Jailbreaking iOS: From past to present

By: tihmstar
Topics:

• What is jailbreaking
• Jailbreak entry point overview / progression
• Terminology: (Tethered/Untethered/Semi-Tethered/Semi-Untethered)
• Hardware mitigations KPP/KTRR/PAC
• Goal of jailbreaking (technical) / kernelpatches
• Future of jailbreaking
Who am i

• tihmstar

• Got my first iPod touch with iOS 5.1
  • Played with jailbreaks ever since

• Been here 2 years ago (iOS Downgrading - From past to present)
  • Kept hacking iOS since then
Projects i worked on

• Downgrading:
  • tsschecker - Gets APITickets for downgrading
  • futurerestore - First tool to downgrade 64bit devices
  • img4tool - Tool for working with firmware images (img4, im4m, im4p ...)

• Jailbreaking (8.4.1-10.3.3)
  • Phoenix, (untether)HomeDepot, jailbreak.me 4.0, etasonJB, h3lix, doubleH3lix (64bit), jelbreakTime (iPad, Apple Watch)
What is jailbreaking

• Gets control over device
  • Escape sandbox
  • Elevate to root/kernel
  • Disable codesigning

• Most popular: install tweaks!

• Do security analysis
Tweaks

• Modifications of built in userspace programs
  • SpringBoard
• Modify UI/functionality
Cydia / DPKG

- Install dpkg/apt (Debian package manager)
- Cydia is a GUI for dpkg (userfriendly)
- (de)centralized package installer
Ages of Jailbreasing

Ages of jailbreaking (IMO):

iOS 1-4: Golden Age (BootROM)
iOS 5-9: Industrial Age (rise of userland)
iOS 10-*: Post-Apocalyptic (KTRR)
First iPhoneOS jailbreaks

- Bufferoverflow in iPhone's libTiff (image parsing library)
  - Exploited through Safari
  - Used as entrypoint to get code execution
- First time a non-Apple software was run on an iPhone
  - Popular applications:
    - Installer, AppTapp .... -> Used to install games/apps
Golden Age

- Attention shifted to BootROM
- DFU (Device-Firmware-Upgrade) Mode (ROM)
- Most famous BootROM exploit: limera1n by geohot
  - Bug in hardware, unpatchable with software
  - Used to jailbreak devices up to iPhone4 (patched in 4s)
Tethered Jailbreak

- limera1n exploits a bug in DFU mode
  - Loading unsigned software only possible through USB
- When rebooting device, a computer is required to re-exploit and load a patched kernel
  - Thus the jailbreak is *tethered* to a computer
- Historically: Tethered jailbroken phones do not boot without re-exploiting
  - Kernel on filesystem patched for performance reasons (tethered boot)
  - Broken chain of trust for bootloader/kernel
Semi-tethered Jailbreak

• Idea: Don't break chain of trust for tethered jailbreak
  • Appeared some time around iOS 5
  • Do not modify kernel on filesystem
  • Can boot into non-jailbroken mode without PC
    • (if no system components were permanently modified by jailbreak)
Ages of Jailbreaking

iOS 1-4: Golden Age (BootROM)
iOS 5-9: Industrial Age (rise of userland)
iOS 10-*: Post-Apocalyptic (KTRR)
Industrial Age

- Release of iPhone 4s and iOS 5
  - Fixed BootROM bug (killed limera1n)
  - Introduction of APTickets (added nonces to bootloader signatures)
  - Throwback for downgrading (killed classic SHSH replay)
Encrypted Bootfiles

- iPhone firmware files are encrypted
- KeyEncryptionKey is fused into the devices
  - Impossible(?) to get through hardware attacks
- All boot files are decrypted on boot by previous bootloader
Industrial Age

- Hardware feature in iPhone4s disabled AES engine after kernel booted
  - Prior to this kernel level code exec was enough
  - iBoot level code execution necessary for decrypting bootloaders/kernel
  - Decrypting bootloaders is a struggle from now on!
Industrial Age

- Attention shifted to userland - jailbreaks *had to be* untethered
- Untethered = device still jailbroken after reboot
  - Achieved through re-exploitation at some point in boot process
- Jailbreaks chained many bugs (sometimes 6 or more!) to get
  - Initial code execution, kernel code execution, persistence
Free Developer Accounts

• Introduced with iOS 9

• Everybody can get a signing certificate valid for 7 days for free
  • Prior only paid dev accs (~100$ per year) could sign apps
  • After 7 days you can get another free certificate
Semi-Untethered

- Initial code execution not an issue anymore
- Jailbreak focus shifted to powerful kernel bugs reachable from sandbox
- Distributed as IPA (installable App) people need to sign themselves
- Semi-Untethered = reboots to non-jailbroken mode, but can get to jailbroken mode by running an app
Apple's game

- iOS 5 - introduction of ASLR (KASLR in iOS 6)
- iPhone5s - introduction of 64bit ARM CPU
- iOS 9 (64bit) - introduction of Kernel Patch Protection (KPP)
- iPhone 7 - Kernel Text Readonly Region (KTRR)
- iOS 11 - removal of 32bit libraries
- iPhone Xs - Pointer Authentication Codes (PAC)
Kernel Patch Protection

• KPP usually refers to what Apple calls *watchtower*

• *Watches* over the kernel and panics when modifications are detected

• Prevents kernel from being patched (does it?)
Watchtower

- Runs in EL3 (ARM exception level 3)
- Exceptions levels are privilege separations (3 highest, 0 lowest)
- Trigger exception to call handler code in higher levels
- *Recurring events* (FPU usage) trigger watchtower inspection of kernel
EL3  Watchtower
EL1  Kernel
EL0  Applications
event occurs from time to time
event triggers watchtower

EL3 Watchtower

EL1 Kernel

EL0 Applications
watchtower scans kernel

EL3
Watchtower

EL1
Kernel

EL0
Applications
watchtower scans kernel

EL3
Watchtower

EL1
Kernel

EL0
Applications
watchtower scans kernel

EL3 Watchtower

scan

EL1 Kernel

EL0 Applications
watchtower scans kernel

- EL3 Watchtower
- EL1 Kernel
- EL0 Applications
executions is transitioned back
With modified kernel

EL3
Watchtower

EL1
Kernel

EL0
Applications
With modified kernel

EL3
Watchtower

EL1
Kernel

EL0
Applications

scan

panic!
Watchtower

- Idea: Kernel is \textit{forced} to call Watchtower
  - Because FPU is blocked otherwise
- Problem: Kernel is in control \textit{before} it calls Watchtower
- Fully defeated by @qwertyoruiop in yalu102
KPP bypass by @qwertyoruiop

- Copy kernel in memory
- Modify the copied kernel
- Modify page tables to use patched kernel
- Switch to unmodified copy before calling Watchtower
- Switch back to patched after kernel was checked by Watchtower
EL3
Watchtower

EL1
Kernel

EL0
Applications

running

XNU
BSD
IOKit
Mach
Create a copy of the kernel in memory

- EL3: Watchtower
- EL1: Kernel
- EL1: Kernel copy
- EL0: Applications
Patch the copied kernel

EL3
Watchtower

EL1
Kernel

EL1
Kernel copy

EL0
Applications
Switch to patched kernel

- EL3 Watchtower
- EL1 Kernel
- EL1 Kernel copy
- EL0 Applications
event occurs at some point

EL3 Watchtower

EL1 Kernel

EL1 Kernel copy

EL0 Applications
Switch kernel to unmodified copy (pagetables)
Watchtower scans unmodified kernel

- EL3: Watchtower
- EL1: Kernel
- EL1: Kernel copy
- EL0: Applications
Executions is returned to patched kernel

return
KPP bypass by @qwertyoruiop

- Problem: Time of Check ≠ Time of Use (TOCTOU)
- Works on iPhone 5s, 6, 6s
- Not really patchable
- iPhone 7 (and higher) use KTRR :(
Kernel Text Readonly Region (KTRR)

- Functionality described by Siguza (https://siguza.github.io/KTRR/)
- Extra memory controller (AMCC) traps all writes to Readonly-Region (RoRgn)
- Extra CPU registers mark executable range (KTRR)
  - Subsection of RoRgn
  - Hardware enforcement at boot time for
    - Readonly memory region
    - Executable memory region
KTRR

CPU

Memory
- RoRgn set at boot
KTRR

- Enforced by hardware memory controller

CPU

not-writeable

writeable

RoRgn

Memory
KTRR

- CPU got KTRR registers
KTRR

- Mark *begin* and *end* of executable region
• Executable region is a subsection of readonly region
KTRRR

Can not execute code here

CPU

KTRR begin

KTRR end
KTRR

Can not be modified here
KTRR

- Has not been *truly* bypassed yet
- Jailbreaks work around kernel-patches
- KPPIless jailbreaks evolved
Jailbreak kernel patches

• General goals:
  • Disable codesigning
  • Disable sandbox
  • Make rootfs writeable
  • Make tweaks (substrate/substitue) work

• Techniques/patches vary across individual jailbreaks
• No general set of patches
Jailbreak patches (h3lix)

• i_can_has_debugger -- relax sandbox

• (iOS7+) Patch mount -- remount / as rw

• (iOS10.3+) Patch mount -- mount / without nosuid

• (iOS 9-10.3) Patch LWVM -- be able to write to /

• proc_enforce -- set to 0 (codesigning related)
Jailbreak patches (h3lix)

- `cs_enforcement_disable` -- disable codesigning (amfi)
- `amfi_memcmp_stub_return_0` -- ??? (amfi)
- Add `get-task-allow` to every process -- allows rwx mappings (for substrate tweaks)
- `(10.3+) label_update_execve` patch -- seems to completely nuke sandbox
  - Fixes "process-exec denied while updating label"
  - Breaks sandbox containers :
- Kill a bunch of check in `mac_policy_ops` -- sandbox related
Jailbreak patches (h3lix)

- Closely related open-source projects:
  - doubleH3lix (64bit version of h3lix)
    [github.com/tihmstar/doubleH3lix](https://github.com/tihmstar/doubleH3lix)
  - jelbrekTime (watchOS-iOS11-equivalent of h3lix)
    [github.com/tihmstar/jelbrekTime](https://github.com/tihmstar/jelbrekTime)
KPPlless Jailbreaks

• Idea: don't patch kernel code, patch data instead!

• Remount root filesystem?
  • Patch kernel data to make rootfs temporary not being seen as rootfs

• Disable codesigning / sandbox?
  • Trustcache injection
  • Patch process structure in kernel (jailbreakd)
  • Take over amfid in userspace (demoed by @bazad)
Future Jailbreaks

- Kernel code patches are not possible anymore
  - Not even required
- We still can
  - Patch kernel data
  - Don't go for kernel at all
Ages of Jailbreaking

Ages of jailbreaking (IMO):

iOS 1-4: Golden Age (BootROM)
iOS 5-9: Industrial Age (rise of userland)
iOS 10-*: Post-Apocalyptic (KTRR)

Post-ApoCalyptic (KTRR PAC)

- Pointer-Authentication-Codes introduced with iPhone Xs
- "Stronger version of stack protection" - Qualcomm
- Message-Authentication-Codes for pointers
- Protects data-in-memory in relation to context with a secret-key
  - Return value, stack pointer
  - Function pointer, vtable
## Post-ApoCalyptic (KTRR PAC)

<table>
<thead>
<tr>
<th>Function Prologue</th>
<th>No stack protection</th>
<th>With Pointer Authentication</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SUB sp, sp, #0x40</td>
<td>PACIASP</td>
</tr>
<tr>
<td></td>
<td>STP x29, x30, [sp,#0x30]</td>
<td>SUB sp, sp, #0x40</td>
</tr>
<tr>
<td></td>
<td>ADD x29, sp, #0x30</td>
<td>STP x29, x30, [sp,#0x30]</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>ADD x29, sp, #0x30</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>Function Epilogue</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>LDP x29,x30,[sp,#0x30]</td>
<td>LDP x29,x30,[sp,#0x30]</td>
</tr>
<tr>
<td></td>
<td>ADD sp,sp,#0x40</td>
<td>ADD sp,sp,#0x40</td>
</tr>
<tr>
<td></td>
<td>RET</td>
<td>AUTIASP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RET</td>
</tr>
</tbody>
</table>
Pointers in AArch64 (with authentication)

- PAC embedded in reserved pointer bits
  - e.g. 7 bits with 48-bit VA with tagging
  - leaving remaining bits intact
Pointers in AArch64 (with authentication)

- PAC embedded in reserved pointer bits
  - e.g. 15 bits with 48-bit VA without tagging
  - leaving remaining bits intact
Post-ApoCalyptic (KTRR PAC)

- Kills ROP like code reuse attacks
- You can not:
  - Modify return value
  - Swap two signed values on stack (unless SP is same for both)
Can we bypass it?
Maybe
Pointer Authentication Codes

- Each PAC is derived from:
  - A pointer value
  - A 64-bit context value
  - A 128-bit secret key

- PAC algorithm $P$ can be:
  - QARMA\(^1\)
  - IMPLEMENTATION DEFINED

- Instructions hide the algorithm details

\(^1\)https://eprint.iacr.org/2016/444.pdf
Attack Strategies for PAC

- Attack cryptographic primitive
- Attack implementation
Attack PAC Implementation

- Signing primitives
- Arbitrary context signing gadget
- Same context signing gadget
- Use unauthenticated code
- Signed pointer replacement attacks (same context)
- Other???
Attacking cryptographic primitive in PAC does not make much sense! (in my opinion)
QARMA

- Proposed by ARM (PAC can be qarma or custom)
- Tweakable Block Cipher (TBC)
  - input - tweak (PAC context) - output
- Practical crypto attacks on QARMA (if there will be any in future) will likely be irrelevant for PAC security
Crypto attacks on PAC

- We define PAC as: $f : \mathbb{F}_{2}^{128} \times \mathbb{F}_{2}^{128} \rightarrow \mathbb{F}_{2}^{15}$ or $f : \mathbb{F}_{2}^{96} \times \mathbb{F}_{2}^{128} \rightarrow \mathbb{F}_{2}^{15}$

- We define the attacker with following capabilities:
  
  - Observe *some* pointer/signature pairs (info leaks)
  
  - *Might* tweak context *slightly*
    
    - Shifting stack before signing (through more function calls)
Crypto attacks on PAC

• Point is: cryptographic attacker is super weak!

• Collision is a problem: \[ 2^{48}_{\text{pointer}} \times 2^{48}_{\text{context}} \times 2^{128}_{\text{key}} \div 2^{15}_{\text{PAC}} = 2^{209}_{\text{collisions}} \]

• With 34bit pointer/context plenty of collisions \( (2^{181}) \)

• But: random collisions not very useful :(
Cryptographic Definition of a MAC

- Let $\Pi$ be a MAC with following components:
  - $Gen() : \; k \leftarrow Gen(1^n)$
  - $Mac(m) : \; t \leftarrow Mac_k(m) \; \text{with} \; m \in \{0,1\}^n$
  - $Vrfy_k(m, t) : \text{true if } (t = Mac_k(m)) \; \text{else} \; \text{false}$
Mac is secure if: $\Pr[Mac - forge_A, \Pi(n) = 1] \leq \text{negl}(n)$
Mac-forge Game

Mac is secure if: \( \Pr[Mac - forge_A, \Pi(n) = 1] \leq \text{negl}(n) \)

PAC attacker does not have an oracle!
Cryptographic Security of PAC

- PAC attacker weaker than MAC attacker

- Every secure MAC is a secure PAC

- Even an insecure MAC *might* still be a sufficiently secure PAC!

- Secure MACs have been around for a while, thus a PAC designed today will likely be secure (in my opinion)

- Go for implementation attacks instead, those will be around forever!
Future iPhone Hax

• Likely not gonna try to bypass KTRR / not patch kernel code

• Gonna struggle with PAC when exploiting

• Might avoid kernel after all

• Need to re-calculate what the low-hanging fruits are

• Maybe go back to iBoot?
Questions?