Behind the Scenes with iOS Security

Black Hat 2016

Ivan Krstić
Head of Security Engineering and Architecture, Apple
Decrypted Kernel Caches
Component Encryption

iOS 10

User data—No change to encryption

Image3 (pre-iPhone 5S)—iBoot, kernel caches, boot logos no longer encrypted

Image4 kernel caches—No longer encrypted

Changes made as part of wider set of performance optimizations

Encryption for these objects was no longer adding a lot of value

No impact to platform security or encryption of user data
Hardened WebKit JIT Mapping
Hardened WebKit JIT Mapping

Background

Just-in-time compilation is necessary for high-performance JavaScript

iOS normally requires all executable pages to be signed

Code signing policy is relaxed in Safari through `dynamic-codesigning` entitlement to support JIT compilation
Hardened WebKit JIT Mapping
iOS 9

32MB RWX JIT memory region

Write-anywhere primitive sufficient for arbitrary code execution

Attacker can write shell code into JIT region and jump to it without ROP
Hardened WebKit JIT Mapping

Execute-only memory protection

Hardware support introduced in ARMv8
Kernel implementation added in iOS 10
Allows us to emit code containing secret data, not readable within the process
Hardened WebKit JIT Mapping

Split view

Create two virtual mappings to the same physical JIT memory
One executable, one writable
The location of the writable mapping is secret
Hardened WebKit JIT Mapping

Tying it all together

Writable mapping to JIT region is randomly located

Emit specialized `memcpy` with base destination address encoded as immediate values

Make it execute-only

Discard the address of the writable mapping

Use specialized `memcpy` for all JIT write operations
void initializeSeparatedWXHeaps(void* stubBase, size_t stubSize, void* jitBase, size_t jitSize)
{
    mach_vm_address_t writableAddr = 0;

    // 1. Create a second mapping of the JIT region at a random address.
    vm_prot_t cur, max;
    kern_return_t ret = mach_vm_remap(mach_task_self(), &writableAddr, jitSize, 0,
                                       VM_FLAGS_ANYWHERE | VM_FLAGS_RANDOM_ADDR,
                                       mach_task_self(), (mach_vm_address_t)jitBase, FALSE,
                                       &cur, &max, VM_INHERIT_DEFAULT);

    bool remapSucceeded = (ret == KERN_SUCCESS);
    if (!remapSucceeded)
        return;

    // 2. Assemble specialized memcpy function for writing into the JIT region.
    MacroAssemblerCodeRef writeThunk =
        jitWriteThunkGenerator(reinterpret_cast<void*>(writableAddr), stubBase, stubSize);

    int result = 0;

#if USE(EXECUTE_ONLY_JIT_WRITE_FUNCTION)
    // 3. Prevent reading the memcpy code we just generated.
    result = mprotect(stubBase, stubSize, VM_PROT_EXECUTE_ONLY);
    RELEASE_ASSERT(!result);
#endif

    // 4. Prevent writing into the executable JIT mapping.
}
// 4. Prevent writing into the executable JIT mapping.
result = mprotect(jitBase, jitSize, VM_PROT_READ | VM_PROT_EXECUTE);
RELEASE_ASSERT(!result);

// 5. Prevent execution in the writable JIT mapping.
result = mprotect((void*)writableAddr, jitSize, VM_PROT_READ | VM_PROT_WRITE);
RELEASE_ASSERT(!result);

// 6. Zero out writableAddr to avoid leaking the address of the writable mapping.
memset_s(&writableAddr, sizeof(writableAddr), 0, sizeof(writableAddr));

jitWriteFunction =
reinterpret_cast<JITWriteFunction>(writeThunk.code().executableAddress());
}
iOS 9
iOS 10
iOS 10

```
mov x0, #0
```

```
mov x0, #0
```
iOS 10

mov x0, #0

mov x0, #0
Hardened WebKit JIT Mapping

iOS 10

Write-anywhere primitive now insufficient for arbitrary code execution
Attacker must subvert control flow via ROP or other means or find a way to call execute-only JIT write function
Mitigation increases complexity of exploiting WebKit memory corruption bugs
Data Protection with the Secure Enclave Processor
Data Protection

Goals

User data protected by strong cryptographic master key derived from user passcode
No offline attack on user passcode—Hardware-bound master key derivation
No brute force—Hard limit on number of passcode attempts
Hardware keys for master key derivation not directly exposed to any mutable software
Secure support for alternative unlock mechanisms (Touch ID, Auto Unlock)
Data Protection

Goals—Sidestep AP attack surface

Authentication policy enforcement even under adversarial AP
Master (long-term) key material never exposed to AP
Non-master key material exposed to AP must be ephemeral and session-bound
Secure Enclave Processor
Secure Enclave Processor

Overview

Dedicated SoC core provides trusted environment for handling cryptographic material
Arbitrates all user data access
Hardware accelerators for AES, EC, SHA
Manages its own encrypted memory and communicates with the AP using mailboxes
Factory-paired secure channels to Touch ID sensor and Secure Element
Device UID Key

Background

Each SEP has reference access to a unique private key (UID)

UID generated by SEP itself immediately after fabrication, using its own free-running oscillator TRNG

Available for cryptographic operations via commands exposed by the Secure ROM

No access to UID key material from SEP or other mutable software after fuses blown
User Keybags

Background

Sets of keys generated for each user to protect their data at rest.
Keys wrapped by master key derived from user passcode and SEP UID.
After 10 incorrect passcode entries, SEP will not process any further attempts.
Different policy associated with each keybag key—Usage, availability.
### User Keybags

#### Class keys

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A (256-bit AES)</td>
<td>Only available while the device is unlocked</td>
</tr>
<tr>
<td>B (Curve 25519)</td>
<td>Public key always available, private key only available when device is unlocked</td>
</tr>
<tr>
<td>C (256-bit AES)</td>
<td>Available after the user unlocked the phone at least once after boot</td>
</tr>
<tr>
<td>D (256-bit AES)</td>
<td>Always available</td>
</tr>
</tbody>
</table>
Keybag version 4

KDF Salt

Iteration Count

Key Identifier: Class B
Key Type: Curve25519
Wrapped Private Key Bytes

Public Key Bytes

Key Identifier: Class A
Key Type: AES
Wrapped Private Key Bytes
Master Key Derivation

Userland

Passcode → KDF

Salt

SEP

Mk_i = KDF_2(E(UID, MK_{i-1})) → Master Key

Timed Iterations (100-150ms)
Filesystem Data Protection

Overview

File blocks are encrypted using AES-XTS with 128-bit keys

Each file on the user partition is encrypted using a unique random key chosen by SEP

Raw file keys are never exposed to the AP

• Wrapped with a key from the user keybag for long-term storage
• Wrapped with an ephemeral key while in use, bound to boot session
1. Fetch wrapped file_key from metadata
2. Unwrap file_key using keybag key
3. Wrap file_key using ephemeral_key, return ephemeral wrapped file_key to kernel
4. Send IO command with ephemeral wrapped file_key
1) kernel boots (system partition)
2) AppleKeyStore loads D key (before user partition is mounted)
3) decrypt class D key, load into keyring
4) decrypt HFS metadata with media key
5) launchd mounts user partition
6) launchd starts keybagd
7) keybagd loads system keybag (class D key available)
8) decrypts keybag with prot key from effaceable
9) keybag loaded into sks memory, class B public loaded into keyring
10) class keys cannot be decrypted until we get the passcode
11) launchd permits userspace to start loading

Userspace

XNU (Kernel)

SEP

class D

SEP UID

class D

SKS

master key + SEP UID

class A
class B (priv)
class C

SKS memory

SKS keyring

class B (public)
class D

systembag.kb

/var/keybags/systembag.kb

contains the device keybag

AppleKeyStore

SEP endpoint to SKS

HFS

Effaceable

media key
keybag prot key
Class D

boot
Userspace

1) SpringBoard acquires the passcode

XNU (Kernel)

7) first unlock notification sent

SEPT

2) generate master key

bio unlock token only created if bio unlock is enabled steps 4 & 5

3) decrypt class keys, add to keyring

master key

4) encrypt master key with random secret – this encrypted master key never leaves SKS

5) send random secret to SBIO, destroy it in SKS

6) securely destroy raw master key

Userspace

Darwin lock/unlock Notifications

XNU (Kernel)

UserEventAgent

AppleKeyStore

SEP Endpoint to SKS

2) generate master key

master key

3) decrypt class keys, add to keyring

4) encrypt master key with random secret – this encrypted master key never leaves SKS

5) send random secret to SBIO, destroy it in SKS

6) securely destroy raw master key
1) device locks

2) purge class A and class B priv keys

3) send lock notification
pressing home button starts Touch ID sensor

1) pressing home button starts Touch ID sensor

2) template match sent to SBIO

3) upon successful match send random secret to SKS

4) decrypt master key

5) decrypt class keys, add to keyring

6) securely destroy master key

7) send unlock notification

Userspace

XNU (Kernel)

SEP

Touch ID Sensor

Userspace

UserEventAgent

Darwin lock/unlock Notifications

Notifications

UserEventAgent

SpringBoard

MobileKeybag.framework

AppleKeyStore

SEP endpoint to SKS

AppleMesa

SEP endpoint to SBIO

SKS

random secret

master key

master key + SEP UID

class A

class B (priv)

class C

class D

SKS memory

master key

SKS keyring

class A

class B (priv)

class B (public)

class C

class D

Touch ID unlock
Unattended Update—Install Later

- **First Unlock**
- **Notify Update**
- **Lock**
- **Last Unlock Before Update**
- **Software Update**
- **Lock**
- **Update Window**

**Timeline:**
- 1:00 PM: Notify Update
- 10:00 PM: Last Unlock Before Update
- 10:15 PM: Software Update
- 10:30 PM: Lock (OTUT persisted)
- 3:00 AM: Update Window

**Timers:**
- 8 hr persist timer
- 20 min commit timer

**Events:**
- Prompt for passcode
- OTUT creation enabled
- OTUT created
- OTUT persist enabled
- OTUT persisted
- OTUT committed
- Reboot
SoC Security Mode

Demotion

Production devices can be “demoted” to enable some debugging features like JTAG and loading development software on the AP (but not the SEP)

Requires full OS erase and device explicitly authorized by the personalization server

Forces a different UID on the SEP, no access to existing user data after demotion

<table>
<thead>
<tr>
<th>SoC mode</th>
<th>AP status</th>
<th>SEP status</th>
<th>SEP UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development fused</td>
<td>Development</td>
<td>Development</td>
<td>Development</td>
</tr>
<tr>
<td>Production fused</td>
<td>Production</td>
<td>Production</td>
<td>Production</td>
</tr>
<tr>
<td>AP demoted</td>
<td>Development</td>
<td>Production</td>
<td>Development</td>
</tr>
</tbody>
</table>
Goals

- User data protected by strong cryptographic master key derived from user passcode
- No offline attack on user passcode—hardware-bound master key derivation
- Hardware keys for master key derivation not directly exposed to any mutable software
- Secure support for alternative unlock mechanisms (Touch ID, Auto Unlock)
- Sidesteps AP attack surface, SEP policy enforced under adversarial AP
Synchronizing Secrets
Synchronizing Secrets

Uses

Passwords and credit card information available on all of a user’s devices
Auto Unlock cryptographic keys shared between Apple Watch and Mac
HomeKit cryptographic keys available on all devices
Synchronizing Secrets
Traditional approaches

Wrap user secrets with strong random key
• User has to take care of a printed “sock drawer key” and enter it on each device
• If printed key is lost, losing devices means loss of secrets

Wrap user secrets with KDF-derived key from their password
• Account provider backend is privileged, can intercept or brute-force account password
• If user resets their account password, must prompt for old password to recover secrets
• Anyone else in possession of wrapped user secrets can launch a brute-force attack
“Humans are incapable of securely storing high-quality cryptographic keys, and they have unacceptable speed and accuracy when performing cryptographic operations.”

C. Kaufman, R. Perlman, M. Speciner
Synchronizing Secrets
Goals—Inspired by Data Protection

Selected user secrets available on all of the user’s devices
Synchronization protected with strong cryptographic keys
User can recover secrets even if they lose all their devices
User secrets not exposed to Apple
No brute-force—backend not in a privileged position
iCloud Keychain

Approach

Each device locally generates iCloud Keychain synchronization key pair
User explicitly approves new devices joining the sync circle from a device already in it
Sync circle uses Apple cloud backend for storage and message passing
• No data is accessible to Apple
• Backend not in a privileged position since key pair is strong and random

What if all devices are lost, or need to configure new device without access to old one?
iCloud Keychain Backup

Premise

New credential—iCloud Security Code (commonly device passcode), unknown to Apple

Generate strong random backup ("escrow") key, wrap with KDF-derived key from iCSC

Back up copy of iCloud Keychain secrets to the Apple cloud, encrypted with escrow key

Send wrapped escrow key to Apple

In case of device loss or new device, user can recover secrets with their iCloud password and the iCSC
Synchronizing Secrets
Goals—Inspired by Data Protection

- Selected user secrets (passwords, credit cards, …) available on all of the user’s devices
- Synchronization protected with strong cryptographic keys
- User can recover secrets even if they lose all their devices
- User secrets not exposed to Apple

Backend not in a privileged position to brute-force keys protecting user secrets
iCloud Keychain Backup

Backend attack surface

In naive implementation, backend could brute force the iCSC to access escrow key.

Just like with SEP, need to enforce policy over escrow key.

No brute-force—Want hard limit on escrow recovery attempts under adversarial cloud.

What if escrow key unwrapping only took place in Hardware Security Modules?
Cloud Key Vault

Overview

HSMs running custom secure code connected to Apple cloud
Key vault fleet operates its own CA, private key never leaves the hardware
Each iOS device hardcodes key vault fleet CA cert
## Cloud Key Vault

### HSM keys

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSCIK</td>
<td>RSA-2048, allows signing custom secure code to run on the HSM</td>
</tr>
<tr>
<td>AK</td>
<td>256-bit for HMAC-SHA-256, to authenticate messages between vault units</td>
</tr>
<tr>
<td>CA</td>
<td>RSA-2048, to certify service key (SK)</td>
</tr>
<tr>
<td>SK</td>
<td>RSA-2048, allows unwrapping escrow records</td>
</tr>
</tbody>
</table>
Cloud Key Vault

Escrow record generated on iOS device

- SRP verifier for iCSC
- User’s escrow key
- Maximum failure count

Encrypted to public service key
Cloud Key Vault

Understanding the design

A kind of SEP Data Protection approach for escrow records

Vault service private key material not available to mutable software, just like SEP UID key

User attempting escrow recovery sends previous escrow record, establishes SRP session

Vault unwraps escrow record, SRP with user device against iCSC verifier in the record, return escrow key if successful

If SRP fails due to incorrect iCSC, vault unit bumps a secure counter

- If maximum failure count (10) is exceeded for the record, record is marked terminal
Cloud Key Vault
Redundancy and attack surface

Users can’t escrow to only a single vault unit, need redundancy
Escrowing to multiple units means multiplying the maximum failure count providing more opportunity for brute force attack
Cloud Key Vault

Club

Host | HSM
---|---
Host | HSM
Host | HSM
Host | HSM
Host | HSM
Cloud Key Vault

Club

Group of five vault units that share a single SK—Service Key
User generates escrow record for a particular club, certified by the fleet CA
Solves redundancy, but not the brute force limiting problem
Club members would still be subject to partitioning attacks
Cloud Key Vault

Quorum commit

Each club member maintains its own failure count for each escrow record

Escrow recovery prompts a vote, majority quorum required to proceed

Provides redundancy and breaks membership partitioning attacks
Cloud Key Vault
Fleet

Authenticated object storage

iCloud escrow service
Cloud Key Vault
Escrow Recovery Transaction

1. Proposal
2. Vote
3. Schedule
4. Ack
5. Perform
6. Confirm

“I propose an update. Please give me your failure count for this record.”
Cloud Key Vault
Escrow Recovery Transaction

1. Proposal
2. Vote
3. Schedule
4. Ack
5. Perform
6. Confirm

“My counter is 4. I vote yes to update since I’m not in another transaction.”
Cloud Key Vault
Escrow Recovery Transaction

1. Proposal
2. Vote
3. Schedule
4. Ack
5. Perform
6. Confirm

“We have majority quorum and no one’s record is terminal, prepare to update failure count to 5.”
“OK, I’ve verified majority quorum and stand ready to increase failure count to 5.”
Cloud Key Vault

Escrow Recovery Transaction

1. Proposal
2. Vote
3. Schedule
4. Ack
5. Perform
6. Confirm

“Please proceed with increasing failure count to 5 for this record.”
Cloud Key Vault

Escrow Recovery Transaction

1. Proposal
2. Vote
3. Schedule
4. Ack
5. Perform
6. Confirm

“OK, my failure count is now 5 for this record.”
Cloud Key Vault

Who watches the watchers?

A given vault fleet runs code signed by its CSCIK (custom secure code signing key)

Use of this signing key requires a quorum of physical vault admin smart cards

Admin cards are created in a secure ceremony when fleet is commissioned, stored in separate physical safes in custody of three different organizations at Apple in tamper-proof evidence bags
Cloud Key Vault

Who watches the watchers?

If someone got their hands on all the admin cards…

 Couldn’t they sign a malicious custom secure code image that can brute-force iCSC for arbitrary escrow records?
No.
Cloud Key Vault

Before a fleet goes into production

Members of all three admin card-carrying organizations meet in a secure facility
Cross-check serial numbers on evidence bags and on cards

Attestations

• Card carriers present at creation of the admin cards
• No other cards were created
• Evidence bags remained sealed since creation
• Cards present today are the ones originally created
Physical One-Way Hash Function
(We Run the Cards Through a Blender)
Cloud Key Vault

Final attestation

All admin cards originally created are now destroyed.
The cards were not used to sign any other custom secure code that can be loaded.
No other mechanism is known for changing custom secure code or loading new code.
This enables us to make the unequivocal commitment that user secrets are not exposed to Apple.
Some News
Apple Security Bounty
Apple Security Bounty

Great help from researchers in improving iOS security all along

iOS security mechanisms continue to get stronger

Feedback from Apple Red Team and external researchers: as iOS security has advanced, increasingly difficult to find the most critical security issues
Apple Security Bounty

Rewards researchers who share critical issues with Apple
We make it a priority to resolve confirmed issues as quickly as possible
Provide public recognition, unless asked otherwise
## Initial Categories

<table>
<thead>
<tr>
<th>Category</th>
<th>Max. Payment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secure boot firmware components</td>
<td>$200,000</td>
</tr>
<tr>
<td>Extraction of confidential material protected by the Secure Enclave Processor</td>
<td>$100,000</td>
</tr>
<tr>
<td>Execution of arbitrary code with kernel privileges</td>
<td>$50,000</td>
</tr>
<tr>
<td>Unauthorized access to iCloud account data on Apple servers</td>
<td>$50,000</td>
</tr>
<tr>
<td>Access from a sandboxed process to user data outside of that sandbox</td>
<td>$25,000</td>
</tr>
</tbody>
</table>
Apple Security Bounty

Payments

We require a clear report and working proof of concept.

Vulnerability must affect latest shipping iOS, and where relevant, latest hardware.

Exact payment amount determined after review by engineering team, criteria include novelty and likelihood of exposure/degree of user interaction required.

Researcher can elect to donate reward to charity of their choice, Apple will consider matching 1:1.
Apple Security Bounty

Few dozen initially invited researchers

If vulnerabilities that would be covered under the program are submitted by researchers outside of the program, we will review the submission and, if the work merits it, invite researcher into the program and reward the vulnerability.
September
Thank You!

product-security@apple.com