iOS Platform Security

Mobile Security 2024

Florian Draschbacher
florian.draschbacher@iaik.tugraz.at

Some slides based on material by **Johannes Feichtner**

Outline

- Low-level System Security
- Updates
- Encryption Systems
- Key Management & Passcodes
- Backup

iOS Platform Fundamentals

iOS Device Architecture

- The device is comprised of a main (ARM) CPU and several coprocessors

- Secure Enclave Processor (SEP)
  - Separate processor for cryptographic operations
  - Key storage, management, encryption / decryption
    - Group ID (GID) key shared between SoC family
    - Unique ID (UID) key generated by SEP at factory
  - Securely paired to FaceID and TouchID sensors

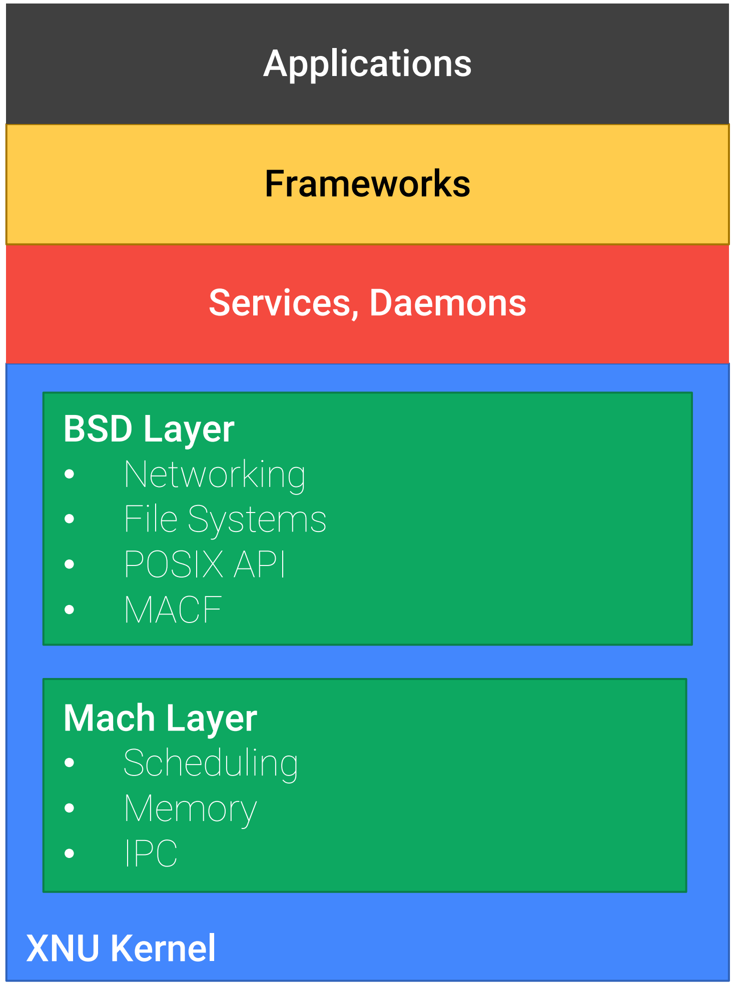
- Secure Element
  - Separate chip for Apple Pay and NFC

Picture: Google / Apache 2.0

IAIK TU Graz

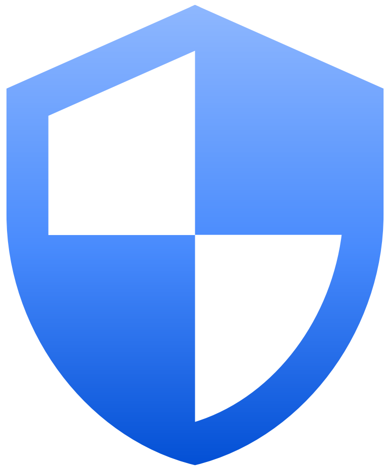
iOS

- XNU Kernel
  - Based on Mach microkernel
  - Added FreeBSD layer for POSIX compatibility
  - IOKit device drivers
  - Shared with macOS
  - Open source!
- Userspace
  - Partly open-source (Darwin)
  - Frameworks (e.g. Cocoa Touch)
  - Daemons, Services, Programs, Apps
    - launchd
    - SpringBoard



Mandatory Access Control Framework (MACF)

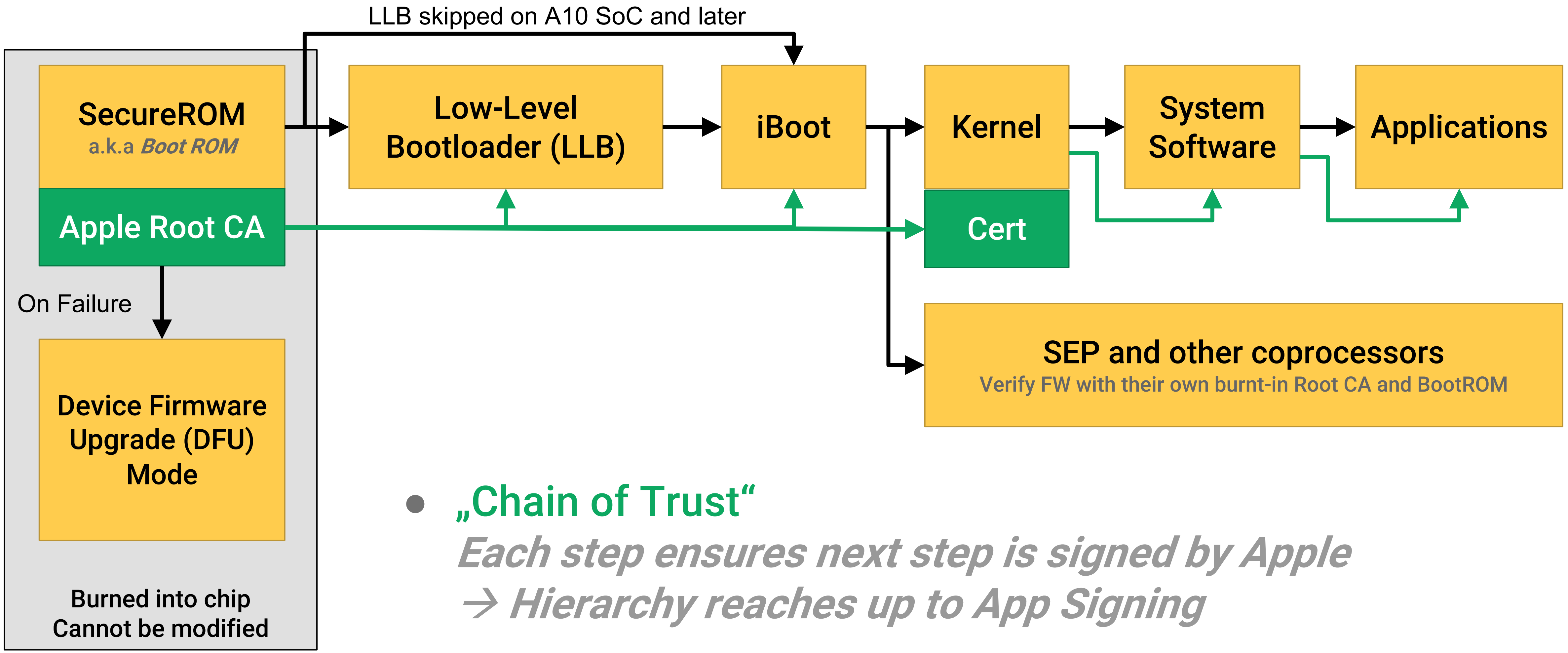
- MAC extends Discretionary Access Control (DAC = file permissions)
- Various hooks scattered throughout syscall implementations in kernel
- Hooks call out to Policy Modules for checking if operation permitted
- Foundation for central iOS security features
  - Code Signing Policy Module: AppleMobileFileIntegrity.kext
  - Sandbox Policy Module: Sandbox.kext



Source: J. Levin: "*OS Internals"

Low-Level System Security

Secure Boot Chain („iBoot Chain“)



LLB skipped on A10 SoC and later
SecureROM
a.k.a Boot ROM
Apple Root CA
On Failure
Device Firmware Upgrade (DFU) Mode
Burned into chip
Cannot be modified
Low-Level Bootloader (LLB)
iBoot
Kernel
Cert
System Software
Applications
SEP and other coprocessors
Verify FW with their own burnt-in Root CA and BootROM

- „Chain of Trust“
  Each step ensures next step is signed by Apple
  → Hierarchy reaches up to App Signing

- From LLB/iBoot to Applications → can be updated

Sources: J. Levin: "*OS Internals", apple.com

Secure Boot Chain

Starting with simple boot loader…

- Burnt into hardware: *„Hardware Root of Trust“*
- Prevent tampering of lowest software levels

- Similar (separate) boot process for coprocessors
  - Baseband processor (cellular access)
  - Secure Enclave coprocessor
  - Started by iBoot

→ Error if load / verify next step failed

- Enter DFU (Recovery mode)
- Connect to iTunes and restore factory defaults

iOS Downgrades?

Apple prevents them using „*System Software Authorization*“!

- Signatures alone would enable replay attacks

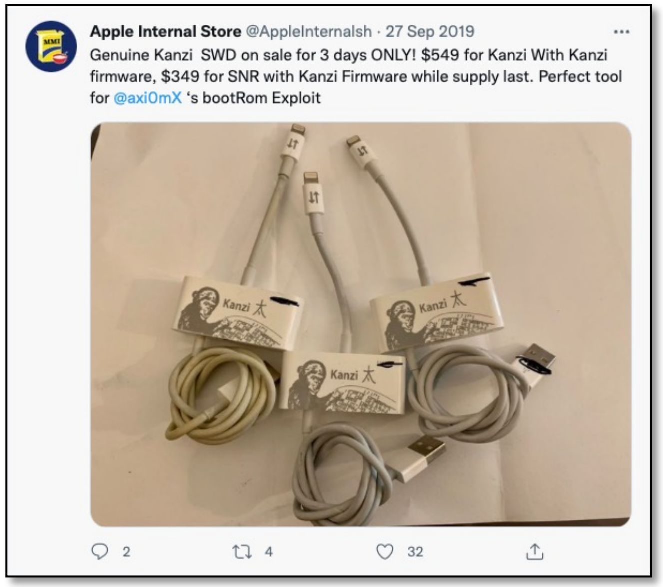
- Online process
  - Device generates nonce *(„anti-replay value“)*
  - Sends Exclusive Chip ID (ECID) + nonce to Apple server
  - Apple generates signature for (OS image + ECID + nonce)
  - Device checks if signature ok, nonce / ECID matches
  - If fine: Install software

- Prevent installation of old OS images by revoking old signatures

Chip Fuse Mode (CPFM)

- A write-only register controls hardware debuggability
  - Burned in factory, enforced by SecureROM
- Two flags: (Production/Development), (Secure/Insecure)
  - Controls CPU and SEP strictness
- Apple-internal development devices:
  - Development: Allow JTAG debug access for CPU
  - Insecure: SEP JTAG + Booting unverified OS image

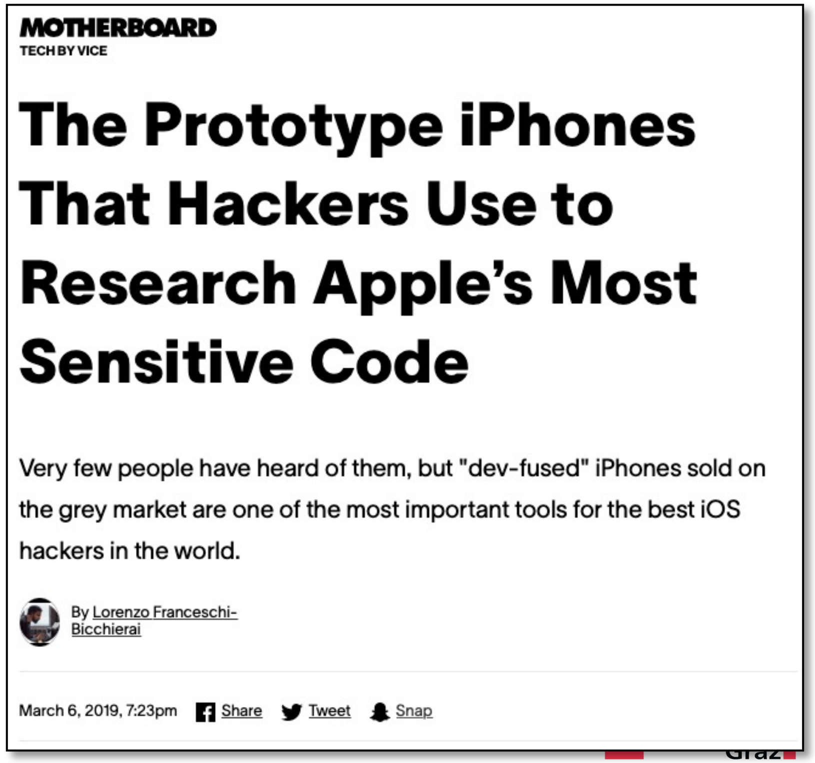
- Leaked "Dev-Fused" iPhones used by hackers
  - Available from gray market
- 2020: Apple Security Research Device Program
  - Only for high-profile security researchers


Apple Internal Store @AppleInternalsh · 27 Sep 2019

Genuine Kanzi SWD on sale for 3 days ONLY! $549 for Kanzi With Kanzi firmware, $349 for SNR with Kanzi Firmware while supply last. Perfect tool for @axi0mX 's bootRom Exploit

2 4 32

Source: twitter.com


MOTHERBOARD
TECH BY VICE

The Prototype iPhones That Hackers Use to Research Apple's Most Sensitive Code

Very few people have heard of them, but "dev-fused" iPhones sold on the grey market are one of the most important tools for the best iOS hackers in the world.

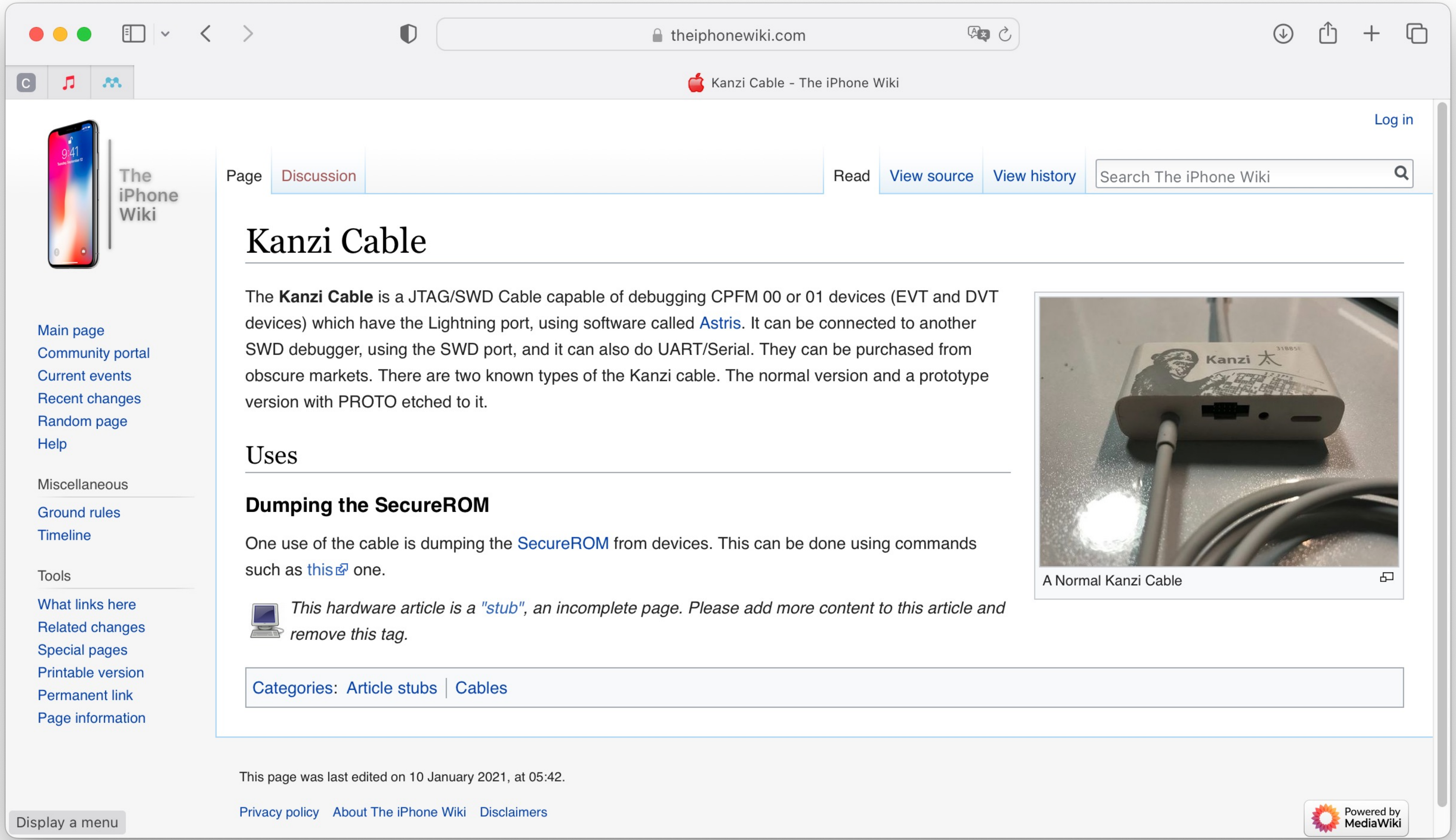
By Lorenzo Franceschi-Bicchierai

March 6, 2019, 7:23pm Share Tweet Snap

Sources: J. Levin: "*OS Internals", vice.com

Source: vice.com

Kanzi Cable



theiphonewiki.com

Kanzi Cable - The iPhone Wiki

Log in

Page | Discussion | Read | View source | View history | Search The iPhone Wiki

The iPhone Wiki

Main page
Community portal
Current events
Recent changes
Random page
Help

Miscellaneous

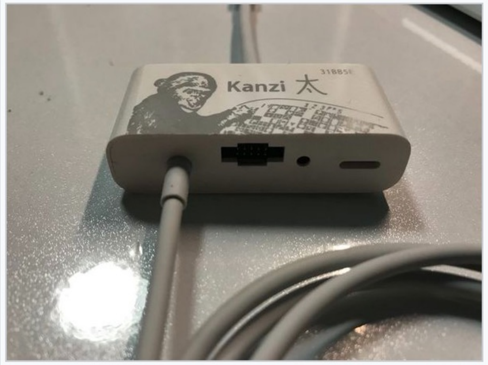
Ground rules
Timeline

Tools

What links here
Related changes
Special pages
Printable version
Permanent link
Page information

Kanzi Cable

The **Kanzi Cable** is a JTAG/SWD Cable capable of debugging CPFM 00 or 01 devices (EVT and DVT devices) which have the Lightning port, using software called Astris. It can be connected to another SWD debugger, using the SWD port, and it can also do UART/Serial. They can be purchased from obscure markets. There are two known types of the Kanzi cable. The normal version and a prototype version with PROTO etched to it.



A Normal Kanzi Cable

Uses

Dumping the SecureROM

One use of the cable is dumping the SecureROM from devices. This can be done using commands such as this one.

This hardware article is a "stub", an incomplete page. Please add more content to this article and remove this tag.

Categories: Article stubs | Cables

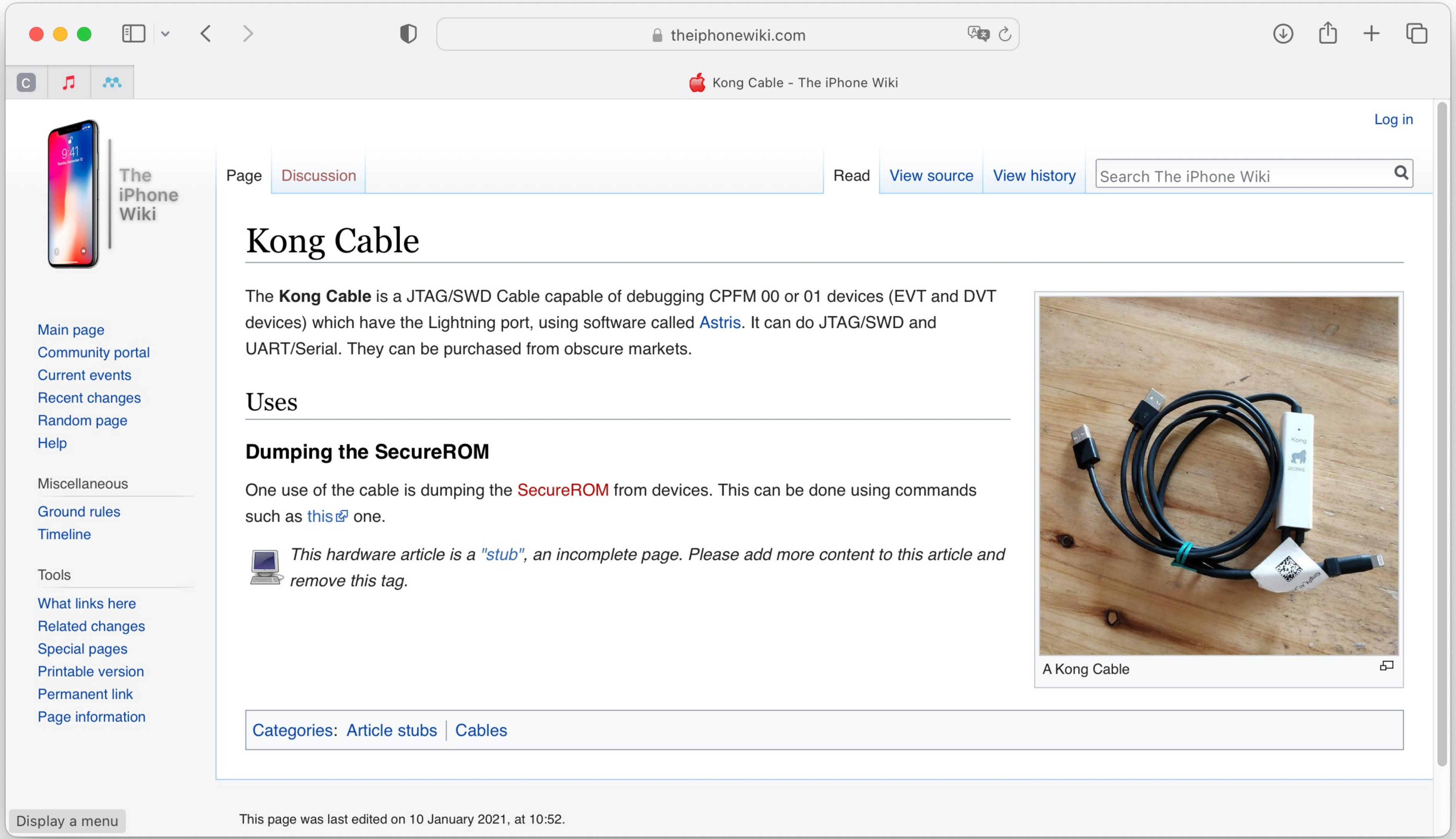
This page was last edited on 10 January 2021, at 05:42.

Privacy policy About The iPhone Wiki Disclaimers

Powered by MediaWiki

Display a menu

Kong Cable



theiphonewiki.com

Kong Cable - The iPhone Wiki

Log in

Page | Discussion | Read | View source | View history | Search The iPhone Wiki

Main page
Community portal
Current events
Recent changes
Random page
Help

Miscellaneous
Ground rules
Timeline

Tools
What links here
Related changes
Special pages
Printable version
Permanent link
Page information

Kong Cable

The **Kong Cable** is a JTAG/SWD Cable capable of debugging CPFM 00 or 01 devices (EVT and DVT devices) which have the Lightning port, using software called Astris. It can do JTAG/SWD and UART/Serial. They can be purchased from obscure markets.

A Kong Cable

Uses

Dumping the SecureROM

One use of the cable is dumping the SecureROM from devices. This can be done using commands such as this one.

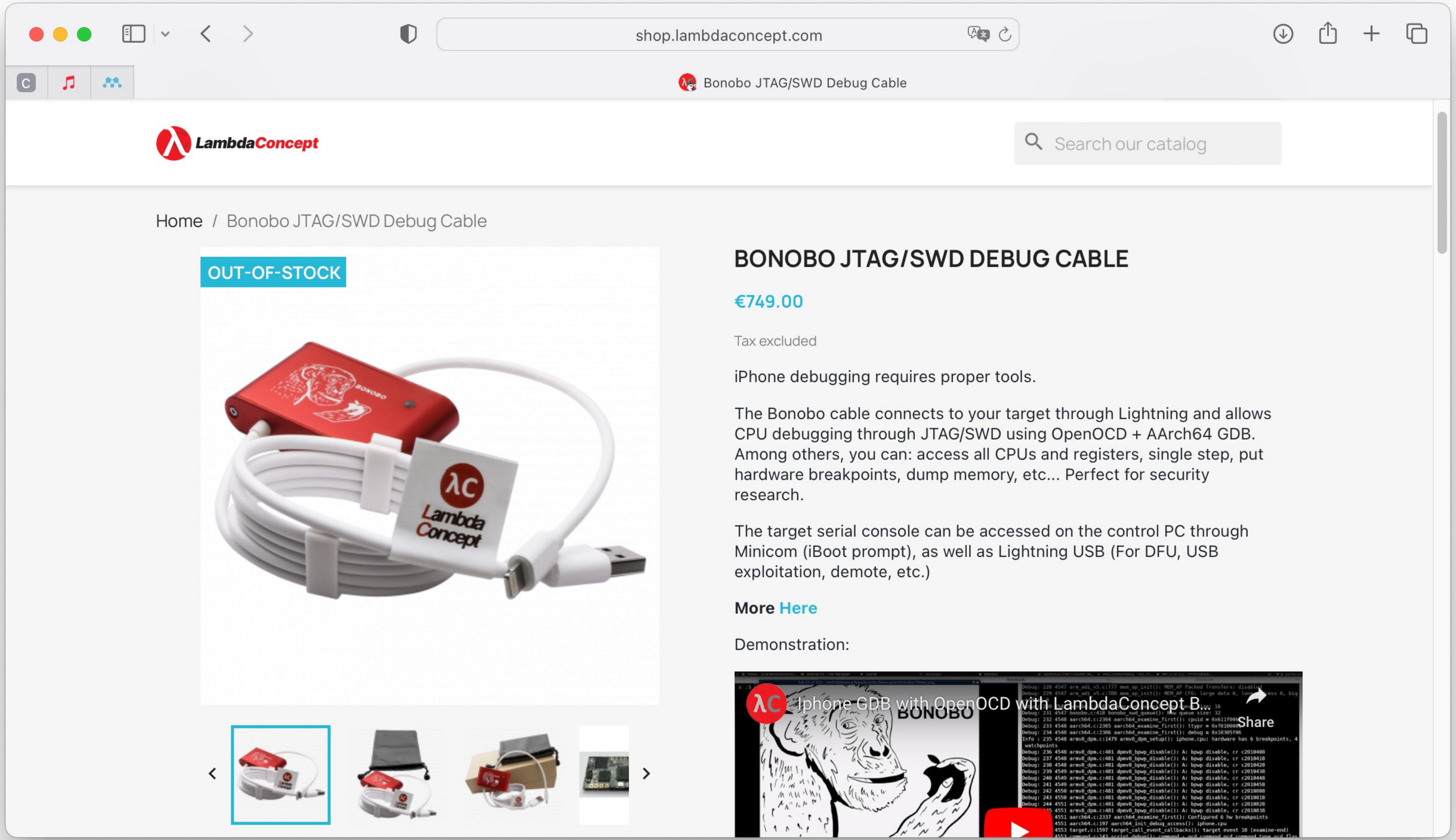
This hardware article is a "stub", an incomplete page. Please add more content to this article and remove this tag.

Categories: Article stubs | Cables

This page was last edited on 10 January 2021, at 10:52.

Display a menu

Bonobo Cable


shop.lambdaconcept.com
Bonobo JTAG/SWD Debug Cable
LambdaConcept
Search our catalog
OUT-OF-STOCK
Iphone GDB with OpenOCD with LambdaConcept B...
Share

Home / Bonobo JTAG/SWD Debug Cable

BONOBO JTAG/SWD DEBUG CABLE

€749.00

Tax excluded

iPhone debugging requires proper tools.

The Bonobo cable connects to your target through Lightning and allows CPU debugging through JTAG/SWD using OpenOCD + AArch64 GDB. Among others, you can: access all CPUs and registers, single step, put hardware breakpoints, dump memory, etc... Perfect for security research.

The target serial console can be accessed on the control PC through Minicom (iBoot prompt), as well as Lightning USB (For DFU, USB exploitation, demote, etc.)

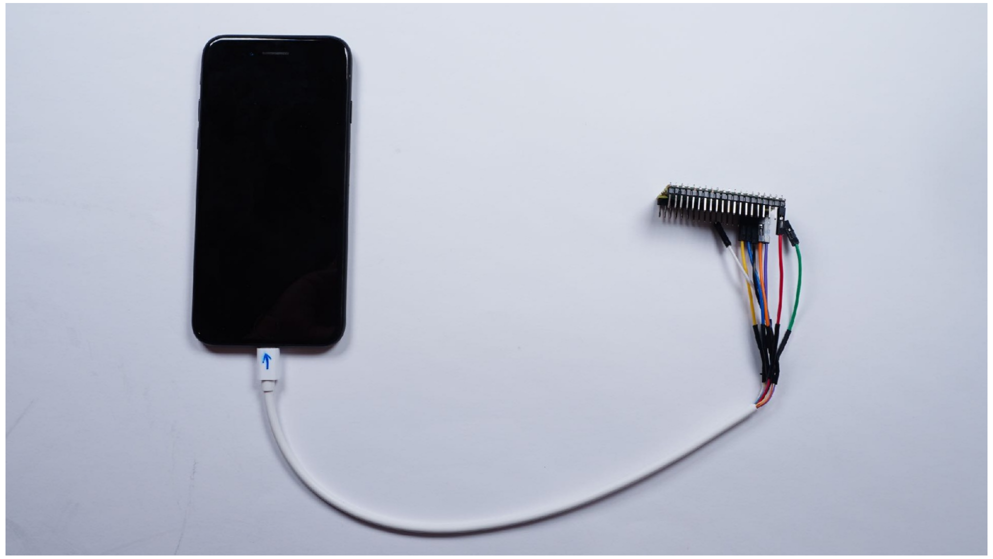
More Here

Demonstration:

Tamarin Cable

- The Checkm8 exploit found in 2019 allows demoting production devices
  - iPhone 4S through X
  - Only AP fuse may be manipulated

- There is an open-source debug cable!
  - Based on Raspberry Pi Pico
  - Same functionality as Apple-internal tools



Sources: DEFCON 30: The Hitchhiker's Guide to iPhone Lightning and JTAG Hacking

Firmware Encryption

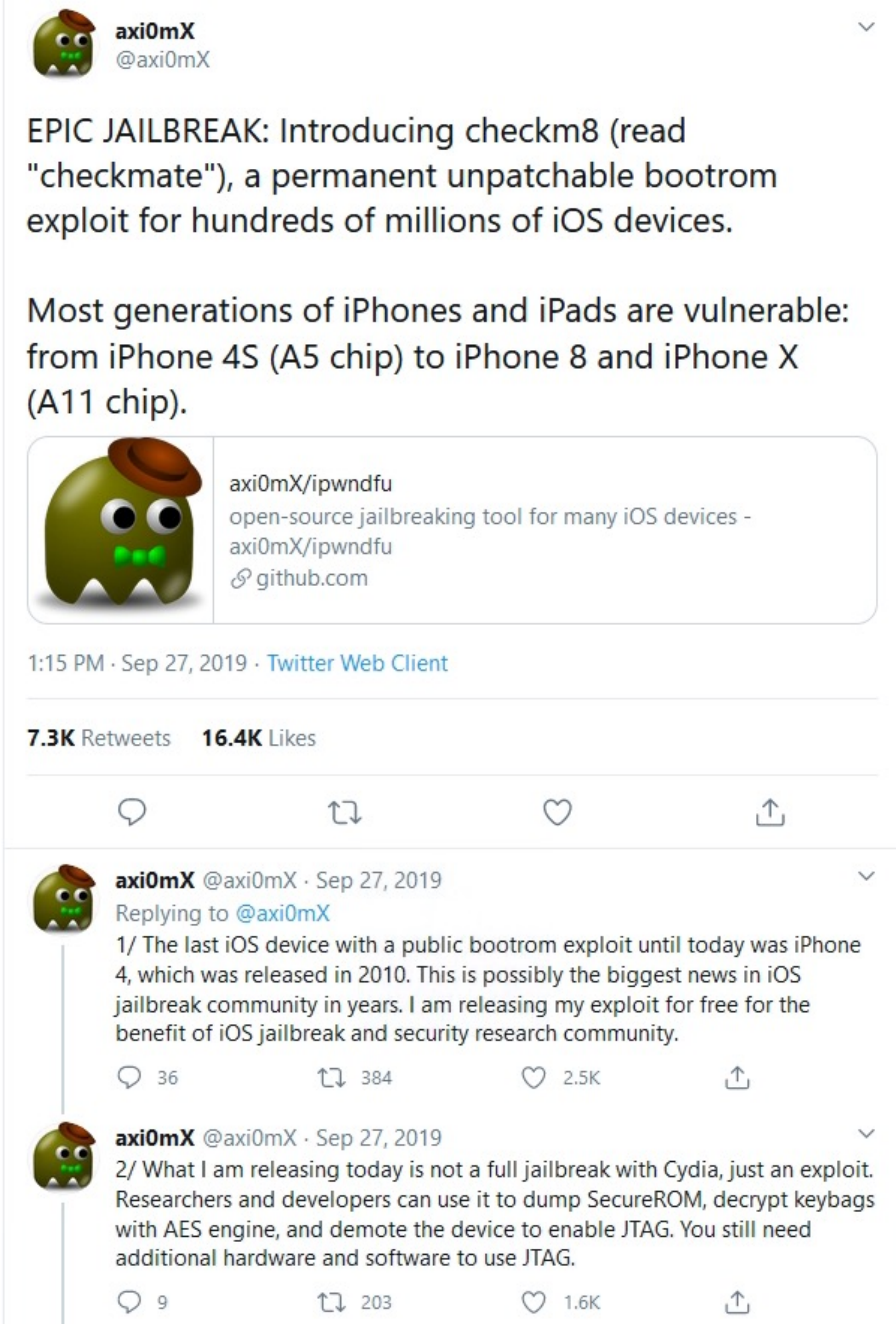
- Firmware is stored on the device in encrypted form
  - Prevent analysis and reverse-engineering
  - Decrypted during boot, using embedded key and IV
    - Wrapped with GID key only available to SEP

→ Access to SEP decryption needed for accessing raw firmware
  - SecureROM exploit
  - SEP exploit
  - Dev-Fused device

Jailbreak

- Boot chain is an interesting attack target
  - Cut the „Chain of Trust“
  - Modify subsequently loaded components
  - E.g. Remove code signature checks from kernel
- Exploits in LLB, iBoot or kernel
  - Software patchfix possible!
- SecureROM exploits
  - Can not be updated → deploy new chips
  - Checkm8 exploit published in 2019

Source: twitter.com


axi0mX
@axi0mX

EPIC JAILBREAK: Introducing checkm8 (read "checkmate"), a permanent unpatchable bootrom exploit for hundreds of millions of iOS devices.

Most generations of iPhones and iPads are vulnerable: from iPhone 4S (A5 chip) to iPhone 8 and iPhone X (A11 chip).

axi0mX/ipwndfu
open-source jailbreaking tool for many iOS devices - axi0mX/ipwndfu
github.com

1:15 PM · Sep 27, 2019 · Twitter Web Client

7.3K Retweets 16.4K Likes

axi0mX @axi0mX · Sep 27, 2019
Replying to @axi0mX
1/ The last iOS device with a public bootrom exploit until today was iPhone 4, which was released in 2010. This is possibly the biggest news in iOS jailbreak community in years. I am releasing my exploit for free for the benefit of iOS jailbreak and security research community.
36 384 2.5K

axi0mX @axi0mX · Sep 27, 2019
2/ What I am releasing today is not a full jailbreak with Cydia, just an exploit. Researchers and developers can use it to dump SecureROM, decrypt keybags with AES engine, and demote the device to enable JTAG. You still need additional hardware and software to use JTAG.
9 203 1.6K

Secure Enclave

Goals?

- Store and manage sensitive user data
  - Data protection keys
  - Biometric information (FaceID, TouchID)

- Separate from main Application Processor (AP ≈ CPU)
  - Even privileged iOS exploits can not access key material

- Enforce strict security policies
  - Prevent brute-force attacks
  - Prevent offline attacks

Secure Enclave Processor (SEP)

Implementation

- Dedicated separate processor core within SoC running its own sepOS
- Transparently encrypted access to external RAM
  - Replay-protected authenticated encryption in hardware!
- AP has no access to SEP memory
- Mailbox interface for exposing services to AP

- Core primitives:
  - Embedded GID and UID keys
  - AES engine hardened against multiple side channel attacks
  - Public Key Accelerator for asymmetric cryptography
  - True Random Number Generator

Source: Mandt et al.: "Demystifying the Secure Enclave Processor", apple.com

TouchID

- Unlock device without having to enter passcode
  - Passcode still required for first unlock after boot
    - And 48 hours after last unlock
- Sensor is securely paired to SEP in factory
  - Establishes a protected communication channel
  - Sensor sends “hash” of fingerprint image to SEP
- Matching fingerprint unlocks access to user data
  - Implemented in SEP

Source: Ivan Krstic: “Behind the Scenes of iOS Security”

TouchID (similar procedure also for FaceID)

How does it work?

- Interaction between two programs on SEP
  - SKS: Secure Key Service
  - SBIO: Secure Biometrics

Encrypts user data on device
Details in a few minutes

1. On Code Unlock: SKS derives Master Key (MK) from passcode and UID key
   1. SKS encrypts MK with Random Secret (RS) ➔ Encrypted MK (EMK)
   2. RS sent to SBIO, MK purged from SKS storage
2. On Touch Unlock:
   1. SBIO obtains fingerprint hash from sensor and compares it to registered values
   2. If match: Send RS to SKS
   3. SKS can now decrypt the wrapped MK from the EMK again

Baseband Processor

- A separate chip sitting on the PCB
  - Supplied by Intel or Qualcomm
  - Communicates with AP via UART/I2C/USB/SDIO
  - Originally used AT commands, now more sophisticated binary protocols

- Manages the cellular communication
  - Internet traffic, calls and messages
  - Responsible for carrier lock

- Early versions could be exploited from AP
  - No exploits from network side are known

Encryption Systems

iOS Data Encryption Systems

- File system encryption
  - Alias: „Full disk encryption“, „Storage encryption“
  - Introduced with iOS 3 and iPhone 3GS
  - Keys were not dependent on passcode, so protection was very limited
- Data Protection
  - Introduced with iOS 4 (2010)
  - Encrypts individual files
  - Improved in newer version (new Protection classes, KeyChain features)

Data Protection

- Upon file creation, a fresh file encryption key is generated 256-bit AES
- The key is wrapped with 1 of 4 class keys of varying protection
  - Wrapped key and class stored in file metadata
- Class keys are wrapped with SEP UID key and/or user passcode

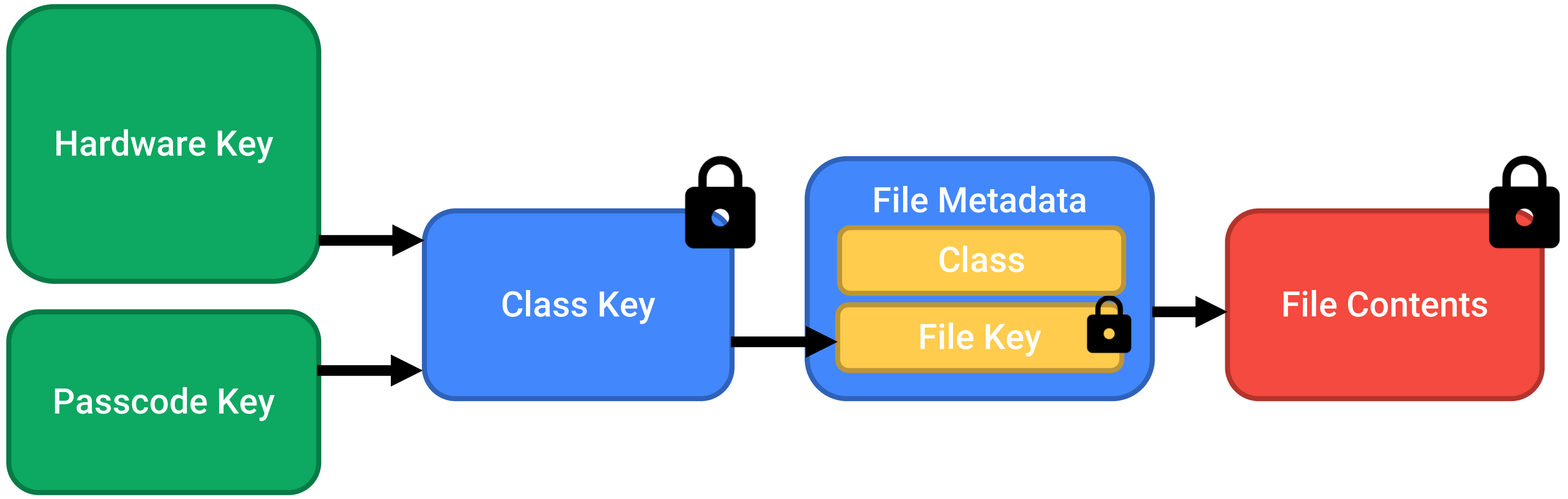
Benefits

- Passcode strength alone depends on user choice
  - Brute-force attacks (offline = on desoldered NAND chip)
- Combined with UID key that never leaves SEP
  - Brute-force attacks have to be carried out on-device!
  - Enforce security policy in SEP
    - Max attempts, delays, …

Data Protection

***Change** file class? Just rewrap file key!*

***Change** passcode? Just rewrap class key!*



Picture: [Google](#) / [Apache 2.0](#)

Hint: To keep it simple… read from right to left ;)

Source: [iOS Security Guide (Q2 / 2019)](#)

Data Protection Classes (a.k.a. „User Keybag Classes")

Class	Class Key Wrapping	Class Name
A	**Passcode + UID**	`NSFileProtectionComplete`
Can only be accessed while device is unlocked		
B	**Special Case**	`NSFileProtectionCompleteUnlessOpen`
Asymmetric Key Pair: Public key always available, Private key only while unlocked (*)		
C	**Passcode + UID**	`NSFileProtectionCompleteUntilFirstUserAuthentication`
Only accessible after user authenticated once (since last boot)		
D	**UID Only**	`NSFileProtectionNone`
Always accessible		

(*) Exception for file descriptors acquired already while device unlocked

Sources: J. Levin: "*OS Internals", Ivan Krstic (Apple): "Behind the Scenes of iOS Security"

IAIK TU Graz

Data Protection: Implementation

What happens behind the scenes?

- Passcode-dependant Class keys stored in an encrypted file in device storage
  - „System Key Bag“ file
- Upon boot:
  - SEP loads and decrypts Class D key from Flash (using UID key)
  - System Key Bag sent to SEP, where the class B public key is unwrapped
  - Unwrapped Class Keys are stored in ***SKS Key Ring*** in SEP
- Upon unlock:
  - Remaining class keys unwrapped using Master Key (derived from passcode and UID key)
- Upon lock:
  - Class A and Class B private key removed from ***SKS Key Ring***

Data Protection: Storage Controller

- Hardware assists in hiding class keys from AP

- At boot: SEP generates ephemeral key and sends it to the Storage Controller

- File access:
  - Kernel fetches wrapped file key from metadata and sends it to SEP
  - SEP unwraps key using corresponding class key
  - Rewraps it using ephemeral key and returns result to kernel
  - Kernel sends rewrapped key to Storage Controller to retrieve Flash content

Kernel never gets access to any secret of long-term value!
Ephemerally wrapped key is only valid until reboot

Data Protection – Where is the problem?

- Every new file gets assigned a protection class by an app (!)
  - Handled by the developer!
  - User cannot know which apps encrypt their data and which do not

- Consider the scenario
  - Getting email with PDF attachment (mail app uses data protection)
  - Opening the mail in a PDF reader (not using data protection)

How to find out? → Application Analysis

- Dynamic approach: Monitor live file access using jailbroken device
- Static approach: Look for file API calls + parameters in binary dump

Data Protection – In Practice

```
let fileManager = FileManager.default
fileManager.createDirectory(atPath: folder.path, withIntermediateDirectories: true,
attributes: [FileAttributeKey.protectionKey: FileProtectionType.complete])
…
fileManager.createFile(atPath: databaseKeyURL.path, contents: nil,
attributes: [FileAttributeKey.protectionKey: FileProtectionType.complete])
```

```
let data = Data(count: count)
data.write(to: fullCachePath, options: [.atomic, .completeFileProtection])
```

Since iOS 7 default protection class: *„Protected until first user authentication“*

Effaceable Storage

A section of the Flash storage that can be completely erased

- Note that the process displayed so far is still simplified!
- Complete file system is also encrypted using key stored in effaceable storage
  - "Media Key"
  - Similar to legacy Full Disk Encryption (FDE)
  - Protects file metadata

- System Key Bag file additionally encrypted with key from effaceable storage
  - Yet another key

Source: Ivan Krstic (Apple): "Behind the Scenes of iOS Security"

File System Encryption – Remote Wipe

From the Apple Platform Security Guide (Q1 / 2021):

```
The metadata of all files in the data volume file system are encrypted with a random volume key, which is created when the operating system is first installed or when the device is wiped by a user ... When stored, the encrypted file system key is additionally wrapped by an “effaceable key” ... This key doesn’t provide additional confidentiality of data. Instead, it’s designed to be quickly erased on demand (by the user with the “Erase All Content and Settings” option, or by a user or administrator issuing a remote wipe command from a mobile device management (MDM) solution, Microsoft Exchange ActiveSync, or iCloud). Erasing the key in this manner renders all files cryptographically inaccessible.
```

→ Erase the file system key to avoid further access to any file!

→ Remote Wipe does not actually *delete* the file…

Key Management
& Passcodes

iOS KeyChain

What for?

Mobile OS needs to handle passwords, login tokens, PINs, certificates, etc

What does it look like?

- 1 SQLite database stored on file system
- Entries can be shared between apps from same developer *(app group)*
- Access from apps using ordinary API
- Protection classes similar to those for files

Side note:

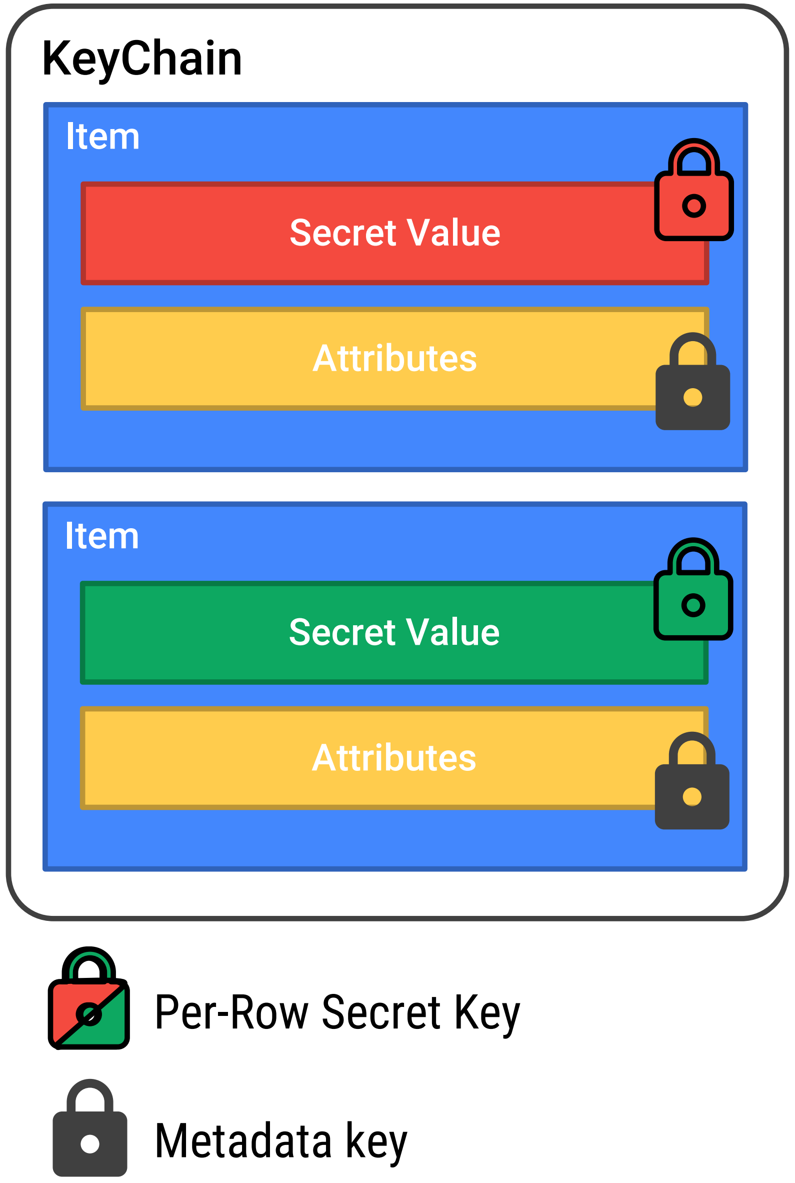
Uninstalling an app does not remove KeyChain data!

iOS KeyChain Items

Every entry has…

- Access control list (ACL)
- Key wrapped with protection class key,
- Protection class affiliation
- Attributes describing the entry
- Version number

→ Every aspect is encrypted (AES-256 GCM)!
E.g. also usernames (= attribute), not only passwords!



Source: Apple Platform Security Guide (Q2 / 2021)

iOS KeyChain Access Control

Every entry has an Access Control List (ACL) specifying

- **Accessibility**
  - When is item readable?
  - Similar to protection class for Data Protection

- **Authentication**
  - What authentication is needed for access?
  - Confirm user presence through TouchID, FaceID, passcode
  - Ensure TouchID or FaceID enrollment unchanged since entry stored

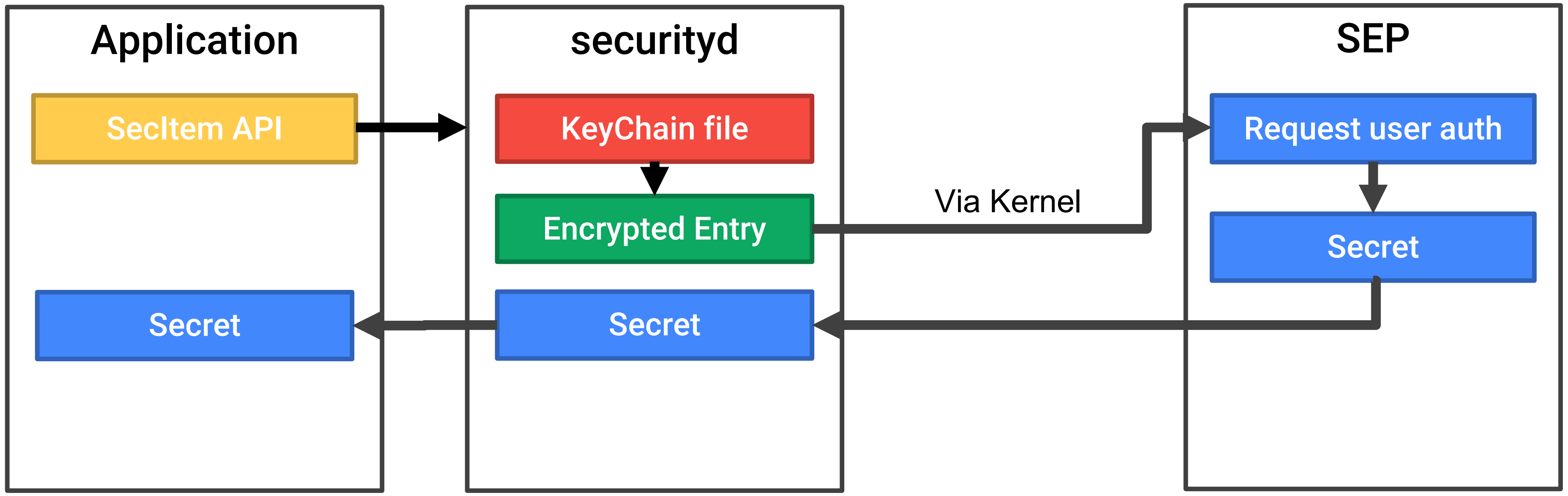
Source: Apple Platform Security Guide (Q2 / 2021)

KeyChain Protection Classes

Secret Availability	Keychain Data Protection
When unlocked	`kSecAttrAccessibleWhenUnlocked`
Protected by user passcode and SEP UID key	
After first unlock	`kSecAttrAccessibleAfterFirstUnlock`
Suitable e.g. for apps that refresh data even while device is locked	
Always	`kSecAttrAccessibleAlways`
Only protected by SEP UID key	
Passcode-enabled	`kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly`
Same as *When unlocked*, except unavailable if no passcode configured	

Source: [Apple Platform Security Guide (Q2 / 2021)]()

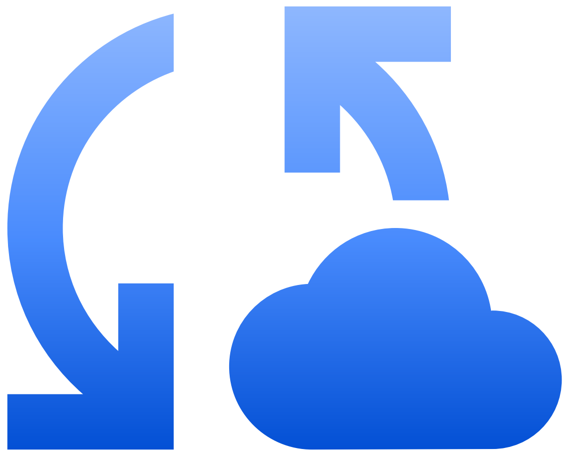
iOS KeyChain: App Access Workflow



Backups and Sync

Local or Remote Backup

- Local iTunes backups (WiFi or USB cable)
  - Encrypted (AES-256 CBC)
  - Plain
- iCloud backups
  - Sync data to iCloud server
  - Access from other Apple devices



Keybags on iOS

Keys for file and KeyChain Protection classes are managed in „Keybags“

- System Keybag (= User Keybag)
  - Contains wrapped keys for passcode-dependent protection classes
  - File encrypted with class D key (protected by SEP UID)
  - See Data Protection slides

- Backup Keybag
  - Transferred (exported) system keybag in backups
    - Non-migratory KeyChain entries remain wrapped with SEP UID key
  - Backup encrypted: Key derived from user-specified iTunes password
  - Backup plain: KeyChain still protected by UID-derived key
    → To migrate backup to new device: encrypt the backup!

Sources: P. Teufl et al: "iOS Encryption Systems", Apple Platform Security Guide (Q1 / 2021)

Keybags on iOS Cont'd

- Escrow Keybag
  - Allows iTunes to backup and sync without user passcode!
    - Upon connection, iOS device creates escrow keybag and wraps it with fresh key
    - Key stays on device classified as *Protect Until First User Authentication*
    - Encrypted escrow keybag stored on computer running iTunes
    - iTunes communicates with device to obtain key when needed

Sources: Apple Platform Security Guide (Q1 / 2021)

iCloud Keychain

Allows syncing Keychain entries between multiple Apple devices

- Every device generates an iCloud Keychain synchronization key pair
- User approves new device from a device already in this sync cycle
- Apple servers just relay encrypted messages between devices

What if the user loses all their devices?

Source: Ivan Krstic (Apple): "Behind the Scenes of iOS Security"

iCloud Keychain Backup

Only concerns iCloud Keychain backups!
No information available on encryption of full iCloud backups!

- Encrypted using backup („escrow“) key
  - Randomly generated key
  - Wrapped using a key that is derived from iCSC
    → iCSC = iCloud Security Code = User passcode
    - Unknown to Apple!
- iCloud Keychain backup encrypted with escrow key
  - Encrypted backup + wrapped escrow key sent to Apple
  - In case of device loss or new device
    → User can recover secrets with iCloud password and iCSC
- Main problem in practice: iCloud account security

Source: Ivan Krstic (Apple): "Behind the Scenes of iOS Security"

iCloud Keychain Backup

„Who watches the watchers?“ :-)

iCloud backend could brute-force iCSC to access escrow key!

Apple‘s solution: Cloud Key Vault

- Enforce policy over escrow key
  - Want hard limit on escrow recovery attempts under adversarial cloud
  - What if escrow key unwrapping only happens in Hardware Security Modules?

- Cloud Key Vault = HSM running custom secure code
  - Key vault runs own certificate authority
    - Private key never leaves HSM
  - Each iOS device hardcodes key vault CA cert
  - Escrow key wrapped with Cloud Key Vault certificate, unwrapping only in HSM

Source: Ivan Krstic (Apple): "Behind the Scenes of iOS Security"

iOS Platform Security

- Low-level System Security
- Updates
- Encryption Systems
- Key Management & Passcodes
- Backup

Questions?

Outlook

- 17.05.2024
  - iOS Application Security
- 24.05.2024
  - Mobile Hardware Security