

原文：<https://geosn0w.github.io/Debugging-macOS-Kernel-For-Fun/>

内核调试是一件非常有趣的事情，但是，这件事情却并不简单——尤其是对于Apple来说。当然，网络上面已经有许多这方面的文章了，然而，随着时间的推移，许多内容已

在MacOS上开始内核调试

首先，我们要搭建一个实验环境。为此，我们需要有一个要调试其内核的设备（在本文中，以iMac 2011作为调试对象），以及一个要在其上进行调试的设备（在本文中，为MacBook Pro 2009）。当然，我们可以用本文中介绍的各种方式将两者连接起来，但对我来说，最合适的方法（也是最可靠的方法），似乎是通过两者之间的Firewire电缆进行连接（这

硬件配置好之后，我们需要准备一些软件。从理论上说，我们可以调试RELEASE版内核，但是对于初学者时，调试Development版的内核要更合适一些。默认情况下，MacOS提供64-bit executable x86_64文件。我们可以访问Apple Developer页面，并下载内核调试工具包，这样就可以获得MacOS版本的Development内核了。

实际上，只要导航至[Apple Developer Portal Downloads](#)，将看到如下所示的内容：

More Downloads for Apple Developers

Hi, John Smith ▾

CATEGORIES

- Developer Tools 420
- macOS 184
- macOS Server 9
- Applications 12
- iOS 10
- Safari 1

Description	Release Date
+ Kernel Debug Kit 10.14.1 build 18B75	Nov 16, 2018
+ Kernel Debug Kit 10.12.6 build 16G1618	Nov 16, 2018
+ Kernel Debug Kit 10.13.6 build 17G3025.dmg	Nov 16, 2018
+ Kernel Debug Kit 10.12.6 build 16G1707	Nov 7, 2018
+ Kernel Debug Kit 10.14.2 build 18C38b	Nov 7, 2018
+ Kernel Debug Kit 10.13.6 build 17G4008	Nov 7, 2018
+ Kernel Debug Kit 10.14.2 build 18C31g	Nov 1, 2018
+ Kernel Debug Kit 10.13.6 build 17G4005	Oct 31, 2018
+ Kernel Debug Kit 10.12.6 build 16G1704	Oct 31, 2018
+ Kernel Debug Kit 10.14.1 build 18B73a	Oct 29, 2018
+ Kernel Debug Kit 10.14.1 build 18B67a	Oct 19, 2018
+ Kernel Debug Kit 10.14.1 build 18B50c	Oct 2, 2018
+ Kernel Debug Kit 10.14.1 build 18B45d	Sep 25, 2018
+ Kernel Debug Kit 10.14 build 18A391	Sep 24, 2018
+ Kernel Debug Kit 10.14 build 18A389	Sep 12, 2018
+ Kernel Debug Kit 10.14 build 18A384a	Sep 9, 2018
+ Kernel Debug Kit 10.14 build 18A377a	Aug 27, 2018
+ Kernel Debug Kit 10.14 build 18A371a	Aug 20, 2018
+ Kernel Debug Kit 10.14 build 18A353d	Aug 6, 2018

1-19 of 123

需要注意的是，我们需要根据自己的MacOS版本来下载相应的内核调试工具包！之后，我们就可以引导下载的内核了，如果内核与我们的MacOS版本不匹配的话，将无法正

查找适合自己MacOS版本的内核调试工具包

为了找到正确的内核调试工具包，您必须知道您的MacOS版本和实际构建编号（actual build number）。要想了解当前所用的MacOS版本，只需转至Apple logo处，单击“About This Mac”，就能从窗口中看到相应的版本（例如，我的系统为“Version 10.13.6”）。

为了获取实际构建编号，既可以单击“About This Mac”窗口中的“Version”标签，也可以运行终端命令`sw_vers | grep BuildVersion`。对于我的系统来说，运行命令后，输出结果为`buildversion:17g65`。

```
Last login: Sun Dec  2 03:58:16 on ttys000
Isabella:~ geosn0w$ sw_vers | grep BuildVersion
BuildVersion:    17G65
Isabella:~ geosn0w$
```

因此，就我来说，运行的是MacOS High Sierra(10.13.6) build number 17G65。这样，就可以根据自己的系统版本下载合适的、包含安装文件的.DMG文件了。

More Downloads for Apple Developers

Hi, John Smith

Kernel Debug Kit

CATEGORIES

Developer Tools

macOS

macOS Server

Applications

iOS

Safari

420

184

9

12

10

1

Description	Release Date
+ Kernel Debug Kit 10.14 build 18A389	Sep 12, 2018
+ Kernel Debug Kit 10.14 build 18A384a	Sep 9, 2018
+ Kernel Debug Kit 10.14 build 18A377a	Aug 27, 2018
+ Kernel Debug Kit 10.14 build 18A371a	Aug 20, 2018
+ Kernel Debug Kit 10.14 build 18A353d	Aug 6, 2018
+ Kernel Debug Kit 10.14 build 18A347e	Jul 30, 2018
+ Kernel Debug Kit 10.13.6 build 17G2208	Jul 24, 2018
+ Kernel Debug Kit 10.14 build 18A336e	Jul 18, 2018
- Kernel Debug Kit 10.13.6 build 17G65	Jul 9, 2018
<div>This package contains development & debug versions of the macOS kernel and many I/O Kit families for use with LLDB remote (two-machine) kernel debugging. These files contain full symbolic information, unlike the equivalent files in a normal macOS installation. Also included are LLDB macros useful for kernel debugging and DEVELOPMENT (for every day use) and DEBUG (for much more error checking) kernels built with more assertions and fewer compiler optimizations.</div>	
+ Kernel Debug Kit 10.14 build 18A326g	Jul 4, 2018
+ Kernel Debug Kit 10.13.6 build 17G62a	Jul 2, 2018
+ Kernel Debug Kit 10.14 build 18A314h	Jun 26, 2018
+ Kernel Debug Kit 10.14 build 18A314k	Jun 26, 2018
+ Kernel Debug Kit 10.13.6 build 17G47b	Jun 18, 2018

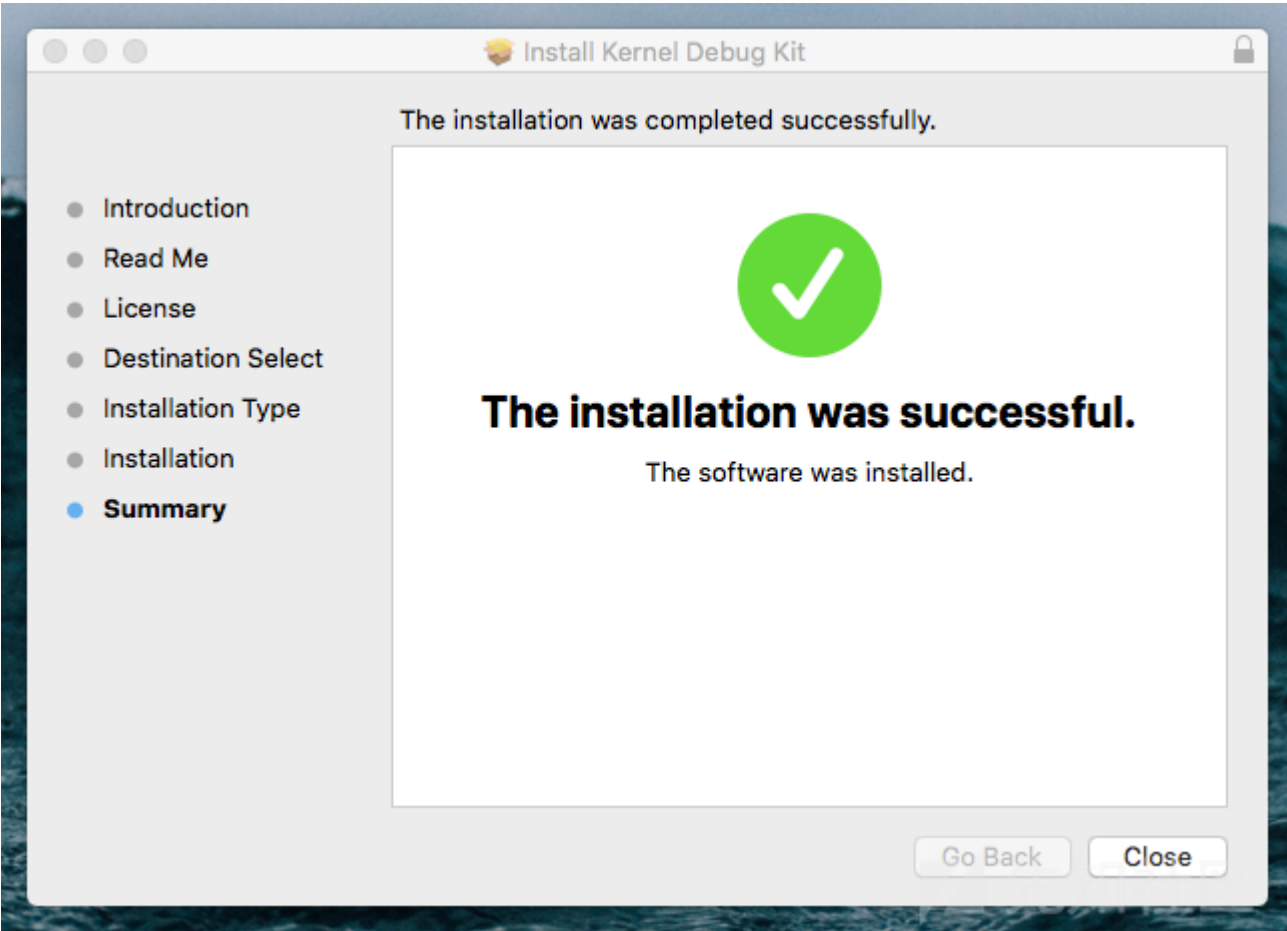
Kernel Debug Kit 10.13.6 build 17G65.dmg

69.7 MB

15-33 of 123

准备调试对象以供调试器调试

将调试工具包下载到调试对象（即要调试其内核的计算机）后，双击DMG文件即可挂载该DMG文件。在DMG中，您可以看到一个名为kerneldebugkit.pkg的文件。双击该文件，当安装完成时，它将如下所示。



安装完成后，请导航到/library/developer/kdks。在那里，有一个名为kdk_your_version_buildnumber.kdk的文件夹。对于我来说，该文件夹名为kdk_10.13.6_17g65.kdk。然后，请将kernel.development复制粘贴到/system/library/kernels/下面，跟RELEASE内核二进制文件放在一起。完成上述操作后，MacOS上就有了两个内核，一个是

为了正确调试，需要在要调试其内核的计算机上禁用SIP（系统完整性保护）。为此，重新启动计算机并进入恢复模式。为此，请重新启动计算机，当听到“boong！”的声音Mode用户界面，然后，从顶部栏打开“Terminal”。

在Recovery Terminal中，运行csrutil disable命令。然后重新启动计算机，并使其正常启动。

自2018/2019年起设置NVRAM boot-args的正确方法

这些年来，苹果已经改变了其boot-args，所以，我们在互联网上找到的相关内容可能有用，也可能没用，这主要取决于文章的发表的时间。截至2018年，以下boot-args在High Sierra上面测试通过。

注意！以下boot-args假设通过Firewire，或通过基于Thunderbolt适配器的Firewire来执行该操作的。

如果您通过物理FireWire端口（较旧的Mac机器）使用FireWire电缆的话：

在Terminal中运行以下命令：

```
sudo nvram boot-args="debug=0x8146 kdp_match_name=firewire fwdebug=0x40 pmuflags=1 -v"
```

如果您通过ThunderBolt适配器使用FireWire：

在Terminal中运行以下命令：

```
sudo nvram boot-args="debug=0x8146 kdp_match_name=firewire fwkdp=0x8000 fwdebug=0x40 pmuflags=1 -v"
```

区别在于fwkdp=0x8000，起作用是让ioFireWireFamily.kext::applefwohci_kdp使用非内置FireWire<->Thunderbolt适配器进行调试会话。

差不多就是这样了，调试对象在重新启动之后就可以调试了，下面，让我来解释一下启动参数的作用。

- debug=0x8146 ->启用调试，并允许我们按下电源按钮来触发NMI（表示不可屏蔽中断），以允许调试器连接。
- kdp_match_name=firewire ->这允许我们通过Firewire进行调试。
- fwkdp=0x8000 ->如前所述，这告诉KEXT使用Thunderbolt至FireWire适配器。如果使用普通FireWire端口，则不要设置该参数。
- fwdebug=0x40 ->启用applefwohci_kdp驱动程序的详细输出，这对于故障排除非常有用。
- pmuflags=1 ->该选项禁用看门狗定时器。
- -v -> 这一选项告诉计算机启动冗长，而不是正常的苹果标志和进度条，这对于排除故障非常有用。

除了我们设置的这些引导参数之外，MacOS还支持在/osfmk/kern/debug.h中定义的其他参数，我将在下面列出这些参数。这些截图来自XNU-4570.41.2。

```
...
/* Debug boot-args */
#define DB_HALT          0x1
// #define DB_PRT          0x2 -- obsolete
#define DB_NMI           0x4
#define DB_KPRT          0x8
#define DB_KDB           0x10
#define DB_ARP           0x40
#define DB_KDP_BP_DIS    0x80
// #define DB_LOG_PI_SCRN  0x100 -- obsolete
#define DB_KDP_GETC_ENA  0x200

#define DB_KERN_DUMP_ON_PANIC      0x400 /* Trigger core dump on panic */
#define DB_KERN_DUMP_ON_NMI       0x800 /* Trigger core dump on NMI */
#define DB_DBG_POST_CORE          0x1000 /* Wait in debugger after NMI core */
#define DB_PANICLOG_DUMP          0x2000 /* Send paniclog on panic, not core */
#define DB_REBOOT_POST_CORE       0x4000 /* Attempt to reboot after
    * post-panic crashdump/paniclog
    * dump.
    */

#define DB_NMI_BTN_ENA            0x8000 /* Enable button to directly trigger NMI */
#define DB_PRT_KDEBUG             0x10000 /* kprintf KDEBUG traces */
#define DB_DISABLE_LOCAL_CORE     0x20000 /* ignore local kernel core dump support */
#define DB_DISABLE_GZIP_CORE      0x40000 /* don't gzip kernel core dumps */
#define DB_DISABLE_CROSS_PANIC    0x80000 /* x86 only - don't trigger cross panics. Only
    * necessary to enable x86 kernel debugging on
    * configs with a dev-fused co-processor running
    * release bridgeOS.
    */

#define DB_REBOOT_ALWAYS          0x100000 /* Don't wait for debugger connection */
```

...

准备调试器计算机

好了，现在调试对象已经准备好了，我们需要配置运行调试器的计算机。为此，我将使用另一台运行El Capitan的MacOS机器，但这无关紧要。还记得我们在调试对象上安装的内核调试工具包吗？我们也需要将它安装到调试器计算机上。不同之处在于，我们不会鼓捣其内核。

注意：您应该在调试器所在机器上安装相同的MacOS内核调试工具包，即使该机器的MacOS版本不同于调试对象的macOS版本，这是因为，我们不会在调试器上启动任何。

安装工具包后，就可以连接了。

调试内核

首先，请重新启动调试对象。您将看到它引导进入一个文本模式控制台，该控制台会输出详细的引导信息。等到屏幕上显示“DSMOS has arrived!”消息后，需要按一次电源按钮，但是，不要按住不动。在调试对象上，您将看到它正在等待连接调试器完成。

在调试器计算机上：

打开终端窗口，并启动fwkdp-v，这是FireWire KDP

Tool，它将侦听FireWire接口并将数据重定向到本地主机，以便我们可以将KDP目标设置为localhost或127.0.0.1地址。这时，我们应该得到类似下面的输出：

```
MacBook-Pro-van-Mac:~ mac$ fwkdp -v
FireWire KDP Tool (v1.6)
Matched on device 0x00002403
Created plugin interface 0x7f9e50c03548 with result 0x00000000
Created device interface 0x7f9e50c0d508 with result 0x00000000
Opened device interface 0x7f9e50c0d508 with result 0x00000000
Added callback dispatcher with result 0x00000000
Created pseudo address space 0x7f9e50c0d778 at 0xf0430000
Address space enabled.
2018-12-02 05:51:05.453 fwkdp[5663:60796] CFSocketSetAddress listen failure: 102
Created KDP socket listener 0x7f9e50c0d940 with result 0
KDP Proxy and CoreDump-Receive dual mode active.
Use 'localhost' as the KDP target in gdb.
Ready.
```

现在，在不关闭该窗口的情况下，打开另一个终端窗口，并通过将kernel.development文件作为内核调试工具包的一部分传递给它，以启动LLDB调试器。请记住，内核文件在kernel.development。

因此，就我来说，需要在新终端窗口中执行的命令是xcrun lldb/library/developer/kdks/kdk_10.13.6_17G65.kdk/system/library/kernels/kernel.development。

```
Last login: Sun Dec  2 10:37:51 on ttys000
MacBook-Pro-van-Mac:~ mac$ xcrun lldb /Library/Developer/KDKs/KDK_10.13.6_17G65.kdk/System/Library/Kernels/kernel.development
(lldb) target create "/Library/Developer/KDKs/KDK_10.13.6_17G65.kdk/System/Library/Kernels/kernel.development"
warning: 'kernel' contains a debug script. To run this script in this debug session:
command script import "/Library/Developer/KDKs/KDK_10.13.6_17G65.kdk/System/Library/Kernels/kernel.development.dSYM/Contents/R
```

To run all discovered debug scripts in this session:

```
settings set target.load-script-from-symbol-file true
```

Current executable set to '/Library/Developer/KDKs/KDK_10.13.6_17G65.kdk/System/Library/Kernels/kernel.development' (x86_64).

正如您所看到的，LLDB指出“kernel”中含有一个调试脚本。在当前打开的LLDB窗口中，执行settings set target.load-script-from-symbol-file true命令，以运行该脚本。

```
Last login: Sun Dec  2 10:37:51 on ttys000
MacBook-Pro-van-Mac:~ mac$ xcrun lldb /Library/Developer/KDKs/KDK_10.13.6_17G65.kdk/System/Library/Kernels/kernel.development
(lldb) target create "/Library/Developer/KDKs/KDK_10.13.6_17G65.kdk/System/Library/Kernels/kernel.development"
warning: 'kernel' contains a debug script. To run this script in this debug session:
command script import "/Library/Developer/KDKs/KDK_10.13.6_17G65.kdk/System/Library/Kernels/kernel.development.dSYM/Contents/R
```

To run all discovered debug scripts in this session:

```
settings set target.load-script-from-symbol-file true
```

Current executable set to '/Library/Developer/KDKs/KDK_10.13.6_17G65.kdk/System/Library/Kernels/kernel.development' (x86_64).

```
(lldb) settings set target.load-script-from-symbol-file true
```

```
Loading kernel debugging from /Library/Developer/KDKs/KDK_10.13.6_17G65.kdk/System/Library/Kernels/kernel.development.dSYM/Contents/Resources/Symbolic/LLDB version lldb-360.1.70
settings set target.process.python-os-plugin-path "/Library/Developer/KDKs/KDK_10.13.6_17G65.kdk/System/Library/Kernels/kernel.development.dSYM/Contents/Resources/Symbolic/PythonOSPlugin.py"
settings set target.trap-handler-names hndl_allintrs hndl_alltraps trap_from_kernel hndl_double_fault hndl_machine_check _fleh
command script import "/Library/Developer/KDKs/KDK_10.13.6_17G65.kdk/System/Library/Kernels/kernel.development.dSYM/Contents/Resources/Symbolic/xnu_debug_macros.py"
xnu debug macros loaded successfully. Run showlldbtypesummaries to enable type summaries.
```

```
settings set target.process.optimization-warnings false
(lldb)
```

现在，我们终于可以通过kdp-remote

localhost命令将LLDB连接到实时内核了。如果一切顺利的话，连接内核成功后，将看到如下所示的输出。刚开始的时候，LLDB窗口会冒出大量的文本，然后，它会停下来。

```
(lldb) kdp-remote localhost
Version: Darwin Kernel Version 17.7.0: Wed Oct 10 23:06:14 PDT 2018; root:xnu-4570.71.13~1/DEVELOPMENT_x86_64; UUID=1718D865-98B4-3F6E-97CF-42BF0D02ADD7
Kernel UUID: 1718D865-98B4-3F6E-97CF-42BF0D02ADD7
Load Address: 0xffffffff802e800000
Kernel slid 0x2e600000 in memory.
Loaded kernel file /Library/Developer/KDKs/KDK_10.13.6_17G3025.kdk/System/Library/Kernels/kernel.development
Loading 152 kext modules warning: Can't find binary/dSYM for com.apple.kec.Libm (BC3F7DA4-03EA-30F7-B44A-62C249D51C10)
.warning: Can't find binary/dSYM for com.apple.kec.corecrypto (B081B8C1-1DFF-342F-8DF2-C3AA925ECA3A)
.warning: Can't find binary/dSYM for com.apple.kec.pthread (E64F7A49-CBF0-3251-9F02-3655E3B3DD31)
.warning: Can't find binary/dSYM for com.apple.iokit.IOACPIFamily (95DA39BB-7C39-3742-A2E5-86C555E21D67)
[...]
.Target arch: x86_64
.. done.
Target arch: x86_64
Instantiating threads completely from saved state in memory.
Process 1 stopped
* thread #2: tid = 0x0066, 0xffffffff802e97a8d3 kernel.development`DebuggerWithContext [inlined] current_cpu_datap at cpu_data.h:401 [opt]
  frame #0: 0xffffffff802e97a8d3 kernel.development`DebuggerWithContext [inlined] current_cpu_datap at cpu_data.h:401 [opt]
```

现在我们已经连接到了实时内核。您可以看到，该进程已终止，这意味着内核已冻结，这就是上面看到停止涌出文本的原因，但现在调试器已附加好了，所以，我们就可以让进程继续运行了。

```
(lldb) c
Process 1 resuming
Process 1 stopped
* thread #2: tid = 0x0066, 0xffffffff802e97a8d3 kernel.development`DebuggerWithContext [inlined] current_cpu_datap at cpu_data.h:401 [opt]
  frame #0: 0xffffffff802e97a8d3 kernel.development`DebuggerWithContext [inlined] current_cpu_datap at cpu_data.h:401 [opt]
(lldb) c
```

一旦调试器所在系统引导进入MacOS，我们会进入桌面，这时我们就可以开始我们的调试了。若要运行调试器命令，我们必须再次触发NMI，为此，需按一次电源按钮。

内核调试实例

示例1：使用LLDB读取所有寄存器的值，并将“AAAAAAA”写入其中一个寄存器中。

要读取所有寄存器，需按下电源按钮以触发NMI，并在打开的LLDB窗口中键入register read --all命令。

```
(lldb) register read --all
General Purpose Registers:
  rax = 0xffffffff802f40ba40  kernel.development`processor_master
  rbx = 0x0000000000000000
  rcx = 0xffffffff802f40ba40  kernel.development`processor_master
  rdx = 0x0000000000000000
  rdi = 0x0000000000000004
  rsi = 0xffffffff7fb1483ff4
  rbp = 0xffffffff817e8ccd50
  rsp = 0xffffffff817e8ccd10
  r8 = 0x0000000000000000
  r9 = 0x0000000000000001
  r10 = 0x000000000000004d1
  r11 = 0x000000000000004d0
  r12 = 0x0000000000000000
  r13 = 0x0000000000000000
  r14 = 0x0000000000000000
  r15 = 0xffffffff7fb1483ff4
```

```
rip = 0xffffffff802e97a8d3 kernel.development`DebuggerWithContext + 403 [inlined] current_cpu_datap at cpu.c:220
kernel.development`DebuggerWithContext + 403 [inlined] current_processor at debug.c:463
kernel.development`DebuggerWithContext + 403 [inlined] DebuggerTrapWithState + 46 at debug.c:537
kernel.development`DebuggerWithContext + 357 at debug.c:537
rflags = 0x0000000000000046
cs = 0x0000000000000008
fs = 0x0000000000000000
gs = 0x0000000000000000
```

Floating Point Registers:

```
fcw = 0x0000
fsw = 0x0000
ftw = 0x00
fop = 0x0000
ip = 0x00000000
cs = 0x0000
dp = 0x00000000
ds = 0x0000
mxcsr = 0x00000000
mxcsrmask = 0x00000000
stmm0 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
stmm1 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
stmm2 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
stmm3 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
stmm4 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
stmm5 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
stmm6 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
stmm7 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
xmm0 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
xmm1 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
xmm2 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
xmm3 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
xmm4 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
xmm5 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
xmm6 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
xmm7 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
xmm8 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
xmm9 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
xmm10 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
xmm11 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
xmm12 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
xmm13 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
xmm14 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
xmm15 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
```

Exception State Registers:

```
3 registers were unavailable.
(lldb)
```

现在，让我们对其中一个寄存器执行写操作。注意，不要向其值为0x0000000000000000的寄存器中写入内容，因为这会覆盖某些有用的内容。所以，我们需要找一个空write r13 0x4141414141414141。现在，如果我们再次读取寄存器的内容，就会发现其中的变化情况：

```
(lldb) register write R13 0x4141414141414141
(lldb) register read --all
```

General Purpose Registers:

```
rax = 0xffffffff802f40ba40 kernel.development`processor_master
rbx = 0x0000000000000000
rcx = 0xffffffff802f40ba40 kernel.development`processor_master
rdx = 0x0000000000000000
rdi = 0x0000000000000004
rsi = 0xffffffff7fb1483ff4
rbp = 0xffffffff817e8ccd50
rsp = 0xffffffff817e8ccd10
r8 = 0x0000000000000000
r9 = 0x0000000000000001
r10 = 0x000000000000004d1
r11 = 0x000000000000004d0
r12 = 0x0000000000000000
r13 = 0x4141414141414141 <-- Yee overwritten this.
```

```

    r14 = 0x0000000000000000
    r15 = 0xffffffff7fb1483ff4
    rip = 0xffffffff802e97a8d3 kernel.development`DebuggerWithContext + 403 [inlined] current_cpu_datap at cpu.c:220
kernel.development`DebuggerWithContext + 403 [inlined] current_processor at debug.c:463
kernel.development`DebuggerWithContext + 403 [inlined] DebuggerTrapWithState + 46 at debug.c:537
kernel.development`DebuggerWithContext + 357 at debug.c:537
    rflags = 0x00000000000000046
    cs = 0x0000000000000008
    fs = 0x0000000000000000
    gs = 0x0000000000000000
```

Floating Point Registers:

```

    fcw = 0x0000
    fsw = 0x0000
    ftw = 0x00
    fop = 0x0000
    ip = 0x00000000
    cs = 0x0000
    dp = 0x00000000
    ds = 0x0000
    mxcsr = 0x00000000
mxcsrmask = 0x00000000
    stmm0 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
    stmm1 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
    stmm2 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
    stmm3 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
    stmm4 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
    stmm5 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
    stmm6 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
    stmm7 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
    xmm0 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
    xmm1 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
    xmm2 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
    xmm3 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
    xmm4 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
    xmm5 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
    xmm6 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
    xmm7 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
    xmm8 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
    xmm9 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
    xmm10 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
    xmm11 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
    xmm12 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
    xmm13 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
    xmm14 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
    xmm15 = {0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00}
```

Exception State Registers:

3 registers were unavailable.

(lldb)

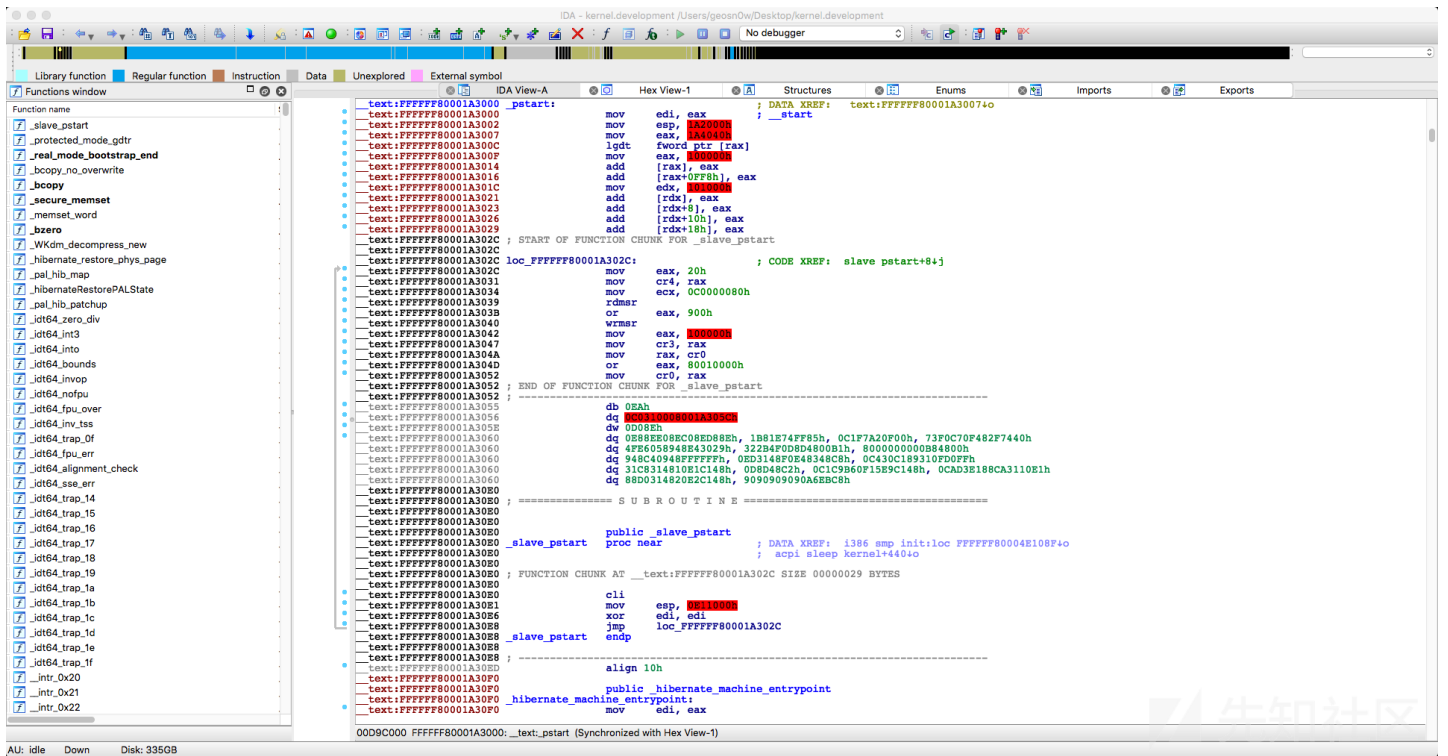
注意：当我们想要读取单个寄存器时，不必运行register read --all命令，相反，我们可以用 register read [register]形式的命令来指定寄存器，例如register read r13。

示例2：通过运行uname -a来修改内核的版本和名称

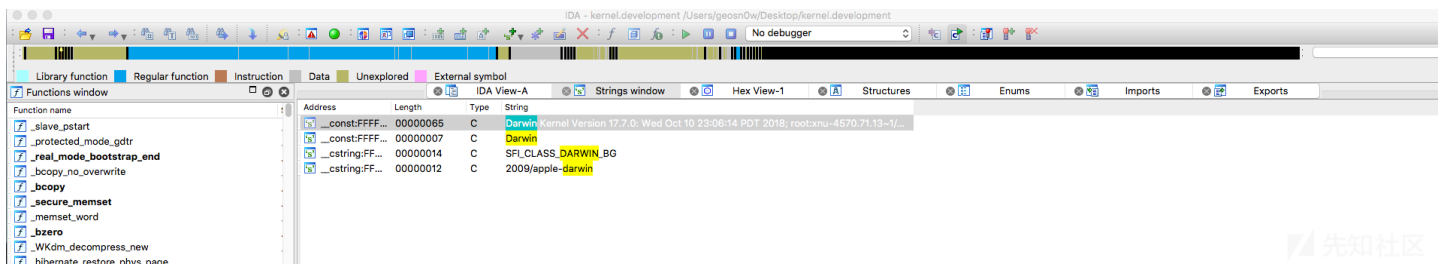
下面，我们将对内核进行一些真正的内存读写操作。大家可能知道，通过在终端中输入命令uname -a可以显示内核的名称、版本和构建日期。那么，我们是否可以按需修改这些内容呢？

首先，我们不知道内核将这些信息存储在哪里，所以，我们需要找到这些地址信息。为此，我们可以借助反汇编程序，如IDA Pro、Hopper Disassembler、JTool、Binary Ninja等等。

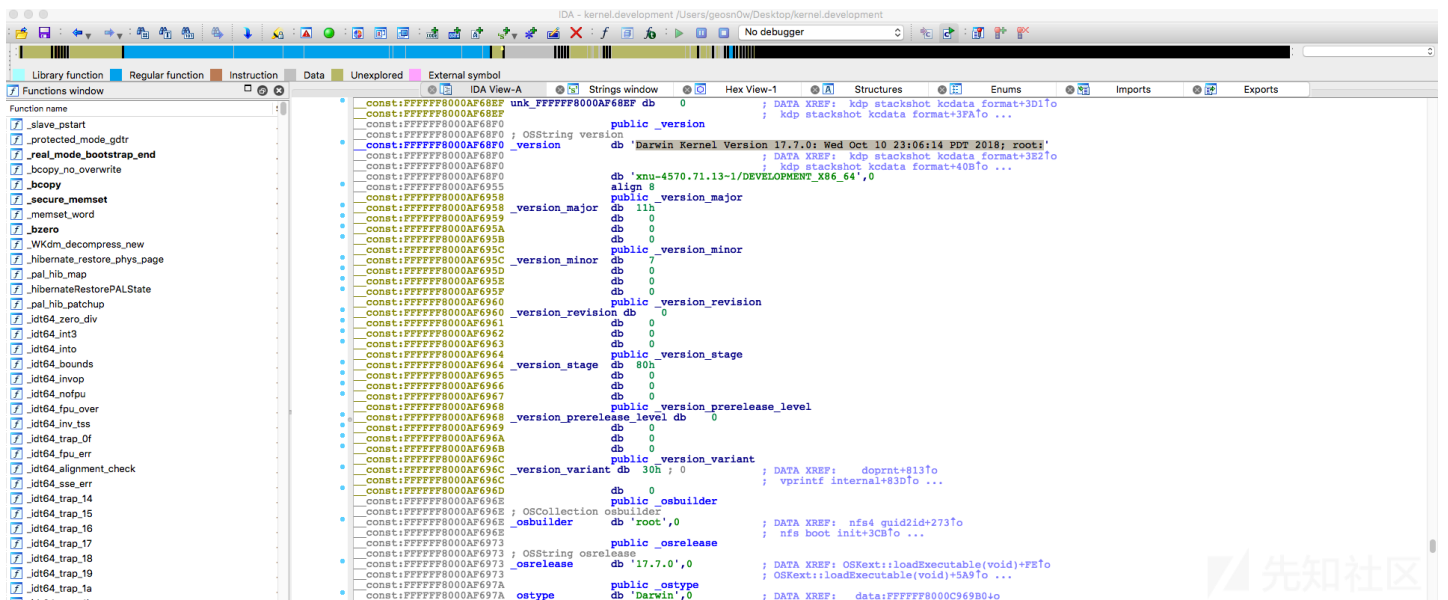
在本例中，我将使用IDA Pro来完成这项任务。为此，需要首先将kernel.development文件加载到IDA Pro中，以便让IDA完成相应的分析工作。当然，这个分析过程可能需要一段时间，所以，还需要大家请耐心等待。毕竟，内核的块头可不小。当IDA完成分析工作后，将输出idle”时，表明分析工作已经完成了。



现在，我们必须找到该字符串。在终端执行uname -a命令时，我们看到内核名为darwin，因此，为了在IDA中找到它，需选择顶部的bar->view->open subviews->strings选项。这时，将出现一个新的Strings窗口，在该窗口中按Ctrl+F组合键，搜索框将出现在窗口的底部，这样就可以从中搜索Darwin了。



双击找到的字符串，将会将我们重定向到一个名为version的常量那里。现在，我们就能找到版本号了。这个常量叫做“version”，这就是我们要寻找的目标。也许您喜欢从ID



获取“version”常量的地址

其实，事情很简单：按下电源按钮以触发NMI（如果您已经让该进行继续运行的话），并键入print &(version)命令即可。

```
(lldb) print &(version)
(const char (*)[101]) $8 = 0xfffff802f0f68f0
(lldb)
```


AHAM ! 就本例来说, const char version位于地址0xfffff802f0f68f0处。所以, 如果我们显示该处的字符数组的话, 将会看到:

```
(lldb) print version
(const char [101]) $9 = {
  [0] = 'D'
  [1] = 'a'
  [2] = 'r'
  [3] = 'w'
  [4] = 'i'
  [5] = 'n'
  [6] = ' '
  [7] = 'K'
  [8] = 'e'
  [9] = 'r'
  [10] = 'n'
  [11] = 'e'
  [12] = 'l'
  [13] = ' '
  [14] = 'V'
  [15] = 'e'
  [16] = 'r'
  [17] = 's'
  [18] = 'i'
  [19] = 'o'
  [20] = 'n'
  [21] = ' '
  [22] = 'l'
  [23] = '7'
  [24] = '.'
  [25] = '7'
  [26] = '.'
  [27] = '0'
  [28] = ':'
  [29] = ' '
  [30] = 'W'
  [31] = 'e'
  [32] = 'd'
  [33] = ' '
  [34] = 'O'
  [35] = 'c'
  [36] = 't'
  [37] = ' '
  [38] = 'l'
  [39] = '0'
  [40] = ' '
  [41] = '2'
  [42] = '3'
  [43] = ':'
  [44] = '0'
  [45] = '6'
  [46] = ':'
  [47] = '1'
  [48] = '4'
  [49] = ' '
  [50] = 'P'
  [51] = 'D'
  [52] = 'T'
  [53] = ' '
  [54] = '2'
  [55] = '0'
  [56] = '1'
  [57] = '8'
  [58] = ';'
  [59] = ' '
  [60] = 'x'
  [61] = 'O'
  [62] = 'O'
  [63] = 't'
  [64] = ':'
```

```

[65] = 'x'
[66] = 'n'
[67] = 'u'
[68] = '-'
[69] = '4'
[70] = '5'
[71] = '7'
[72] = '0'
[73] = '.'
[74] = '7'
[75] = '1'
[76] = '.'
[77] = '1'
[78] = '3'
[79] = '~'
[80] = '1'
[81] = '/'
[82] = 'D'
[83] = 'E'
[84] = 'V'
[85] = 'E'
[86] = 'L'
[87] = 'O'
[88] = 'P'
[89] = 'M'
[90] = 'E'
[91] = 'N'
[92] = 'T'
[93] = '_'
[94] = 'X'
[95] = '8'
[96] = '6'
[97] = '_'
[98] = '6'
[99] = '4'
[100] = '\0'
}
(lldb)

```

实际上，借助于x <address>命令，我们可以将该内存地址处的内容转储出来。那好，让我们开始下手吧。

```

(lldb) x 0xffffffff802f0f68f0
0xffffffff802f0f68f0: 44 61 72 77 69 6e 20 4b 65 72 6e 65 6c 20 56 65  Darwin Kernel Ve
0xffffffff802f0f6900: 72 73 69 6f 6e 20 31 37 2e 37 2e 30 3a 20 57 65  rsion 17.7.0: We
(lldb)

```

看起来，这些内容延续至地址0xffffffff802f0f6900处，所以，我们继续进行转储：

```

(lldb) x 0xffffffff802f0f6900
0xffffffff802f0f6900: 65 72 73 69 6f 6e 20 36 39 2e 30 30 20 57 65 65  rsion 17.7.0: We
0xffffffff802f0f6910: 64 20 4f 63 74 20 31 30 20 32 33 3a 30 36 3a 31  d Oct 10 23:06:1
(lldb)

```

太好了！看见44617277696E没？这是单词Darwin的十六进制表示形似。如果我们将其改为十六进制表示的“geosn0w”，就相当于修改了内核名称了。对于版本号的修改，为此，我们需要一个文本到十六进制的转换工具。不要担心，这类工具在网上随处可见，例如我用的[这个](#)。需要注意的是，如果写入的字符串太长的话，就会覆盖其他内容。

最后，我是用的十六进制数据是这样的：

```

47 65 6f 53 6e 30 77 20 4b 65 72 6e 65 6c 20 56 = "GeoSn0w Kernel V"

65 72 73 69 6f 6e 20 36 39 2e 30 30 20 57 65 65 = "ersion 69.00 Wee"

```

现在，我们还不能把它写到这两个地址，因为必须先在所有字符前添加“0x”。最终，它们变为下面的样子：

```

0x47 0x65 0x6f 0x53 0x6e 0x30 0x77 0x20 0x4b 0x65 0x72 0x6e 0x65 0x6c 0x20 0x56 = "GeoSn0w Kernel V"

0x65 0x72 0x73 0x69 0x6f 0x6e 0x20 0x36 0x39 0x2e 0x30 0x30 0x20 0x57 0x65 0x65 = "ersion 69.00 Wee"

```

现在，我们就可以将这些字节写入内存中了。下面，让我们先从第一个地址开始。就我而言，所用命令如下所示：

```
(lldb) memory write 0xffffffff802f0f68f0 0x47 0x65 0x6f 0x53 0x6e 0x30 0x77 0x20 0x4b 0x65 0x72 0x6e 0x65 0x6c 0x20 0x56
(lldb) x 0xffffffff802f0f68f0
0xffffffff802f0f68f0: 47 65 6f 53 6e 30 77 20 4b 65 72 6e 65 6c 20 56  GeoSn0w Kernel V
0xffffffff802f0f6900: 72 73 69 6f 6e 20 31 37 2e 37 2e 30 3a 20 57 65  rsion 17.7.0: We
(lldb)
```

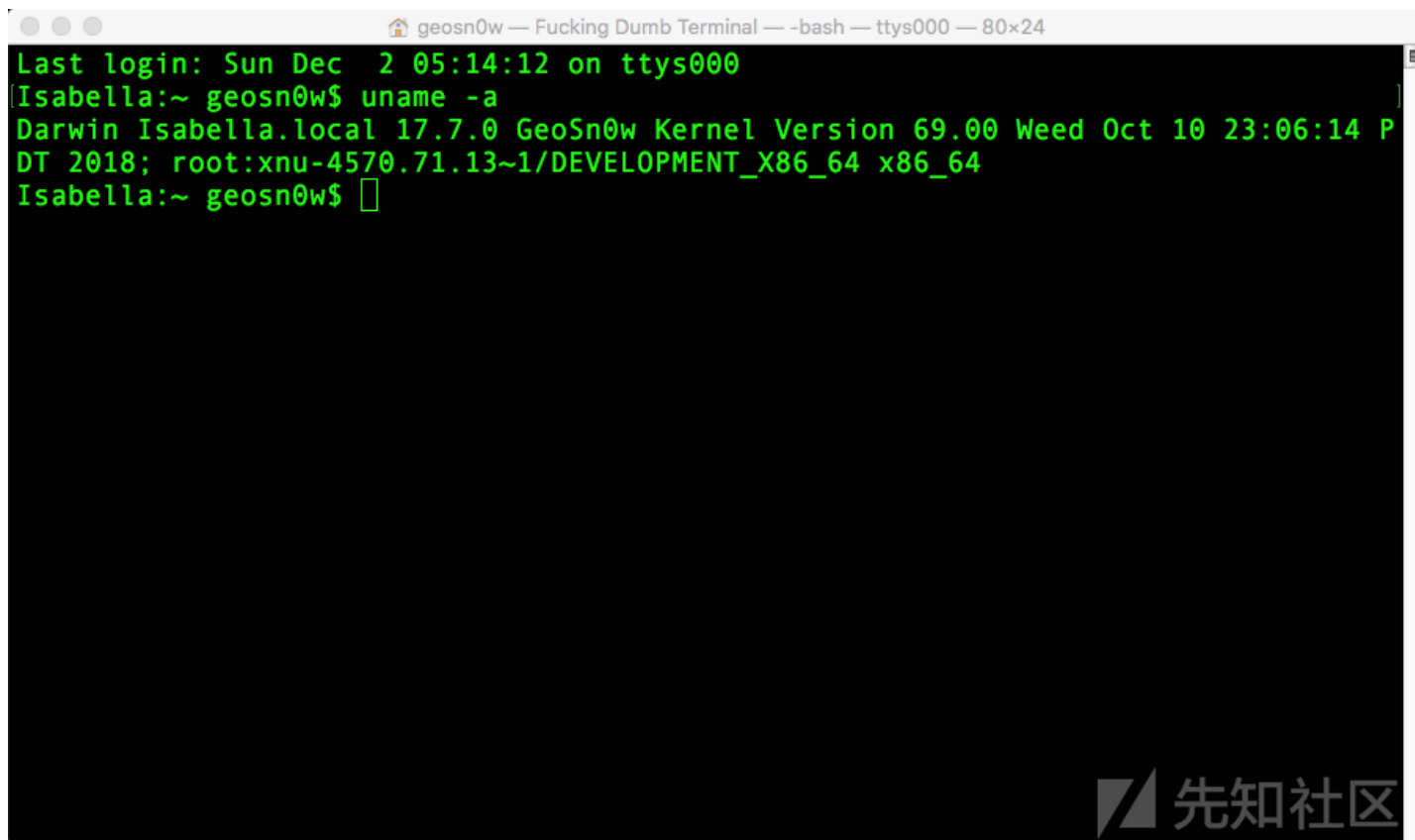
现在，我们已经将这个字符串完整地存放到了0xffffffff802f0f6900地址处：

```
(lldb) memory write 0xffffffff802f0f6900 0x65 0x72 0x73 0x69 0x6f 0x6e 0x20 0x36 0x39 0x2e 0x30 0x30 0x20 0x57 0x65 0x65
(lldb) x 0xffffffff802f0f6900
0xffffffff802f0f6900: 65 72 73 69 6f 6e 20 36 39 2e 30 30 20 57 65 65  version 69.00 Weed
0xffffffff802f0f6910: 64 20 4f 63 74 20 31 30 20 32 33 3a 30 36 3a 31  d Oct 10 23:06:1
(lldb)
```

现在，让我们解冻调试对象上的内核：

```
(lldb) c
Process 1 resuming
(lldb) Loading 1 kext modules warning: Can't find binary/dSYM for com.apple.driver.AppleXsanScheme (79D5E92F-789E-3C37-BE0E-7D
. done.
Unloading 1 kext modules . done.
Unloading 1 kext modules . done.
(lldb)
```

然后，让我们在调试对象的终端中运行uname -a命令：



```
geosn0w — Fucking Dumb Terminal — -bash — ttys000 — 80x24
Last login: Sun Dec  2 05:14:12 on ttys000
Isabella:~ geosn0w$ uname -a
Darwin Isabella.local 17.7.0 GeoSn0w Kernel Version 69.00 Weed Oct 10 23:06:14 P
DT 2018; root:xnu-4570.71.13~1/DEVELOPMENT_X86_64
Isabella:~ geosn0w$
```

现在，它将显示我们植入的字符串：

```
Last login: Sun Dec  2 07:12:19 on ttys000
Isabella:~ geosn0w$ uname -a
Darwin Isabella.local 17.7.0 GeoSn0w Kernel Version 69.00 Weed Oct 10 23:06:14 PDT 2018; root:xnu-4570.71.13~1/DEVELOPMENT_X86
Isabella:~ geosn0w$
```

好了，我们已经通过示例讲解了如何在macOS上进行内核调试，希望对大家有所帮助。需要注意的是，在完成调试之后，需要将boot-args重新设置为stock，这样才能启动nvram boot-args=""。然后，转至/System/Library/Kernels/目录下面，并删除kernel.development文件。

```
Isabella:~ geosn0w$ sudo nvram boot-args=""
Password:
Isabella:~ geosn0w$
```

现在，在终端中执行以下两个命令，使KextCache无效：

```
sudo touch /Library/Extensions
```

以及：

```
sudo touch /System/Library/Extensions
```

在此之后，重新启动，计算机就能正常启动RELEASE版本的内核了。

点击收藏 | 1 关注 | 2

[上一篇：滥用 GPO 攻击活动目录—Part 1](#) [下一篇：HackFest靶机实战](#)

1. 0 条回复

- 动动手指，沙发就是你的了！

[登录](#) 后跟帖

先知社区

[现在登录](#)

热门节点

[技术文章](#)

[社区小黑板](#)

目录

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)