



## iPhone Rootkit? There's an App for that!

Eric Monti – Sr. Security Researcher

# \$ id

---

```
uid=501(emonti) gid=501(emonti) groups=0(wheel),42(tw_spiderlabs),69(security_nerds)
```

## **Eric Monti**

Senior Security Researcher

Trustwave - SpiderLabs Research

~15 years in the industry

Worn hats for everything from sysadmin to software engineering

## **Nowadays:**

Interests mostly in vulnerability research and reverse engineering

Mobile device security is my latest obsession

# My Motivation

---

*I am not a iPhone Jailbreak team member - Just an avid fan*

## **JailbreakMe.com 2.0 Launched around August 2010**

- Whole security community got really intrigued
- I'd been focused on product engineering for several months
  - Really enjoy it (not knocking it!!!) but sometimes I missed vulns
- Pen-testers officially propose a weaponized exploit
  - “It'd be cool to demo an iPhone rootkit to clients”
- !!! \o/ !!! Sounds fun!
- Dropped everything. Started reverse engineering jailbreakme.com
- Soon found myself experimenting with iPhone and App-Store backdoors

# Presentation Overview

---

## Understanding Jailbreaks

- Understand iOS security measures and challenges
- See how JB teams apply attack patterns
- Get some fun reversing in

## ... to leverage the same techniques for rootkit purposes

- “Malicious” exploits - in the PoC sense
- Rootkits - also in the PoC sense ... *please don't root my phone "for reals"*

## No 0-day, the “star” of the show isn't even mine

- The bug is old news since research began
- Wasn't found by me to begin with
- Jailbreak team members are pretty rockstar!!!
- This is one of those “process” presentations

***This time out, I'm going into more post-exploitation and app reversing***





## iPhone/iOS Security Overview

# iOS Security From 1,000 Feet

---

## Nutshell:

- Bootloader verifies...
- signed firmware verifies...
- signed kernel verifies...
- signed Applications installed from the app store
  
- Apple signed everything!

Actually a pretty sound design (barring implementation problems)

# Architecture Overview

---

## Applications Processor

- ARM (6 or 7 depending on idevice/version)
- XNU Based Kernel (like OS X-lite on ARM)
- Implements Kernel and Application Signing from bootloader down.

## Baseband Modem

- Another ARM, handles GSM connectivity
- Separated from App. Processor with its own RAM and FLASH
- Mostly interesting to carrier unlocks, but not my rootkits (yet?)

## Hardware Encryption Introduced in iPhone 3GS

- Low-level data encryption on NAND storage –
  - Idea is that you drop the key, and FS can't be read.
- A silly feature, really:
  - 'The remote wipe feature as well as 'Find my iPhone' can be disabled by removing the iPhone's SIM card.' – Jonathan Zdziarski

# OS Environment

---

## Two partitions make up filesystem

- Root partition at / (read-only from factory)
  - Kernel, Base OS, Core APIs
- User Partition at /private/var (read-write)
  - All third party apps
  - User data

## Two users for pretty much everything

- “root” - system services, kernel
- “mobile” - apps and data running as you, the user
- Basic Unix security model applies

## System libraries and APIs approximate OS X / Darwin

# Application Security

---

## Code signing

- All exe's from the AppStore must be signed by Apple
- Signatures stored in mach-o header section
- Check implemented in kernel as through an enhanced exec() system call

## Sandbox

- Applications run as user "mobile"
- Chroot sandbox (ostensibly) restricts apps to their own data
- Can't access the OS or other apps' data
- Entitlements also restrict some functionality
  - Programs need special entitlements for things like debugging
  - Entitlements are trivial to add, but during exploitation this factors in

# Reality

---

## **Apple's .app authorization process is probably the biggest iOS security feature**

- Private APIs are accessible but apps using them are usually rejected
- So low-level functionality is almost all there, just not "approved of"

*Meaning...*

*Exploit code running in signed apps or on jailbroken devices can still do lots of interesting things. The system is pretty much a full XNU-based darwin platform and entitlement restrictions leave us a lot of elbow room.*

**Apple "audits" every app in the store. Think they don't miss bugs?**



## Jailbreaking Overview

# Jailbreak Landscape

---

## **Remote client-side exploits are few and far between**

- Highly valuable
- Obviously more potential for abuse
- Obviously more exciting for security research

## **Most jailbreaks exploit vulns in restore and FW updates over USB**

- Fertile territory for jailbreaks, JB nerds, and regular nerds to follow (like me)
- Security impact for 'evil maid' style bad-guy attacks

## ***Very impressive work is consistent from the JB community***

- It takes a real !\$\$-hole to taint their awesome efforts...
- But this is just how I do adoration and idolization

## **Internets have loads of tech details for learning**

- Patience! Gotta wade through lots of Apple fanboi and tech writer blogs to find the good stuff
- JB teams have cool info on wikis, but it's not always up to date
- Github!!! Jailbreak-team stalker's paradise!



# Jailbreakme.com A Thing to Behold...

---

## Author: Comex backed up by other jailbreak team members

- The actual exploit and jailbreak package dubbed “star”
- Worked on every iDevice Apple made and across almost all modern versions.
- iPhone 4G out for just a month or so.
  - Jailbreak users had been waiting patiently and were not disappointed
- Released right after a crucial US legal decision on jailbreaking
  - It’s now officially legal in US. Prior status was fuzzy
- Source for exploit released after Apple releases security fix (iOS 4.0.2)
  - See <http://github.com/comex/star>
- Handled everything like pros
  - Implementation, to presentation, to disclosure, to the timing of the release

# The What

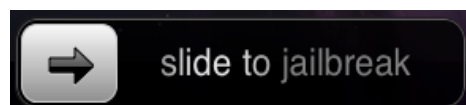
---

Instant, convenient jail-breaking using nothing more than your phone's data connection and the built-in Safari web browser

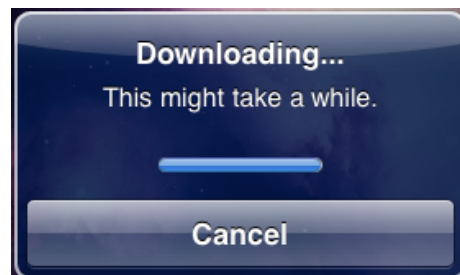
- No iPhone is safe. Guy walked into apple store and filmed himself jailbreaking their floor model! Posts to You Tube, hilarity ensues
- Actually the second published web-based jailbreakme.com exploit
  - First was a based on a libtiff vulnerability from Tavis Ormandy
  - Similar security analysis and exploitation undertaken for libtiff by HD Moore/metasploit team and others

# What it looks like

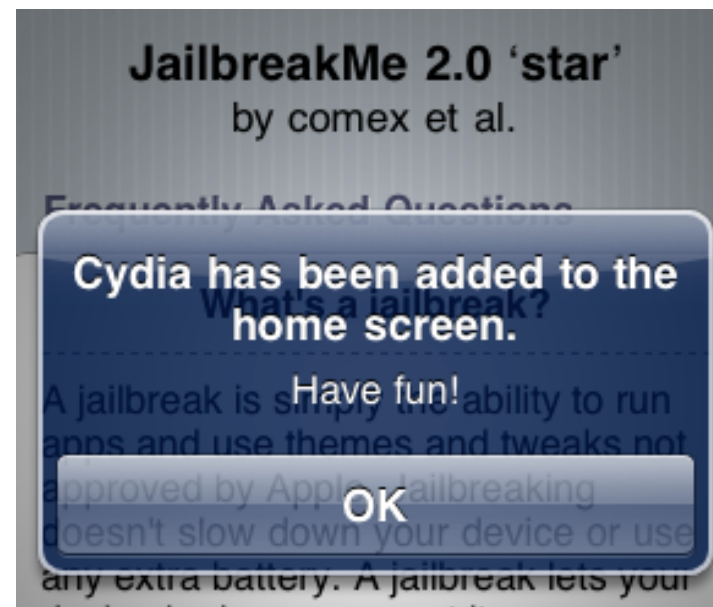
Visit  
<http://jailbreakme.com>



Star exploit execution



Finished. Pretty safe and easy!



*Note: If we want a "stealth jailbreak", we need to do away with all these prompts.*

# The How

---

## The “star” PDF Exploit – Code execution

- Classic stack overflow
- BoF in CoreGraphics CFF(Compact Font Format) handling long strings
- Overwrites \$pc (EIP for ARM)
- Code still running as “mobile” at this point
- Leverages IOSurface (IOKit) bug for privilege escalation and sandbox escape

## The IOKit Vulnerability – Priv. escalation / breaks out of the sandbox

- Kernel integer overflow in handling of IOSurface properties
- Calls setuid(0) making Safari and all its loaded exploit code run as root
- Dominoes all fall down from there

## The Jailbreak Phase – Setting up residence on the iDevice

- Patches out restrictive kernel functionality
- Installs a basic jailbreak filesystem along with Cydia (APT based package manager)
- “Polite” and clean - *Even calls setuid(501) back to “mobile” once it’s finished.*



**Weaponizing**

# Reversing the "star" Exploit (pre-source)

## First few weeks, no source was released for JailbreakMe.com

- I was curious and impatient. Wasn't sure if comex would release
- Began reversing the binaries within a few days of the JB release
  - Staring at strange hex-dumps and peeling the onion one layer at a time
  - Fun and soothing – *Like catnip for my O.C.D.*

```

1 25 50 44 46 2D 31 2E 33 %PDF-1.3
} 0A 25 C4 E5 F2 E5 EB A7 .%......
1 F3 A0 D0 C4 C6 0A 34 20 .....4
} 30 20 6F 62 6A 0A 3C 3C 0 obj.<<
1 20 2F 4C 65 6E 67 74 68 /Length
} 20 36 33 31 20 3E 3E 0A 631 >>.
    
```

```

13 0 obj
<<
/Subtype/Type1C
/Filter[/FlateDecode]
/Length 10908
>>
stream
x<9c>1}^MPTx<95>ame^E<8d>hU-#L^Y^Kc!d<90>1" [
8>-gGNH<96>î<90>Düh<84>^S<9c>0VÊ^N<8e>=a26ii
mypp;<8d>e!Q0AY^GAB^_yî^Ei^V^70^<89>|<83>Á
/η_î-ýý:èη;ñVú<8c>[0L^3;Èd^E;<93>%00<91>
8b>^ei?.7B6<97>^Pμ<89>à^Vú#U<8d>η<92>^<8d>
!)ÿ<89>^W^Kfá^GD.<92>]N0<94>0^Á<83>È{)/W^Á<9
0f>0xf^Mm5i<97>f^1_4^1<93>^Fu#<85>1<8b>78
    
```

```

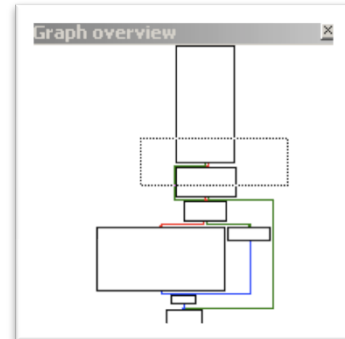
Terminal — bash — 126x23
13 41 42 43 44 45 46 2b |.....ABCDEF+|
6d 61 6e 00 01 01 01 1f |Times-Roman.....|
03 f8 19 04 1c 6f 00 0d |.....0..|
05 e9 11 8b 8b 12 00 03 |<.<n.<|.....|
2e 30 30 37 54 69 6d 65 |.....001.007Time|
69 6d 65 73 00 00 00 02 |s RomanTimes....|
05 00 00 04 dc 0e 0e 0e |.....|
00 00 00 ff 00 00 00 00 |.....|
    
```

```

niko:dump_iPod1,1_3.1.3 emonti$ file egg macho_1 macho_2
egg: data
macho_1: Mach-O dynamically linked shared library arm
macho_2: Mach-O dynamically linked shared library arm
niko:dump_iPod1,1_3.1.3 emonti$
    
```

```

_text:0000209c
_text:0000209c _lui_go
_text:0000209c var_10
_text:0000209c = -0x10
_text:000020a0 STMFD SP!, {R4-R7,LR}
_text:000020a4 ADD R7, SP, #0xc
_text:000020a8 SUB SP, SP, #4
_text:000020ac MOV R4, R0
_text:000020b0 LDR R0, =(cfstr_lui_go0neP0ne_ - 0x2000)
_text:000020b4 MOV R6, R2
_text:000020b8 MOV R5, R1
_text:000020bc ADD R0, PC, R0 ; "lui_go: one=3p one_len=3d"
_text:000020c0 BL _NSLog
_text:000020c4 LDR R0, =(cfstr_0ne0 - 0x2000)
_text:000020c8 LDRB R1, [R5]
_text:000020cc ADD R0, PC, R0 ; "one = %d"
_text:000020d0 BL _NSLog
_text:000020d4 LDR R1, =(bus - 0x2000)
_text:000020d8 MOV R0, R0x0 ; int
_text:000020dc ADD R1, PC, R1 ; void (*)(int)
_text:000020e0 BL _signal
_text:000020e4 LDR R0, =(off_397c - 0x20f0)
_text:000020e8 LDR R1, =(off_39f4 - 0x20f4)
_text:000020ec LDR R0, [PC,R1]
_text:000020f0 BL _objc_msgSend
_text:000020f4 LDR R1, =(off_393c - 0x2100)
_text:000020f8 MOV R2, R5
    
```



# Patch Plan

---

Reversing the installui.dylib and wad.bin provided guidance.  
To quickly turn around a weaponized jailbreak, we'd need to...

- Patch out a "security" check comex had incorporated
  - The jailbreakme.com PDFs' installui.dylib had code to ensure they'd been downloaded from "jailbreakme.com". I couldn't leave that
  - Not sure what motivation Comex had for this
- Patch out all the gui pop-ups
  - Didn't want the victim to realized they were being `kitted
  - I hadn't learned the wonders of usbmuxd and libimobiledevice for live syslog yet so I left a single popup for debugging/troubleshooting
  - Would patch it out last
- Prepare a modified wad.bin containing our "rootkit"
  - Started out just shooting for getting the actual jailbreak to download and install quietly from a server I controlled

# “All for naught”?

---

**Got it working. Super happy! Then it turned out to be a total waste of time. Or was it?**

- Comex released the source about a week after I'd finished testing my first PoC

See: *<http://github.com/comex/star>*

- No use crying over spilled code. Now we just fork and patch the github project

See: *<http://github.com/emonti/star>*

- But I'd still had more fun the other way. My custom “star” reversing tools for historic sake

See: *[http://github.com/emonti/star\\_reversing\\_tools](http://github.com/emonti/star_reversing_tools)*



# My "Big Fat Rootkit"... so far

Custom-written and patched 3<sup>rd</sup> party code for backdoors and kit

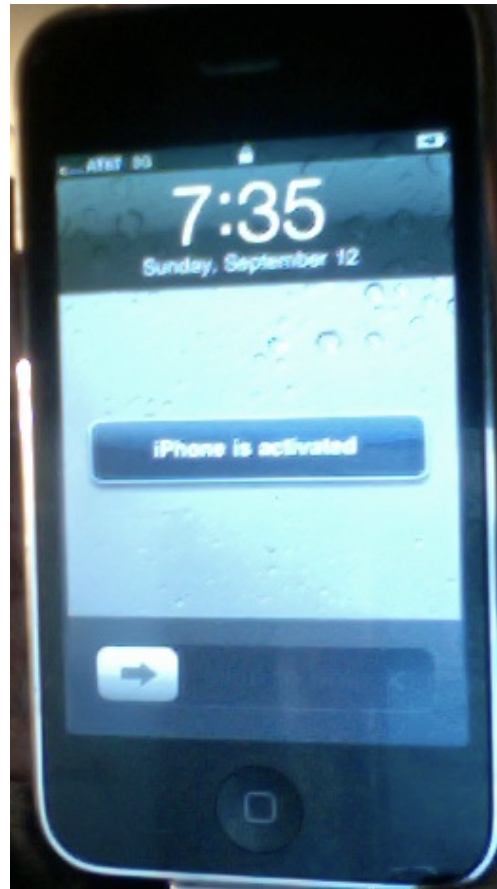
- I call it "Fat" because it weighs in too large to be considered "stealthy"
- Userland rockin' it like it's 1990
  - Why not: Apple did most of the hard work hiding the underlying system for us
- Includes lots of the jailbreak base, but no cydia or other obvious signs of entry
  - MobileSubstrate and other components turn out to be very handy (more later)
- UDP knockd called "bindwatch" fakes its name on argv[0]
  - Knockd Spawns a bind shell called, wait for it .... "bindshell" also fakes argv[0]
- Patched "veency" to stay under the hood
  - Nice opensource iPhone VNC server by saurik
  - Runs via a DYLIB in MobileSubstrate
  - Mostly just removed the GUI config plist from System Preferences
  - Coded a trivial CLI tool to strap and start veency via darwin notifications without the GUI
- Developing some targeted backdoors for "interesting" App Store application classes
  - More on this later...



## Rootkitting Demo

# Set-up

**A vanilla un-jailbroken iPhone 3g running iOS 4.0.1**



# So What Now?

## Plenty of interesting locations to explore on a rooted iPhone filesystem

- Emails
  - /var/mobile/Library/Mail/
    - “Protected Index”
      - (SQLite for message metadata)
    - “Envelope Index”
      - (SQLite for email folders metadata)
    - IMAP-victim@victimco.com@imapserver.victimco.com/
      - (mailbox for your IMAP accounts)
    - ExchangeActiveSyncXXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXXX/
      - (mailbox for victim’s exchange GUID XX...)
    - ...etc.
  - Voicemails
    - /var/mobile/Library/Voicemail/
      - voicemail.db
        - (SQLite for message metadata)
      - /var/mobile/Library/Voicemail/\*.amr
        - (Audio – Download and open in Quicktime)
  - SMS Messages
    - /var/mobile/Library/SMS/
      - sms.db
        - (SQLite for message metadata)
      - Parts/...
        - (TXT msg’s and msg parts for MMS)

# Hardware: Mic and Audio

---

## Audio recording apps are the “hello world” for the iPhone

Google Core Audio and AudioToolBox.framework

### Audio File Services

Audio File services lets you read or write audio data to and from a file or buffer. You use it in conjunction with Audio Queue Services to record or play audio. In iOS and Mac OS X, Audio File Services consists of the functions, data types, and constants declared in the `AudioFile.h` header file in `AudioToolbox.framework`.

My first rootkit app was “playaudio”

*First rootkit fart app... “playaudio fart.aif”*

My second was “recordaudio”

*Only slightly harder. Lots of tutorials.*

# Hardware Capture: Location

- CoreLocation is cool, but requires user approval for GPS location
  - Can hijack a process that already has approval
  - ... or go under the API layer that actually enforces approval
  - ... or ... hack approval with a patch.
  - Investigating lower CoreLocation layers.
- Poor-mans approach:
  - Dump a recent LAT/LONG from the locationd cache file
  - Extracted from `/var/root/Library/Caches/locationd/cache.plist`

```
WifiLocationNearby = {  
    Altitude = 0;  
    HorizontalAccuracy = 80;  
    Latitude = "41.882041";  
    Lifespan = 144;  
    Longitude = "-87.628489";  
    Timestamp = "304907008.940135";  
    Type = 4;  
    VerticalAccuracy = "-1";  
};
```

# Dumping Process Data

## Several interesting persistent Apple processes running

- For example: "dataaccessd" handles email connectivity
- Memory regions with 'rw' set are more likely to be program data
- **0x100000** happened to be an interesting region in this case

```
iPhone:/var/mobile root# ps -hax |grep dataaccessd
 38 ??          0:05.33 /System/Library/PrivateFrameworks/DataAccess.framework/Support/dataaccessd
...
iPhone:var/mobile root# gdb --quiet --pid=38
Attaching to process 38.
Reading symbols for shared libraries . done
Reading symbols for shared libraries ..... done
0x3404c658 in mach_msg_trap ()
(gdb) info mach-regions
Region from 0x0 to 0x1000 (---, max ---; copy, private, not-reserved)
... from 0x1000 to 0x2000 (r-x, max r-x; copy, private, not-reserved)
... from 0x2000 to 0x3000 (rw-, max rw-; copy, private, not-reserved)
...
... from 0xfe000 to 0xff000 (r--, max rwx; share, private, reserved)
... from 0x100000 to 0x400000 (rw-, max rwx; copy, private, not-reserved) (3 sub-regions)
...
(gdb) dump memory dump_100000.bin 0x100000 0x400000
```

# Oh !@#\$. My Cached Corp. OWA Login

... continued

```
(gdb) dump memory dump_100000.bin 0x100000 0x400000
```

...

```
$ strings dump_100000.bin |grep -B6 -A2 'Authorization: Basic'
```

```
POST /Microsoft-Server-ActiveSync?User=emonti&DeviceId=ApplnnnnNP&DeviceType=iPhone&Cmd=Ping HTTP/1.1
```

```
Host: owa.mycompany.com
```

```
Content-Length: 0
```

```
Ms-Asprotocolversion: 12.1
```

```
User-Agent: Apple-iPhone2C1/801.306
```

```
X-Ms-Policykey: 550504473
```

```
Authorization: Basic bXljb21wYW55LmNvbVxlbW9udGk6bm90ZnVja2luZ2xpa2VseSE=
```

```
Accept: */*
```

```
Accept-Language: en-us
```

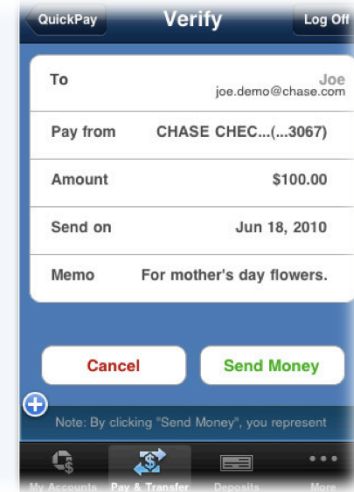
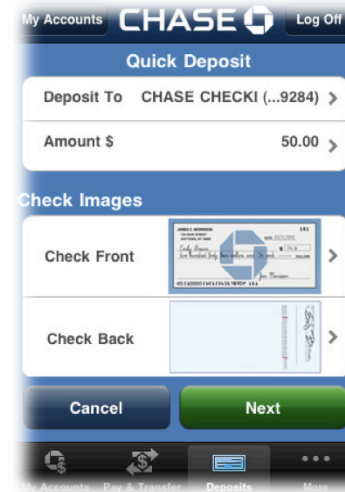
**\* Alternately See:** [http://github.com/emonti/gcore\\_arm](http://github.com/emonti/gcore_arm)

*Still working out minor kinks, but it (usually) gets a full core-dump without gdb and without killing the target. Downside, full core-dumps are usually huge.*



# Targeting iOS Applications

iOS isn't just for fart apps anymore...



## ScadaMobile

Logic Controller data at your fingertips

# Pick an “interesting” target in the App Store

**I chose “Square” because... well it’s free.**

**As in free!**

... and I the promo video cracked me up





## Demo App Backdoor

# Decrypt The Binary for Reversing

## Crackulous, xcrack, etc. But how do they work?

1. Find the app binary on the iDevice

```
# find /var/mobile/Applications -name SomeApp.app  
/var/mobile/Applications/0578A160-.../SomeApp.app
```

2. Get the size of the encrypted code/data section

```
# otool -l .../0578A.../SomeApp.app/SomeApp |grep -A4 LC_ENCRYPTION_INFO  
cmd LC_ENCRYPTION_INFO  
    cmdsize 20  
    cryptoff 4096  
    cryptsize 4096  
    cryptid 1
```

3. Load the executable in a debugger (gdb is a cydia package)

```
# gdb .../SomeApp.app/SomeApp
```

*See: [dvlabs.tippingpoint.com/blog/2009/03/06/reverse-engineering-iphone-appstore-binaries](http://dvlabs.tippingpoint.com/blog/2009/03/06/reverse-engineering-iphone-appstore-binaries)*

# Binary Reversing Prep (continued...)

4. Observe the encrypted section "before" -- garbage instructions:

```
(gdb) x/3i 0x2000
0x2000:      addge   r4, r7, r4, asr r11
0x2004:      bl      0xfe147a48
0x2008:      ldrbcc  r5, [r4, #3476]
```

5. Set a BP to fire after decryption, and let iOS decrypt for us just by running it

```
(gdb) break *0x2000
Breakpoint 1 at 0x2000
(gdb) r
...
Breakpoint 1, 0x00002000 in ?? ()
gdb) x/3i 0x2000
0x2000:      ldr     r0, [sp]
0x2004:      add    r1, sp, #4      ; 0x4
0x2008:      add    r4, r0, #1      ; 0x1
```

# Binary Reversing Prep (continued...)

6. Dump the decrypted section to a file based on "cryptsize" from step #2 (4096)  
(gdb) dump memory decrypted.bin 0x2000 (0x2000 + 4096)

7. Merge decrypted.bin back into SomeApp using your favorite hex editor.

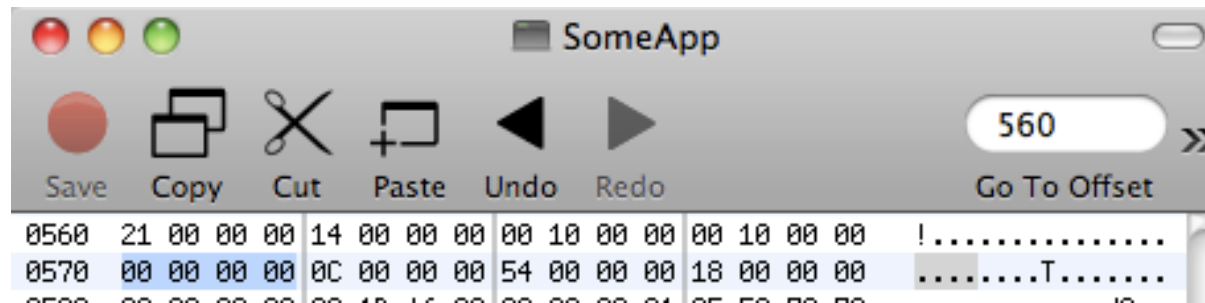
- *Relative file offset for the encrypted data will be 0x1000 away from the mach header (aka cryptoff) .*

8. Don't forget to disable "cryptid" so that class-dump and other tools will work on our fixed up binary. Change Cryptid to 0.

# No More Secrets

*Hint: find the LC\_ENCRYPTION\_INFO(0x21) load command by converting “otool -l” output back to 32-bit little endian hex and searching for it.*

```
cmd LC_ENCRYPTION_INFO (21000000h)
  cmdsize 20           (14000000h)
  cryptoff 4096        (00100000h)
  cryptsize 4096      (00100000h)
  cryptid 1           (00000000h)
```



# Still Curious About Apple's Binary Encryption?


So was I

**TIP:** Encryption code portions are in the open XNU source

[http://opensource.apple.com/source/xnu/xnu-1228.9.59/bsd/kern/mach\\_loader.c](http://opensource.apple.com/source/xnu/xnu-1228.9.59/bsd/kern/mach_loader.c)

Code responsible for parsing mach-headers on loading mach or FAT binaries

```
#if CONFIG_CODE_DECRYPTION
    case LC_ENCRYPTION_INFO:
        if (pass != 2)
            break;
        ret = set_code_unprotect(
            (struct encryption_info_command *) lcp,
            addr, map, vp);
        if (ret != LOAD_SUCCESS) {
            printf("proc %d: set_code_unprotect() error %d "
                "for file \"%s\"\n",
                p->p_pid, ret, vp->v_name);
            /* Don't let the app run if it's
             * encrypted but we failed to set up the
             * decrypter */
            psignal(p, SIGKILL);
        }
        break;
#endif
```





# Mach Binary Encryption (continued...)

[http://opensource.apple.com/source/xnu/xnu-1228.9.59/bsd/kern/mach\\_loader.c](http://opensource.apple.com/source/xnu/xnu-1228.9.59/bsd/kern/mach_loader.c)

```
#if CONFIG_CODE_DECRYPTION

static load_return_t
set_code_unprotect(
    struct encryption_info_command *eip,
    caddr_t addr,
    vm_map_t map,
    struct vnode *vp)
{
    int result, len;
    char vpath[MAXPATHLEN];
    pager_crypt_info_t crypt_info;
    const char * cryptname = 0;

    size_t offset;
    struct segment_command_64 *seg64;
    struct segment_command *seg32;
    vm_map_offset_t map_offset, map_size;
    kern_return_t kr;

    ...

    /* set up decrypter first */
    if(NULL==text_crypter_create) return LOAD_FAILURE;
    kr=text_crypter_create(&crypt_info, cryptname, (void*)vpath);

    if(kr) {
        printf("set_code_unprotect: unable to create decrypter %s, kr=%d\n",
            cryptname, kr);
        return LOAD_RESOURCE;
    }
}
```



# Mach Binary Encryption (continued...)

[http://www.opensource.apple.com/source/xnu/xnu-1228.9.59/osfmk/kern/page\\_decrypt.h](http://www.opensource.apple.com/source/xnu/xnu-1228.9.59/osfmk/kern/page_decrypt.h)

```
/*
 *Interface for text decryption family
 */
struct pager_crypt_info {
    /* Decrypt one page */
    int      (*page_decrypt)(const void *src_vaddr, void *dst_vaddr,
                           unsigned long long src_offset, void *crypt_ops);
    /* Pager using this crypter terminates - crypt module not needed anymore */
    void      (*crypt_end)(void *crypt_ops);
    /* Private data for the crypter */
    void      *crypt_ops;
};
typedef struct pager_crypt_info pager_crypt_info_t;

typedef int (*text_crypter_create_hook_t)(struct pager_crypt_info *crypt_info,
                                         const char *id, void *crypt_data);
extern void text_crypter_create_hook_set(text_crypter_create_hook_t hook);
//extern kern_return_t text_crypter_create(pager_crypt_info_t *crypt_info, const char *id,
//                                         void *crypt_data);
extern text_crypter_create_hook_t text_crypter_create;
```

[http://www.opensource.apple.com/source/xnu/xnu-1228.9.59/osfmk/kern/page\\_decrypt.c](http://www.opensource.apple.com/source/xnu/xnu-1228.9.59/osfmk/kern/page_decrypt.c)

```
text_crypter_create_hook_t text_crypter_create=NULL;
void text_crypter_create_hook_set(text_crypter_create_hook_t hook)
{
    text_crypter_create=hook;
};
```

# What's your point?

**For starters... text encryption is an OS X feature also!**

```
MySnowLeopardMac$ nm /mach_kernel |grep text_crypter  
ffffff8000623410 D _text_crypter_create  
ffffff800027cdac T _text_crypter_create_hook_set
```

*Will we see this used for the Mac App store?*

Not the same thing as DSMOS ("**Don't Steal Mac OS X**") binary protection but bears some resemblance to it:

<http://www.osxbook.com/book/bonus/chapter7/binaryprotection/>

All I'll say is beyond that is "ongoing research"

Anybody knows more about this stuff, I'd like to buy you some beers.  
(extra beers if you are under an Apple NDA and "shouldn't" talk about it!)

# But Remember...

**DBADB == "Don't Be A D-Bag"**



Stealing is lame.

# Objective-C Method Hooking

It's called a "swizzle"



# Examine our “prepared” Binary

## Did I mention class-dump?

```
/*
 *   Generated by class-dump 3.3.2 (64 bit).
 *
 *   class-dump is Copyright (C) 1997-1998, 2000-2001, 2004-2010 by Steve Nygard.
 */
/* ... */

@interface SomeApp_AppDelegate <UIApplicationDelegate>
{
    UIWindow *window;
}

- (void)applicationDidFinishLaunching:(id)arg1;
- (id)doSomethingWithArg:(id)arg1;
- (void)dealloc;
@property(retain) UIWindow *window; // @synthesize window;

@end
```

# Objective-C Method Hook Example

Lets "swizzle" [NSObject -init]

```
#import <Foundation/Foundation.h>
#import <objc/runtime.h>
#import <objc/message.h>

// This macro sets up a hook into the objective-C runtime
#define HookObjC(cl, sel, new, bak) \
    (*(bak) = method_setImplementation(class_getInstanceMethod((cl), (sel)), (new)))

// Holds a pointer to the original [NSObject init]
static IMP orig_init;

// our overridden [NSObject init] hook
id hook_init(id _self, SEL _cmd) {
    NSLog(@"Class Initialized: %@", [_self class]);
    return orig_init(_self, _cmd);
}

// compile with -dynamiclib -init _override_init
void override_init(void) {
    // hook [NSObject init]
    HookObjC(objc_getClass("NSObject"),
             @selector(init),
             (IMP) hook_init,
             (IMP *) &orig_init);
}
```



# Objective-C Method Hook Example

Lets "swizzle" [NSObject -init]

```
#import <Foundation/Foundation.h>
#import <objc/runtime.h>
#import <objc/message.h>

// This macro sets up a hook into the objective-C runtime
#define HookObjC(cl, sel, new, bak) \
    (*(bak) = method_setImplementation(class_getInstanceMethod((cl), (sel)), (new)))

// Holds a pointer to the original [NSObject init]
static IMP orig_init;

// our overridden [NSObject init] hook
id hook_init(id _self, SEL _cmd) {
    NSLog(@"Class initialized: %@", [_self class]);
    return orig_init(_self, _cmd);
}

// compile with -dynamiclib -init _override_init
void override_init(void) {
    // hook [NSObject init]
    HookObjC(objc_getClass("NSObject"),
             @selector(init),
             (IMP) hook_init,
             (IMP *) &orig_init);
}
```

Produces something like:

```
...
SomeApp[850] <Warning>: Class initialized: UIRuntimeOutletConnection
SomeApp[850] <Warning>: Class initialized: UIProxyObject
SomeApp[850] <Warning>: Class initialized: SomeApp_AppDelegate
SomeApp[850] <Warning>: Class initialized: UIRuntimeOutletConnection
...
```



# Injecting Our Library

---

Conventionally, we'd use DYLD\_INSERT\_LIBRARIES (think LD\_PRELOAD)

But MobileSubstrate, is even better (dpkg available from cydia)

*<http://www.iphonedevwiki.net/index.php/MobileSubstrate>*

We can associate our injected dylib to a single target

```
Filter = {  
    Bundles = (com.apple.springboard);  
};
```

Or many

```
Filter = {  
    Bundles = (com.apple.UIKit);  
};
```

But, we avoid MSHook\* in the lib for use with other injection techniques.

# Other Library Injection Techniques

For things running from “launchd”, we can use a trick from Charlie Miller’s SMS fuzzing playbook.

**Find your /System/Library/LaunchDaemons/\*.plist and add**

```
<key>EnvironmentVariables</key>
<dict>
  <key>DYLD_FORCE_FLAT_NAMESPACE</key>
  <string>1</string>
  <key>DYLD_INSERT_LIBRARIES</key>
  <string>/path/to/your.dylib</string>
</dict>
```

Dino Dai Zovi’s “Machiavellian” bundle-inject’ion also works on iOS.

*(porting in progress... stay tuned)*

# Conclusions

---

## **Lots of security research to be done on iPhones and mobiles in general**

- Objective-C runtime is a ripe area for rootkits and backdoors
- Mach kernel features are just as intriguing on iOS as they are on OS X
- In general, if you get good at hacking OS X and you'll also be getting good at IOS

## **Mitigation**

- Conventional wisdom is that Jailbroken devices are more vulnerable. I think it's more nuanced than that.
- My take-away:  
Jailbreak your iPhone/iPad/iPod before someone else does it for you!
- Once jailbroken treat it just like the other computers you own
  - Patches
  - Stripped services
  - Monitoring (periodic md5 filesystem checks are probably even that crazy)

# Conclusions (continued)

We need to see more AV and defense-ware for iOS

- Don't expect Apple to facilitate this very much
- Any reasonable AV solution will fail App Store approval on multiple counts
- Google Kaspersky's take on this. (Yet another frustrated iPhone developer)

We need Apple to adopt a better relationship with security.





## Reversing Redux: The Binary "star" Exploit

# Reversing Steps

|   |             |             |          |
|---|-------------|-------------|----------|
| 3 | 25 50 44 46 | 2D 31 2E 33 | %PDF-1.3 |
| 3 | 0A 25 C4 E5 | F2 E5 EB A7 | %......  |
| 3 | F3 A0 D0 C4 | C6 0A 34 20 | .....4   |
| 3 | 30 20 6F 62 | 6A 0A 3C 3C | 0 obj.<< |
| 3 | 20 2F 4C 65 | 6E 67 74 68 | /Length  |
| 3 | 20 36 33 31 | 20 3E 3E 0A | 631 >>.  |

```
13 0 obj
<<
/Subtype/Type1C
/Filter[/FlateDecode]
/Length 10908
>>
stream
x<9c>1}^MpT<95>am0^E<8d>hU-#L^Y^Kç!d<90>1" [
8>-gGNH<96>Í<90>DÜh<84>^S<9c>ÖVE^N<8e>=a!óii
mybμ;<8d>e!QðAý^GÁ8^_şýí^Ei^V^7ð³·<89>|<83>Á
/¶·_l-ýý±è¶ññWú<8c>[01¹¾Èð^E;<93>%0σ<91>
8b>¹ši?.78ð<97>^Pμ<89>à^Vú#Ú<<8d>¶<92>^]<8d>
!)ÿ<89>^W^Kfá^GD.<92>]Ñó<94>0^Á<83>È()/*^Á<9
<9f>ðwí^Kσ5í<97>8¹.4^1/<93>»5v8<85>1<8b>?é
```

## Analyzed the PDF

- Barebones PDF. Viewer shows one "empty" page
- Compared PDFs between iOS device/version
  - A single zlib deflated font section is the only difference
- Deflate this section - strings indicate a whole MACH-O dylib in there someplace
- Wrote a quick file splitter "extract\_payload"
- Found 3 parts
  - CFF Font egg
  - Macho\_1
  - Macho\_2

# ... continued: egg

## Malformed Times-Roman CFF Font

```
00000000 01 00 04 01 00 01 01 01 13 41 42 43 44 45 46 2b |.....ABCDEF+|
00000010 54 69 6d 65 73 2d 52 6f 6d 61 6e 00 01 01 01 1f |Times-Roman....|
00000020 f8 1b 00 f8 1c 02 f8 1d 03 f8 19 04 1c 6f 00 0d |.....O..|
00000030 fb 3c fb 6e fa 7c fa 16 05 e9 11 8b 8b 12 00 03 |.<.n.|.....|
00000040 01 01 08 13 18 30 30 31 2e 30 30 37 54 69 6d 65 |.....001.007Time|
00000050 73 20 52 6f 6d 61 6e 54 69 6d 65 73 00 00 00 02 |s RomanTimes....|
00000060 04 00 00 00 01 00 00 00 05 00 00 04 dc 0e 0e 0e |.....|
00000070 0e ff 00 00 00 00 ff 00 00 00 00 ff 00 00 00 00 |.....|
00000080 ff 34 04 f9 31 ff 00 00 00 00 ff 00 00 00 00 ff |.4..1.....|
00000090 00 00 00 00 ff 30 17 15 bf ff 09 00 00 00 ff 00 |.....0.....|
000000a0 10 7f 38 ff 00 00 00 03 ff 00 00 10 12 ff 30 0e |..8.....0..|
000000b0 18 ad ff 00 00 00 00 ff 00 00 00 00 ff 00 00 00 |.....|
000000c0 00 ff 30 01 4e d9 ff ff ff f9 98 ff 33 c4 3f f1 | 0 1 3 2
```

...

Compiled code extracted from dylib chunk at end of payload:  
(aka macho\_2 or 'one.dylib')

```
00001070 69 76 61 74 65 2f 76 61 72 2f 6d 6f 62 69 6c 65 |ivate/var/mobile|
00001080 2f 00 00 00 bf 2f 70 72 69 76 61 74 65 2f 76 61 |/....private/va|
00001090 72 2f 6d 6f 62 69 6c 65 2f 4c 69 62 72 61 72 79 |r/mobile/Library|
000010a0 2f 50 72 65 66 65 72 65 6e 63 65 73 2f 00 00 00 |/Preferences/...|
000010b0 bf 00 04 00 00 b9 92 05 80 71 77 08 80 15 d6 3e |.....qw....>|
000010c0 80 f9 d9 3e 80 2f 64 65 76 2f 6d 65 6d 00 00 00 |...>./dev/mem...|
000010d0 00 2f 64 65 76 2f 6b 6d 65 6d 00 00 00 2f 64 65 |./dev/kmem.../de|
000010e0 76 2f 6b 6d 65 6d 00 00 00 00 00 00 90 b5 01 |v/kmem.....|
000010f0 af 2f 74 6d 70 2f 69 6e 73 74 61 6c 6c 75 69 2e |./tmp/installui.|
00001100 64 79 6c 69 62 00 00 00 00 |dylib....|
00001109
```



# IOKit Integer Overflow XML Extract

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
3 <plist version="1.0">
4   <dict>
5     <key>IOSurfaceAllocSize</key>
6     <integer>119888</integer> ←
7     <key>IOSurfaceBufferTileMode</key>
8     <false/>
9     <key>IOSurfaceBytesPerElement</key>
10    <integer>4</integer>
11    <key>IOSurfaceBytesPerRow</key>
12    <integer>885952832</integer> ←
13    <key>IOSurfaceHeight</key>
14    <integer>2147500567</integer> ←
15    <key>IOSurfaceIsGlobal</key>
16    <true/>
17    <key>IOSurfaceMemoryRegion</key>
18    <string>PurpleGfxMem</string>
19    <key>IOSurfacePixelFormat</key>
20    <integer>1095911234</integer> ←
21    <key>IOSurfaceWidth</key>
22    <integer>3442713680</integer> ←
23  </dict>
24 </plist>
```



# class-dump on installui.dylib (aka macho\_1)

```
@interface Dude
{
    UIAlertView *progressalertView;
    UIAlertView *choicealertView;
    UIAlertView *donealertView;
    UIProgressView *progressBar;
    NSMutableData *wad;
    long long expectedLength;
    char *freeze;
    int freeze_len;
    char *one;
    unsigned int one_len;
    NSURLConnection *connection;
}

- (id)initWithOne:(char *)arg1 oneLen:(int)arg2;
- (void)setProgress:(id)arg1;
- (void)setProgressCookie:(unsigned int)arg1;
- (void)doStuff;
- (void)bored;
- (void)bored2;
- (void)connection:(id)arg1 didReceiveResponse:(id)arg2;
- (void)connection:(id)arg1 didReceiveData:(id)arg2;
- (void)connectionDidFinishLoading:(id)arg1;
- (void)connection:(id)arg1 didFailWithError:(id)arg2;
- (void)keepGoing;
- (void>alertView:(id)arg1 clickedButtonAtIndex:(int)arg2;
- (void)start;

@end

@interface (null) (DDData)
- (id)inflatedData;
@end
```

# Wad.bin

## What gets downloaded and installed for the jailbroken device?

- Wad.bin pseudo-code structure

```
{
  uint32  magic           // "magic" value of BBBB / 0x42424242
  uint32le totlen        // total len (including magic)
  uint32le liblen        // size of install_dylib_chunk
  char install_dylib_chunk[] // liblen sized zlib deflated install.dylib
  char filesystem_txz_chunk[] // rest is XZ(lzma) compressed FS tarball
}

00000000 42 42 42 42 99 a6 3b 00 15 b5 01 00 78 9c ec 7d BBBB...;.....x..}
00000010 0d 9c 54 c5 95 ef bd dd 3d 43 33 34 70 81 46 87 ..T.....=C34p.F.
...
0001b520 6c fd 37 7a 58 5a 00 00 04 e6 d6 b4 46 02 00 21 1.7zXZ.....F..!
0001b530 01 16 00 00 00 74 2f e5 a3 e1 8d 95 ef fe 5d 00 .....t/.....].
...
003ba690 30 03 00 00 00 00 04 59 5a                                0.....YZ
003ba699
```

- XZ'ed tarball contents
  - Stripped down Unix dir structure and CLI programs (bash et al)
  - Cydia.app for downloading more packages