Trail of Bits

# Apple iOS Security Evaluation

Dino A. Dai Zovi
Principal
Trail of Bits LLC

Version: DRAFT

# Table of Contents

# ASLR

## Overview

Address Space Layout Randomization (ASLR) is an important protection that makes the remote exploitation of memory corruption vulnerabilities significantly more difficult. In particular, when it is fully applied, it usually requires that attackers find and exploit one or more memory disclosure vulnerabilities in order to enable the exploitation of a memory corruption vulnerability. On many operating systems, however, the implementation of ASLR may be incomplete and attackers can make often make use of executable or writable memory regions at fixed or predictable locations.

ASLR was introduced in iOS 4.3 and there are two levels of completeness of ASLR in iOS 4.3, depending on whether the application was compiled with support for Position Independent Executables (PIE). If the application was compiled without PIE support, it will run with limited ASLR. Specifically, the main executable binary (including its code and data sections) and the dynamic linker (dyld) will be loaded at fixed locations. The main thread's stack will also always begin at the same location in memory. This is presumably to maintain compatibility with existing iOS applications. If the application is compiled with PIE support, then the application will be able to make full use of ASLR and all memory regions will be randomized. In iOS 4.3, all built-in applications are compatible with full ASLR. The table below summarizes which segments of memory will be found at randomized locations depending on whether the application was compiled with or without PIE support.

**Memory Region Randomization by Deployment Target Version**

| PIE | Executable | Data | Heap | Stack | Libraries | Linker |
|---|---|---|---|---|---|---|
| No | Fixed | Fixed | Randomized per execution | Fixed | Randomized per device boot | Fixed |
| Yes | Randomized per execution | Randomized per execution | Randomized per execution | Randomized per execution | Randomized per device boot | Randomized per execution |

## Observing ASLR

In order to detect which memory regions are loaded at randomized locations and how often that location changes, a small command-line executable[1] was written to observe and output a memory address within those regions. The program captures the memory addresses of the executable's main function, a variable in its initialized data segment, a variable in its uninitialized data segment, a single heap allocation, a variable on the stack, the address of a function in a loaded shared library, and the load address of the dynamic linker (dyld). This program was used to observe memory addresses for five invocations before rebooting the device and gathering data from five more invocations. This was performed for the executable compiled with and without support for Position Independent Executables (PIE). The results of these tests are shown in the table below.

**Observed Memory Addresses**

| Main 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 |
| | | ***Compile with -fPIE*** | | |
| 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 | 0x2fe5f2a8 | 0x36146dd1 | 0x2fe60000 |
| 0x9ee48 | 0x1d577490 | 0x2fe9b2a8 | 0x36146dd1 | 0x2fe9c000 |
| 0xa0e48 | 0x1e506130 | 0x2fe9d2a8 | 0x36146dd1 | 0x2fe9e000 |
| 0xcde48 | 0x1fd1d130 | 0x2feca2a8 | 0x36146dd1 | 0x2fecb000 |

The memory addresses shown in the table above clearly show which memory regions are randomized and how often for both non-PIE and PIE executables. It also shows that the amount of entropy in heap memory randomization is greater for PIE executables than for non-PIE executables. In this way, the PIE compatibility of an executable can be used to determine its level of ASLR compatibility in iOS 4.3: limited or full.

[1] Included in the supplementary materials as src/aslr/aslr.c

### Third-Party Applications

In order to determine the prevalence of PIE support in third-party applications, the top ten free applications as of the time of this writing were downloaded and examined. As summarized in the table below, none of these applications were compiled with PIE support.

**PIE Support in Top Ten Free Applications**

| Application | 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 |

### Xcode Automatic Deployment Target

In Xcode, the “iOS Deployment Target” application setting is the lowest version of iOS that the application will run on. The default setting is “Compiler Default”, indicating that Xcode will automatically choose the minimum version of iOS based on the APIs that are used by the application. This may cause more applications to automatically support PIE if they use an API that is only available in iOS 4.3 or later. More likely, however, is that applications that make use of new APIs introduced in the upcoming release of iOS 5 will automatically be compiled with PIE support.

## Assessment

In order to preserve application compatibility, applications compiled without PIE support do not run with full ASLR on iOS 4.3. This leaves both the application and dynamic linker text segments at fixed known locations. The presence of known code sequences at known locations facilities the use of code reuse techniques such as return-oriented programming in the exploitation of memory corruption vulnerabilities. The most likely targets for remote exploits, however, are the built-in MobileSafari and MobileMail applications, which are both compiled with PIE support and make use of full ASLR in iOS 4.3.

Scriptable applications such as MobileSafari, provide more opportunities for an attacker to identify and exploit memory disclosure vulnerabilities that may be used to weaken or defeat the randomization provided by ASLR. Other applications that do not provide malicious content as many opportunities to inspect their runtime environment and adapt to it are significantly more difficult to remotely exploit. The presence of full ASLR, especially in combination with the other runtime protections described in this document such as Code Signing Enforcement and Sandboxing, makes the remote exploitation of memory corruption vulnerabilities in iOS applications significantly more difficult than on other mobile or desktop platforms.

# Code Signing

## Mandatory Code Signing

In order to verify the authenticity of all executable code running on the device, iOS requires that all native code (command-line executables and graphical applications) is signed by a known and trusted certificate. This protection is referred to as Mandatory Code Signing. The Mandatory Code Signing system also forms the basis of the code signing security model in iOS whereby particular developers or software may be granted specific additional privileges.

There are several important components to the code signing security model:

- Developer Certificates
- Provisioning Profiles
- Signed Applications
- Entitlements

We will describe what these components are and how they interact in turn.

### Developer Certificates

Any developer may freely download Apple's Xcode developer tools and test their applications within the included iOS Simulator. The iOS Simulator runs on the developer's Mac and allows them to interact with their application by simulating touch events and other device hardware features using the mouse and keyboard. While this allows basic development and testing, it may not be sufficient for many applications that need to ensure they run fast enough on actual hardware (i.e. games) or interact properly with location-based services, the digital compass, or accelerometer. In order to run custom applications on an actual iOS device, even just for testing, the developer must be granted a Developer Certificate from Apple by being an approved member of Apple's iOS Developer Program[2].

Developers may apply for this program as an individual, company, enterprise, or university. Depending on how the developer applies, they may enroll in either the Standard, Enterprise, or University Programs and they each have separate verification requirements and developer privileges, including how and whether developers may distribute their applications. An individual or company enrollment for the Standard Program is typically used to publish free or paid applications on Apple's App Store. The Enterprise Program is for larger companies and organizations to develop and distribute custom in-house applications. Finally, the University Program is designed to facilitate educational institutions teaching iOS development and it only permits on-device testing (no application distribution is permitted).

[2] http://developer.apple.com/programs/ios/

| Developer Program | Device Testing | Ad Hoc | App Store | In-House |
|---|---|---|---|---|
| Apple Developer | No | No | No | No |
| University Program | Yes | No | No | No |
| Standard Program | Yes | Yes | Yes | No |
| Enterprise Program | Yes | Yes | No | Yes |

All of the developer programs require verification of the individual developer's or developer organization's real-world identity. For an individual, the use of a credit card to pay for the yearly membership fee is sufficient verification. In order to apply as a company, the developer must submit the Articles of Incorporation or Business License. An Enterprise must submit their DUNS Number[3] and pay a higher yearly membership fee. The University program is only open to qualified, degree granting, higher education institutions. For the purposes of this paper, we will use the term "developer" to refer to an individual or organizational member of Apple's iOS Developer Program and only specify which type of developer when it is significant.

## Provisioning Profiles

A Provisioning Profile is an XML plist[4] file signed by Apple that configures the iOS device to permit the execution of code signed by the embedded developer certificate. It also lists the entitlements that the developer is permitted to grant to applications signed by their certificate. The provisioning profile may also include a list of the Unique Device Identifiers (UDIDs) of the devices that the profile may provision. For on-device testing, the provisioning profile will only list the developer's testing device that they have configured via the iOS Developer Portal. For Ad-Hoc Distribution, an Ad-Hoc Distribution Provisioning Profile may list up to 100 devices for wider application testing. Enterprise Provisioning Profiles are significantly more powerful since they are not restricted to a list of devices in the provisioning profile.

An example Development Provisioning Profile for on-device developer testing is shown in Figure XXX. All provisioning profiles are stored on the filesystem in the directory /var/MobileDevice/ProvisioningProfiles/. The provisioning profile validation is performed by the MISProvisioningProfileCheckValidity function in /usr/lib/libmis.dylib. If the provisioning profile is properly validated, then it will be displayed in the System Preferences application and usable for verifying signed applications. In order to be considered valid, all of the following conditions must hold:

- The signing certificate must be issued by the built-in "Apple iPhone Certification Authority" certificate
- The signing certificate must be named "Apple iPhone OS Provisioning Profile Signing"
- The certificate signing chain must be no longer than three links long
- The root certificate (referred to as the "Apple CA") must have a particular SHA1 hash
- The provisioning profile version number must be 1
- The provisioning profile must contain the UDID of this device or the profile must contain the key "ProvisionsAllDevices"
- The current time is before the expiration date of the profile

[3] DUNS numbers are issued by Dun and Bradstreet, a business credit reporting firm

[4] A plist file is a standard Apple XML file format for storing property lists (nested key-value pairs)

```
<?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>
```

*Figure XXX: Sample Provisioning Profile*

## Signed Applications

All iOS executable binaries and applications must be signed by a trusted certificate.  While Apple's certificates are inherently trusted, any other certificates must be installed via a properly signed Provisioning Profile, as described above. Command-line executables may contain either an ad-hoc signature, as shown for an iOS binary in Figure XXX or a traditional signature as shown for a Mac OS X built-in executable in Figure XXX.

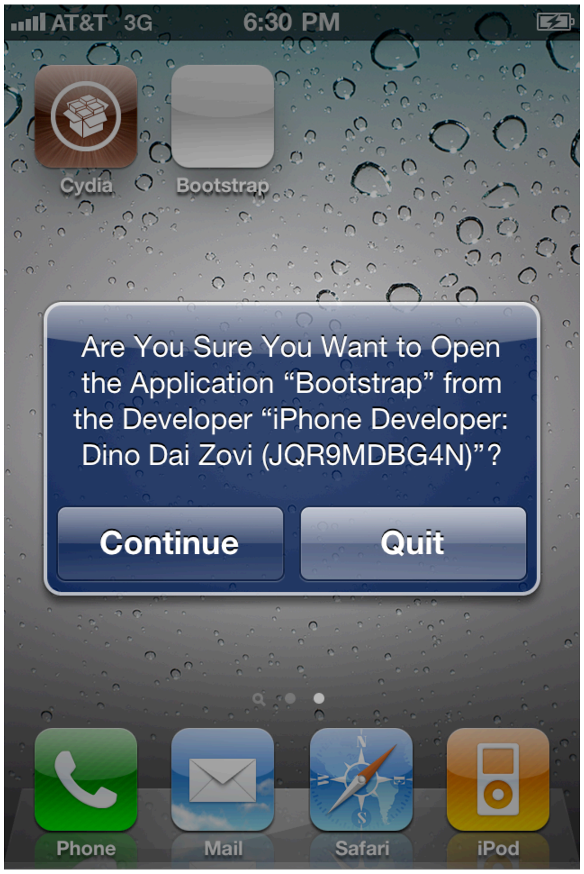
```
% codesign -dvvv debugserver
Executable=/Developer/usr/bin/debugserver
Identifier=com.apple.debugserver
Format=Mach-O universal (armv6 armv7)
CodeDirectory v=20100 size=1070 flags=0x2(adhoc) hashes=45+5 location=embedded
CDHash=6a2a1549829f4bff9797a69a1e483951721ebcbd
Signature=adhoc
Info.plist=not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

*Figure XXX: Ad-hoc code signed executable*

```
% codesign -dvvv /bin/ps
Executable=/bin/ps
Identifier=com.apple.ps
Format=Mach-O universal (i386 ppc7400 x86_64)
CodeDirectory v=20100 size=281 flags=0x0(none) hashes=9+2 location=embedded
CDHash=45cb78d0aee8457ae1e3593a0aff1cc3de109e63
Signature size=4064
Authority=Software Signing
Authority=Apple Code Signing Certification Authority
Authority=Apple Root CA
Info.plist=not bound
Sealed Resources=none
Internal requirements count=1 size=68
```

*Figure XXX: Ad-hoc code signed executable*

An ad-hoc signature does not contain an actual certificate and is typically only seen on Apple-supplied command-line executables for iOS. These ad-hoc signatures are validated by searching for its code directory hash (CDHash) in the *trust cach*e. The kernel contains a static set of known CDHashes for all iOS built-in executables and applications that we refer to as the *static trust cache*. The kernel also maintains a linked list of the CDHashes for all applications and executables that have the passed full certificate validation before their first execution. We call this data structure the *dynamic trust cache.* The dynamic trust cache contains all the unique executables or applications that have been executed since boot. There does not appear to be any garbage collection or pruning of this linked list.



Are You Sure You Want to Open the Application “Bootstrap” from the Developer “iPhone Developer: Dino Dai Zovi (JQR9MDBG4N)”?

Figure XXX: Warning message on first launch of a custom application

The first time that a new custom application (i.e. not one signed by Apple and downloaded from the App Store) is run, iOS will prompt the user whether they are sure that they would like to run the application. The warning box, as shown in Figure XXX, displays both the application name and the name of the Developer Certificate that it is signed with.

## Entitlements

Signed executable binaries and applications may also contain an XML plist file specifying a set of *entitlements* to grant the application. This plist file contains a dictionary of keys with each representing a special privilege to be granted to the application. This mechanism allows Apple to make some processes more privileged than others, even if they are all running as the same Unix user id (i.e. user mobile).

The entitlements that may be granted to a particular application depend on which entitlements have been granted to the signing developer’s certificate in its associated provisioning profile.  Since Apple signs every provisioning profile, this allows them to carefully control which entitlements they will allow other developers to grant to that developer’s custom applications. The provisioning profile shown above in Figure XXX has an entitlements dictionary with three keys: application-identifier, get-task-allow, and keychain-access-groups. The application-identifier entitlement specifies a unique prefix for applications they develop. The Application Identifiers in iOS are used to uniquely refer to particular applications. While third-party developers are assigned unique prefixes, Apple has reserved com.apple.* for themselves. The get-task-allow entitlement permits applications signed with the embedded developer certificate to be debugged, indicating that this provisioning profile is intended to permit on-device custom application testing. Applications that are built for distribution (ad-hoc, App Store, or in-house) must omit this key in their entitlements as the associated distribution provisioning profiles will not permit the key to be in applications signed by the certificate in the profile.  The third entitlement, keychain-access-groups, defines a list of the iOS Keychain namespaces that the developer may permit their applications to access. Note that this list is set to only include the developer's Application Identifier prefix. This allows all of their apps to share iOS Keychain items if the developer wishes. The developer may also subdivide this namespace between their applications in order to prevent different applications that they develop from accessing the same items in the iOS Keychain.

As another example of how entitlements are specified, consider the Entitlements plist for the Apple-provided debugserver as shown in Figure XXX. This entitlements dictionary contains three entitlements: com.apple.springboard.debugapplications, run-unsigned-code, and seatbelt-profiles. The first entitlement is an application-specific entitlement that informs Springboard that debugserver should be permitted to debug applications. The second entitlement, run-unsigned-code, appears to indicate that the debugserver should be allowed to run unsigned code. This does not appear, however, to be sufficient to allow debugserver to run arbitrary unsigned code, as the authorization checks performed by AppleMobileFileIntegrity (described below) to run with an invalid code signature require additional entitlements. The seatbelt-profiles entitlement lists which built-in sandbox profile to apply to the process (the iOS sandbox is described in detail in a later chapter).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>com.apple.springboard.debugapplications</key>
        <true/>
        <key>run-unsigned-code</key>
        <true/>
        <key>seatbelt-profiles</key>
        <array>
                <string>debugserver</string>
        </array>
</dict>
</plist>
```

*Figure XXX: Entitlements plist for debugserver*

## Application Distribution Models

### Device Testing

The simplest form of application distribution is on-device testing using a Development Provisioning Profile. The Development Provisioning Profile created for the developer through the iOS Developer Portal or automatically by Xcode provisions a single device to run software signed by that developer's certificate. This provisioning profile must be installed onto the device before any of the developer's applications will run on it. The first time that a particular custom application is run using the developer's certificate, the user will be warned with the dialog box shown in Figure XXX. This model is intended to allow the developer to continually test their applications on their devices as they are developing them.

### Ad-Hoc Distribution

In the later stages of testing, the developer may wish to beta test their application on a wider range of devices and by a wider range of users.  The ad-hoc distribution model is designed for this scenario and allows a developer to distribute their application for testing to up to 100 users. This distribution model requires that the developer create a separate Ad-Hoc Provisioning Profile. In order to do so, the developer must collect the UDIDs from all of their testers devices. The developer must then enter these UDIDs into the iOS Developer Portal and create an Ad-Hoc Provisioning Profile that includes all of them in the ProvisionedDevices array. Finally, the developer must send each tester the provisioning profile and an iOS application as an .ipa file signed with the developer's Distribution Certificate. The testers may then drag both of these files into iTunes in order to install the application on their device.

**App Store Distribution**
In order to submit an application to Apple's App Store, the developer must build the application for distribution and sign it with their Distribution Certificate. They may then upload it through iTunes Connect for Apple's review. After the application is approved, Apple will re-sign the application with an Apple signing certificate and make the application available for download on the App Store. Since the application is now signed with an Apple certificate, users do not need to install the original developer's provisioning profile in order to run the application.

**In-House Distribution**
Enterprise developers may take advantage of a special distribution model that bypasses Apple's App Store review process and permits installation of the applications on devices without configuring their UDIDs in the provisioning profile. This model, In-House Distribution, is designed for large organizations to distribute their own custom applications to their employees.

The Enterprise Provisioning Profile may be preloaded on users' devices when they install their organization's iOS Configuration Profile or configure their device to use their organization's Mobile Device Management (MDM) server. Alternatively, the provisioning profile may be installed manually just as is done with Ad-Hoc Distribution. The custom application may be pushed out to user's devices through MDM or sent to individual users via Over-the-Air (OTA) Distribution.

**Over-the-Air (OTA) Distribution**
OTA Distribution is designed to allow Enterprise Developers to send applications to individual users in their organization through e-mail or by hosting the application on a web server. The link to the application can be sent to the users via e-mail, SMS, or linked to from another web page. Although it was not designed for it, this method can also be used to facilitate Ad-Hoc Distribution[5]. This is done by simply packaging up the Ad-Hoc Provisioning Profile with the application instead of an Enterprise Provisioning Profile.

While OTA Distribution greatly facilitates sending software to enterprise users and beta testers, it does create a potential avenue for social-engineering based attacks that may convince users to install undesirable applications. If an attacker is able to obtain an Enterprise Distribution Certificate (perhaps fraudulently), they may use this to build a malicious application that will not be reviewed by Apple. They can then proceed to attempt to convince users to install this application through SMS, e-mail, or other web pages linking to it.

As a partial mitigation against malicious use of this feature, iOS prompts the user before installing the application with "<Developer> would like to install <Application>: Cancel / Install". If the enterprise organization is a trusted name and the application name seems plausible the user will likely install it. The second mitigation against this is Apple's purposed "kill-switch" that can remove malicious applications from all iOS devices globally[6]. The exact functioning of this mechanism has not yet been identified.

[5] "Distribute Ad Hoc Applications Over the Air (OTA)", http://iphonedevelopertips.com/xcode/distribute-ad-hoc-applications-over-the-air-ota.html

[6] "Apple iPhone 'Kill Switch' Discovered", http://www.telegraph.co.uk/technology/3358115/Apple-iPhone-kill-switch-discovered.html

## Code Signing Enforcement

In order to prevent the introduction of new executable code at runtime, iOS implements a security protection called Code Signing Enforcement (CSE). This prevents applications from loading unsigned libraries, downloading new code at runtime, and using self-modifying code. It is also a strong protection against remote attacks that inject new native code into the application (i.e. classic buffer overflow and many other memory corruption attacks). It is, however, significantly stronger than protections such as Data Execution Prevention (DEP) found on Microsoft Windows and non-executable memory protections found on other operating systems. With these systems, an attacker can repurpose already loaded application code in order to enable the execution of an injected native code payload. With iOS CSE, the attacker must use already-loaded code in order to achieve all of their objectives, significantly raising the cost of attack development.

CSE is built into the iOS kernel's virtual memory system and most of its implementation is visible in Apple's open source xnu kernel[7], which is shared between iOS and Mac OS X. In essence, the virtual memory system tracks the validity of executable memory pages and the process as a whole using the "dirty" bit used to implement Copy-on-Write (COW) semantics and virtual memory page-ins. When an executable memory page is paged-in and is marked as being "dirty", its signature may have been invalidated and it must be reverified. New executable memory pages are always "dirty". If a single memory page is found to be invalid, then the entire process' code signing validity is also set to be invalid.

The code signing validity of the process is tracked with the CS_VALID flag in the csflags member of the kernel's proc structure for that process. If the executable's code signing signature has been validated prior to it being executed, the process begins execution with the CS_VALID flag set. If the process becomes invalid, then this flag will be cleared. What happens next depends on the CS_KILL flag. If this flag is set, the kernel will forcibly kill the process once it becomes invalid. On Mac OS X, the default is to not set this flag so that processes created from signed binaries may become invalid. On iOS, however, this flag is set by default so the process is killed once it becomes invalid. The flags defined for this field and a system call (csops) for getting and setting them from user space are documented in bsd/sys/codesign.h. The defined flags are also summarized in the table below:

| Flag Name | 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 |

## AppleMobileFileIntegrity

The AppleMobileFileIntegrity kernel extension uses the Mandatory Access Control Framework (MACF) to implement the security protections related to Code Signing.  It does so by installing MAC policy hook functions to check the signatures on executed binaries, set default code signing-related process flags, and add extra authorization checks to the task_for_pid, execve, and mmap system calls.

For each function, a summary of its purpose and usage by AMFI are given in the table below (the MAC label lifecycle management functions have been omitted for brevity).

[7] http://www.opensource.apple.com/source/xnu/

**AppleMobileFileIntegrity MAC Policy Hooks**

| MAC Policy 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 static and dynamic 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. |
| 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. |
| mpo_cred_check_label_update_execve | Indicate desire to change the process label at exec time | Returns true (non-zero) to indicate that AMFI needs to update credentials at exec time. |
| mpo_cred_label_update_execve | Update credential at exec time | Updates the credential with the entitlements of the newly executed binary or Application. |

In order to implement some more expensive operations outside of the kernel, the AMFI kernel extension communicates with a userspace daemon, amfid, over Mach RPC. The amfid RPC interface is simply two routines, as shown in the table below.

**AMFI Daemon Mach RPC Interface**

| 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. |

## Dynamic Code Signing

In order to support the “Nitro” native Just-in-Time (JIT) compilation JavaScript engine, MobileSafari possesses the dynamic-codesigning entitlement. This allows it to loosen the default Code Signing Enforcement mechanism and allow it to generate native executable code at runtime. Without this entitlement, the MobileSafari process would not be permitted to create new executable memory pages. On iOS 4.2 and earlier, MobileSafari was restricted to using an interpreted bytecode JIT JavaScript engine, which has much poorer performance than a native code JIT. With the release of iOS 4.3, however, MobileSafari began using the Nitro native code JavaScript JIT engine for increased performance.

## Assessment

The Mandatory Code Signing and Code Signing Enforcement security protections are a stronger defense against the execution of unauthorized native code than the protections found on other common desktop and mobile operating systems. In particular, the runtime Code Signing Enforcement mechanism is a unique feature that is not found on these other systems. It is a significantly stronger protection against the execution of injected native code than Microsoft’s Data Execution Prevention (DEP) and similar non-executable data memory protections in Linux and Mac OS X. In addition, these desktop operating systems do not have any similar features to Mandatory Code Signing and, by design, permit the execution of unsigned binaries and applications. Code Signing Enforcement makes remotely executing native code on iOS significantly more difficult than on desktop operating systems and Mandatory Code Signing similarly makes installing unauthorized software on iOS-based devices significantly more difficult than doing so on a desktop operating system.

While Google’s Android and RIM’s BlackBerry OS mobile operating systems implement similar features to the Mandatory Code Signing found in iOS, neither of them implement any non-executable data memory protections. Again, this makes remote injection and execution of native code easier on these platforms than on Apple’s iOS. In addition, it should be noted that all three platforms include mobile web browsers based on the same open-source WebKit HTML rendering engine. This means that all three platforms will likely be affected by any vulnerabilities identified in this component and, in fact, many such vulnerabilities have been identified over the last several years[8].

With iOS 4.3 and presumably later versions, the dynamic-codesigning entitlement in MobileSafari that is required to permit native code JavaScript JIT compilation also allows remote browser-based exploits to inject and execute native code. On previous versions of iOS and within applications that do not posses this entitlement, an attacker may only

[8] http://osvdb.org/search?search%5Bvuln_title%5D=WebKit

repurpose already-loaded native code in their attack. While this has been shown to be Turing-complete and therefore equivalent to arbitrary native code execution[9], it is significantly more work and not as reusable across target versions or applications as native code. In addition, the introduction of Address Space Layout Randomization in iOS 4.3 significantly complicates code-reuse attacks as well as any taking advantage of Dynamic Code Signing by requiring the attacker to also discover and exploit a memory disclosure vulnerability.

[9] Shacham, Hovav. “The Geometry of Innocent Flesh on the Bone”, Proceedings of CCS 2007, ACM Press.

# Sandboxing

## Introduction

The iOS application-based security model requires that applications and their data are isolated from other applications. The iOS Sandbox is designed to enforce this application separation as well as protect the underlying operating system from modification by a potentially malicious application. It does so by assigning each installed application a private area of the filesystem for its own storage and applying fine-grained process-level runtime security policies. These security policies enforce file and system access restrictions on the application.

The iOS sandboxing mechanism used for built-in applications, background processes, and third-party applications. In this paper, however, we only concern ourselves with the security policy applied to third-party applications in order to assess the risk posed by installing and running potentially malicious third-party applications.

## Application Containers

Third-party applications on iOS are each assigned their own *Application Home Directory* or *container* on the device filesystem. These containers are stored within /var/mobile/Applications/*UUID* and the Application UUID is randomly generated dynamically when the application is installed.  If an application is deleted and the re-installed, the old container will be deleted entirely and the application will be assigned a brand new random UUID when it is re-installed. In general, the application may only read and write files within their container, but there are a number of exceptions allowed by the runtime sandbox (described later in this chapter). There are several pre-defined subdirectories within the application's container with the intended purposes described in the table below.

**Container Subdirectories**

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

## Sandbox Profiles

Since all applications on iOS run as the same Unix user (mobile), the normal Unix-based security model is not able to provide sufficient application isolation and system protection. Instead, iOS uses the Sandbox kernel extension to enforce more customized security policies.  The Sandbox kernel module does this using the MAC Framework to install MAC policy hooks that apply a finer-grained security policy for accessing system objects and resources. These hooks make their access determination by evaluating the per-process security policy (referred to as its “profile”).

When an application is launched, its sandbox profile (if any) is determined by the value of its seatbelt-profiles entitlement. iOS includes 35 built-in profiles usually named for the single application that uses them. For example, the MobileSafari profile is only used by the MobileSafari application. The built-in profiles include profiles for graphical applications such as MobileSafari, MobileSMS, and MobileMail as well as non-graphical background processes such as apsd, ntpd, and ptpd.

The precise internals of Apple’s sandbox implementation have been well documented by other public research[10] and will only be covered at a basic level here. Instead we will focus on the sandbox profile applied to third-party applications (called “container”). All of the sandbox profiles in iOS have been extracted from the kernel and converted into a more human-readable from using Dion Blazakis’ XNUSandbox tools[11] with some updates and modifications. These converted sandbox profiles from iOS are included in the supplementary materials.

## Sandboxed Operations

The sandboxed system operations are listed in the table beginning on the next page. This list of operations was extracted from the iOS kernel and the purpose of each is described to the best of our knowledge.  The purpose of the operations were derived from their names, descriptions of similarly named kernel system calls, the xnu kernel source code, and sandbox profile source files found in Mac OS X.

For each operation, the sandbox profile in effect defines an ordered sequence of rules that are evaluated by the MAC policy hook installed on that event. The list of sandbox operations is somewhat symmetrical to the list of defined MAC policy “check” hooks like mpo_file_check_create, mpo_file_check_fcntl[12], etc. The MAC policy hooks are more specific than the list of sandbox operations, so several related MAC policy hooks are often folded into a single sandbox operation. Unfortunately, many of the iOS-specific sandbox operations are not documented in the Mac OS X header files and use “reserved” members in the mac_policy_ops structure.

Each sandbox operation rule may contain a boolean filter to determine whether the defined result (allow or deny) applies. The first rule with a matching filter determines the result for the requested operation and there is often a final default rule with a filter that always evaluates to true. These filter functions are statically defined in the kernel and they perform regular expression and string matches against resource names or specific resource type comparisons (i.e. socket type).

[10] Blazakis, Dionysus. “The Apple Sandbox”. The BlackHat Briefings DC 2011

[11] https://github.com/dionthegod/XNUSandbox

[12] See /usr/share/security/mac_policy.h

| Operation | Description |
|---|---|
| default | Evaluated if there are no rules defined for the specific operation |
| appleevent-send | Send an Apple Event |
| file* | Wildcard operation for all defined file operations below |
| file-chroot | Change the root directory of the current process to the given directory |
| file-ioctl | Control a device through an open device special file |
| file-issue-extension* | *Unknown* |
| file-issue-extension-read | *Unknown* |
| file-issue-extension-write | *Unknown* |
| file-mknod | Make a special file node (i.e. device file) |
| file-read* | Wildcard operation for all file read operations |
| file-read-data | Read data from a particular file |
| file-read-metadata | Read the metadata (owner, permissions, size, etc) associated with a file |
| file-read-xattr | Read the extended attributes on a file |
| file-revoke | Invalidate open file descriptors for the file at the given path |
| file-search | Search a file system volume quickly |
| file-write* | Wildcard operation for all file write operations |
| file-write-create | Create a new file |
| file-write-data | Write data to a file |
| file-write-flags | Change the flags for a file |
| file-write-mode | Change the file’s permissions |
| file-write-mount | Mount a filesystem |
| file-write-owner | Change the ownership of a file |
| file-write-setugid | Set set-user-id or set-group-id bits on a file |
| file-write-times | Change the file’s access times |
| file-write-unlink | Delete a file |
| file-write-unmount | Unmount a filesystem |
| file-write-xattr | Write to the file’s extended attributes |
| iokit* | Wildcard operation for IOKit operations |
| iokit-open | Open a connection to an IOKit User Client |
| iokit-set-properties | Set properties on an IOKit object |
| ipc* | Wildcard operation for IPC operations |
| ipc-posix* | Wildcard operation for POSIX IPC |
| ipc-posix-sem | Access POSIX semaphores |
| ipc-posix-shm | Access POSIX shared memory |
| ipc-sysv* | Wildcard for SysV IPC operations |
| ipc-sysv-msg | Access SysV messages |
| ipc-sysv-sem | Access SysV semaphores |
| ipc-sysv-shm | Access SysV shared memory |
| job-creation | Create a new job in launchd |

| Operation | Description |
|---|---|
| mach* | Wildcard operation for Mach operations |
| mach-bootstrap | Access mach bootstrap port |
| mach-lookup | Lookup a particular Mach RPC server over the bootstrap port |
| mach-priv* | Wildcard operation for Mach privileged operations |
| mach-priv-host-port | Access the Mach host port for the current host |
| mach-priv-task-port | Access the Mach task port for the current task |
| mach-task-name | Access the Mach task name port for the current task |
| network* | Wildcard operation for network operations |
| network-inbound | Accept an inbound network connection |
| network-bind | Bind a network or local socket |
| network-outbound | Establish an outbound network connection |
| priv* | Wildcard operation for privileged operations |
| priv-adjtime | Adjust the system time |
| priv-netinet* | Wildcard operation for privileged network operations |
| priv-netinet-reservedport | Listen for connections on a reserved network port (< 1024) |
| process* | Wildcard operation for process operations |
| process-exec | Execute a binary via execve() or posix_spawn() |
| process-fork | Create a new process |
| signal | Send a Unix signal |
| sysctl* | Wildcard operation for sysctl operations |
| sysctl-read | Read kernel state variable |
| sysctl-write | Write kernel state variable |
| system* | Wildcard operation for system operations |
| system-acct | Enable or disable process accounting |
| system-audit | Submit a BSM audit record to the system audit log |
| system-chud | Access the CHUD (Computer Hardware Understanding Development) Tools system call |
| system-fsctl | Control filesystems |
| system-lcid | Access login context |
| system-mac-label | Access MAC labels |
| system-nfssvc | Access NFS services system calls |
| system-reboot | Reboot the system |
| system-sched | *Unknown* |
| system-set-time | Set the system time |
| system-socket | Create a socket |
| system-suspend-resume | Suspend or resume the system |
| system-swap | *Manipulate system swap files?* |
| system-write-bootstrap | *Write boot blocks to a disk?* |
| mach-per-user-lookup | Lookup per-user Mach RPC servers |

## Container Sandbox Profile

The “container” sandbox profile is the profile that gets applied to all third-party applications. The profile, in general, restricts file access to the application’s container, some necessary system files, and the user’s address book.  The applications are generally allowed to read the user’s media (photos and music), but not write to them.  In addition, the application is allowed to read and delete files from the container Documents/Inbox directory, but not write to them. There are a few IOKit User Clients that the applications may interact with. All outbound network connections, except for connecting to launchd’s unix domain sockets, are allowed. Applications are also allowed to execute binaries from within their application bundle directory and send signals to themselves. The applications are also allowed to create sockets to receive kernel events and the system routing table. Finally, the profile does not restrict actions related to POSIX semaphores, shared memory, file IOCTLs, Mach bootstrap servers, network socket binding and accepting inbound connections, certain classes of privileged actions, and reading kernel state information through the kernel sysctl interface.

We have made an attempt to simplify and present the sandbox profile without sacrificing any accuracy in the table below. In reading the table, keep in mind that rules are evaluated in order. In that case, the first matching rule for a particular operation will determine the result of the security policy: allow or deny. If there are no rules defined for a particular operation, the rules for the related operation wildcard may be evaluated instead. If there are no rules for the applicable wildcard operations, then the rules for the “default” operation will be evaluated.

Many of the rules use regular expressions to specify applicable resource names. If the rule applies to all resources, we denote this with a ‘*’ in the table. If the rule only supports matching exact strings, the exact string argument is listed rather than a regular expression representing an exact string match.

| Action | Resource | Result |
|---|---|---|
| default | * | Deny |
| ipc-posix-sem | * | Allow |
| ipc-posix-shm | * | |
| file-ioctl | * | |
| mach-bootstrap | * | |
| mach-lookup | * | |
| network* | * | |
| network-inbound | * | |
| network-bind | * | |
| priv* | * | |
| priv-adjtime | * | |
| priv-netinet* | * | |
| priv-netinet-reservedport | * | |
| sysctl-read | * | |
| file-read*, file-write* | ^/private/var/mobile/Library/AddressBook(/\|$) | Allow |
| | ^/private/var/ea/ea([.0-9])+(out\|in)$ | |
| | ^/dev/null$ | |
| | ^/dev/dtracehelper$ | |
| | ^/dev/(ttys[0-9][0-9][0-9]\|ptmx)$ | |
| | ^/dev/(ptyp[0-9a-f]\|ttyp[0-9a-f])$ | |
| | ^/dev/(sha1_0\|aes_0)$ | |
| | ^/dev/(urandom\|random)$ | |
| | ^/dev/zero$ | |
| | ^/dev/(.)*$ | Deny |
| file-read* | ^/System/Library/Carrier Bundles/(.)*.png$ | Allow |
| | ^/private/var/mobile/Library/Carrier Bundles/(.)*/carrier.plist$ | |
| | ^/System/Library/Carrier Bundles/(.)*/carrier.plist$ | |
| | ^/private/var/mobile/Media/iTunes_Control/Artwork($\|/) | |
| | ^/private/var/mobile/Media/iTunes_Control/iTunes($\|/) | |
| | ^/private/var/mobile/Library/ConfigurationProfiles/PublicInfo($\|/) | |
| | ^/private/var/mobile/Library/Preferences/com.apple.carrier.plist | |
| | ^/private/var/mobile/Library/Carrier Bundles/(.)*.png$ | |
| | ^/private/var/logs(/\|$) | Deny |
| | ^/private/var/mobile/Library/Carrier Bundles($\|/) | |
| | ^/System/Library/Carrier Bundles($\|/) | |
| | ^/private/var/mobile/Media/Photos/Thumbs$ | Allow |
| | ^/private/var/mobile/Library/Caches/MapTiles(/\|$) | |
| | ^/private/var/mobile/Library/Caches/com.apple.IconsCache(/\|$) | |
| | ^/private/var/mobile/Media/Photos/Thumbs/([^/])+.ithmb$) | |

| Action | Resource | Result |
|---|---|---|
| | ^/private/var/mobile/Media/Photos/Videos($\|/) | Deny |
| | ^/private/var/mobile/Media/Photos/Thumbs($\|/) | |
| | ^/private/var/mobile/Media/Photos/com.apple.iPhoto.plist$ | |
| | ^/private/var/mobile/Media/com.apple.itdbprep.postprocess.lock$ | Allow |
| | ^/private/var/mobile/Media/(PhotoData\|Photos\|PhotoStreamsData)(/\|$) | |
| | ^/private/var/mobile/Media/com.apple.itunes.lock_sync$ | |
| | ^/private/var/mobile/Library/Preferences/com.apple.(books\|commcenter\|itunesstored\|springboard\|youtube\|AppStore\|MobileStore).plist | Deny |
| | ^/private/var/mobile/Library/FairPlay(/\|$) | |
| | ^/usr/sbin/fairplayd$ | |
| | ^/private/var/mobile/Media/ | |
| | ^/private/var/mobile/Library/Caches/com.apple.keyboards(/\|$) | Allow |
| | ^/private/var/mobile/Library/Preferences(/\|$) | |
| | ^/private/var/mobile/Library/Keyboard(/\|$) | |
| | ^/private/var/mobile/Library/ | Deny |
| | ^/private/var/mobile/Applications/(([-0-9A-Z])*)($\|/) | Allow |
| | ^/private/var/mnt/ | Deny |
| | ^/private/var/tmp(/\|$) | |
| | ^/private/var/mobile/Applications/(.)*$ | |
| file-write* | ^/private/var/mobile/Media(/\|$) | Deny |
| | /private/var/mobile/Library/Preferences/com.apple.Preferences.plist.[0-9A-Za-z][0-9A-Za-z][0-9A-Za-z][0-9A-Za-z][0-9A-Za-z][0-9A-Za-z][0-9A-Za-z]$ | Allow |
| | ^/private/var/mobile/Library/Preferences/com.apple.Preferences.plist$ | |
| | ^/private/var/mobile/Library/Keyboard/ | |
| | /private/var/mobile/Applications/(([-0-9A-Z])*)/Library/Preferences/.GlobalPreferences.plist$ | Deny |
| | ^/private/var/mobile/Applications/(([-0-9A-Z])*)/Library/Preferences/com.apple.PeoplePicker.plist$ | |
| | ^/private/var/mobile/Applications/(([-0-9A-Z])*)/Documents/Inbox/ | |
| | ^/private/var/mobile/Applications/(([-0-9A-Z])*)/(tmp\|Library\|Documents)(/\|$) | Allow |
| | ^/private/var/mobile/Applications/(.)*$ | Deny |
| file-write-unlink | ^/private/var/mobile/Applications/(([-0-9A-Z])*)/Documents/Inbox/ | Allow |
| iokit-open | AppleKeyStoreUserClient | Allow |
| | AppleMBXShared | |
| | IMGSGXShared_A0 | |
| | IMGSGXGLContext | |
| | IMGSGXDevice | |
| | IOMobileFramebufferUserClient | |
| | IOSurfaceRootUserClient | |

| Action | Resource | Result |
|---|---|---|
| | IOSurfaceSendRight | |
| | IMGSGXShared | |
| | IMGSGXGLContext_A0 | |
| | AppleJPEGDriverUserClient | |
| | AppleM2ScalerCSCDriverUserClient | |
| | AppleMBXDevice | |
| | AppleMBXUserClient | |
| network-outbound | ^/private/tmp/launchd-([0-9])+.([^/])+/sock$ | Deny |
| | ^/private/var/tmp/launchd/sock$ | |
| process-exec | ^/private/var/mobile/Applications/(([-0-9A-Z])*)/([^/])*.app($\|/) | Allow |
| signal | self | |
| system-socket | PF_SYSTEM, SYSPROTO_CONTROL | |
| | PF_SYSTEM | |
| | PF_ROUTE | |
| | * | Deny |

## Assessment

The sandbox profiles for third-party applications and applications that handle potentially untrusted data (MobileSafari and MobileMail) enforce tight restrictions on which files the applications may read or write. Very little sensitive data on the device is directly accessible on the filesystem from within the sandbox. The most sensitive files on the filesystem that are accessible through the sandbox are the SQLite file containing the user's address book contacts, the user's photo library, and iTunes library.

The container profile allows unrestricted access to local Mach RPC servers over the bootstrap port. There are currently 141 accessible bootstrap services, each providing an interface of one or more subroutines. These local RPC servers are used to implement a number of features, but documenting each one and all of their interfaces is a significant task and beyond the scope of this paper. The large number of Mach RPC services accessible from within the sandbox is a relatively large attack surface. If an attacker is able to exploit a memory corruption vulnerability in the implementation of any of these RPC interfaces, they may be able to inject a return-oriented payload into a process with a less restrictive sandbox environment or a process running without a sandbox. While the locations of most memory segments in the remote service process will be different than in the current process due to ASLR, the memory locations of the system libraries will be the same. This may aid the attacker in constructing their exploit and return-oriented payload. This is still a significantly more complex attack than exploiting a memory corruption vulnerability in the iOS kernel or an accessible IOKit User Client. The more likely risk is that these RPC interfaces may contain insufficient authorization checks and this may allow malicious third-party applications to perform undesirable actions on the user's device.

The most likely attack surface to be used in a sandbox escape is the iOS kernel, including the IOKit User Clients that are allowed by the sandbox profile. In fact, IOKit User Client vulnerabilities were exploited by both JailbreakMe 2.0[13] (IOSurface) and JailbreakMe 3.0 (IOMobileFramebuffer) in order to escape the MobileSafari sandbox. The kernel interfaces accessible from within the sandbox are likely to remain the most attractive target for a sandbox escape due to the fully unrestricted privileges yielded by obtaining kernel code execution.

[13] https://github.com/comex/star/

Finally, most other system-level actions are well restricted. Applications cannot use the fork system call to create a background process, but they can use posix_spawn to execute a binary from their application bundle directory. Any such binary, however, is part of the signed application bundle and cannot be written to at runtime (this is prevented by the sandbox profile).

# Data Encryption

## Overview

Mobile devices face an increased risk of sensitive data compromise through a lost or stolen device than is faced by traditional desktop workstations. While traditional workstations and laptops may be protected by Full Disk Encryption with pre-boot authentication, most mobile platforms cannot perform any pre-boot authentication. The data encryption facilities provided by the mobile platform, if any, are only available after the device has booted up. The limited data input possibilities on a touch screen or mobile device keyboard also make entering long passphrases infeasible. All of this makes data protection on mobile devices more challenging.

Apple's iOS 4 uses a system called Data Protection in order to protect the user's sensitive data in files on the filesystem and items in the Keychain. Data Protection uses the user's passcode and device-specific hardware encryption keys to derive encryption keys for designated protected data. This is done in order to hamper an attacker's efforts to recover the protected data from a lost or stolen device.

The iOS Data Protection internals were documented in precise detail by researchers at Sogeti and presented at the Hack in the Box Amsterdam conference in May 2011[14]. Since it has been publicly described in detail elsewhere, this chapter will only provide an overview of the data encryption facilities and limitations in iOS. In particular, users should be aware of the currently very limited coverage of the Data Protection API and the ease with which the default simple four-digit passcodes can be guessed on a lost or stolen device.

## Hardware Encryption

The iPhone 3GS and later devices include a hardware AES cryptographic accelerator. This crypto accelerator is used for realtime filesystem encryption and various other encryption tasks by iOS. In addition to providing high-performance data encryption and decryption capabilities, it also provides many security services through its use of hardware-protected AES keys.

The AES accelerator includes both a unique per-device key (referred to as the UID Key) and a globally shared key (referred to as the GID Key) that are not accessible to the CPU. They may only be used for encryption or decryption through the AES accelerator itself. The GID Key is primarily used to decrypt iOS firmware images provided by Apple. The UID Key is used to derive a number of device-specific AES keys that are used to encrypt the filesystem metadata, files, and Keychain items.

[14] "iPhone Data Protection in Depth", http://esec-lab.sogeti.com/dotclear/public/publications/11-hitbamsterdam-iphonedataprotection.pdf

## Data Protection API

The Data Protection API is designed to let applications declare when files on the filesystem and items in the keychain should be decrypted and made accessible by passing newly defined *protection class* flags to existing APIs. The protection class instructs the underlying system when to automatically decrypt the indicated file or keychain item.

In order to use enable Data Protection for files, the application must set a value for the NSFileProtectionKey attribute using the NSFileManager class. The supported values and what they indicate are described in the table below. By default, all files have the protection class NSFileProtectionNone, indicating that they may be read or written at any time.

**File Protection Classes**

| Protection Class | Description |
|---|---|
| `NSFileProtectionNone` | The file does not need to be protected and can be read from or written to at any time |
| `NSFileProtectionComplete` | The file is to be encrypted using a key derived, in part, from the user's passcode and should only be made available for reading and writing while the device is unlocked. |

The protection class of items in the Keychain are similarly indicated by specifying the protection class to the SecItemAdd or SecItemUpdate functions. Compared with files where the protection is either non-existent or complete, the Keychain item protection classes offer somewhat more control. Keychain items can be accessible at any time, after the device has been first unlocked after boot, or anytime that the device is unlocked. In addition, the application may specify whether the Keychain item can be migrated onto other devices or not. If one of the "ThisDeviceOnly" protection classes are used, then the Keychain item will be encrypted with a key derived from the Device Key. This ensures that only the device that created the Keychain item can decrypt it. By default, all Keychain items are created with a protection class of kSecAttrAccessibleAlways, indicating that they can be decrypted at any time and migrated onto other devices.

**Keychain Item Protection Classes**

| Protection Class | Description |
|---|---|
| `kSecAttrAccessibleAlways` | The keychain item is accessible at any time |
| `kSecAttrAccessibleAfterFirstUnlock` | The keychain item may be accessed anytime after the device is unlocked for the first time after being powered on. |
| `kSecAttrAccessibleWhenUnlocked` | The keychain item may only be accessed while the device is unlocked. |
| `kSecAttrAccessibleAlwaysThisDeviceOnly` | The keychain item is accessible at any time, but it may not be migrated onto another device. |
| `kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly` | The keychain item may be accessed anytime after the device is unlocked for the first time after being powered on and may not be migrated onto another device. |
| `kSecAttrAccessibleWhenUnlockedThisDeviceOnly` | The keychain item may be accessed while the device is unlocked and may not be migrated onto another device. |

## Filesystem Encryption

The iPhone 3GS and later devices use the embedded encryption accelerators to perform block-level encryption on both the System and Data partitions. This is primarily to support a quick remote wipe operation. On earlier versions of iOS, the remote wipe command would force the device to overwrite each block of the flash storage, which could take hours to complete. Now, the entire filesystem can be rendered unreadable by simply wiping a single encryption key (referred to here as the File System Key).

This level of filesystem encryption does not prevent the recovery of data from a lost device. If the remote wipe command is not received, an attacker may boot the device using a custom RAM disk and use the device itself to decrypt the encrypted partitions. This was demonstrated against the iPhone 3GS running iOS 3 by using iPhone jailbreak utilities to exploit the Boot ROM and boot a custom RAM disk enabling remote SSH access to the device[15]. This same style of attack can be used against iOS 4, but it will only give access to the filesystem metadata (directory structure, file names, permissions, owners, file sizes, etc) and files protected using the NSProtectionNone protection class.

If the raw filesystem is forensically copied off the device by directly accessing the device file, however, only the filesystem metadata will be accessible. The file contents, even files protected with NSProtectionNone, are encrypted using keys derived from the Device Key and are not readable off of the original device unless the NSProtectionNone protection class key (referred to as the “Class D Key”) has also been recovered from the device. This Class D Key is stored on the device flash storage but encrypted using the Device Key, which is in turn derived from the device-specific UID Key. The UID Key is only usable through the encryption accelerator and is not directly accessible by the CPU. Even with this key, however, files protected with NSProtectionComplete will not be accessible without the User Passcode Key.

### File Protection

All files are encrypted with a unique File Key stored in an extended attribute of the file. Since this is stored within the filesystem metadata, it is effectively protected by the File System Key. The File Key itself is protected by the Class Key, this Class Key is protected by the Device Key for NSProtectionNone and is protected by both the Device Key and the User Passcode Key for NSProtectionComplete. The FIle Key is stored in the com.apple.system.cprotect extended attribute with the format described in the table below.

**com.apple.system.cprotect Extended Attribute Format**

| Type | Name | Description |
|---|---|---|
| uint32_t | xattr_version | Version 2 |
| uint32_t | unknown | Appears unused |
| uint32_t | persistent_class | Protection Class A-E (1-5) |
| uint32_t | key_and_integrity_length | Always 40 bytes |
| uint8_t | persistent_key[32] | 256-bit AES key |
| uint8_t | persistent_integrity[8] | 160-bit SHA1 hash |

## iOS Passcodes

The Springboard application is responsible for presenting the lock screen and accepting the user’s passcode. The passcode is verified by passing it through the MobileKeyBag framework and to the AppleKeyStoreUserClient KeyBagUnlock method. In the kernel, the AppleKeyStore kernel extension derives the User Passcode Key through the PBKDF2 key stretching algorithm using AES with the UID Key as the pseudorandom function. This User Passcode Key is then used to unwrap the Class Keys from the System KeyBag. If the unwrapping fails, then the KeyBagUnlock method returns that the passcode is incorrect.

After successive incorrect passcode guesses, Springboard enforces a increasing “back-off” delay in order to deter casual passcode guessing. In addition, the device may be configured to wipe itself after a chosen number of incorrect passcode

[15] http://www.zdziarski.com/blog/?p=516

guesses. These protections, however, are only enforced by the Springboard application, not the iOS kernel. With a custom command-line application on a jailbroken device, the passcodes can be guessed directly against the AppleKeyStoreUserClient without encountering an increasing delay or a forced device wipe.

In the course of this assessment, such a tool was written and tested on a jailbroken iPhone 4. On this configuration, passcodes could be guessed at an average rate of 9.18 guesses per second. This yields a maximum of 18 minutes to guess a simple 4-digit passcode.

## Data Protection API Coverage

An enumeration of all of the files metadata on a raw filesystem image identified only three files protected with NSProtectionComplete:

- Data:/mobile/Library/Caches/Snapshots/com.apple.Preferences/UIApplicationAutomaticSnapshotDefault-Portrait@2x.jpg
- Data:/mobile/Library/Caches/Snapshots/com.apple.VoiceMemos/UIApplicationAutomaticSnapshotDefault-Portrait@2x.jpg
- Data:/mobile/Library/Mail/Protected Index

These files indicated the only two aspects of iOS that make use of Data Protection: automatic screenshots and MobileMail. The automatic screenshots are taken when the user is in an application and hits the home button to return to the Springboard screen. This causes a screen shrinking transition to be displayed using a screenshot of the current application view. These screenshots are written to the filesystem protected using the Data Protection API. In addition, MobileMail uses Data Protection for stored e-mail messages and attachments. No other files or aspects of iOS were found to use Data Protection for files stored on the filesystem.

As was demonstrated by a research report from the Fraunhofer Institute for Secure Information Technology[16], many passwords in the Keychain are protected using kSecAttrAccessibleAlways and are accessible from a lost phone without requiring knowledge of the user's passcode. Their results are summarized for common built-in password types in the table below.

**Password Keychain Item Protection Classes**

| Password Type | Keychain Item Protection Class |
|---|---|
| MobileMail AOL Email Account | kSecAttrAccessibleWhenUnlocked |
| CalDav | kSecAttrAccessibleAlways |
| MobileMail IMAP Account | kSecAttrAccessibleWhenUnlocked |
| MobileMail SMTP Account | kSecAttrAccessibleWhenUnlocked |
| MobileMail Exchange Account | kSecAttrAccessibleAlways |
| iOS Backup Password | kSecAttrAccessibleWhenUnlocked |
| LDAP | kSecAttrAccessibleAlways |
| Voicemail | kSecAttrAccessibleAlways |
| VPN IPsec Shared Secret | kSecAttrAccessibleAlways |

[16] "Lost iPhone? Lost Passwords! Practical Consideration of iOS Device Encryption Security", http://sit.sit.fraunhofer.de/studies/en/sc-iphone-passwords.pdf

| Password Type | Keychain Item Protection Class |
|---|---|
| VPN XAuth Password | kSecAttrAccessibleAlways |
| VPN PPP Password | kSecAttrAccessibleAlways |
| Safari Forms Auto-Fill | kSecAttrAccessibleWhenUnlocked |
| WiFi WPA Enterprise with LEAP | kSecAttrAccessibleAlways |
| WiFi WPA | kSecAttrAccessibleAlways |

## Assessment

The Data Protection API in iOS is a well designed foundation that enables iOS applications to easily declare which files and Keychain items contain sensitive information and should be protected when not immediately needed. There are no obvious flaws in its design or use of cryptography. It is, however, too sparingly used by the built-in applications in iOS 4, let alone by third-party applications. This leaves the vast majority of data stored on a lost device subject to recovery if a remote wipe command is not sent in time.

The default iOS simple four-digit passcodes can be guessed in under 20 minutes using freely available tools, which will allow the attacker will physical access to the device to also decrypt any files or items in the Keychain that are protected using the Data Protection API. For this reason, it is crucial that sufficiently complex passcodes be used on all iOS devices.

Even with sufficiently complex passcodes, there are a number of sensitive passwords that may be recovered from a lost device, including passwords for Microsoft Exchange accounts, VPN shared secrets, and WiFi WPA passwords. This should be taken into account and these passwords should be changed if an iOS device storing them is lost.