Apple iOS 4 Security Evaluation

Dino A. Dai Zovi
Trail of Bits LLC

Disclaimers

- I have never worked for Apple, but I have a crippling addiction to buying and tinkering with their products.
- I have never received any monetary compensation from Apple, but they have sent me some free schwag
- I have been mistaken by strangers on the street for being an off-duty Apple Store employee
- I hacked a Mac once, but don’t worry, it wasn’t yours.
- Charlie Miller and I wrote an entire book on hacking the Mac and I still have never met Steve Jobs. I blame Charlie.

Acknowledgements

- iPhone jailbreak developer community
  - Chronic Dev Team, Comex for releasing tools with source
  - The iPhone Wiki for excellent up-to-date documentation
- Other security researchers with great iOS research
  - Jean-Baptiste Bedrune, Jean Sigwald (SOGETI ESEC)
  - Dion Blazakis
  - Stefan Esser

Overview

- What businesses need to know about iOS security features and properties to make informed deployment, configuration, usage, and procedure decisions
- How iOS security compares to competing mobile platforms (even with freely available jailbreak tools)
- Assorted iOS implementation details and internals
- Interesting places for reverse engineers and vulnerability researchers to look (if they pay close attention)

Background

iOS Security Concerns

- Sensitive data compromise from lost/stolen device
  - What data can be recovered by attacker?
- Malicious Apps
  - i.e. DroidDream for iOS?
- Remote attacks through web browser, e-mail, etc.
  - Is that a desktop in your pocket?

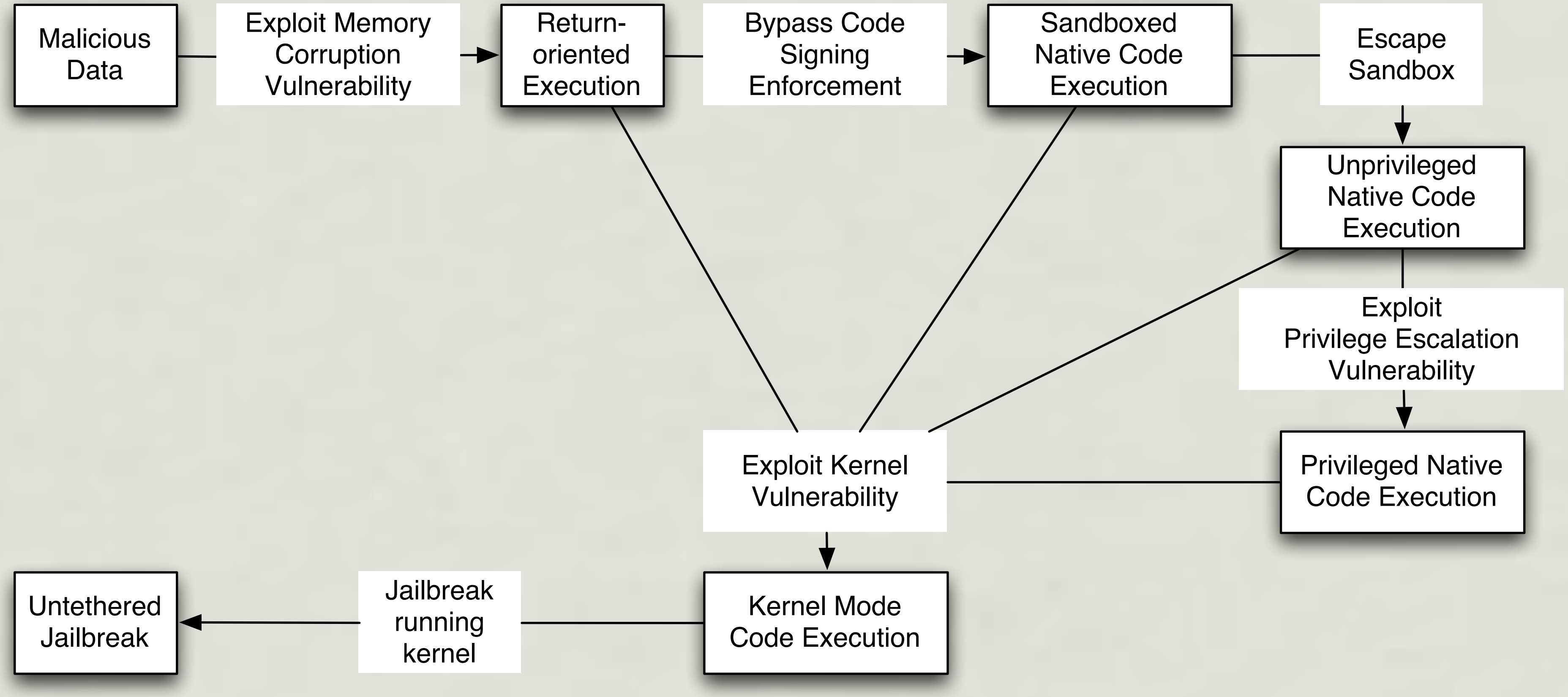
Takeaways

- Modern iOS jailbreaks are very sophisticated and contain many exploits
  - Needed to break various levels of protection in iOS
- Why Android gets a new malware variant every day while entire teams spend months working on one iOS jailbreak that gets patched in 2 weeks

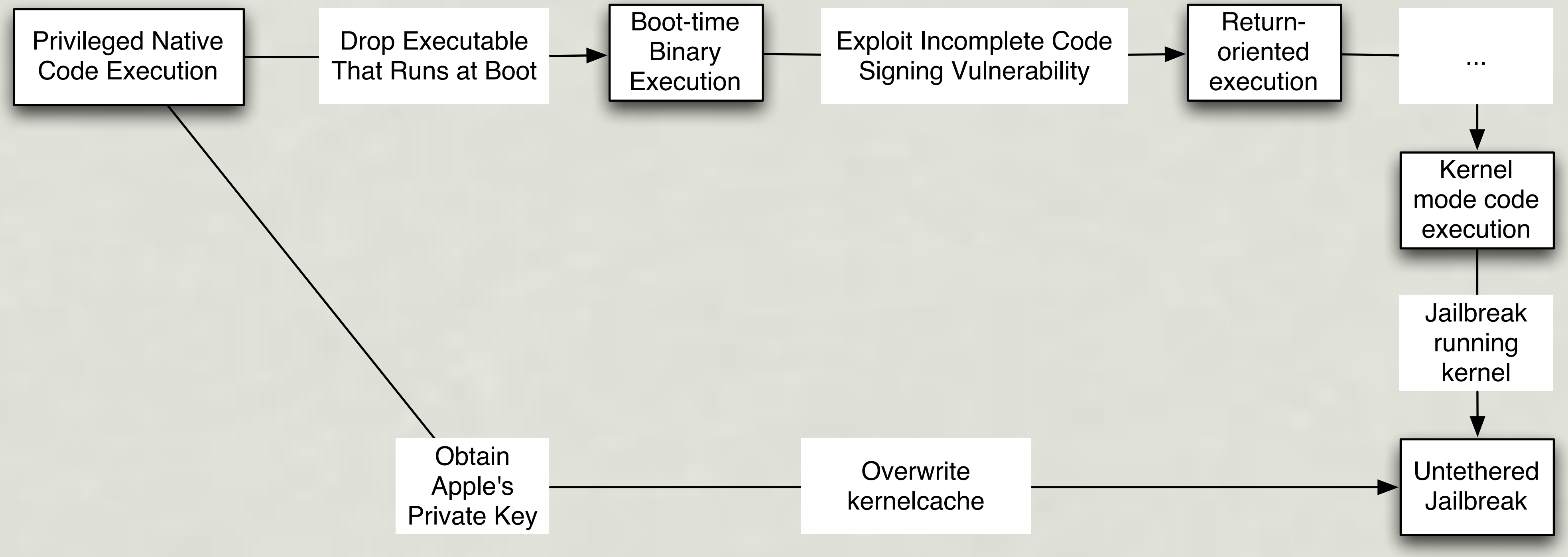
Unique to iOS

- Dynamic Code Signing Enforcement
  - Stronger defense against remote native code injection than DEP, NX, etc.
- Kernel is secured against user mode
  - Root user has to exploit kernel to run kernel mode code

Remote Attack Graph


Malicious Data
Exploit Memory Corruption Vulnerability
Return-oriented Execution
Bypass Code Signing Enforcement
Sandboxed Native Code Execution
Escape Sandbox
Unprivileged Native Code Execution
Exploit Privilege Escalation Vulnerability
Privileged Native Code Execution
Exploit Kernel Vulnerability
Kernel Mode Code Execution
Jailbreak running kernel
Untethered Jailbreak

Persistence Attack Graph


Privileged Native Code Execution
Drop Executable That Runs at Boot
Boot-time Binary Execution
Exploit Incomplete Code Signing Vulnerability
Return-oriented execution
...
Kernel mode code execution
Jailbreak running kernel
Obtain Apple's Private Key
Overwrite kernelcache
Untethered Jailbreak

Address Space Layout Randomization

iOS 4.3 ASLR

- iOS 4.3 introduced ASLR support
    - ASLR is common on desktop and server operating systems and is a good genetic protection against remote exploits
    - iOS 4.3 requires iPhone 3GS and later (ARMv7)
    - Apps must be compiled with PIE support for full ASLR

ASLR without PIE

Executable	Heap	Stack	Libraries	Linker
0x2e88	0x15ea70	0x2fdff2c0	0x36adadd1	0x2fe00000
0x2e88	0x11cc60	0x2fdff2c0	0x36adadd1	0x2fe00000
0x2e88	0x14e190	0x2fdff2c0	0x36adadd1	0x2fe00000
0x2e88	0x145860	0x2fdff2c0	0x36adadd1	0x2fe00000
0x2e88	0x134440	0x2fdff2c0	0x36adadd1	0x2fe00000
Reboot				
0x2e88	0x174980	0x2fdff2c0	0x35e3edd1	0x2fe00000
0x2e88	0x13ca60	0x2fdff2c0	0x35e3edd1	0x2fe00000
0x2e88	0x163540	0x2fdff2c0	0x35e3edd1	0x2fe00000
0x2e88	0x136970	0x2fdff2c0	0x35e3edd1	0x2fe00000
0x2e88	0x177e30	0x2fdff2c0	0x35e3edd1	0x2fe00000

ASLR with PIE

Executable	Heap	Stack	Libraries	Linker
0xd2e48	0x1cd76660	0x2fecf2a8	0x35e3edd1	0x2fed0000
0xaae48	0x1ed68950	0x2fea72a8	0x35e3edd1	0x2fea8000
0xbbe48	0x1cd09370	0x2feb82a8	0x35e3edd1	0x2feb9000
0x46e48	0x1fd36b80	0x2fe432a8	0x35e3edd1	0x2fe44000
0xc1e48	0x1dd81970	0x2febe2a8	0x35e3edd1	0x2febf000
Reboot				
0x14e48	0x1dd26640	0x2fe112a8	0x36146dd1	0x2fe12000
0x62e48	0x1dd49240	0x2fe112a8	0x36146dd1	0x2fe60000
0x9ee48	0x1d577490	0x2fe9b2a8	0x36146dd1	0x2fe9c000
0xa0e48	0x1e506130	0x2fe9d2a8	0x36146dd1	0x2fe9e000
0xcde48	0x1fd1d130	0x2feca2a8	0x36146dd1	0x2fecb000

Partial vs. Full ASLR

PIE	Main Executable	Heap	Stack	Shared Libraries	Linker
No	Fixed	Randomized per execution	Fixed	Randomized per device boot	Fixed
Yes	Randomized per execution	Randomized per execution	Randomized per execution	Randomized per device boot	Randomized per execution

Top 10 Free Apps

App	Version	Post Date	PIE
Songify	1.0.1	June 29, 2011	No
Happy Theme Park	1.0	June 29, 2011	No
Cave Bowling	1.10	June 21, 2011	No
Movie-Quiz Lite	1.3.2	May 31, 2011	No
Spotify	0.4.14	July 6, 2011	No
Make-Up Girls	1.0	July 5, 2011	No
Racing Penguin, Flying Free	1.2	July 6, 2011	No
ICEE Maker	1.01	June 28, 2011	No
Cracked Screen	1.0	June 24, 2011	No
Facebook	3.4.3	June 29, 2011	No

Identifying PIE support

- otool ...
- hexdump ...

Bottom Line

- All built-in apps in iOS 4.3 have full ASLR with PIE support
- Third-party apps are rarely compiled with PIE support and run with partial ASLR
  - Static location of dyld facilitates return-oriented programming in exploits
  - Applications using UIWebView are likely highest risk (embedded browser in Twitter, Facebook, etc)
  - Direct remote attacks against 3rd-party apps are relatively low risk

Code Signing

Code Signing

- Why?
  - Validates the authenticity of native code run on the device
  - Verifies the integrity of the application code at rest and at runtime
  - Prevents AppStore application piracy

Code Signing

- Mandatory Code Signing
  - Every executable binary, library, or application must have a valid and trusted signature
  - Enforced when application or binary is executed
- Code Signing Enforcement
  - Processes may only execute code that has been signed with a valid and trusted signature
  - Enforced at runtime

Mandatory Code Signing

- Code Signing security model
  - Certificates
  - Provisioning Profiles
  - Signed Applications
  - Entitlements

Certificates

- Identify the author or publisher of a piece of software
- Must be issued by and signed by Apple
- Developers are assigned unique application identifier prefixes
- Developer Certificates
- Distribution Certificates

Provisioning Profiles

- The Provisioning Profile itself must be signed by Apple
- Configures an iOS device to trust software signed by the embedded certificate
    - Defines which entitlements the developer is permitted to give to applications they sign
- Profile may be tied to one specific device or global

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>ApplicationIdentifierPrefix</key>
        <array>
                <string>9ZJJSS7EFV</string>
        </array>
        <key>CreationDate</key>
        <date>2010-08-20T02:55:55Z</date>
        <key>DeveloperCertificates</key>
        <array>
                <data>...</data>
        </array>
        <key>Entitlements</key>
        <dict>
                <key>application-identifier</key>
                <string>9ZJJSS7EFV.*</string>
                <key>get-task-allow</key>
                <true/>
                <key>keychain-access-groups</key>
                <array>
                        <string>9ZJJSS7EFV.*</string>
                </array>
        </dict>
        <key>ExpirationDate</key>
        <date>2010-11-18T02:55:55Z</date>
        <key>Name</key>
        <string>Development</string>
        <key>ProvisionedDevices</key>
        <array>
                <string>e757cfc725783fa29e8b368d2e193577ec67bc91</string>
        </array>
        <key>TimeToLive</key>
        <integer>90</integer>
        <key>UUID</key>
        <string>BDE2CA16-499D-4827-BB70-73886F52D30D</string>
        <key>Version</key>
        <integer>1</integer>
</dict>
</plist>
```

Application Distribution

- ***Device Testing*** allows developers to build and test applications on their own devices
- ***Ad-Hoc Distribution*** allows developers to beta test applications on up to 100 other users' devices
- ***AppStore Distribution*** allows developers to publish applications on the iTunes AppStore
- ***In-House Distribution*** allows Enterprise Developers to distribute their custom applications to any device

OTA App Distribution

- Ad-Hoc and In-House Provisioning Profiles can be distributed with the Application in a single archive
- Developer must host the manifest on a web server
- Link to manifest can be sent via e-mail, SMS, or other web page
- When user clicks the link, iOS displays developer and application name in a cancel/allow dialog

Normal Code Signature

```
Executable=/.../9D3A8D85-7EDE-417A-9221-1482D60A40B7/iBooks.app/iBooks
Identifier=com.apple.iBooks
Format=bundle with Mach-O universal (armv6 armv7)
CodeDirectory v=20100 size=14585 flags=0x0(none) hashes=721+5 location=embedded
Hash type=sha1 size=20
CDHash=ac93a95bd6594f04c209fb6bf317d148b99ac4d7
Signature size=3582
Authority=Apple iPhone OS Application Signing
Authority=Apple iPhone Certification Authority
Authority=Apple Root CA
Signed Time=Jun 7, 2011 11:30:58 AM
Info.plist entries=36
Sealed Resources rules=13 files=753
Internal requirements count=2 size=344
```

Ad-Hoc Code Signature

```
Executable=/Developer/usr/bin/debugserver
Identifier=com.apple.debugserver
Format=Mach-O universal (armv6 armv7)
CodeDirectory v=20100 size=1070 flags=0x2(adhoc) hashes=45+5 location=embedded
CDHash=6a2a1549829f4bff9797a69a1e483951721ebcbd
Signature=adhoc
Info.plist=not bound
Sealed Resources=none
Internal requirements count=1 size=152
```

Ad-Hoc Code Signature

- iOS kernel has a static set of CDHashes (*static trust cache*)
- AMFI IOKit UserClient lets root load trust caches
  - Must be signed
- Kernel caches the CDHash of each verified binary in an *MRU trust cache*

Code Signing Verification

- Is CDHash in static trust cache?
- Is CDHash in dynamically loaded trust caches?
- Is CDHash in MRU trust cache?
  - Move entry to front of the linked list
- RPC call to amfid verify_code_directory()
  - Add CDHash to MRU trust cache

AMFI Daemon

- /usr/libexec/amfid

Message ID	Subroutine	Description
1000	verify_code_directory	Verifies the given code directory hash and signature for the executable at the given path. This checks whether the signature is valid and that it should be trusted based on the built-in Apple certificates and installed provisioning profiles (if any).
1001	permit_unrestricted_debugging	Enumerates the installed provisioning profiles and checks for a special Apple-internal provisioning profile with the UDID of the current device that enables unrestricted debugging on it.

Bypassing Code Signing

- Incomplete Code Signing[1] Exploits
  - Manipulate dynamic linker to perform stack pivot and execute return-oriented payload
  - Interposition exploit (4.0), Initializers exploit (4.1)
  - More recent exploits use relocations to dynamically adjust ROP payloads to compensate for ASLR
  - iOS 4.3.4 strengthens iOS defenses against these

[1] http://theiphonewiki.com/wiki/index.php?title=Incomplete_Codesign_Exploit

Code Signing Enforcement

- Ensure that process stays dynamically valid
  - No introduction of executable code
  - Already loaded executable code can’t be changed

csops

```
int csops(pid_t pid, uint32_t ops, user_addr_t useraddr, user_size_t usersize);
```

- System call to interact with a process’ code signing status
  - Get code signing status
  - Set code signing flags (CS_HARD, CS_KILL)
  - Get executable pathname, code directory hash, active running slice

CS Ops

Flag	Value	Description
CS_OPS_STATUS	0	Return process CS status
CS_OPS_MARKINVALID	1	Invalidate process
CS_OPS_MARKHARD	3	Set CS_HARD flag
CS_OPS_MARKKILL	4	Set CS_KILL flag
CS_OPS_PIDPATH	5	Get executable's pathname
CS_OPS_CDHASH	6	Get code directory hash
CS_OPS_PIDOFFSET	7	Get offset of active Mach-O slice

CS Status Flags

Flag	Value	Description
CS_VALID	0x00001	Process is dynamically valid
CS_HARD	0x00100	Process shouldn’t load invalid pages
CS_KILL	0x00200	Process should be killed if it becomes dynamically invalid
CS_EXEC_SET_HARD	0x01000	Process should set CS_HARD on any exec’d child
CS_EXEC_SET_KILL	0x02000	Process should set CS_KILL on any exec’d child
CS_KILLED	0x10000	The process was killed by the kernel for being dynamically invalid

CS_HARD and CS_KILL

- CS_HARD
  - Enforce W^X (Writable XOR Executable) memory page policy
  - Do not allow invalid memory pages to be loaded
  - `mprotect(addr, len, ... | PROT_EXEC) => EPERM`
- CS_KILL
  - Kill the process if the process becomes invalid

  -
    ```
    mprotect(text, len, PROT_READ | PROT_WRITE)
    ... Modify code page ...
    mprotect(text, len, PROT_READ | PROT_EXEC) => SIGKILL
    ```

AppleMobileFileIntegrity

- Kernel extension responsible for implementing code signing security policy
- Installs MAC Framework policy hooks to enforce Mandatory Code Signing and Code Signing Enforcement

MAC Hook	API Description	AMFI Usage
mpo_vnode_check_signature	Determine whether the given code signature or code directory SHA1 hash are valid.	Checks for the given CDHash in the trust caches. If it is not found, the full signature is validated by performing an RPC call to the userspace amfid daemon. If a particular global flag is set (amfi_get_out_of_my_way), then any signature is allowed.
mpo_vnode_check_exec	Determine whether the subject identified by the credential can execute the passed vnode.	Sets the code signing CS_HARD and CS_KILL flags, indicating that the process shouldn't load invalid pages and that the process should be killed if it becomes invalid.
mpo_proc_check_get_task	Determine whether the subject identified by the credential can get the passed process's task control port.	Allows task port access if the process has the get-task-allow and task_for_pid-allow entitlements.

MAC Hook	API Description	AMFI Usage
mpo_proc_check_run_cs_invalid	Determine whether the process may execute even though the system determined that it is untrusted (unidentified or modified code)	Allow execution if the process has the get-task-allow, run-invalid-allow, and run-unsigned-code entitlements and an RPC call to amfid returns indicating that unrestricted debugging should be allowed.
mpo_proc_check_map_anon	Determine whether the subject identified by the credential should be allowed to obtain anonymous memory with the specified flags and protections.	Allows the process to allocate anonymous memory if and only if the process has the dynamic-codesigning entitlement.

Dynamic Code Signing

- The **dynamic-codesigning** entitlement allows the process to map anonymous memory *with any specified protections*.
- Only MobileSafari has this entitlement in iOS 4.3
  - Necessary for JavaScript native JIT (“nitro”)
  - Previously MobileSafari did bytecode JIT

Bottom Line

- Mandatory Code Signing in iOS is strong defense against execution of unauthorized binaries
  - Requires incomplete code signing exploits to bypass and obtain return-oriented execution
- Code signing forces attackers to develop fully ROP payloads
  - DEP, NX, etc. only require a ROP stage
- JIT support in Safari reduces ROP requirements to stage

Sandboxing

Sandboxing in iOS

- Based on same core technologies as Mac OS X sandbox
    - See Dion's "The Apple Sandbox" from BHDC 2011 for more information on internals
    - Modified his tools to decompile iOS 4.3 profiles
- iOS only supports static built-in profiles
- Process' sandbox profile is determined by seatbelt-profiles entitlement

Sandbox Kernel Extension

- Installs MAC Hooks on all secured operations
- MAC hooks evaluate a binary decision tree to make access determination
- Sandbox profiles consist of the set of decision trees defined for each defined operation with conditional filters based on requested resource
  - i.e. does file name match this regex?

Built-in Sandbox Profiles

- Background daemons: accessoryd, apsd, dataaccessd, iapd, mDNSResponder, etc.
- Built-in Apps: MobileMail, MobileSafari, MobileSMS, Stocks, YouTube, etc.
- Third-party Apps: container and container2 (iBooks)

Third-Party Applications

- Assigned a dedicated portion of the file system ("container" or "application home directory") each time it is installed

- Can a rogue application escape the sandbox and read other applications' data or modify the device firmware?

App Home Directory

Subdirectory	Description
<AppName>.app/	The signed bundle containing the application code and static data
Documents/	App-specific user-created data files that may be shared with the user's desktop through iTunes's "File Sharing" features
Library/	Application support files
Library/Preferences/	Application-specific preference files
Library/Caches/	App-specific data that should persist across successive launches of the application but not needed to be backed up
tmp/	Temporary files that do not need to persist across successive launches of the application

Container Sandbox Profile

- See whitepaper for detailed description and tarball for fully decompiled profile
- Summary:
  - File access is generally restricted to app's home directory
  - Can read media: songs, photos, videos
  - Can read and write AddressBook
  - Some IOKit User Clients are allowed
  - All Mach bootstrap servers are allowed

Mach Bootstrap Servers

- All Mach tasks have access to a bootstrap port to lookup service ports for Mach RPC services
  - On iOS, this is handled by launchd
- 141 RPC servers accessible from apps
  - May present risk of allowing apps to perform unauthorized or undesirable actions
  - Lesser risk of being exploited over RPC

Bottom Line

- Rogue applications would need to exploit and jailbreak the kernel to escape sandbox
  - Could repurpose kernel exploits from Jailbreaks
  - Apple’s review will likely catch this
  - OTA app distribution bypasses Apple’s review (target user interaction required)
- Apply upgraded firmwares quickly, preferably from DFU mode
  - Side benefit of unjailbreaking the device

Data Encryption

Overview

- What you need to know about Data Encryption in iOS to make informed deployment and configuration decisions
- For more internals and implementation details, refer to excellent “iPhone Data Protection in Depth”[1] from HITB Amsterdam 2011

[1] http://esec-lab.sogeti.com/dotclear/public/publications/11-hitbamsterdam-iphonedataprotection.pdf

Encryption Layers

- Entire filesystem is encrypted using block-based encryption with **File System Key**
    - FSK is stored on the flash
- Each file has a unique encryption key stored in an extended attribute, protected by **Class Key**
    - Class Keys are stored in the System KeyBag
- Some Class Keys are protected by a key derived from the user's passcode (**Passcode Key**)
- Certain Class Keys are also protected by the device-specific **UID Key** that is embedded in hardware and inaccessible to code running on CPU

Data Protection API

- Applications must specifically mark files on the filesystem and Keychain items with a Protection Class in order for them receive greater protection
  - Files and Keychain items can be made inaccessible when the device is locked (protected by Passcode Key)
  - Keychain items can be made non-migratable to other devices (protected by UID Key)

Data Protection Coverage

- In iOS 4, DP is only used by the built-in Mail app
  - Protects all mail messages, attachments, and indexes
  - Protects passwords for IMAP, SMTP servers
  - Protected items are only accessible when device is unlocked
  - Exchange ActiveSync passwords are accessible always to preserve remote wipe functionality
- DP is also used for automatic UI screenshots generated when user hits the Home Button.

Attacking Passcode

- With knowledge of passcode, you can decrypt the data protected by iOS Data Protection
- Increasing incorrect passcode delay and forced device wipe after too many incorrect guesses are enforced by UI
  - Springboard -> MobileKeyBag Framework -> MobileKeyBag UserClient -> MKB Kernel Extension
- On a jailbroken device, you can guess passcodes directly using the MKB Framework or MKB IOKit User Client
  - Jailbreak device using BootROM exploit, install SSH bundle, restart, and log in via SSH over USBMUX

Passcode Key

- Passcode Key is derived using PBKDF2 using AES with the Device Key as the hashing function
  - Cannot derive key off of the device that created it unless you can extract the UID Key from the hardware
  - Iteration count of PBKDF2 is tuned to hardware speed
  - Roughly 9.18 guesses/second on iPhone4

Worst-Case Passcode Guessing Time (iPhone4)

Passcode Length	Complexity	Time
4	Numeric	18 minutes
4	Alphanumeric	51 hours
6	Alphanumeric	8 years
8	Alphanumeric	13 thousand years
8	Alphanumeric, Complex	2 million years

Assuming 26 lowercase letters + 10 digits + 34 complex characters = 70 chars

Bottom Line

- 6-character alphanumeric passcodes are sufficient
  - Unless attacker can extract UID Key from hardware
- Lack of thorough Data Protection coverage is a serious issue
  - Wait to see what iOS 5 covers
  - Audit third-party apps for Data Protection usage
- iPad2 and later have no public Boot ROM exploits, making attacks on lost devices much more difficult and unlikely

Evaluation

Attacks You Should Actually Care About

- Lost/stolen device
- Repurposed remote jailbreak tools
  - JailbreakMe PDF attacks via e-mail or web
- Stolen Enterprise In-House Distribution Certificate and social engineering OTA app links

Hardware Advice

- iPhone 3G and earlier shouldn't be allowed
  - No longer supported by iOS
  - No device encryption support
  - Permanently jailbroken via Boot ROM exploits
- iPad 2 has no public Boot ROM exploits, making it safer than earlier iOS devices

Bottom Line

- Should I deploy or allow employee-owned iOS devices for business use?
- Is iOS safer than BlackBerry and/or Android?
  - ***Answers to be revealed at BH presentation***