A Potpourri of Pwn

Jevin Sweval
For Apple Inc.
2018-12-06

About Me

- Worked at Arxan Technologies for six years
- Anti-tamper and obfuscation protection
- EnsureIT - LLVM/Clang based product operating on LLVM IR
    - PS3 port to Power 64 and Sony’s custom ABI
    - Countless hours spent understanding libc’s/runtime linkers and reversing iOS internals
- GuardIT - Binary based project decompiling customer binaries, protecting, and reassembling/relinking
    - Port x86 engine to support ARM/Thumb v7 for Android protection
- Research Team
    - ENABLE_BITCODE - Port EnsureIT to work on bitcode only without a binary-based post-linker step
    - Automatic App Protection for Android APKs
    - Discover and develop PoC of new protection techniques

About Me

- Lots of hacking as a hobby
- PS3
  - Reverse engineering undocumented dev-kit internals
  - Working on exploiting Flash 9 after Sony fixed WebKit based CFW vector in latest PS3 firmware update
- PS4
  - Working with a global team on replicating failoverfl0w's hardware SCA to recovery root console keys from south bridge and APU security module
- Android/iOS
  - Tegra SoC bootrom attacks
  - Tweaks to unlock restricted functionality in applications
- Whitebox Cryptography
  - iTunes album art encryption
  - FairPlay application DRM

iOS Debugger Detection

- dyld/src/glue.c

```
void _dyld_debugger_notification(enum dyld_notify_mode mode, unsigned long count, uint64_t
machHeaders[])
{
    // Do nothing.  This exists for the debugger to set a break point on to see what images
    have been loaded or unloaded.
}
```

  - Just RET in ARM/ARM64
  - When LLDB attaches, it finds this RET and replaces it with BKPT
  - Find this symbol at runtime and see if it is RET or BKPT to detect LLDB
- dyld/src/dyld_gdb.cpp

```
static void gdb_image_notifier(enum dyld_image_mode mode, uint32_t infoCount, const
dyld_image_info info[])
{
  switch ( mode ) {
      case dyld_image_adding:
          _dyld_debugger_notification(dyld_notify_adding, infoCount, machHeaders);
          break;
```

  - The notifier subroutine is called when dyld loads or unloads an image

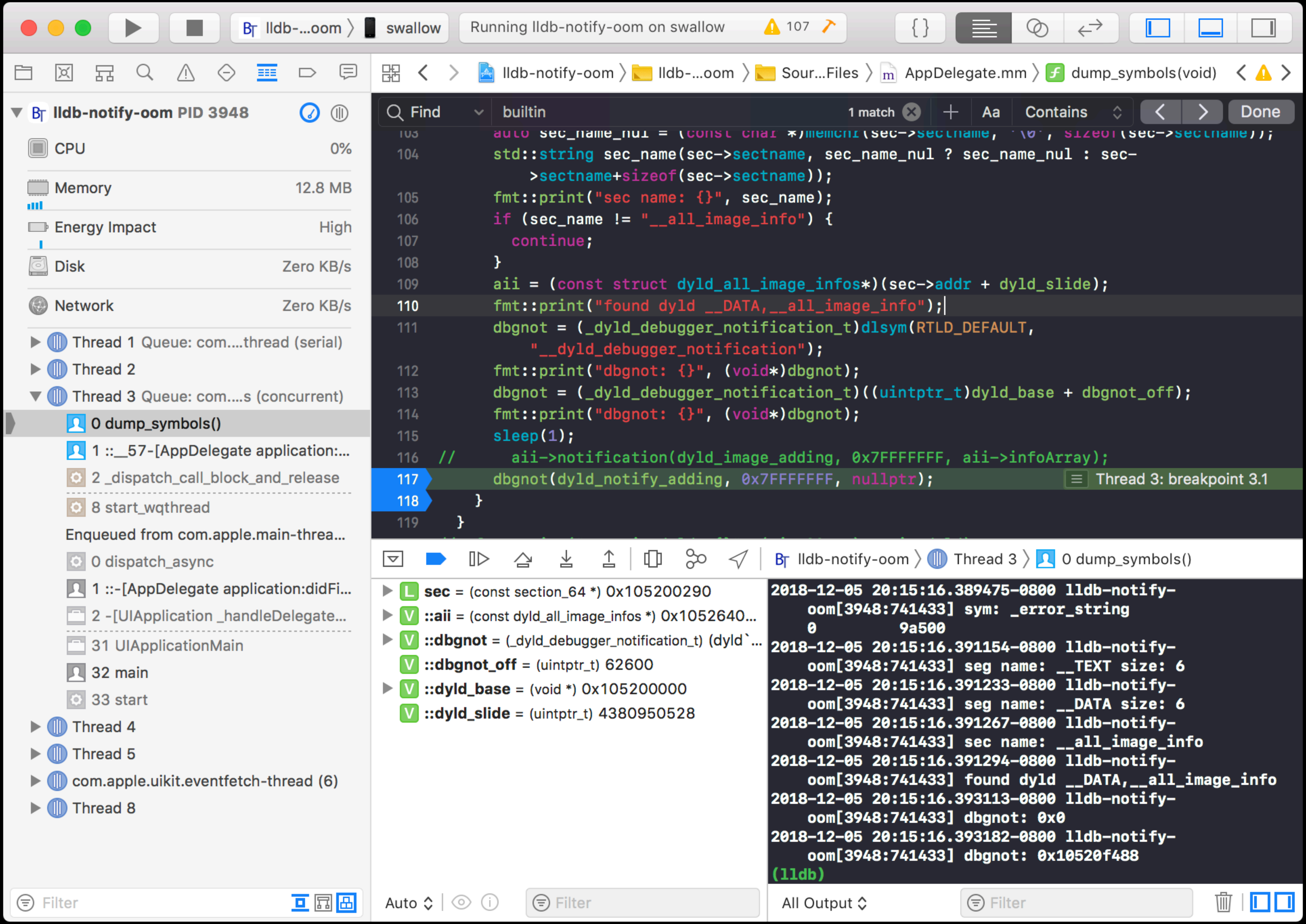
iOS Debugger Detection

- Finding `_dyld_debugger_notification` is a simple matter of walking dyld’s Mach-O load commands to find the __TEXT,__text segment symbol table info and then iterating over the symtab to find the symbol
  - I use LIEF in this example but wrote a freestanding Mach-O and ELF parser in C++ at Arxan that avoided heap allocations and never left the target symbols as plaintext in memory
- How do you find dyld’s Mach-O header to parse?
  - `_dyld_get_image_header()` but that can be hooked
  - It can be replicated by parsing `dyld_all_image_infos` but that must be found with dyld’s Mach-O header as well
  - Just call `__builtin_return_address(0)` in a constructor to get a pointer into dyld’s __TEXT,__text
  - Mask that pointer to 4 KB and walk back 4 KB until you find Mach-O magic (works with 16 KB page devices too)

iOS Anti-Debug

- What if we called `_dyld_debugger_notification` ourselves instead of letting dyld have all the fun?
- And passed in **lies** for the parameters?
- `dbg_notify_fptr(dyld_notify_adding, 0x7FFFFFFF, nullptr);`
- Xcode is not happy.
  - Older versions of Xcode consumed > 32 GB of RAM before crashing
  - Current Xcode beachballs while lldb-rpc-server chews up an entire core

iOS Anti-Debug



Running lldb-notify-oom on swallow

lldb-notify-oom PID 3948

CPU	0%
Memory	12.8 MB
Energy Impact	High
Disk	Zero KB/s
Network	Zero KB/s

- Thread 1 Queue: com....thread (serial)
- Thread 2
- Thread 3 Queue: com....s (concurrent)
  - 0 dump_symbols()
  - 1 ::__57-[AppDelegate application:...
  - 2 _dispatch_call_block_and_release
  - 8 start_wqthread
  - Enqueued from com.apple.main-threa...
  - 0 dispatch_async
  - 1 ::-[AppDelegate application:didFi...
  - 2 -[UIApplication _handleDelegate...
  - 31 UIApplicationMain
  - 32 main
  - 33 start
- Thread 4
- Thread 5
- com.apple.uikit.eventfetch-thread (6)
- Thread 8

lldb-notify-oom 〉 lldb-...oom 〉 Sour...Files 〉 AppDelegate.mm 〉 dump_symbols(void)

Find: builtin — 1 match

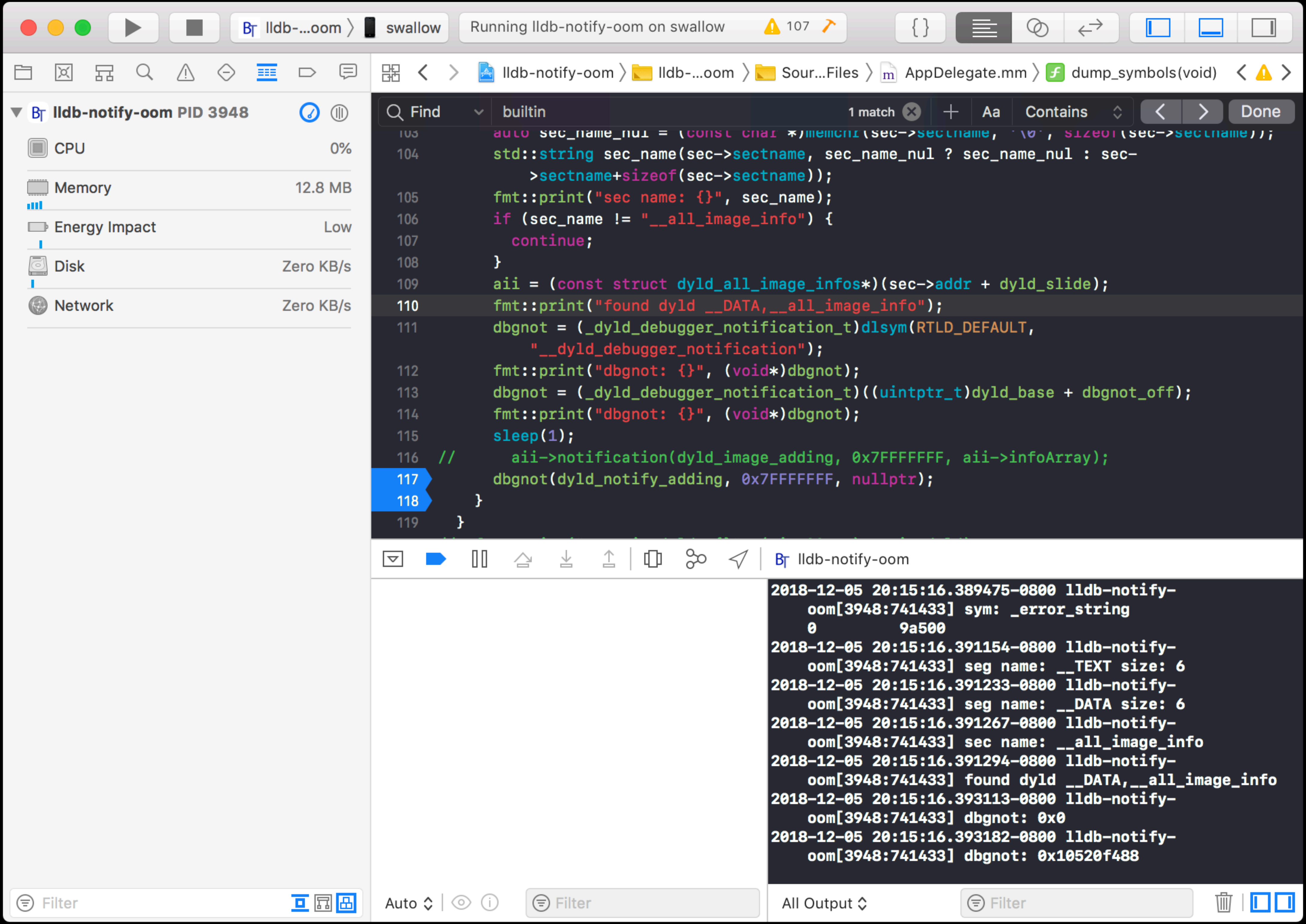
```cpp
        auto sec_name_nul = (const char *)memchr(sec->sectname, '\0', sizeof(sec->sectname));
        std::string sec_name(sec->sectname, sec_name_nul ? sec_name_nul : sec-
            >sectname+sizeof(sec->sectname));
        fmt::print("sec name: {}", sec_name);
        if (sec_name != "__all_image_info") {
          continue;
        }
        aii = (const struct dyld_all_image_infos*)(sec->addr + dyld_slide);
        fmt::print("found dyld __DATA,__all_image_info");
        dbgnot = (_dyld_debugger_notification_t)dlsym(RTLD_DEFAULT,
            "__dyld_debugger_notification");
        fmt::print("dbgnot: {}", (void*)dbgnot);
        dbgnot = (_dyld_debugger_notification_t)((uintptr_t)dyld_base + dbgnot_off);
        fmt::print("dbgnot: {}", (void*)dbgnot);
        sleep(1);
//        aii->notification(dyld_image_adding, 0x7FFFFFFF, aii->infoArray);
        dbgnot(dyld_notify_adding, 0x7FFFFFFF, nullptr);        Thread 3: breakpoint 3.1
      }
    }
```

lldb-notify-oom 〉 Thread 3 〉 0 dump_symbols()

- sec = (const section_64 *) 0x105200290
- ::aii = (const dyld_all_image_infos *) 0x1052640...
- ::dbgnot = (_dyld_debugger_notification_t) (dyld`...
- ::dbgnot_off = (uintptr_t) 62600
- ::dyld_base = (void *) 0x105200000
- ::dyld_slide = (uintptr_t) 4380950528

```
2018-12-05 20:15:16.389475-0800 lldb-notify-
    oom[3948:741433] sym: _error_string
    0          9a500
2018-12-05 20:15:16.391154-0800 lldb-notify-
    oom[3948:741433] seg name: __TEXT size: 6
2018-12-05 20:15:16.391233-0800 lldb-notify-
    oom[3948:741433] seg name: __DATA size: 6
2018-12-05 20:15:16.391267-0800 lldb-notify-
    oom[3948:741433] sec name: __all_image_info
2018-12-05 20:15:16.391294-0800 lldb-notify-
    oom[3948:741433] found dyld __DATA,__all_image_info
2018-12-05 20:15:16.393113-0800 lldb-notify-
    oom[3948:741433] dbgnot: 0x0
2018-12-05 20:15:16.393182-0800 lldb-notify-
    oom[3948:741433] dbgnot: 0x10520f488
(lldb)
```

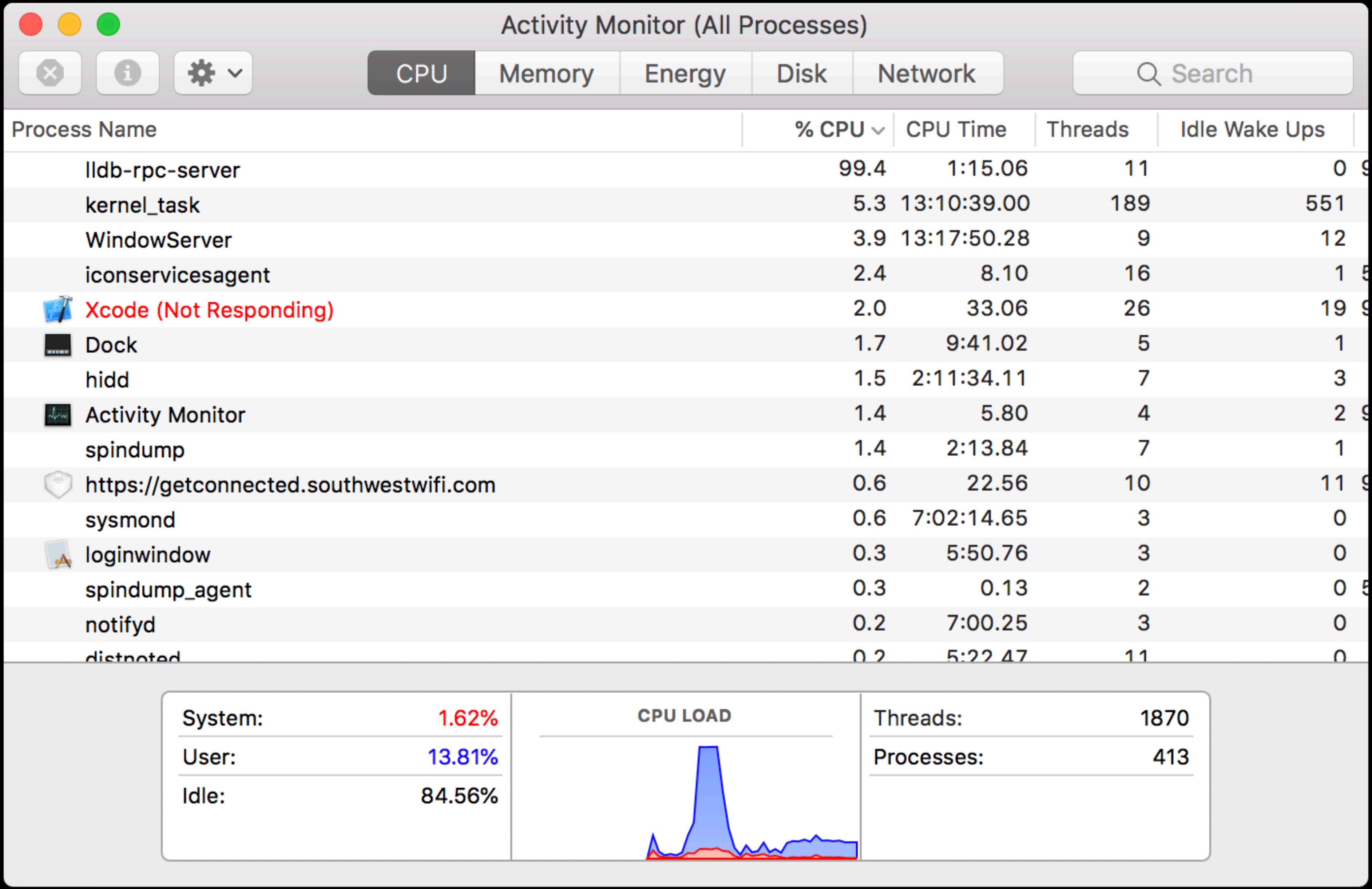
iOS Anti-Debug



```
      auto sec_name_nul = (const char *)memchr(sec->sectname, '\0', sizeof(sec->sectname));
      std::string sec_name(sec->sectname, sec_name_nul ? sec_name_nul : sec->sectname+sizeof(sec->sectname));
      fmt::print("sec name: {}", sec_name);
      if (sec_name != "__all_image_info") {
        continue;
      }
      aii = (const struct dyld_all_image_infos*)(sec->addr + dyld_slide);
      fmt::print("found dyld __DATA,__all_image_info");
      dbgnot = (_dyld_debugger_notification_t)dlsym(RTLD_DEFAULT, "__dyld_debugger_notification");
      fmt::print("dbgnot: {}", (void*)dbgnot);
      dbgnot = (_dyld_debugger_notification_t)((uintptr_t)dyld_base + dbgnot_off);
      fmt::print("dbgnot: {}", (void*)dbgnot);
      sleep(1);
//      aii->notification(dyld_image_adding, 0x7FFFFFFF, aii->infoArray);
      dbgnot(dyld_notify_adding, 0x7FFFFFFF, nullptr);
    }
  }
```

```
2018-12-05 20:15:16.389475-0800 lldb-notify-
    oom[3948:741433] sym: _error_string
    0          9a500
2018-12-05 20:15:16.391154-0800 lldb-notify-
    oom[3948:741433] seg name: __TEXT size: 6
2018-12-05 20:15:16.391233-0800 lldb-notify-
    oom[3948:741433] seg name: __DATA size: 6
2018-12-05 20:15:16.391267-0800 lldb-notify-
    oom[3948:741433] sec name: __all_image_info
2018-12-05 20:15:16.391294-0800 lldb-notify-
    oom[3948:741433] found dyld __DATA,__all_image_info
2018-12-05 20:15:16.393113-0800 lldb-notify-
    oom[3948:741433] dbgnot: 0x0
2018-12-05 20:15:16.393182-0800 lldb-notify-
    oom[3948:741433] dbgnot: 0x10520f488
```

iOS Anti-Debug



Process Name	% CPU	CPU Time	Threads	Idle Wake Ups
lldb-rpc-server	99.4	1:15.06	11	0
kernel_task	5.3	13:10:39.00	189	551
WindowServer	3.9	13:17:50.28	9	12
iconservicesagent	2.4	8.10	16	1
Xcode (Not Responding)	2.0	33.06	26	19
Dock	1.7	9:41.02	5	1
hidd	1.5	2:11:34.11	7	3
Activity Monitor	1.4	5.80	4	2
spindump	1.4	2:13.84	7	1
https://getconnected.southwestwifi.com	0.6	22.56	10	11
sysmond	0.6	7:02:14.65	3	0
loginwindow	0.3	5:50.76	3	0
spindump_agent	0.3	0.13	2	0
notifyd	0.2	7:00.25	3	0
distnoted	0.2	5:22.47	11	0

System:	1.62%
User:	13.81%
Idle:	84.56%

Threads:	1870
Processes:	413

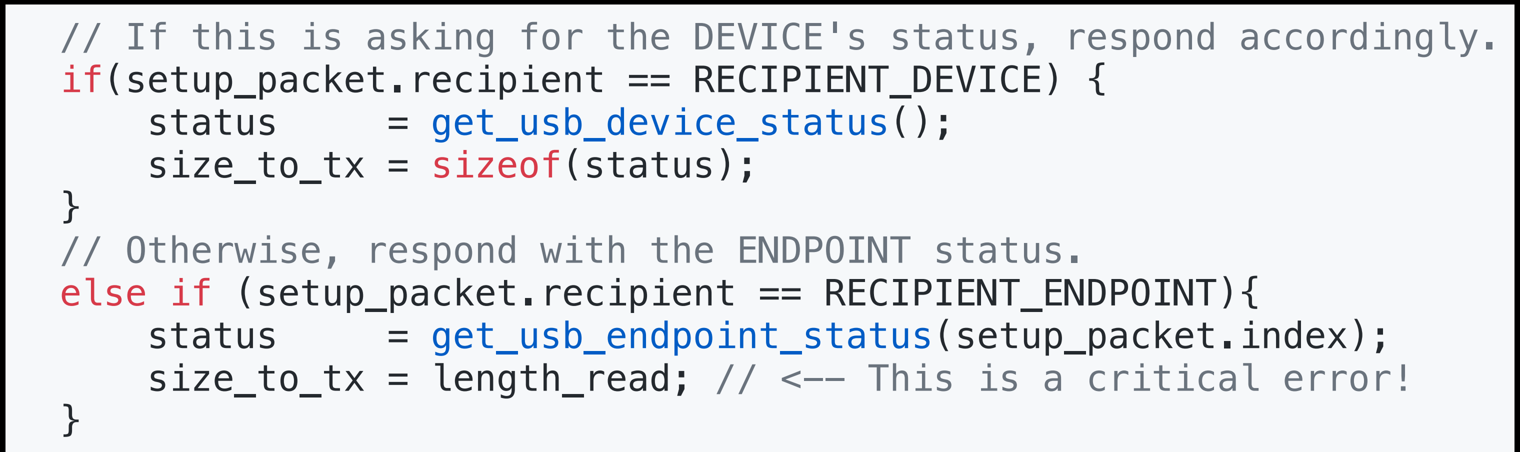
A Fun Heisenbug

- Customer reports protected app *rebooting* phones
  - Can’t reproduce locally, customer won’t provide artifacts, log issue and move on since customer can work around using a different protection seed
- Second customer reports a similar issue
  - Provides artifacts, we find it only reproduces on certain devices
  - Again very sensitive to the protection seed but seems to happen in FP-heavy code
- Reproduce using Fhourstones, vastly simplifying further debugging
- Used delta debugging (LLVM bugpoint and Delta) to somewhat minimize repro

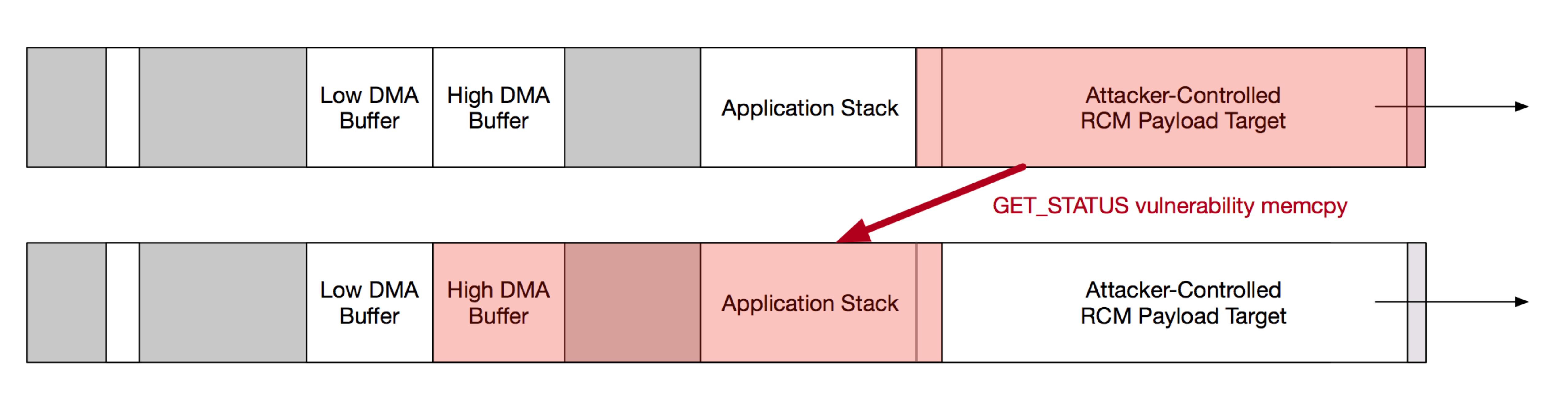
A Fun Heisenbug

- No API calls, no syscalls around reboot. No kernel panics, just WDT timeout!
- Instrumenting app with logging and tracing would usually make the issue disappear
- Issue is found to only occur on Apple A8 CPU… is this an erratum?
  - Nothing found by digging around LLVM source, even Apple FOSS releases
- Using Xcode clang to assemble asm fixes the issue. Differences in bitcode between FOSS and Xcode determined not to matter
- Disassembly of the Xcode assembled asm reveals that clang is assembling `movi.2d vX, #0` as `movi.16b vX, #0`
- Further reversing reveals that the above transformation is the complete erratum workaround
  - Fixed by LLVM flag `-fix-16473581` "Fix for rdar://16473581"
  - Still not part of open source LLVM/Clang releases today. Unsure if App Store checks ever started looking for this DoS.

Tegra Bootrom Exploitation



```
// If this is asking for the DEVICE's status, respond accordingly.
if(setup_packet.recipient == RECIPIENT_DEVICE) {
    status     = get_usb_device_status();
    size_to_tx = sizeof(status);
}
// Otherwise, respond with the ENDPOINT status.
else if (setup_packet.recipient == RECIPIENT_ENDPOINT){
    status     = get_usb_endpoint_status(setup_packet.index);
    size_to_tx = length_read; // <-- This is a critical error!
}
```


Low DMA Buffer
High DMA Buffer
Application Stack
Attacker-Controlled RCM Payload Target
GET_STATUS vulnerability memcpy
Low DMA Buffer
High DMA Buffer
Application Stack
Attacker-Controlled RCM Payload Target

Simplified code and diagram by Katherine Temkin (@ktemkin)

Tegra Bootrom Exploitation

- Bootrom readout after first 1 KB is blocked by a write to a PMGR security register as last step in bootrom before jumping to next stage bootloader

- Ported Tegra X1 bootrom exploit originally developed for Nintendo Switch to Denver (Nexus 9), 3 (Nexus 7, Honda CRV, Tesla Model S), 2 (Asus TF101, WIP)

    - Different offsets, packet header sizes

    - Debugging difficult, lack of UART, use reboot to signal success, store values in always-on PMGR registers and SRAM that survive reset

    - Teams spent much time dumping bootrom. It turns out Nvidia reused bootrom code in miniloaders (think iBEC) that are publicly available and contain the same vulnerability

- Tegra 2 contains the vulnerability but is not easily exploited

    - Working on two phased attack where the payload loaded to SRAM, bootrom manipulated to reset SoC, then exploit the vulnerability using payload persisted in SRAM from first phase

    - Working on Tegra support in QEMU to discover a method to reset SoC after payload loading

Tegra Bootrom Exploitation

- Tegra uses an SBK encrypted (later encrypted + RSA signed) “warmboot blob” to restore PLLs / DRAM peripherals after waking from deep sleep
- The warmboot blob header specifies the address for the bootrom to load it into SRAM
- Load address is not checked! Set load address before bootrom stack and overwrite return address to return to custom payload in blob
    - Spray copies of bootrom or secure boot key to end of SRAM and dump from Android after boot using /dev/mem or kernel module
- NOP out SBK key disable in Asus TF101 aboot bootloader to enable encryption of exploit warmboot blob, then overwrite original blob with properly encrypted exploit blob
    - aboot is encrypted with SBK but it conveniently encrypts and flashes any unverified bootloader written to an update partition

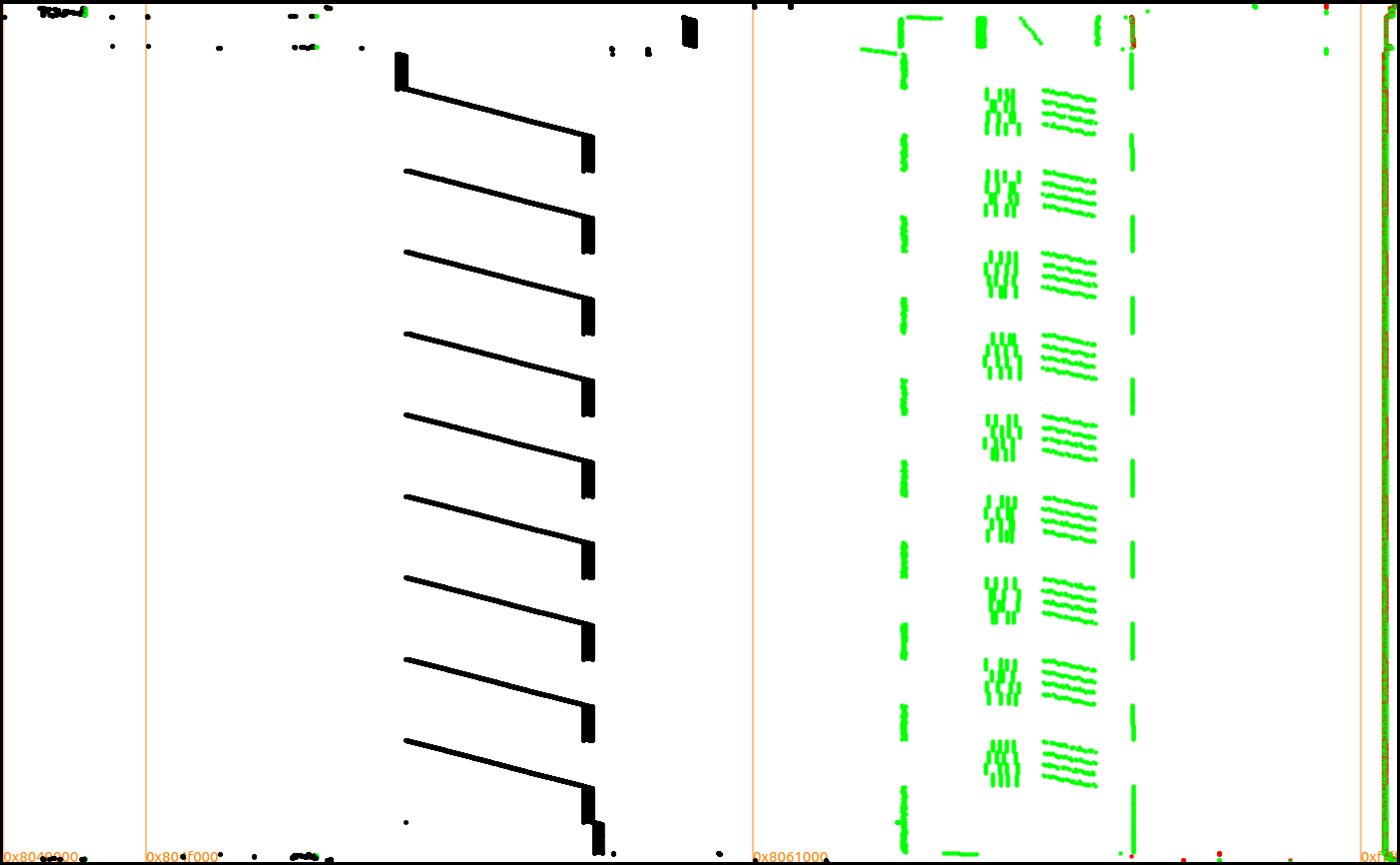
iTunes Album Art Whitebox Crypto

- Back in 2012 when I used to meticulously curate my music library, iTunes provided the highest resolution album art
- Some time prior, Apple started encrypting the album art
  - Encrypted art 20 bytes larger than decrypted art
- Reversed iTunes.dll to discover RC4 decryption of data after 20 byte header
- Header contained some key but not the key fed to RC4
  - My first discovery of whitebox crypto

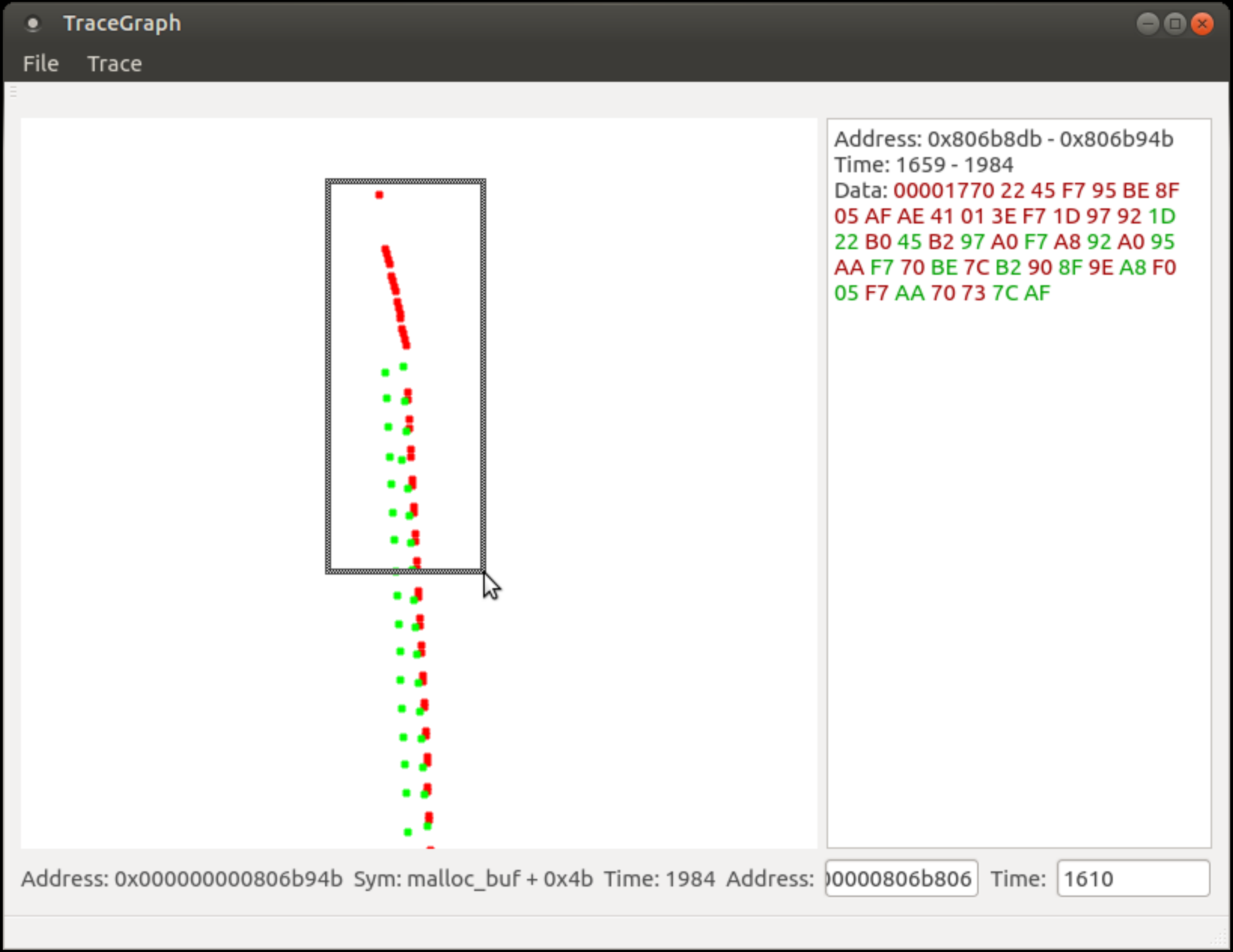
iTunes Album Art Whitebox Crypto

- Used OllyDbg to identify outermost key transformation subroutine
    - Trace every instruction executed and memory accessed during that subroutine
- From traces, all code and data tables (27.5 KB total) used in whitebox were identified and lifted to source form
- With a Python script to search and download encrypted art, I was able to decrypt it using my code lifted whitebox that now ran on any platform

iTunes Album Art Whitebox Crypto



iTunes Album Art Whitebox Crypto



```
Address: 0x806b8db - 0x806b94b
Time: 1659 - 1984
Data: 00001770 22 45 F7 95 BE 8F
05 AF AE 41 01 3E F7 1D 97 92 1D
22 B0 45 B2 97 A0 F7 A8 92 A0 95
AA F7 70 BE 7C B2 90 8F 9E A8 F0
05 F7 AA 70 73 7C AF
```

Address: 0x000000000806b94b Sym: malloc_buf + 0x4b Time: 1984 Address: 0000806b806 Time: 1610

iTunes Album Art Whitebox Crypto

- After getting hired at Arxan and learning about their whitebox product, I became interested in extracting the normal AES keys
- In 2016 I used SideChannelMarvel's Daredevil to attempt key extraction using CPA
    - Could not find the right parameters to get a full key but Daredevil believed it had found every odd byte of key
- Last month I revisited the topic and used the newer JeanGrey tool to perform a DFA attack
    - Keys recovered in fewer than 40 lines of Python and less than 10 seconds (single core) for each of the 64 subkeys
    - My 2016 CPA attempts were on the right track, the odd key bytes were indeed correct

FairPlay App Binary DRM

- As part of the ENABLE_BITCODE project at Arxan, I investigated using Apple’s existing code signing as an alternative for requiring a post-linker step that hashed compiled code

- Soon realized that the code signature is on the FairPlay encrypted binary, not the decrypted version that would be hashed at runtime

- How to get the decrypted binary to hash?

    - Jailbroken device dumping binary from a live process

    - Authenticated backdoor to dump out binary upon receiving a signed request

        - What could go wrong?

    - Reverse engineer FairPlay?

FairPlay App Binary DRM

- Initial investigations focused on iOS 3.0 since it was the first version to include App Store and a quick glance showed it had weaker obfuscations than newer iOS
    - Cloakware identified by “Standard-Eta”, “Standard-Beta”, “Standard-Theta” XOR encrypted strings
- Shelved the project until I improved my iOS kernel hacking skills
- Originally planned to use Kirk Swidowski’s VERTIGO microvisor to trap MMIO access to CDMA peripheral
- Last month I reversed iOS 7.1.2 FairPlayIOKit some more and realized it didn’t directly access CDMA peripheral and went through AppleCDMA.kext

FairPlay App Binary DRM

- Used xerub’s kexty project to get an easy to use kernel mode working environment with my own custom kext

- Overwrite AppleCDMA vtable’s _performAES pointer to point to my dumper subroutine that dumps key, IV, key ID, 16 bytes of plain/ciphertext and any extra data in IOAESAcceleratorRequest

- Load the target application and walk every page to get FairPlayIOKit to decrypt every page, logging keys/IVs over UART

- Result: Every page is encrypted with a unique key/IV pair. Complete offline decryption is possible with dump log.

FairPlay App Binary DRM

- key_id 300
  - IOAESAcceleratorRequest has an extra 32 byte buffer that is some kind of table for a “FairPlay descrambler” implemented in the CDMA hardware
- Still unsure how the descrambler works
  - Decrypted all 0 key/IV/PT with all 3 bit bit flips of all zero scrambler buffer and recorded PTs
  - No difference between CBC and ECB mode so IV is not involved
  - No simple XOR mask, LFSR, GF(2^8) multiplication, or AES S-Box application of scrambler buffer found
  - Working on statistical analysis

Q & A

Source code is available upon request.