Mobile Hacking VI

iOS Jailbreak

@crakun

Sat, 11-9-13, 6am PST Day 8

Mobile Hacking VI
Class Agenda

- Part I:
  - Communications
  - Bug #00000002 Status Update / Review
    - Sulphur27
    - Saelo
  - RE Steps
- Part II
  - Homework

m0bdev
Jailbreak Team

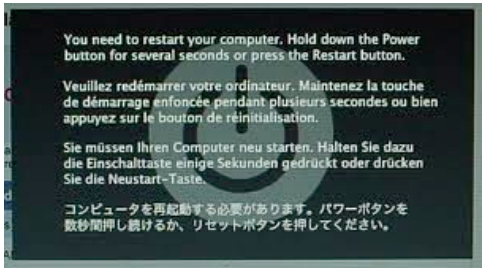
Core Members:

Isa56k
Sulphur27
compilingEntropy
Demito
Saelo
crakun

(…more to come…)

Bugs, Vulns, Exploits we are looking for in class

- Remote Exploits
- Userland Vulnerabilities(to obtain mobile)
- Privilege escalation (mobile to root)
- Escaping sandbox techniques
- Information leaks
- Bypassing code signing techniques
- Kernel Vulnerabilities
- Strategies for dealing with KASLR on 64-bit ARM



Communications

crakun contact Info

Email: crakun@m0bdev.com

Twitter: @crakun

Skype: m0bdev (m0bdev is spelled with a zero)

IRC: #openjailbreak on freenode

**-> If I am slow responding / or no response, ping me again because I may have missed it

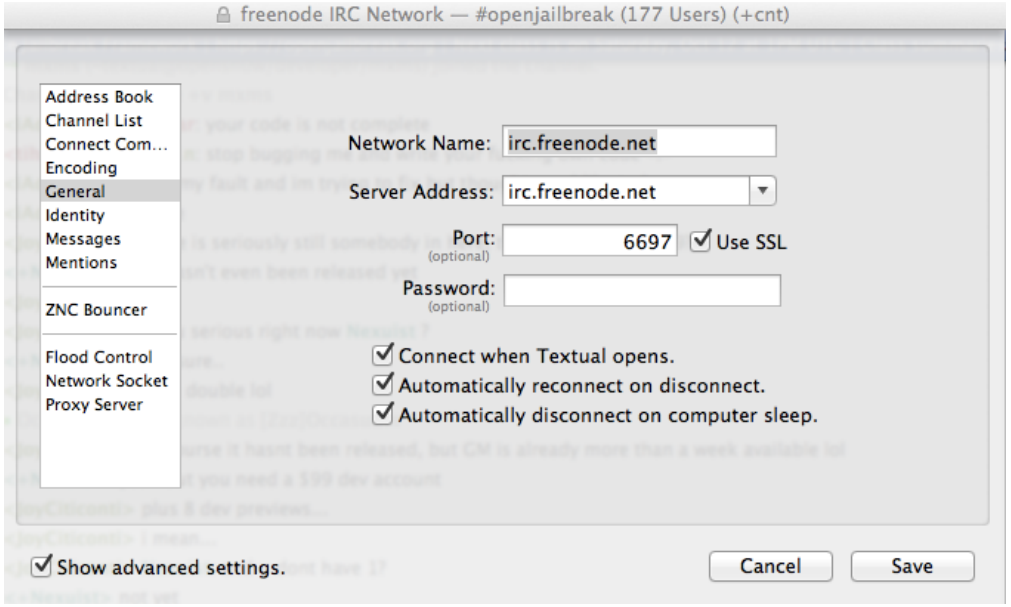
Submittal for Bugs/ Vulns /Exploits

Email Me (crakun@m0bdev.com) with a zip file of:

1. Detail Step-by-Step Method how / what you did so I can reproduce it
2. What devices and firmware does this occur on?
3. Tell me what you think it is?
4. Include Crash Report / Panic Log
5. Include Proof-of-Concept source code
6. Put a date on the submission

Class Communications

Text -> IRC: #openjailbreak on freenode



Skype: m0bdev (m0bdev is spelled with a zero)

Please mute Skype unless you have a question during clazz

Bug # 00000002
Kernel Panic Review for the Class

Current Bug

- Submited By: Sulphur27
- MHVI-Class_Bug: #00000002
- Kernel Panic from .mov file

http://www.hotwan.com/class/mov/sample/2.zip

Reverse Engineering Tips

- We will use a variety of tools such as aptdiff, hexdiff, quicktime file format view, quicktime specs, XNU source code, gdb and IDA Pro, http://www.hopperapp.com/ , xpwntool

- Also take a look at this:
  - http://samdmarshall.com/re.html -Important! (thanks Dirkg)

IDA Pro

- IDA Pro for Mac OSX
- Hex-rays ARM Decompiler for Mac OSX

- Looks like eval works

Source code Review RE Tips

- In reverse engineering some people find bugs thru reviewing source code and specs:
  - OSX XNU, kernelcache in IPSW & runtime (iOS kernel)
  - webkit (safari)
  - File format specification for quicktime https://developer.apple.com/standards/classicquicktime.html , ISO/IEC 14496-12 , ISO/IEC 14496-15 (for .mov files)
- You can then trace backward through the code to whether code fragments expose any vulnerabilities accessible from an application entry point (think sources and sinks). Find a code path to reach vulnerable code.
- Identify the input data and trace the input data while looking for coding errors. Trace input data through the kernel functions while looking for potentially vulnerable locations
- Look for use of unvalidated length or size values.
- Look for vulnerable c functions

More Source Code review RE Tips

- Look for if proper error conditions are defined.
- Look at and validate return values are returned correctly
- Look for a library function that's called directly after the vulnerability happens
- Study the use of explicit type conversions (casts). An explicit conversion occurs between char and a signed int.
- Many vulnerabilities related to type conversions are the result of conversions between unsigned and signed integers (think integer overflows)
- Look for destination addresses for memory-copy operations that are extracted from user-supplied (our specially crafted .mov file) data
- We are looking for memory corruption that occur in a process, thread or kernel

Combining Static Analysis and Dynamic Analysis

- Aside from the static analysis of source code review, reverse engineers may employ fuzzing starting with a valid format sample or creating a file from scratch.
- We will match C source code to assembly when possible including processing .mov files processed
- Using a combination static analysis for disassembly of a binary (Quicktime file format viewers, OxED, Aptdiff, hexdiff, IDA Pro, hopperapp,). Disassemble mediaserverd
- Combined with dynamic analysis (fuzzing –zzuf, debugging –gdb)
- Combined with CVE's, previous bugs posted and other people's work as guides for our analysis, we develop a better understanding of typical software bugs and approaches we can apply for our gain
- Zero in on user-influenced input data enters the software through an interface to the outside world. Combining static and dynamic analysis to find input handling and error handling
- **As a jailbreak team, combining our experience, we will develop a formal methodology for interpreting / evaluating bugs with kernel panic logs and crash reports in class**

Samples of Previous Quicktime Bugs

For example: 2012

http://secunia.com/community/advisories/47447

- Insufficient validation when parsing encoded movie files
- Boundary errors
- Stack-based buffer overflow
- Heap-based buffer overflow
- Buffer underflow
- Off-by-one error causing a single byte buffer overflow.
- Integer overflow MPEG files.
- Use-after-free error
- Type conversions
- Memory leaks that freeze the device

Reverse Engineering Quicktime Bugs

(thank u c0xei for the Info Research)

- A Bug Hunter's Dairy (Chapter 8)
- http://reversemode.com/index.php?option=com_content&task=view&id=69&Itemid=1
- http://shakacon.org/2009/talks/Exploit_or_Exception__DeMott.pdf
- https://www.corelan.be/index.php/2013/02/26/root-cause-analysis-memory-corruption-vulnerabilities/
- https://www.corelan.be/index.php/2013/07/02/root-cause-analysis-integer-overflows/
- http://opensecuritytraining.info/IntroductionToReverseEngineering.html

Tools to view Quicktime Format

Mp4Player (Windows)
Dumpster (Mac)
Atom Explorer (Mac)

Export sections of the file?

Any other tools out there we can leverage?

- Your best bet is to extend MP4 Explorer to do this, or write your own parser. Parsing the atoms is actually pretty simple, things start to get complicated when you need to interpret the content of the atoms and cross-reference them to, for example, locate where the frame data is.

- The QuickTime file format specification is the best resource for Apple generated QuickTime files, but you may need to do some reverse engineering, as the spec is not very complete in some areas, like the handling of MPEG-2 and MPEG-4 video.

- If you have access to ISO specs, the have access to ISO specs, the ISO/IEC 14496-12 is a standardized version of the QuickTime format (or better said, of a subset of it). The ISO/IEC 14496-15 specification builds on top of 14496-12 and defines a specific implementation of this format for the H. 264 format. This is the so called MP4 format.

- http://echoone.com/filejuicer/formats/mov-repair

Bug #00000002
Status Update / Review

- Sulphur27
- Saelo

Intro to Reverse Engineering steps from Userland to Kernel (Defining the Process for RE .mov files)

It's important that we drive more into their understanding of the bug and it's characteristics, plus verify for ourselves as best as possible.

- Working on order, but here is a start…

1. **There could be some complexity that we are missing subtleness of. Such as, it may not be just one magic byte we can leverage.**
2. **Don't stop with "oh, it's a null pointer dereference bug"and stop there. This could be a clue to something more, something still hidden and exploitable.**
3. We need to **understand the entry point of user controlled data into the application**. And **it's flow into the kernel.**
4. **Trace it thru user land**. By **disassembling mediaserverd in IDA pro**. (It's my understanding eval edition works for this). **Seeing the flow of ARM instructions**.
5. Matching these up the mov file with documentation on apple.
6. **Parse and understand the mov file construction as much as possible. Use open source solutions that make mov files can help with this. (Ffmpeg)**
7. **Running it on an iphone4 first.**
8. Pulled from JB phones, **Comparing mediaserverd for iphone4 and iphone5 in IDA and note any differences**. **Try bindiff / hexdiff**
9. **Run gdb and attach to the mediaserverd process. Also safari process. Note libraries used. See how registers flow for normal mov versus fuzzed mov.**
10. **Look at memory utilization, top, and ps on devices. Especially loading original and fuzzed.**
11. **Try to identify entry point into kernel. Use OSX Source and references winocm gives you** and **reference "OSX and iOS internals, the Apple's Core**" Book to **find the lowest level APIs that would be called**.
12. **Boot into kernel debugging with redsn0w and/or use stefan esser's method on the iPhone4.** Also try Stefan's pyKDP.
13. **Load the mov files when kernel debugging is enabled and ready to go.**
14. **Observe all the libraries called. Trace and understand these libraries.**
15. Decrypt and **analyze the libraries used in iPhone 4 for 6.1.2 and 7.0.3 to note differences**.
16. Dump **kernel cache for iphone5 for 6.1.2 and compare to kernel cache differences in iphone 4 of 7.0.3.**
17. **Amass a lot of people's panic logs for comparison.**
18. Also **start with a cold boot every time testing the fuzzed mov file.**
19. **Lightly Fuzz other sections of the fuzzed mov file to see what values we can change in registers /PC.**
20. Get Stefan Esser's input. Also Geohot, p0sixninja, Winocm, iH8sn0w, nikias, mark dowd for reversing ideas.
21. **Analyze multiple devices panic logs to see what they are.**
22. Understand Apple's current day mitigations for leveraging null pointer dereferences and other current day security enhancements for 7.0.3 based on previous talks/ presos.
23. **Is there any return from kernel mode?**
24. **Understand previous quicktime bugs for clues to ours.**
25. **Watch Apple react and Internet for analysis of other security folks ,etc. We will learn more as time goes on.**
26. **Consult other RE methodologies to hone our own as we set the Standard.**

Kernel Panic

Panic Log Analysis

Look at panic log

fault_type & register values

/var/mobile/Library/Logs/CrashReporter/Panics/ for kernel panics

Kernel Panic logs are stored in /private/var/mobile/Library/Logs/panic.log if there is no panic.log file you have to create one***

Finding iOS Kernel Bugs

- http://www.youtube.com/watch?v=HL0_Wfc_iEU&feature=youtu.be
- http://conference.hitb.org/hitbsecconf2013kul/materials/D2T2%20-%20Stefan%20Esser%20-%20Tales%20from%20iOS%206%20Exploitation%20and%20iOS%207%20Security%20Changes.pdf
- http://cansecwest.com/slides/2013/CSW2013_StefanEsser_iOS6_Exploitation_280_Days_Later.pdf
- http://media.blackhat.com/bh-us-11/Esser/BH_US_11_Esser_Exploiting_The_iOS_Kernel_Slides.pdf
- http://www.slideshare.net/seguridadapple/targeting-the-ios-kernel
- http://www.slideshare.net/inggmartinez/find-your-own-ios-kernel-bug
- Review these slides and the videos of these presentations

Reading the Mac OSX XNU

Auditing XNU will reveal a bunch of vulnerabilities already fixed in iOS

http://www.opensource.apple.com/source/xnu/xnu-2050.24.15/

KernelCache

Use kernelcache to determine attack surface

Getting Kernelcache for iPhone4:

Use Xpwntool

Dumping Kernelcache from memory for iPhone 5

<iH8n0w> You can choose two methods, use a gadget in the kernel that loads it into a register, then returns. Then write that register to a file. (4 bytes a time which is very slow), or write a fully working kernel payload that flips the kernel page attributes to RW for the taskforpid0 page. Then patch taskforpid0, and use vm_read to dump the kernel in 2048 chunks.

KernelCache

The 6.1.2 kernel dumps are crucial for locating specific functions within the kernel that are static between iOS 6.1.x kernel builds. This means functions such as "_START" within the kernel, are located at the same location in 6.1.2 kernels and possibly 7.0.3 kernels.

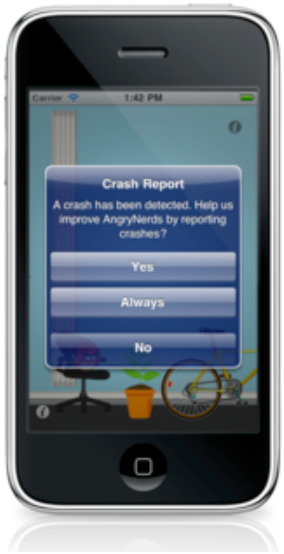
So, why do we need these? Simple. We need some static offsets for functions within the 6.1.2 kernel to utilize them in the 7.0.3 kernel, and dump the actual 7.0.3 kernel.

The 7.0.3 kernel is more essential as some kexts such as the sandbox kext, signature check kexts [AMFI], etc, are *not* static and tend to shift its location on every recompile.

From Overwritten PC to Code Execution

- once we control PC we can jump anywhere in kernel space
- in iOS a lot of kernel memory is executable
- challenge is to put code into kernel memory
- and to know its address
- **nemo's papers** already show ways to do this for OS X

Crash Report Analysis



Crash Reports

- **<ep0k>** Not all crashes are interesting : aborts, timeouts or out of memory kind of crashes are useless.

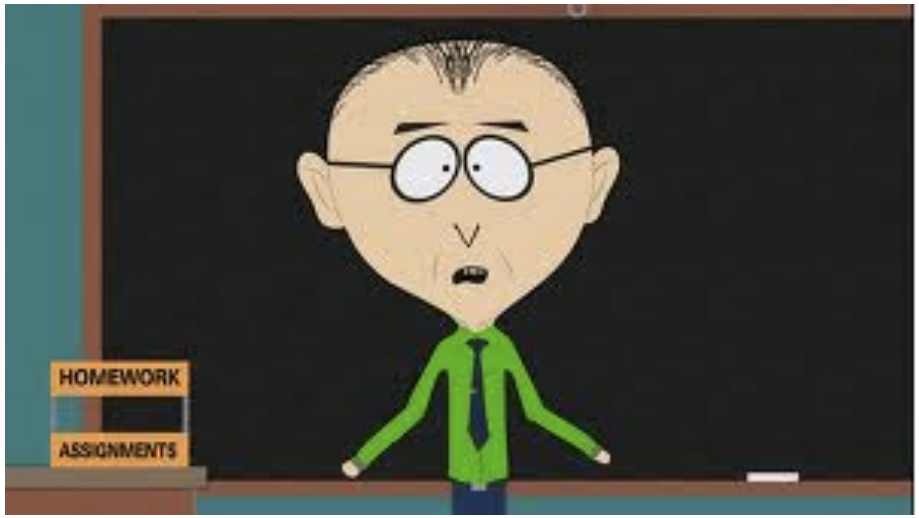
- Verify the crash dump in Settings / General / About / Diagnostics & Usage / Diagnostic & Usage Data that the crash report you created.

Crash Report References

- http://www.plausible.coop/blog/?p=176

Got some better more links / info ?

Homework



Homework Assignment

- Review slide deck.
  - Think about what you can do.

  - Contact me if you want to share bugs, confirmed vulns, exploits via Email: crakun@m0bdev.com

Reverse Engineer today's Class Bug #00000002, (Slide 20). Do what u can.
http://www.hotwan.com/class/mov/sample/2.zip

- Email your panic log generated 1.mov. Note firmware and device. This will be distributed in class for comparison with other submittals.

- New Comers: (see Appendix)
  - Fuzz .mov files / Submit

Next Meeting:
(Day 9): Sat, Nov 16, 2013 6am PST

- Dirkg's presentation on Reverse Engineering

Open Floor Discussion

APPENDIX

1st Steps in Bug Hunting

- Find / Create Fuzzer (http://caca.zoy.org/wiki/zzuf, peach)
- Find / Make sample .mov files
  - Create with your iDevice, FFmpeg, Camtasia, powerpoint, Final Cut Pro, Quicktime Pro
- Generate semi-invalidate samples
- Fuzz Application
  - Log such that you know which file caused fault
- Review Faults
- AptDiff (Good file, Mutated file)
- Change each difference back, to identify the byte that caused the fault
- Verify/ Debug to determine exploitability

Intro to .mov files

Why .mov Files?

- .mov files are old, feature rich, remote, and a quick burn

- it will be patched before we complete our jb simply because we are openly working on the exploit

Complexity with Media Players -Codec

"Codec" is a technical name for "compression/decompression". It also stands for "compressor/decompressor" and "code/decode". All of these variations mean the same thing: a codec is a computer program that both shrinks large movie files, and makes them playable on your computer . Codec programs are required for your media player to play your downloaded music and movies.

"Why do we need codecs?"

ANS: Because video and music files are large, they become difficult to transfer across the Internet quickly. To help speed up downloads, mathematical "codecs" were built to encode ("shrink") a signal for transmission and then decode it for viewing or editing. Without codecs, downloads would take three to five times longer than they do now.

"Is there only one codec I need?"

ANS: Sadly, there are hundreds of codecs being used on the Internet, and you will need combinations that specifically play your files. There are codecs for audio and video compression, for streaming media over the Internet, videoconferencing, playing mp3's, speech, or screen capture. To make matters more confusing, some people who share their files on the Net choose to use very obscure codecs to shrink their files. This makes it very frustrating for users who download these files, but do not know which codecs to get to play these files. If you are a regular downloader, you will probably need ten to twelve codecs to play your music and movies.

What codec is used for .MOV files

- *As far as I know they can only be en/decoded by Quicktime. (**mediaserverd for iPhone**)*

- MOV files have used dozens of different video and audio formats over the years.

- Currently, the most common is H.264 video and AAC audio, and they can in fact be played by nearly everything, mainly because the MOV format is basically the same as MP4. You might have to remux the file though, as there are some subtle differences between MOV and MP4. Yes, this is yet another reason to hate Apple.

Fuzzing Info

Fuzzing .mov Files

- .mov files are old, feature rich, remote, and a quick burn
- Any crashes so far? (don't send them to Apple)
- Do we need to build a custom .mov file fuzzer?
  - http://shakacon.org/2009/talks/Exploit_or_Exception__DeMott.pdf
  - http://www.blackhat.com/presentations/bh-usa-05/bh-us-05-sutton.pdf
  - http://www.cert.org/vuls/discovery/bff.html
  - http://peachfuzzer.com/v3/TutorialFileFuzzing.html
  - Chapter 6: iOS Hacker's Handbook
  - https://developer.apple.com/standards/classicquicktime.html

Tool Talk Showcase

- Tool development for Jailbreak class:

isa56k **http://**www.hotwan.com/class/fuzzers/fuzzyDuck.sh

compilingEntropy http://github.com/compilingEntropy/fuzzycactus

Nexuist http://nexuist.tumblr.com/

Uroboro http://github.com/uroboro/Hexdiff

etelek_1 http://www.expetelek.com/2013/10/automated-ios-safari-file-fuzzing/

Demito https://github.com/Dem8/tcr

Ask these people and others (**iAdam1n, compilingEntropy, sulphur27**) on IRC #openjailbreak to get your fuzzing up and running

More on Fuzzing

- Exploring Code Paths
- Sending invalid data to an interface in hopes of triggering a error condition or fault condition.
- These errors can lead to exploitable vulnerabilities

Submittal for .mov Crashes

Email Me (crakun@m0bdev.com) with a zip file of:

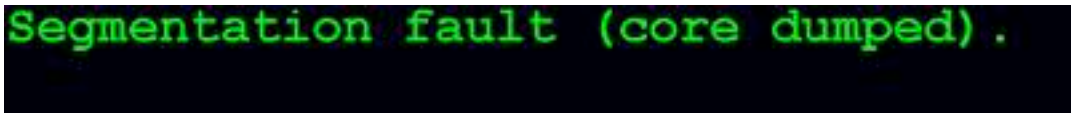
1. Which phones and firmware were the .mov file tested on

2. Include Crash Report / Panic Log

3. Include original, unfuzzed .mov file and fuzzed .mov file

4. Tell me what you think it is?

Signals



Signals

https://developer.apple.com/library/ios/documentation/system/conceptual/manpages_iphoneos/man3/signal.3.html

Signals allow the manipulation of a process from outside its domain, as well as allowing the process to manipulate itself or copies of itself (children). There are two general types of signals: those that cause termination of a process and those that do not. Signals which cause termination of a program might result from an irrecoverable error or might be the result of a user at a terminal typing the `interrupt' character.

Most signals result in the termination of the process receiving them, if no action is taken; some signals instead cause the process receiving them to be stopped, or are simply discarded if the process has not requested otherwise. Except for the SIGKILL and SIGSTOP signals, the **signal**() function allows for a signal to be caught, to be ignored, or to generate an interrupt. These signals are defined in the file <signal.h>:

```
Vortexs-MacBook-Air:~ vortex$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGEMT       8) SIGFPE
 9) SIGKILL     10) SIGBUS      11) SIGSEGV     12) SIGSYS
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGURG
17) SIGSTOP     18) SIGTSTP     19) SIGCONT     20) SIGCHLD
21) SIGTTIN     22) SIGTTOU     23) SIGIO       24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGINFO     30) SIGUSR1     31) SIGUSR2
```

Signals

The following is a list of commonly encountered, process-terminating signals and a brief description:

Signal	Description
SIGILL	Attempted to execute an illegal (malformed, unknown, or privileged) instruction. This may occur if your code jumps to an invalid but executable memory address.
SIGTRAP	Mostly used for debugger watchpoints and other debugger features.
SIGABRT	Tells the process to abort. It can only be initiated by the process itself using the `abort()` C stdlib function. Unless you're using `abort()` yourself, this is probably most commonly encountered if an `assert()` or `NSAssert()` fails.
SIGFPE	A floating point or arithmetic exception occurred, such as an attempted division by zero.
SIGBUS	A bus error occurred, e.g. when trying to load an unaligned pointer.
SIGSEGV	Sent when the kernel determines that the process is trying to access invalid memory, e.g. when an invalid pointer is dereferenced.

SIGNALS

- Exception Type SIGILL, SIGBUS or SIGSEGV are the useful types.

SIGILL

The **SIGILL** signal is raised when an attempt is made to execute an invalid, privileged, or ill-formed instruction.

SIGILL is usually caused by a program error that overlays code with data or by a call to a function that is not linked into the program load module.

<C0deH4cker> SIGILL is very unlikely to encounter but in general the best one to find, as it means its executing code where it should'nt be

SIGSEGV

<C0deH4cker> SIGSEGV is the next best one, and it likely to be exploitable

In SIGSEGV, it's trying to read/write to an invalid address.

Signal 11– SIGSEGV

- Always the hottest because it means we're already dealing with a memory read/write issue

  - We want a SEGV based on a write operation

Ideas toward popular heap exploitation techniques

- Clobber function pointer on Heap with overwrite
- Get that function called

SIGABRT

```
15. Exception Type:  EXC_CRASH (SIGABRT)
16. Exception Codes: 0x0000000000000000, 0x0000000000000000
```

abort() sends the calling process the SIGABRT signal, this is how abort() basically works.

abort() is usually called by library functions which detect an internal error or some seriously broken constraint. For example malloc() will call abort() if its internal structures are damaged by a heap overflow.

Often indicate heap issues when fuzzing

- But that's why were here, so back to the QuickTime crash from my Mac
  - **SIGFPE** is the signal sent to computer programs that perform erroneous arithmetic operations on POSIX compliant platforms. The symbolic constant for SIGFPE is defined in the header file signal.h. Symbolic signal names are used because signal numbers can vary across platforms.
- So….. is it exploitable or not? Probably not.
  - Why? Because the chance for memory corruption, and thus execution redirection looks minimal. See example:

```
int main() {
  int x = 42/0;
  return 0; /* Never reached */
}
```

- I hear that. You really always need to reverse each exception to gain full understanding of the crash
  - ***GET TO THE ROOT CAUSE VIA REing***
    - If the value is used to influence allocation or similar functions, it could still have a chance of being exploitable
- On the other hand, no invalid memory access is directly happening here, so the chances that it would be exploitable seems very thin

Null Pointer Dereference

Null-pointer dereference issue can occur through a number of flaws, including race conditions, and simple programming omissions.

They can be exploitable sometimes. Ask Mark Dowd, Also A Bug Hunter's Diary (p 35, Ch4 and p153)

Good coding practice: Before using a pointer, ensure that it is not equal to NULL

Does userland and kernel land share the same zero page?

I think this is fixed – not possible since 6.x

ARM

ARM Reference Material

ARM Quick reference card:

- http://web.eecs.umich.edu/~prabal/teaching/eecs373-f10/readings/ARM_QRC0001_UAL.pdf
- http://www.eng.auburn.edu/~nelson/courses/elec5260_6260/ARM_AssyLang.pdf
- http://simplemachines.it/doc/arm_inst.pdf
- http://opensecuritytraining.info/IntroARM.html

Reverse Engineering ARM:

- http://www.peter-cockerell.net/aalp/html/frames.html
- http://media.hacking-lab.com/scs3/scs3_pdf/SCS3_2011_Bachmann.pdf
- http://www.raywenderlich.com/37181/ios-assembly-tutorial
- http://www.phrack.com/issues.html?issue=66&id=12

To Begin:

- http://lightbulbone.com/post/27887705317/reversing-ios-applications-part-1
- http://yurichev.com/writings/RE_for_beginners-en.pdf
- http://blog.claudxiao.net/wp-content/uploads/2011/07/Elementary-ARM-for-Reversing.pdf
- http://www.sealiesoftware.com/blog/

Helpful Sites and IRC References

Websites:

- jailbreakqa.com
- www.reddit.com/r/jailbreak
- http://www.hopperapp.com/
- http://theiphonewiki.com
- http://hak5.org
- http://samdmarshall.com/re.html
- http://blog.ih8sn0w.com/2013/10/its-dumping-season.html

- https://github.com/samdmarshall/Daodan
- https://github.com/winocm/monstrosity
- http://winocm.com/research/2013/09/20/resources-for-getting-started/

Can you suggest others?

IRC

- #iphonedev
- #jailbreakqa
- IOSdev
- #openjailbreak
- #metasploit (really ?)

Sk00l Supplies
Software / Hardware

- Xcode (run latest version)
- IDA Pro
- Gdb
- OxED http://www.suavetech.com/0xed/
- Mac book running Mountain Lion
- Serial Debugging Cable
- WiFi
- Jailbroken iPhone4/5 (6.1.2)
- UnJailbroken iPhone4 (7.0.3)
- UnJailbroken 5/5S (7.0.3)

Great Starter Books

- Mac OSX and iOS Internals
- A Bug Hunter's Diary -(alas! , no ROP)
- Reverse Engineering Code with IDA Pro
- Hacking and Securing iOS Applications
- iOS Hacker's Handbook
- Mac Hacker's Handbook
- OSX and iOS Kernel Programming
- Cocoa Application Security
- C in a nutshell
- Learn C on the Mac: For OSX and iOS
- ARM Assembly Language –An Introduction