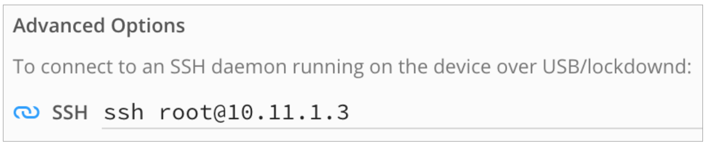
5. Debugging iOS with Corellium

IDA can also debug the iOS kernel, provided you have access to a virtual iOS device from Corellium.

5.1. Quick Start

To get started with debugging iOS, we will perform a simple experiment to patch kernel memory.

The device used in this example is a virtual iPhone XS with iOS 12.1.4, but it should work with any model or iOS version that Corellium supports. Begin by powering on your device and allow it to boot up. In the Corellium UI, look for the line labeled **SSH** under **Advanced options**:



Advanced Options

To connect to an SSH daemon running on the device over USB/lockdownd:

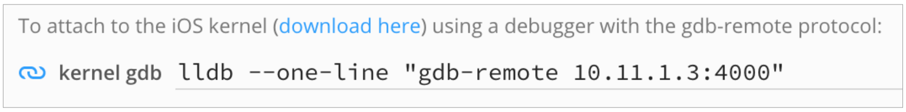
SSH `ssh root@10.11.1.3`

Ensure you can connect to the device by running this command over ssh:

```
$ ssh root@10.11.1.3 uname -v
Darwin Kernel Version 18.2.0 ... root:xnu-4903.242.2~1/RELEASE_ARM64_T8020
```

We will use IDA to patch this version string.

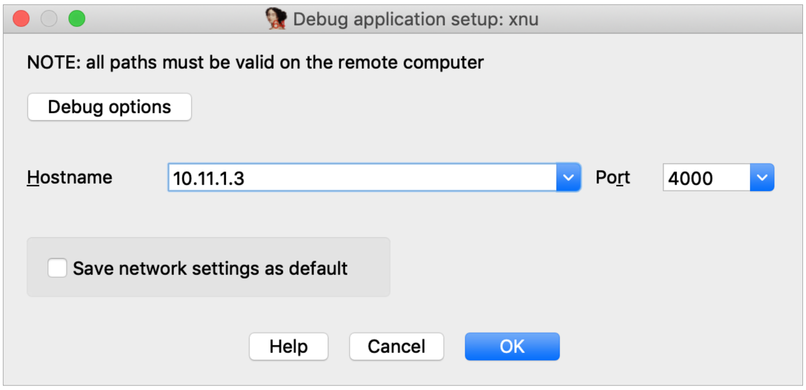
Now launch IDA, and when prompted with the window **IDA: Quick start**, choose **Go** to start with an empty database and open **Debugger>Attach>Remote XNU Debugger**. In the Corellium UI, find the hostname:port used by the kernel GDB stub. It should be specified in the line labeled **kernel gdb**:



To attach to the iOS kernel (download here) using a debugger with the gdb-remote protocol:

kernel gdb `lldb --one-line "gdb-remote 10.11.1.3:4000"`

And set the **Hostname** and **Port** fields in IDA's application setup window:



Debug application setup: xnu

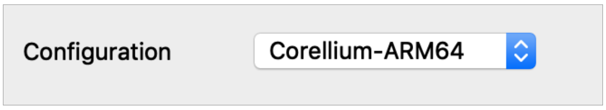
NOTE: all paths must be valid on the remote computer

Hostname: 10.11.1.3

Port: 4000

☐ Save network settings as default

Now click on **Debug options>Set specific options**, and for the **Configuration** dropdown menu, be sure to select **Corellium-ARM64**:

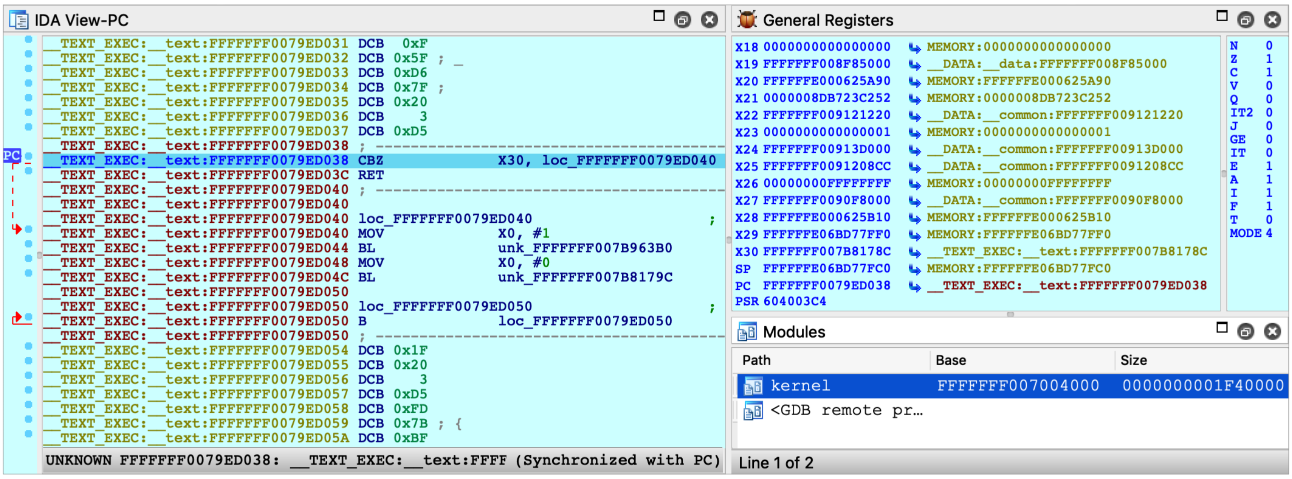


Configuration: Corellium-ARM64

You can ignore the other config options for now, and click OK.

Click OK again, and wait for IDA to establish a connection to Corellium's GDB stub (this may take a few seconds). Then select **<attach to the process started on target>** and wait for IDA to attach. This might take several seconds (we will address this later), but for now simply wait for IDA to perform the initial setup.

If IDA could detect the kernel, it should appear in the Modules list:



```
__TEXT_EXEC:__text:FFFFFFF0079ED031 DCB  0xF
__TEXT_EXEC:__text:FFFFFFF0079ED032 DCB 0x5F ; _
__TEXT_EXEC:__text:FFFFFFF0079ED033 DCB 0xD6
__TEXT_EXEC:__text:FFFFFFF0079ED034 DCB 0x7F ;
__TEXT_EXEC:__text:FFFFFFF0079ED035 DCB 0x20
__TEXT_EXEC:__text:FFFFFFF0079ED036 DCB    3
__TEXT_EXEC:__text:FFFFFFF0079ED037 DCB 0xD5
__TEXT_EXEC:__text:FFFFFFF0079ED038 ; ---------------------------------------
__TEXT_EXEC:__text:FFFFFFF0079ED038 CBZ             X30, loc_FFFFFFF0079ED040
__TEXT_EXEC:__text:FFFFFFF0079ED03C RET
__TEXT_EXEC:__text:FFFFFFF0079ED040 ; ---------------------------------------
__TEXT_EXEC:__text:FFFFFFF0079ED040
__TEXT_EXEC:__text:FFFFFFF0079ED040 loc_FFFFFFF0079ED040                    ;
__TEXT_EXEC:__text:FFFFFFF0079ED040 MOV             X0, #1
__TEXT_EXEC:__text:FFFFFFF0079ED044 BL              unk_FFFFFFF007B963B0
__TEXT_EXEC:__text:FFFFFFF0079ED048 MOV             X0, #0
__TEXT_EXEC:__text:FFFFFFF0079ED04C BL              unk_FFFFFFF007B8179C
__TEXT_EXEC:__text:FFFFFFF0079ED050
__TEXT_EXEC:__text:FFFFFFF0079ED050 loc_FFFFFFF0079ED050                    ;
__TEXT_EXEC:__text:FFFFFFF0079ED050 B               loc_FFFFFFF0079ED050
__TEXT_EXEC:__text:FFFFFFF0079ED050 ; ---------------------------------------
__TEXT_EXEC:__text:FFFFFFF0079ED054 DCB 0x1F
__TEXT_EXEC:__text:FFFFFFF0079ED055 DCB 0x20
__TEXT_EXEC:__text:FFFFFFF0079ED056 DCB    3
__TEXT_EXEC:__text:FFFFFFF0079ED057 DCB 0xD5
__TEXT_EXEC:__text:FFFFFFF0079ED058 DCB 0xFD
__TEXT_EXEC:__text:FFFFFFF0079ED059 DCB 0x7B ; {
__TEXT_EXEC:__text:FFFFFFF0079ED05A DCB 0xBF
UNKNOWN FFFFFFF0079ED038: __TEXT_EXEC:__text:FFFF (Synchronized with PC)
```

```
X18 0000000000000000 ↳ MEMORY:0000000000000000
X19 FFFFFFF008F85000 ↳ __DATA:__data:FFFFFFF008F85000
X20 FFFFFFE000625A90 ↳ MEMORY:FFFFFFE000625A90
X21 0000008DB723C252 ↳ MEMORY:0000008DB723C252
X22 FFFFFFF009121220 ↳ __DATA:__common:FFFFFFF009121220
X23 0000000000000001 ↳ MEMORY:0000000000000001
X24 FFFFFFF00913D000 ↳ __DATA:__common:FFFFFFF00913D000
X25 FFFFFFF0091208CC ↳ __DATA:__common:FFFFFFF0091208CC
X26 00000000FFFFFFFF ↳ MEMORY:00000000FFFFFFFF
X27 FFFFFFF0090F8000 ↳ __DATA:__common:FFFFFFF0090F8000
X28 FFFFFFE000625B10 ↳ MEMORY:FFFFFFE000625B10
X29 FFFFFFE06BD77FF0 ↳ MEMORY:FFFFFFE06BD77FF0
X30 FFFFFFF007B8178C ↳ __TEXT_EXEC:__text:FFFFFFF007B8178C
SP  FFFFFFE06BD77FC0 ↳ MEMORY:FFFFFFE06BD77FC0
PC  FFFFFFF0079ED038 ↳ __TEXT_EXEC:__text:FFFFFFF0079ED038
PSR 604003C4
```

```
N    0
Z    1
C    1
V    0
Q    0
IT2  0
J    0
GE   0
IT   0
E    1
A    1
I    1
F    1
T    0
MODE 4
```

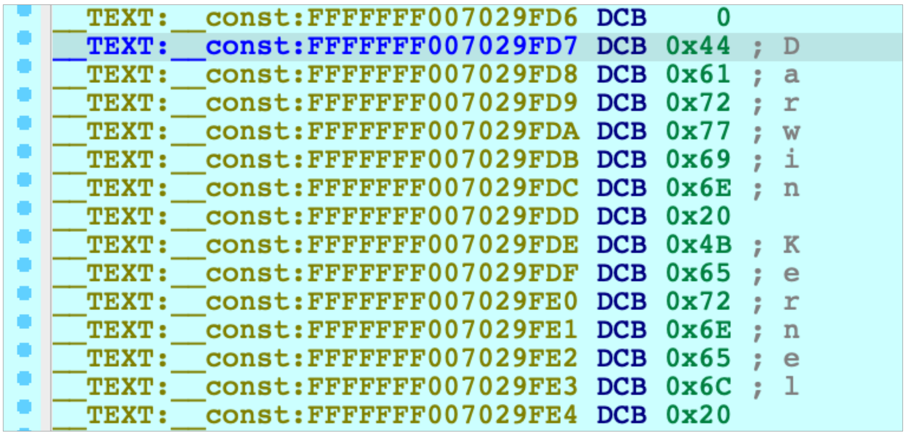
Path	Base	Size
kernel	FFFFFFF007004000	0000000001F40000
<GDB remote pr…		

Line 1 of 2

and the kernel version will be printed to the console:

```
FFFFFFF007029FD7: detected Darwin Kernel Version 18.2.0 ...
```

Navigate to this address and use IDAPython to overwrite the string:



```
__TEXT:__const:FFFFFFF007029FD6 DCB    0
__TEXT:__const:FFFFFFF007029FD7 DCB 0x44 ; D
__TEXT:__const:FFFFFFF007029FD8 DCB 0x61 ; a
__TEXT:__const:FFFFFFF007029FD9 DCB 0x72 ; r
__TEXT:__const:FFFFFFF007029FDA DCB 0x77 ; w
__TEXT:__const:FFFFFFF007029FDB DCB 0x69 ; i
__TEXT:__const:FFFFFFF007029FDC DCB 0x6E ; n
__TEXT:__const:FFFFFFF007029FDD DCB 0x20
__TEXT:__const:FFFFFFF007029FDE DCB 0x4B ; K
__TEXT:__const:FFFFFFF007029FDF DCB 0x65 ; e
__TEXT:__const:FFFFFFF007029FE0 DCB 0x72 ; r
__TEXT:__const:FFFFFFF007029FE1 DCB 0x6E ; n
__TEXT:__const:FFFFFFF007029FE2 DCB 0x65 ; e
__TEXT:__const:FFFFFFF007029FE3 DCB 0x6C ; l
__TEXT:__const:FFFFFFF007029FE4 DCB 0x20
```

```
idaapi.dbg_write_memory(0xFFFFFFF007029FD7, "IDAPRO".encode('utf-8'))
```

Resume the OS, and try running the same command as before:

```
$ ssh root@10.11.1.3 uname -v
IDAPRO Kernel Version 18.2.0 ... root:xnu-4903.242.2~1/RELEASE_ARM64_T8020
```

If we could successfully write to kernel memory, **IDAPRO** should appear in the output.

5.2. Creating a KDK for iOS

Typically a Kernel Development Kit is not available for iOS devices, but we can still utilise the KDK_PATH option in IDA to achieve faster debugging. In the example above, the initial attach can be slow because IDA must parse the kernel image in memory (which can be especially slow if the kernel has a symbol table).

Here's how you can speed things up:

1. create the KDK directory:

   ```
   $ mkdir ~/iPhoneKDK
   ```

2. copy the kernelcache from the virtual device:

```
$ scp root@ip:/System/Library/Caches/com.apple.kernelcaches/kernelcache /tmp
```

3. uncompress the kernelcache with lzssdec:

```
$ lzssdec -o OFF < /tmp/kernelcache > ~/iPhoneKDK/kernelcache
```

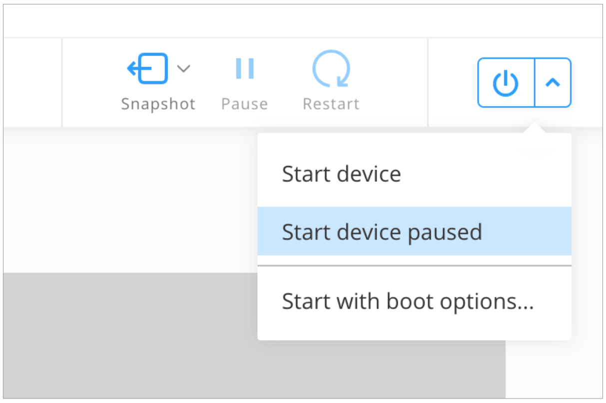
4. set KDK_PATH in dbg_xnu.cfg:

```
KDK_PATH = "~/iPhoneKDK";
```

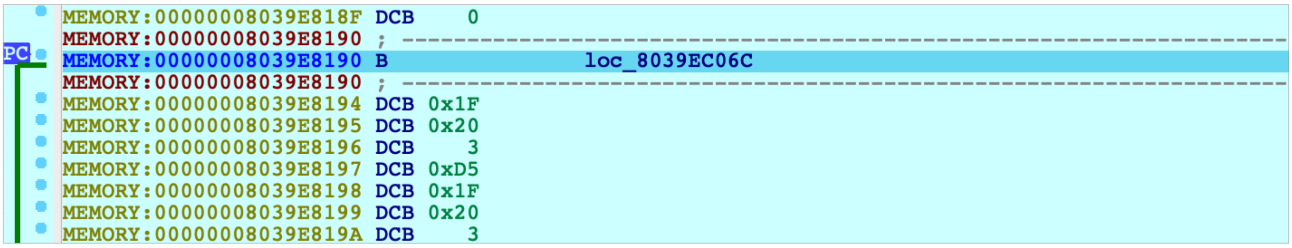
Now whenever the debugger must extract information from the kernel, it will parse the local file on disk. This should be noticeably faster, especially if the device is hosted by Corellium’s web service.

5.3. Debugging the iOS Kernel Entry Point

Corellium allows us to debug the first few instructions of kernel initialization. This can be very useful if we want to gain control of the OS as early as possible. In the Corellium UI, power on your device with the **Start device paused** option:

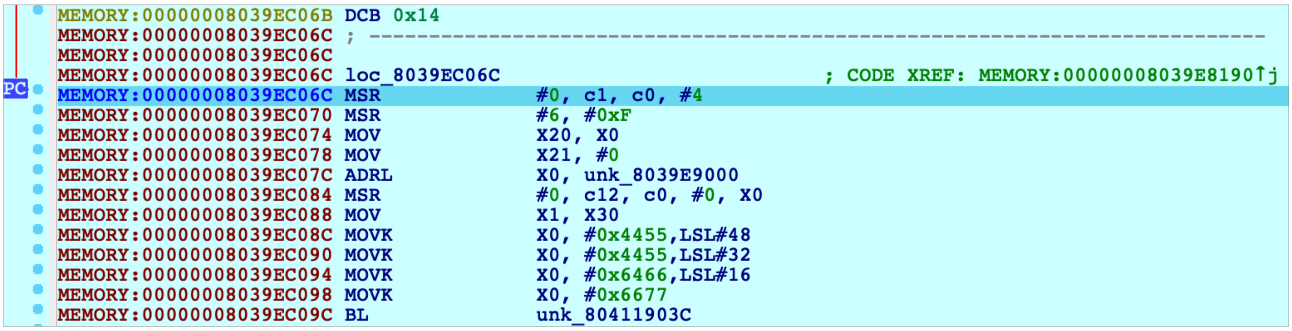


Now start IDA with an empty database and attach to the suspended VM:



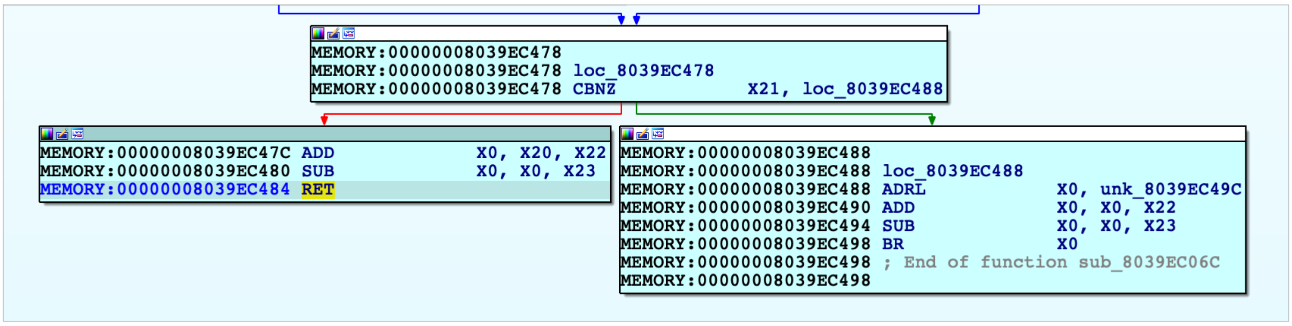
```
MEMORY:00000008039E818F DCB    0
MEMORY:00000008039E8190 ; ---------------------------------------------------------------------------
MEMORY:00000008039E8190 B                 loc_8039EC06C
MEMORY:00000008039E8190 ; ---------------------------------------------------------------------------
MEMORY:00000008039E8194 DCB 0x1F
MEMORY:00000008039E8195 DCB 0x20
MEMORY:00000008039E8196 DCB    3
MEMORY:00000008039E8197 DCB 0xD5
MEMORY:00000008039E8198 DCB 0x1F
MEMORY:00000008039E8199 DCB 0x20
MEMORY:00000008039E819A DCB    3
```

From the XNU source, this is likely the **_start** symbol in osfmk/arm64/start.s, which simply branches to **start_first_cpu**. After stepping over this branch:



```
MEMORY:00000008039EC06B DCB 0x14
MEMORY:00000008039EC06C ; ---------------------------------------------------------------------------
MEMORY:00000008039EC06C
MEMORY:00000008039EC06C loc_8039EC06C                           ; CODE XREF: MEMORY:00000008039E8190↑j
MEMORY:00000008039EC06C MSR             #0, c1, c0, #4
MEMORY:00000008039EC070 MSR             #6, #0xF
MEMORY:00000008039EC074 MOV             X20, X0
MEMORY:00000008039EC078 MOV             X21, #0
MEMORY:00000008039EC07C ADRL            X0, unk_8039E9000
MEMORY:00000008039EC084 MSR             #0, c12, c0, #0, X0
MEMORY:00000008039EC088 MOV             X1, X30
MEMORY:00000008039EC08C MOVK            X0, #0x4455,LSL#48
MEMORY:00000008039EC090 MOVK            X0, #0x4455,LSL#32
MEMORY:00000008039EC094 MOVK            X0, #0x6466,LSL#16
MEMORY:00000008039EC098 MOVK            X0, #0x6677
MEMORY:00000008039EC09C BL              unk_80411903C
```

Press shortcut **P** to analyze **start_first_cpu**. This is where the kernel performs its initial setup (note that the value in X0 is a pointer to the boot_args structure). This function is interesting because it is responsible for switching the kernel to 64-bit virtual addressing. Typically the switch happens when this function sets X30 to a virtual address, then performs a **RET**:

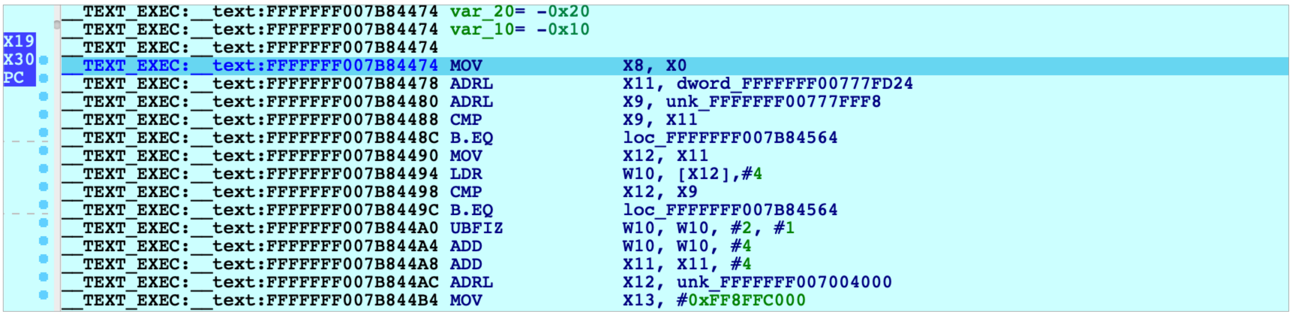


```
MEMORY:00000008039EC478
MEMORY:00000008039EC478 loc_8039EC478
MEMORY:00000008039EC478 CBNZ            X21, loc_8039EC488
```

```
MEMORY:00000008039EC47C ADD             X0, X20, X22
MEMORY:00000008039EC480 SUB             X0, X0, X23
MEMORY:00000008039EC484 RET
```

```
MEMORY:00000008039EC488
MEMORY:00000008039EC488 loc_8039EC488
MEMORY:00000008039EC488 ADRL            X0, unk_8039EC49C
MEMORY:00000008039EC490 ADD             X0, X0, X22
MEMORY:00000008039EC494 SUB             X0, X0, X23
MEMORY:00000008039EC498 BR              X0
MEMORY:00000008039EC498 ; End of function sub_8039EC06C
MEMORY:00000008039EC498
```

Use F4 to run to this RET instruction. In this example X30 will now point to virtual address **0xFFFFFFF007B84474**. After single stepping once more, we end up in **arm_init** in virtual memory:



```
__TEXT_EXEC:__text:FFFFFFF007B84474 var_20= -0x20
__TEXT_EXEC:__text:FFFFFFF007B84474 var_10= -0x10
__TEXT_EXEC:__text:FFFFFFF007B84474
__TEXT_EXEC:__text:FFFFFFF007B84474 MOV             X8, X0
__TEXT_EXEC:__text:FFFFFFF007B84478 ADRL            X11, dword_FFFFFFF00777FD24
__TEXT_EXEC:__text:FFFFFFF007B84480 ADRL            X9, unk_FFFFFFF00777FFF8
__TEXT_EXEC:__text:FFFFFFF007B84488 CMP             X9, X11
__TEXT_EXEC:__text:FFFFFFF007B8448C B.EQ            loc_FFFFFFF007B84564
__TEXT_EXEC:__text:FFFFFFF007B84490 MOV             X12, X11
__TEXT_EXEC:__text:FFFFFFF007B84494 LDR             W10, [X12],#4
__TEXT_EXEC:__text:FFFFFFF007B84498 CMP             X12, X9
__TEXT_EXEC:__text:FFFFFFF007B8449C B.EQ            loc_FFFFFFF007B84564
__TEXT_EXEC:__text:FFFFFFF007B844A0 UBFIZ           W10, W10, #2, #1
__TEXT_EXEC:__text:FFFFFFF007B844A4 ADD             W10, W10, #4
__TEXT_EXEC:__text:FFFFFFF007B844A8 ADD             X11, X11, #4
__TEXT_EXEC:__text:FFFFFFF007B844AC ADRL            X12, unk_FFFFFFF007004000
__TEXT_EXEC:__text:FFFFFFF007B844B4 MOV             X13, #0xFF8FFC000
```

After this single step, IDA detected that execution reached the kernel's virtual address space and automatically initialized the debugging environment. In this case a message will be printed to the console:

```
FFFFFFF007004000: process kernel has started
```

This signifies that IDA successfully detected the kernel base and created a new module in the Modules list. If the kernel has a symbol table, debug names will be available. Also note that PC now points inside the segment **__TEXT_EXEC:__text** instead of **MEMORY**, because the debugger parsed the kernel's load commands to generate proper debug segments.

Now that we know the address of **arm_init**, we can streamline this task:

1. power on the device with **Start device paused**
2. attach to the paused VM
3. set a hardware breakpoint at **arm_init**:

   ```
   idaapi.add_bpt(0xFFFFFFF007B84474, 1, BPT_EXEC)
   ```

4. resume, and wait for the breakpoint to be hit

This gives us a quick way to break at the first instruction executed in virtual memory. You can continue debugging iOS as normal.

6. Known Issues and Limitations

Here is a list of known shortcomings in the XNU Debugger. Eventually we will address all of them, but it is unlikely we will resolve all of them by the next release. If any of the following topics are important to you, please let us know by sending an email to support@hex-rays.com. Issues with vocal support from our users are automatically prioritised.

6.1. iBoot debugging

Debugging the iOS firmware/bootloader is not yet supported. An On-Premise Corellium box is required for this functionality, so we will only implement it if there is significant demand.

6.2. 32-bit XNU

The XNU Debugger does not support debugging 32-bit XNU. Since pure 32-bit OSes are quite outdated it is unlikely we will support them unless there is exceptional demand.

6.3. KDP

The XNU Debugger relies on the Remote GDB Protocol, and currently Apple's Kernel Debugging Protocol (KDP) is not supported. It is possible to add KDP support to IDA in the future.