Mobile Hacking VI (Chapter 2)

m0bdev
iOS Jailbreak

@crakun
Sat, 10-26-13, 6am PST Day 6

About Chapter 2

Chapter 1 involved fuzzing .mov files (Day 2-5 of Jailbreak class)

Chapter 2 involves Reverse Engineering (RE) of Panic Logs [kernel] and Crash Reports [userland] to determine exploitability of .mov files from different perspectives and how we can leverage this in the development of m0bdev's chemrail Jailbreak.

We start with Panic Logs as we gleam into the iOS kernel

We will derive methodologies for RE, provide initial "known" background of the iOS kernel, offer primers in memory management, ARM, ROP in both kernel and userland, plus continue to develop our strategy and design for chemrail.

This will be a very long and difficult chapter for us all, but I know we can make it together if the Community gives back by helping.

This slidedeck will evolve over time as we progress thru this chapter, i.e., work in progress…

Mobile Hacking VI
Class Agenda

- Part I:
  - Goal
  - Communications
  - Our First Bug to Review as a Class
- Part II
  - Homework

Goal

Our Goal

The Goal of this class is to build an iOS untethered Jailbreak from scratch by creating a new Community-based Jailbreak Team

m0bdev
Jailbreak Team

Jailbreak Codename:

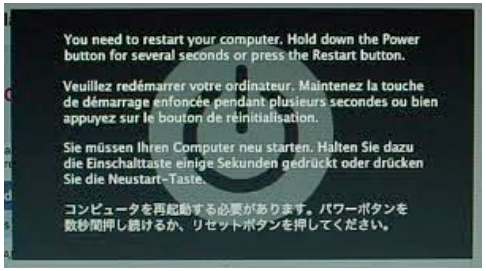
chemrail

An Analogy,…

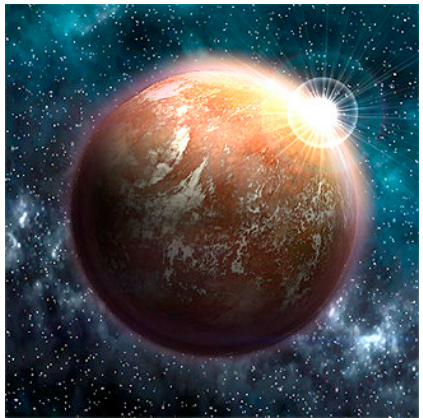
http://www.youtube.com/watch?v=oIBtePb-dGY

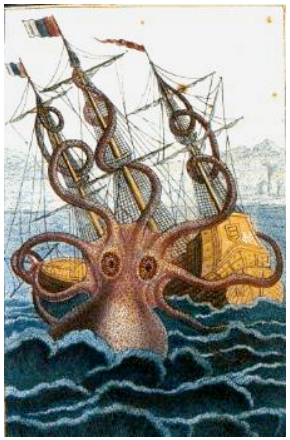
Bugs, Vulns, Exploits we are looking for in class

- 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

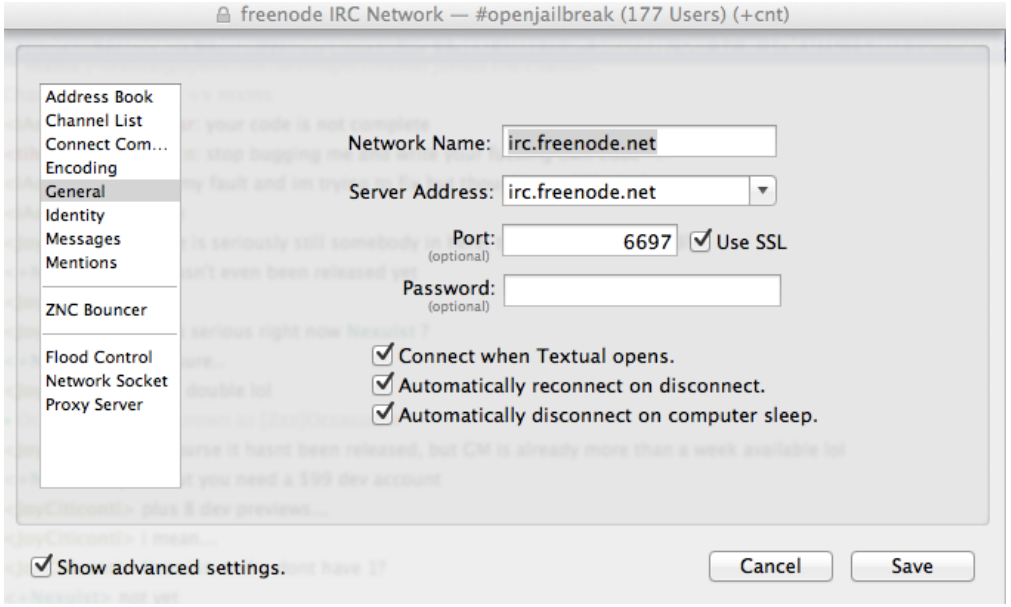
Submittal for Bugs/ Vulns /Exploits

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

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

Skype Communications is ‘limited’

Meaning, s0rry no room for :

sp00kz who don’t contribute Source Code for jailbreak

Flies on the wall

Apple

Our First Bug Review for the Class

Current Approach on Jailbreak and .mov files

p0sixninja advised:

Remote kernel exploits will be very very very hard to exploit

With KASLR, you'll need multiple primitives using different mov exploits

Probably will need to combine it with a webkit exploit to really make it work or javascript exploit

you'll need an exploit to cause a KASLR leak to bring back to user land, and dynamically generate a new mov file based on that info to send back into kernel land to exploit it

it will be hard, but not impossible

at least one KASLR leak, maybe more depending on the kernel and user land bugs

m0bd3v Jailbreak Team Members

Look at this Bug

- Submited By: compiling Entropy
- MHVI-Class_Bug: #00000001
- Kernel Panic from .mov file
- http://www.hotwan.com/class/mov/sample/1.zip

Warning:

Though unlikely , your phone may get bricked in this lab exercise.
We are not responsible for anything that happens to your phone in this lab exercise . You are.
Use at your own risk. **No Guts, no Glory.**

Reverse Engineering Tips

- We will use a variety of tools such as aptdiff, 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)

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, 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 Bruno 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.

Kernel Panic

Panic Log Analysis

Look at panic log

fault_type & register values

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

iOS Kernel Basics / Background Info

(Many Thanks to Stefan Esser for a lot of this content)

 +  = 

Twitter's gift to the Community

iOS Kernel Basics / Background Info Section of slidedeck

is a work in Progress

Finding iOS Kernel Bugs

- 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.

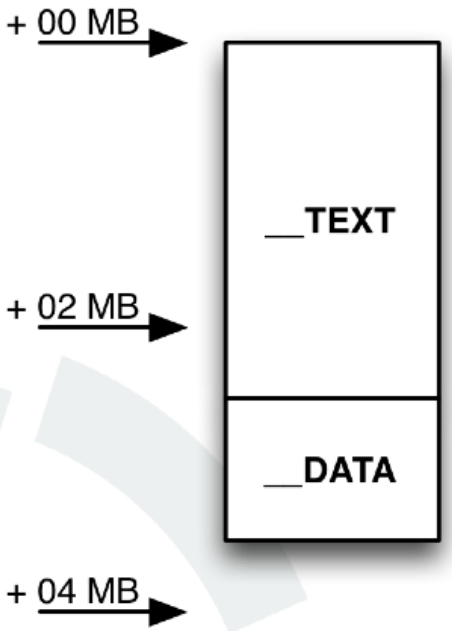
KASLR for iPhone 4 and 5

KASLR

- iOS 6 introduces KASLR - kernel address space layout randomization
- only 256 possible load addresses
- each 2 MB apart
- starting at **`0x81200000`** ending at **`0xA1000000`**

KASLR

KASLR: But why 2 MB Aligned?



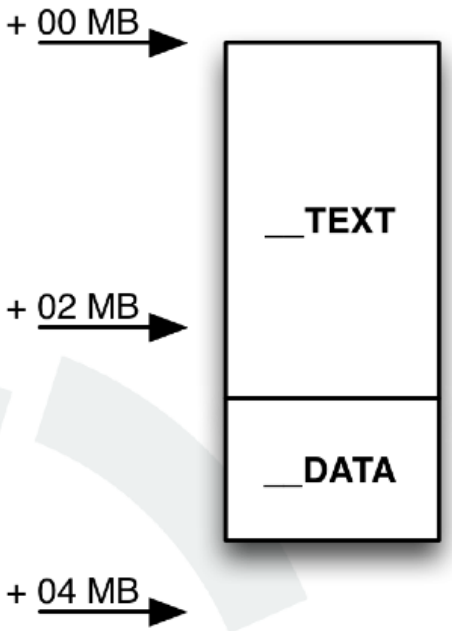
- 2 MB alignment of KASLR seems arbitrary
- why not smaller alignment?
- big alignment is less secure
- right now:
  - leak any address in __DATA and you know the kernel's base address

    (address - 0x200000) & 0xFFE00000
  - leak any address from first 2 MB of kernel __TEXT and know the kernel's base address

    address & 0xFFE00000

KASLR

KASLR: But why 2 MB Aligned?



- 2 MB alignment of KASLR seems arbitrary
- why not smaller alignment?
- big alignment is less secure
- right now:
  - leak any address in __DATA and you know the kernel's base address

    (address - 0x200000) & 0xFFE00000
  - leak any address from first 2 MB of kernel __TEXT and know the kernel's base address

    address & 0xFFE00000

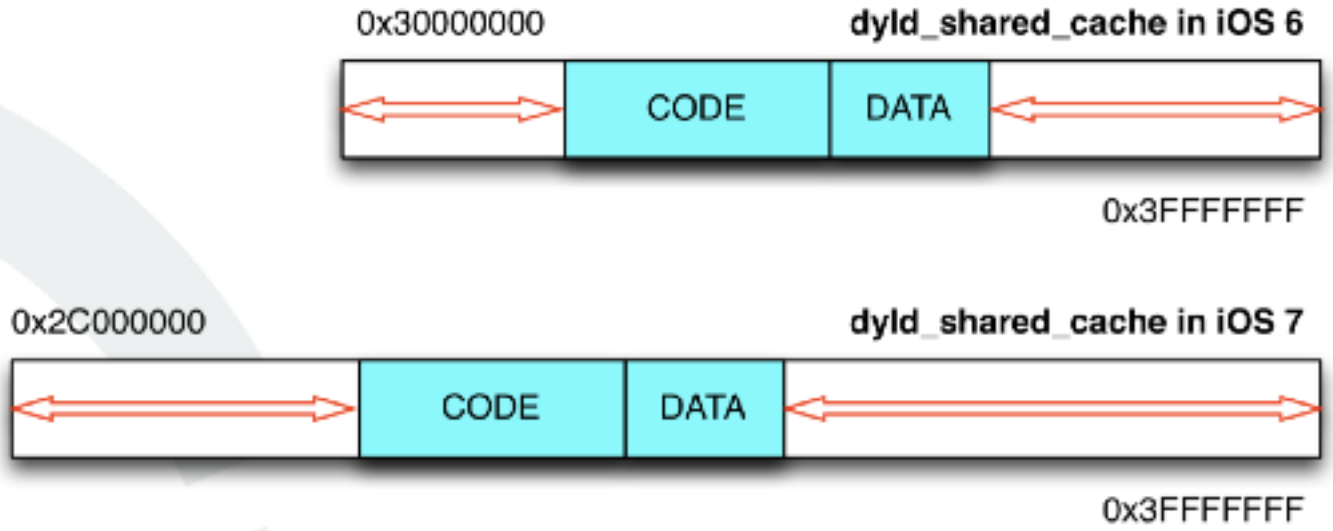
.text is where code resides
.data and .bss are data sections

Kernel Address Space Hardening

- kernel __TEXT no longer writable
  - ➡ to stop kernel code hotpatching
- kernel heap no longer executable
  - ➡ to stop just executing kernel data
- kernel address space is separated from user space processes
  - ➡ to stop return into user space code and offset from NULL-deref attacks

Library Randomization

- iOS 6 slid the dynamic shared cache between 0x30000000 - 0x3FFFFFFF
- in this 256MB window 21500 different base addresses possible (iPod 4G)
- new devices = more code = less random
- iOS 7 now slides between 0x2C000000 - 0x3FFFFFFF adds 2^13 entropy



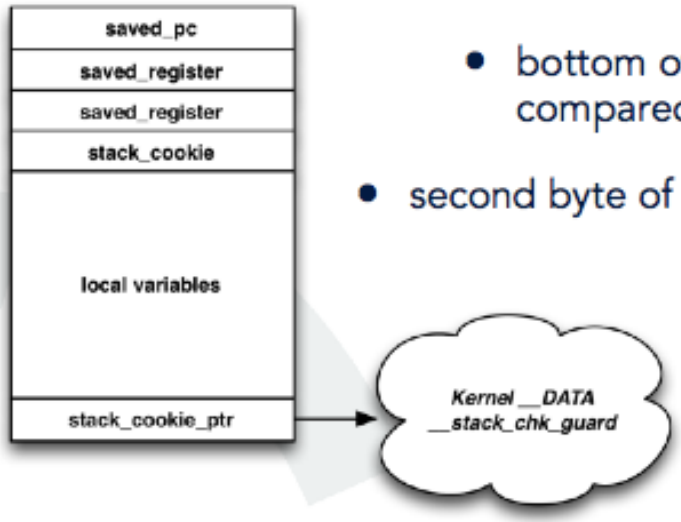
Library Randomization (64 bit)

- iPhone 5S and its 64 bit address space allows for better randomization
- separate 64 bit shared cache file /System/Library/Caches/com.apple.dyld/dyld_shared_cache_arm64
- dynamic shared cache loaded between 0x180000000 - 0x19FFFFFFF
- finally fixes the cache overlap vulnerability

Kernel Stack Cookies

Kernel Stack Cookies

- iOS 6 added stack cookies to protect from kernel stack buffer overflows
- implementation is rather unusual
  - stack cookie on top of stack
  - bottom of local stack contains ptr to the value it is compared against
- second byte of stack cookie is forced to **0x00**



Kernel Heap Cookies

Kernel Heap Cookies

- iOS 4 and iOS 5 kernel heap exploitation has always attacked the free list
- in iOS 6 Apple introduced heap protection cookies to protect free list
- distinguishes between small poisoned and larger non-poisoned blocks
- two different security cookies are used for this

➡ stops attacks against the free list as used before in public jailbreaks

Kernel Heap Cookies

Kernel Heap Cookies after allocation

- on allocation free list pointer and cookie are overwritten with **`0xdeadbeef`**
- most probably as defense in depth against information leaks

Kernel Heap Hardening

Kernel Heap Hardening

- previously **mach_zone_info()** and **host_zone_info()** leaked internal state
- both functions now require debugging kernel boot arguments

- previously **OSUnserializeXML()** allowed fine control over kernel heap
- Apple fixed some bugs in it and put some arbitrary limits on it
- only exact methods described at BlackHat / SyScan were killed
- **other ways to abuse this function for kernel heap feng shui still working**

Kernel Info Leaks

Death to Kernel Info Leaks

- two fold strategy to fight kernel info leaks
  - fix information leak vulnerabilities
  - obfuscate kernel addresses returned to user land

- example of fixed information leaks
  - **BPF** stack data info leak
  - **kern.proc** leak fixed
  - **kern.file** info leak fixed

Still a ton of information leaks available though

Kernel Info Leaks

posix_spawn() File Actions Information Leak

- by carefully crafting the data (and its size) it is possible to leak bytes from the kernel heap with a **PSFA_OPEN** file action
- choose size in a way that the beginning of the filename is from within the buffer and the end of the filename is taken from the kernel heap after it

- with **fcntl(F_GETPATH)** it is then possible to retrieve the leaked bytes

Kernel Address Obfuscation

Kernel Address Obfuscation

- lots of kernel API return kernel addresses to user land processes

  e.g. mach_port_kobject(), mach_port_space_info(), vm_region_recurse(), vm_map_region_recurse(), vm_map_page_info(), proc_info(), fstat(), sysctl()

- protected by adding a random 32 bit cookie *(lowest bit set)*

```
#define VM_KERNEL_ADDRPERM(_v)                                  \
        (((vm_offset_t)(_v) == 0) ?                             \
            (vm_offset_t)(0) :                                  \
            (vm_offset_t)(_v) + vm_kernel_addrperm)
```

```
iin->iin_urefs = IE_BITS_UREFS(bits);
iin->iin_object = (natural_t)VM_KERNEL_ADDRPERM((uintptr_t)entry->ie_object);
iin->iin_next = entry->ie_next;
iin->iin_hash = entry->ie_index;
```

Kernel Image Address Obfuscation

- some API might even return addresses inside the kernel image
- these addresses are additionally **unslid** to protect against **KASLR** leaks

```
#define VM_KERNEL_UNSLIDE(_v)                                   \
        ((VM_KERNEL_IS_SLID(_v) ||                              \
          VM_KERNEL_IS_KEXT(_v)) ?                              \
            (vm_offset_t)(_v) - vm_kernel_slide :               \
            (vm_offset_t)(_v))
#define VM_KERNEL_SLIDE(_u)                                     \
        ((vm_offset_t)(_u) + vm_kernel_slide)

#define VM_KERNEL_ADDRPERM(_v)                                  \
        (((vm_offset_t)(_v) == 0) ?                             \
            (vm_offset_t)(0) :                                  \
            (vm_offset_t)(_v) + vm_kernel_addrperm)
```

```
if (0 != kaddr && is_ipc_kobject(*typep))
    *addrp = VM_KERNEL_ADDRPERM(VM_KERNEL_UNSLIDE(kaddr));
else
    *addrp = 0;
```

System Call Table Hardening (Structure)

- in previous versions of iOS Apple has protected the table by
  - removing symbols
  - moving variables like the system call number around
- this was done to protect against easy detection in memory / in the binary
- in iOS 7 they went a step further and changed the actual structure of the system call table entries

➡ unknown if Apple did this a security protection but it makes all public detectors fail

Just replace Syscall Table completely?

- kernel linking changes in iOS 6 introduced lots of indirect accesses
- syscall table is no longer accessed directly (also true for lots of other stuff)
- instead pointer to syscall table is used from **`__nl_symbol_ptr`** section
- and guess what - this section is writable

```
__text:8021F760        LDR     R10, [R0,#0x30]
__text:8021F764        CMP     R10, #0
__text:8021F768        LDREQ   R10, [R0]
__text:8021F76C        MOV     R2, #(_pNsys - 0x8021F77C) ; _pNsys
__text:8021F774        LDR     R2, [PC,R2] ; _pNsys
__text:8021F778        MOV     R1, #(_pSysent - 0x8021F78C) ; _pSysent
__text:8021F780        UXTH    R5, R10
__text:8021F784        LDR     R1, [PC,R1] ; _pSysent
__text:8021F788        LDR     R2, [R2]
__text:8021F78C        CMP     R5, R2
__text:8021F790        BLT     loc_8021F7A0
```

```
__nl_symbol_ptr:802D2C7C _pNsys          DCD _nsys
__nl_symbol_ptr:802D2C7C
__nl_symbol_ptr:802D2C80 _pSysent        DCD _sysent
```

System Call Table Hardening (Structure)

- in previous versions of iOS Apple has protected the table by
  - removing symbols
  - moving variables like the system call number around
- this was done to protect against easy detection in memory / in the binary
- in iOS 7 they went a step further and changed the actual structure of the system call table entries

➡ unknown if Apple did this a security protection but it makes all public detectors fail

System Call Table Hardening (Access)

- in iOS 6 Apple has moved system call table into **__DATA::__const**
- this section is read-only at runtime
- protects system call table from overwrites
- but the code would access table via a writable pointer in **__nl_symbol_ptr**

- iOS 7 fixes this by using **PC** relative addressing when accessing **_sysent**

System Call Table Hardening (Variables)

- potential attack has always been tampering with the **nsys** variable
- overwriting this allowed referencing memory outside the table
- executing illegal syscalls would have resulted in execution hijack

- iOS 7 fixes this by removing access to the **nsys** variable
- maximum number of system calls is now hardcoded into the code

Sandbox Hardening

- requires more research
- but filesystem access has been locked down once more
- application containers can access fewer files in the filesystem
  - example iOS 7 disallows access to **`/bin`** and **`/sbin`**
  - applications can no longer steal e.g. **`launchd`** from **`/sbin/launchd`**

LaunchDaemon Security

- Apple added code signing for launch daemons in iOS 6.1
- but Apple forgot / or ignored **`/etc/launchd.conf`**
- **`/etc/launchd.conf`** defines commands **launchctl** executes on start
- jailbreaks like evasi0n abused this to execute arbitrary existing commands
- in iOS 7 Apple removed usage of this file

```
bsexec .. /sbin/mount -u -o rw,suid,dev /
setenv DYLD_INSERT_LIBRARIES /private/var/evasi0n/amfi.dylib
load /System/Library/LaunchDaemons/com.apple.MobileFileIntegrity.plist
bsexec .. /private/var/evasi0n/evasi0n
unsetenv DYLD_INSERT_LIBRARIES
bsexec .. /bin/rm -f /private/var/evasi0n/sock
bsexec .. /bin/ln -f /var/tmp/launchd/sock /private/var/evasi0n/sock
```

Partial Code Signing Hardening

- many jailbreaks used partial code signing vulnerabilities for persistence
- basically all those exploited the dynamic linker **dyld**
- with iOS 7 Apple has added a new function called **crashIfInvalidCodeSignature**
- function touches all segments to cause crashes if invalid signature is provided

```
int __fastcall ImageLoaderMachO::crashIfInvalidCodeSignature(int a1)
{
  int v1; // r4@1
  int result; // r0@1
  unsigned int v3; // r5@2

  v1 = a1;
  result = 0;
  if ( *(_BYTE *)(v1 + 72) )
  {
    v3 = 0;
    while ( (*(int (__fastcall **)(int, unsigned int))(*(_DWORD *)v1 + 208))(v1, v3)
         || !(*(int (__fastcall **)(int, unsigned int))(*(_DWORD *)v1 + 200))(v1, v3) )
    {
      ++v3;
      result = 0;
      if ( v3 >= *(_BYTE *)(v1 + 72) )
        return result;
    }
    result = *(_DWORD *)(*(int (__fastcall **)(int, unsigned int))(*(_DWORD *)v1 + 236))(v1, v3);
  }
  return result;
}
```

iOS Kernel for 6.x

iOS Kernel

- user-land dereference bugs are partially exploitable
- privilege escalation to root just a starting point
- memory corruptions or code exec in kernel always required
- kernel exploits only triggerable as root are interesting

Types of Kernel Exploits

normal kernel exploits

- privilege escalation from "mobile" user in applications
- break out of sandbox
- disable codesigning and RWX protection for easier infection
- must be implemented in 100% ROP

untethering exploits

- kernel exploit as "root" user during boot sequence
- patch kernel to disable all security features in order to jailbreak
- from iOS 4.3.0 also needs to be implemented in 100% ROP

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

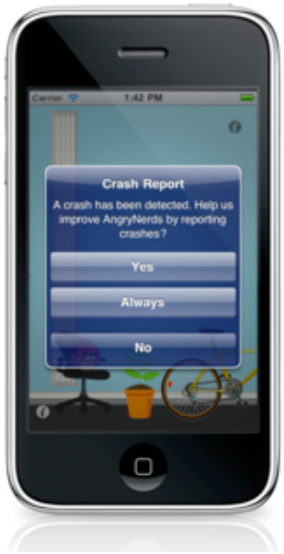
ROP in Kernel

- kernel level ROP very attractive because limited amount of different iOS kernel versions
- just copy data from user space to kernel memory
- and return into it

ROP in Kernel

- integer overflow when allocating kernel memory
- leads to a heap buffer overflow
- requires root permissions

Crash Report Analysis



Crash Reports

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

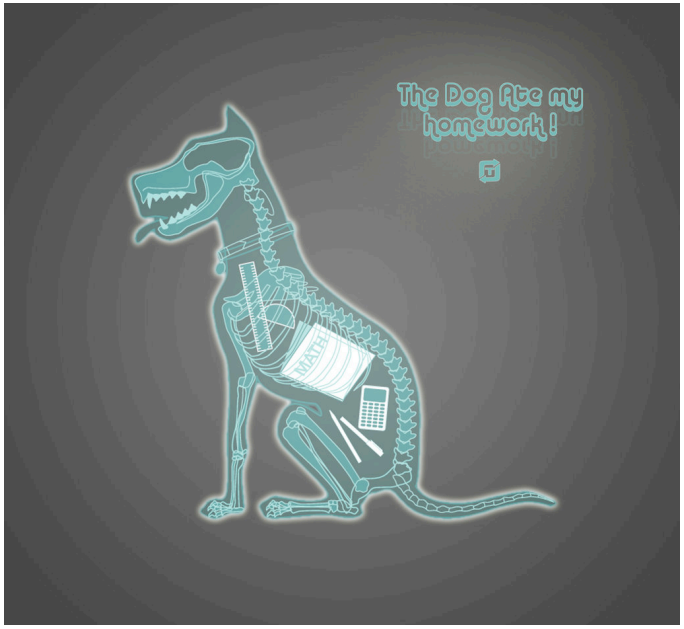
- Verify the crash dump in Settings / General / About / Diagnostics & Usage / Diagnostic & Usage Data that the crash report you created.

Crash Report References

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

Got some better more links / info ?

Homework



Review of YOUR Oct 19 Homework Assignment

.mov Bug Submittals are now Due

New comers and late people should submit

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
- Begin Validate Bug on your device and Begin Reverse Engineering today's Class Bug #00000001
- New Comers:
  - Fuzz .mov files / Submit
- Review previous

Homework Assignment
Your Next Steps -Validation

- **As a precaution, compare original and mutated file with AptDiff first**

- Validate kernel panic should work on:
  - Jailbroken iPhone4/5 (6.1.2)
  - UnJailbroken iPhone4 (7.0.3)
  - UnJailbroken 5/5S (7.0.3)

- If the .mov file is not creating a kernel panic on your device / firmware version, can you fuzz the original file to create a kernel panic on your device?

- Verify/ Debug to determine exploitability

Email Me your findings: crakun@m0bdev.com

Homework Assignment
Next Steps – Reverse Engineering

- Use AptDiff to compare the original good .mov file to the fuzzed, mutated file
- Change each difference back, to identify the byte that caused the fault
- If we can determine that we have some level of control over the instruction pointer (Program Counter, PC in ARM) or structured exception handler, we can quickly determine that the crash is in fact exploitable.
- However, when this is not the case, analysis of crash data becomes significantly more difficult.
- Determining if a crash allows us to corrupt portions of memory or manipulate structures require a great amount of time and understanding of the application.
- Use the following slides (13-15) for examples in reverse engineering

Can you fill PC with 41414141?

Can you determine the root cause of the kernel panic?

Next Meeting:
(Day 6): Sat, Nov 2, 2013 6am PST

- Status check class' Reverse Engineering of selected .mov file.
- Bring your findings and be prepared to talk about it
- Does anyone have control of PC at this time?
- Weekly Class: Saturdays, 6am PST

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/

Cykey https://github.com/Cykey/mobilesafari-fuzzer

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

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

HOtWAN

Submittal for .mov Crashes

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

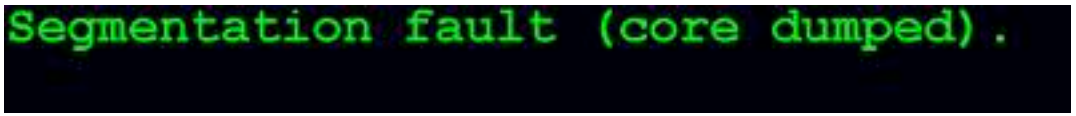
1. Which phones and firmware were the .mov file tested on

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

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

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 (7.0.2)
- UnJailbroken 5/5S (7.0.2)

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