What's in a Jailbreak?

By Mark Dowd

Author of

The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities

Director of Azimuth Security

Who am I?

- Did vulnerability research / exploit dev for 9 years with ISS X-Force (2000 – 2009)
- Founded Azimuth Security (2009, Acquired by L3 Technologies in 2018)
  - Rescued several Internet hackers from Apple
- Done security research and presented on various exploitation topics (Windows/iOS primarily)
- Authored “The Art of Software Security Assessment”

Introduction

- iPhone Jailbreaking is all the rage
  - Jailbreaks released to much fanfare
- Big money

Introduction

"*iOS Jailbreaking is the removing of software restrictions imposed by iOS, Apple's operating system, on devices running it through the use of software exploits*"

- Wikipedia

- An increasingly complicated task
  - 5 years ago, jailbreaking an iPhone was like breaking out of county lockup, now we are trying to bust out of a SuperMax!

What will we cover?

1) iPhone Security model
2) Jailbreak Anatomy
3) Practical Example: Pegasus
4) Post-Pegasus: iOS 12 Jailbreaking

Part I: The iPhone Security Model

iPhone Security Model

- Apple's security model
  - Code Integrity
  - Bug minimization / Isolation
  - Exploit mitigations
  - Environment Preservation (Post-exploit mitigations)
  - (Encryption - Data privacy)

- Won’t consider data privacy
  - More relevant to USB-style attacks
  - Not enough time, read Ivan Krstic’s excellent talk: https://www.blackhat.com/docs/us-16/materials/us-16-Krstic.pdf

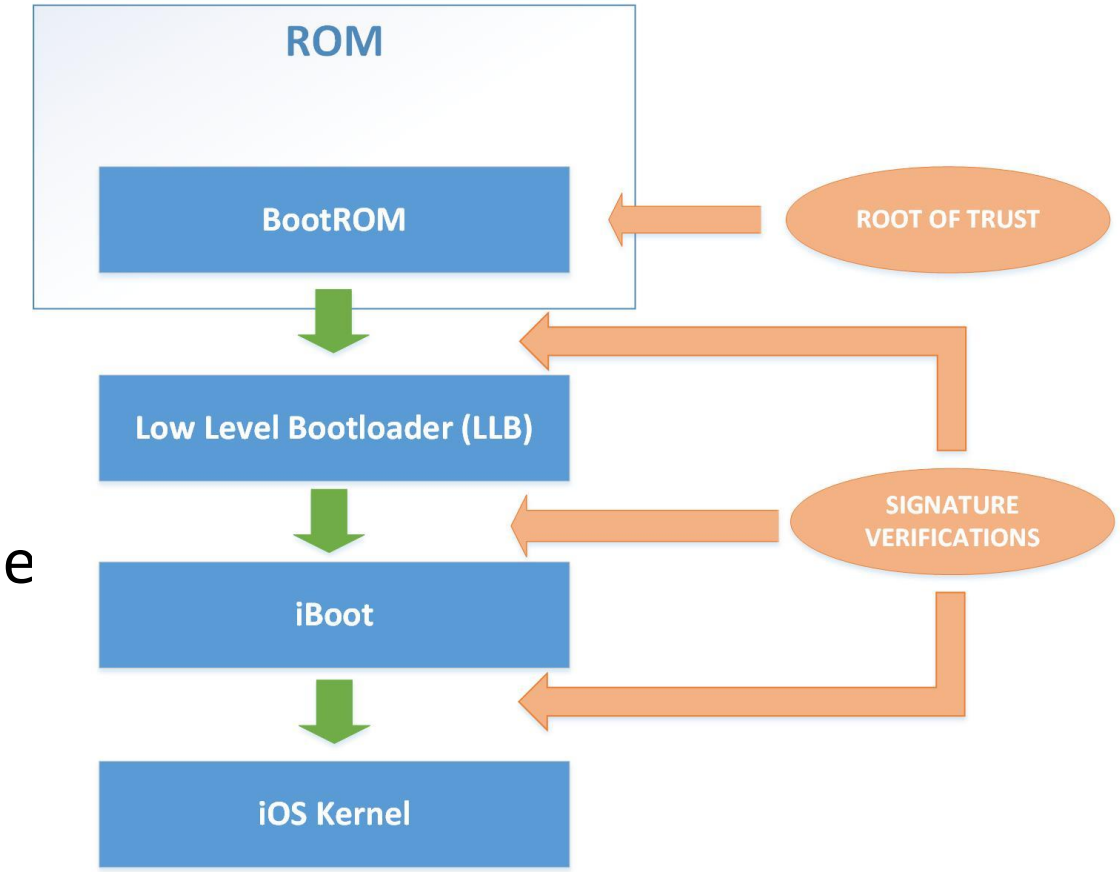
iPhone Security Model

- Code Integrity Goals
  - Ensure only Apple-approved code is ever running on the system

- Multi-pronged approach
  - Trusted boot chain (Boot Integrity)
  - 100% Signed usermode applications / apps (Usermode Integrity)
  - Maintain runtime integrity (Runtime Integrity)

iPhone Security Model: Boot Integrity

- Execution begins with the BootROM
  - Immutable
  - Establish root of trust (Apple)
- Several other boot stages
  - LLB
  - iBoot
  - iOS kernel
- Each stage cryptographically verifies the next stage as it loads it
- End result: iOS kernel is supplied by Apple and is not modified



iPhone Security Model: Usermode Integrity

- All usermode binaries are signed by Apple
  - All binaries contain a code signature
  - Code signature is cryptographically verified by kernel (or amfid)

- Code signature contains information about the executable (or App)
  - A list of hashes of every page in the binary
  - Entitlements granted to the binary (discussed later)

iPhone Security Model: Usermode Integrity

- Two methods for verifying binaries
  - Hash code directory, check for match in static list in the kernel (“trust cache”)
  - Slow method: kernel contacts a system binary named amfid (Apple Mobile File Integrity Daemon)
  - Checks the apps attached signature, that it is signed by Apple
- Each page is checked against code directory for validity as it is faulted in
  - Kill the process (with extreme prejudice) if a discrepancy is found

iPhone Security Model: Runtime Integrity

- Prevent Introduction of new code
  - Disallow RWX mappings (or variations thereof)
  - Executable code must belong to something with a valid code directory
- Disallow library interposing
- Disallow third-party library loading (“TeamID”)
  - System binaries can only load system libraries
  - Apps can only load system libraries and libraries from same vendor
- Disallow interpreter script execution

iPhone Security Model: Isolation

- Isolation Goals
  - Prevent compromised applications from adversely affecting the system
  - Reduce attack surface
- Enforce granular control over system resources and operations
- What an application may access is governed by 3 factors
  - User running the application
  - Entitlements
  - Sandbox restrictions

iPhone Security Model: Isolation

- User is either root or mobile
  - All apps and many services run as mobile (garbage user)
  - A few services run as root

- Entitlements are special privileges granted to the application
  - Entitlements are per-binary
  - Included in the code signature
  - They are immutable; programs can’t be modified or augmented at runtime

iPhone Security Model: Isolation

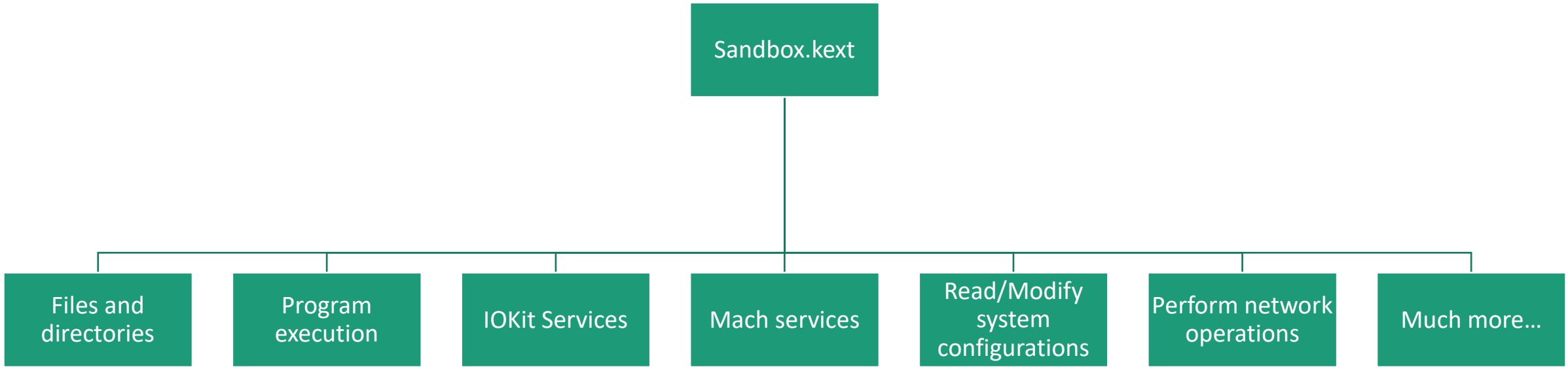
Examples of entitlements

Entitlement Name	Meaning	Application Example
dynamic-codesigning	Can allocate a single executable code region for dynamically generated code (used for JIT primarily)	Com.apple.WebKit.WebContent (MobileSafari's sandboxed process)
com.apple.keystore.access-keychain-keys	Can access the phone's keychain	securityd service
com.apple.private.skip-library-validation	Ignore TeamID protection among other things	

Full DB of entitlements by Jonathan Levin: http://newosxbook.com/ent.jl

iPhone Security Model: Isolation

The Sandbox kernel extension (Sandbox.kext) hooks numerous operations and consults the application's policy to determine access



iPhone Security Model: Isolation

- AppStore apps have a standardized app profile
- System applications / services typically use built-in profiles (or possibly no sandbox)
- Most importantly, prevent access to files, XPC services, and IOKit kernel objects (significant attack surface)
- Excellent in depth discussion on Sandboxing mechanics by Jonathan Levin: http://newosxbook.com/files/HITSB.pdf

iPhone Security Model: Isolation

- Entitlements vs sandboxing
  - Sandboxing generally restricts access that a user would normally have
  - Entitlements generally permit access that the user (or no one) would have
  - Entitlements are sometimes used for whitelist sandboxing
    - Example: Access to a particular XPC service

iPhone Security Model: Exploit Mitigations

- Exploit Mitigation Goals
  - Prevent vulnerabilities from being (reliably) exploitable
- Exploit mitigations are implemented in both hardware and software
- Continually being updated
  - Less mature ones are generally easier to circumvent

iPhone Security Model: Exploit Mitigations

- Early-stage mitigations
  - Encode or detect modification of control structures (heap, saved return addresses, etc)
  - Usually applies to only 1 or 2 bug classes
  - Examples: Heap hardening, ASLR

- Late-stage mitigations
  - Prevent arbitrary code from loading/executing
  - Can hinder most attacks, even unique/new threats
  - There are usually more bypass avenues open to the attacker in later stages
  - Examples: NX, CFI (PAC), JIT Hardening, Code Signing

iPhone Security Model: Exploit Mitigations

iOS Early-Stage Exploit Mitigations

	Affected Bug Classes	Bugs Rendered Unexploitable	Development Cost Increase	Compromise Exploit Reliability
Stack Cookies	Stack overflows	Many	Generally Recurring	No
Safari Heap Isolation / GIGACAGE	Heap Overflows, UAF, Type Confusion	Many	Generally Recurring	No
Kernel Heap Hardening I	Heap overflows	Very few	Generally Recurring	No
Kernel Heap Hardening II	UAF, Type Confusion	Many	One off or recurring	No
Kernel Heap Randomization	Heap overflows, UAF, Type Confusion	Very few	Linear	Yes
ASLR / kASLR	All	Many	Recurring	Yes

iPhone Security Model: Exploit Mitigations

iOS Late-Stage Exploit Mitigations

	Bugs Rendered Unexploitable	Development Cost Increase	Compromises Exploit Reliability
NX	None	Recurring	No
SMEP/SMAP	None	Recurring	No
JIT Hardening	None	Linear	No
PAC	Some (context-dependent)	Linear or Recurring	No
Code signing	Very few	Recurring, very high	No

iPhone Security Model: Exploit Mitigations

- Most relevant mitigations for this talk

kASLR/ASLR	Kernel heap hardening	Safari heap hardening	Safari BulletProofJIT	Pointer Authentication (PAC)
• "slide" binary images by random amount	• Zone isolation • Zone data structure hardening (zone metadata) • Zone / block verification • Block poisoning • Freelist randomization	• Isolation / GIGACAGE	• Dual mapping (A10 X-only) • Fast-permission switching	• Signed pointers / vtables

iPhone Security Model: Environment Preservation

- Environment Preservation Goals:
  - Limit impact of fully compromised kernel (TFP0)
  - Preserve system integrity
  - Attempt to preserve integrity of critical data structures / files in the event of compromise

- Lately, the focus has been here
  - Examples: Code Signing, CFI (PAC), PPL, APFS filesystem protections

iPhone Security Model: Environment Preservation

Late-stage Exploit Mitigations

	Intended Effect	Development Cost Increase
APFS remounting	Can’t write to root F/S	Linear
PPL (+PAC)	Can’t inject unsigned code with TFP0	Recurring
Code signing	Can’t run any unsigned code	Recurring, very high
KPP / KTRR	Can’t modify protected region in kernel space	Recurring

iPhone Security Model: Takeaways

- Apple's two key differentiators
  - Code Integrity
  - Controlling the hardware

- Differentiator 1: Code Integrity
  - Code is whitelisted, not blacklisted
  - Very rigid controls over executing any non-Apple (or Apple-authorized) code
  - Been done before, but only in limited contexts (signed Microsoft ActiveX controls, TPM, etc)

- Differentiator 2: Controlling the Hardware
  - Can enforce security at the hardware level
  - Can generally get hardware mitigations to market more quickly than competitors

iPhone Security Model: Takeaways

- Exploit mitigations increasingly moved from software to hardware
  - KPP => KTRR
  - BulletProofJIT iterations (JIT fast permission switching)
  - Pointer Authentication (PAC)
  - PMAP Protected Layer (PPL)
- Hardware mitigations often much more difficult to bypass
  - Significantly complicated software component can be a good target
- Some gotchas:
  - Need the latest devices to get the latest mitigation benefits
  - A hardware-level bug might not be resolvable through patching

Part II: Jailbreak Anatomy

Jailbreaking Anatomy

- Compromising the phone requires a multitude of vulnerabilities
  - Any broken link in the chain will render the chain partially or entirely useless
- We will consider remote jailbreaks today
  - There are also other kinds of jailbreaks, don't have time to cover them
    - Local Jailbreaks (USB-based)
    - Near-access Jailbreaks (Baseband, WiFi, BTLE)
- Goals:
  - Compromise runtime integrity
  - Compromise trusted boot chain to regain code execution after reboot

Jailbreaking Anatomy: Jailbreak Template

- Compromise Runtime Integrity
  - Stage 1: Get a foothold on the system
  - Stage 2: Extend access
  - Stage 3: Get kernel access
  - Stage 4: Extend code execution capabilities (remove code signing, add entitlements, etc)
- Compromise Trusted Bootchain
  - Stage 5: Place data on the system

Jailbreak Anatomy: Jailbreak Template

- Generally: At least 3 bugs
  - Some stages may not be required
  - Some stages may require more than 1 bug (think: Information Leaks)

- Typical exploit chain
  - Safari exploit
  - Kernel exploit
  - Code-signing bypass and/or boot time exploit

- Variety of potential vectors
  - browser, messaging app, mail client, etc
- Browsers are the most attractive target to attackers by far
  - Large attack surface
  - Interaction with complex state machine
  - Ability to groom memory easily
  - Programmatic feedback (infoleaking)
  - Most importantly: “run-unsigned-code” entitlement for JIT!!
- Alternate vectors have a huge problem: code signing
  - Remember: ROP sux, and also maybe not an option with PAC!

- Optional stage, if direct kernel exploit is not available
  - Break out of sandbox
  - Obtain additional privileges needed for access to larger kernel attack surface

- Escalate to achieve kernel access (that is, access to kernel_task)
  - Commonly referred to as "tfp0" (task_for_pid(0) - ie. getting a kernel_task port)
  - Allows memory to be read from/written to in kernel
  - This used to imply kernel-mode code execution (now doesn't - more on that later)
- Exposed kernel attack surface depends on vantage point, but includes:
  - Mach services (RPC-like services running in-kernel)
  - System calls (Restricted by sandboxing rules)
  - IOKit Drivers (Graphics drivers and the like, also restricted by sandbox rules)
- Some chains can bypass Stage 1 & 2 and go straight to kernel
  - Exploiting a USB driver or similar
  - Remote TCP/IP bug (ICMP example)

- Post-exploitation strategy depends on use case, but includes the following
  - Weakening security controls
  - Installing persistence
  - Implant stuff (not important here)

- Weakening security controls
  - Allow unsigned binaries to run
  - Removing sandboxes selectively
  - Granting entitlements
  - Modifying system-wide configuration data

- Installing Persistence
  - Implant binary or other data to subvert secure bootchain and regain TFP0 after reboot
  - Often involves remounting the root filesystem with write privileges

- Re-gaining code execution after reboot
  - Not straightforward - remember trusted boot chain?

- Attack the bootchain - break the chain of trust
  - Goal is to bypass cryptographic checks
  - Memory corruption flaws: malformed FS, executable parsing, IMG3/IMG4 parsing
  - Logic flaws: Cryptographic weaknesses, segment trickery perhaps

- Option 1: Exploit BootROM
  - Can undermine everything in iOS
  - Memory corruption mitigations mostly not present (ASLR, hardened heap etc)
  - Been done before – 24KPwn (https://www.theiphonewiki.com/wiki/0x24000_Segment_Overflow)
  - Would *not* be able to undermine SEP (need SEPROM or other SEPOS bug)
  - Ever-present bug, can't be fixed
  - Very difficult these days, BootROM is very sparse
- Option 2: Exploit iBoot
  - Similar advantages, although can be patched, and PAC is present
  - Attack surface is larger but still slim pickings
- Option 3: Attack kernel boot-time
  - Very difficult to perform (for memory corruption)
  - Full scale of memory corruption mitigations present
  - Limited ability to do any precise memory grooming since you aren't executing code yet

- Option 4: Usermode
  - Find a logic flaw in code-signing machinery
  - Exploit an Apple binary that runs at startup
  - Alternatively: replace an Apple binary that runs at startup (more on this later)
  - Difficult for memory corruption (not running code yet)
  - A bit less difficult than kernel, might get to try multiple times
  - Failure does not mean reboot (or bootloop) of the device (usually)

- Option 4 is by FAR the most popular vector

- Finding a logic flaw in code-signing is by far the most desirable option
  - Reliable
  - Allows privilege escalation to be performed in native code (NO ROP)
  - Could possibly re-use the component in Stage 1 – “zero click” options become more viable

- Areas to target
  - Kernel binary loader
  - Dynamic loader (DYLD) – loading libraries or the shared cache
  - Code-signing fault-in logic
  - Apple Mobile File Integrity Daemon (amfid)

- Past compromise example
  - Details of Pangu iOS 9 DYLD bypass: https://www.blackhat.com/docs/us-16/materials/us-16-Wang-Pangu-9-Internals.pdf
- Problem: It's been rigorously attacked
  - Undergone a lot of scrutiny
  - Codebase is quite mature – difficult target now
  - Still possible occasionally! Ian Beer's mach_portal amfid bypass: https://bugs.chromium.org/p/project-zero/issues/detail?id=965#c10

Part III: Jailbreak Walkthrough (Pegasus)

Pegasus Jailbreak

- We will examine NSO's "Pegasus" Jailbreak
  - Discovered in the wild by Lookout
  - Awesome writeup by @mbaziley: https://info.lookout.com/rs/051-ESQ-475/images/pegasus-exploits-technical-details.pdf

- Target: iOS 9.3
  - Release date: March 21, 2016
  - MobileSafari version: 9.0 (601.1.46)
  - Kernel version: xnu-3248.41.4~28
  - Current device: iPhone 6S/SE (64-bit)

Pegasus Jailbreak

- iOS is a tough target, but a bit less so back then

- Safari / JavaScriptCore Exploit (CVE-2016-4657)
  - Use After Free (UaF) in the MarkedArgumentBuffer implementation
  - Can be triggered via defineProperties() method

- Protections
  - Limited heap protections

- Stage 1-1: Trigger the bug
  - Call the defineProperties() method on an object with multiple properties
  - One of the properties will a toString() method which will delete a property
  - Also needs to cause garbage collection
- Stage 1-2: Defeat ASLR
  - Have the stale reference and a new object point to the same memory, both objects have different types
  - Use a Uint32Array object to access process memory
- Stage 1-3: Arbitrary code execution
  - Create new function, call it repeatedly so it is JIT'd
  - Use the ASLR primitive to find the JSFunction object
  - Overwrite JIT and call function

- Not needed

- Pair of vulnerabilities:
  - OSUnserializeBinary() Information leak (kASLR bypass) exploit (CVE-2016-4655)
  - OSUnserializeBinary() Use After Free (UaF) exploit (CVE-2016-4656)

- Protections
  - Kernel heap hardening I (Internal randomization, poisoning)
  - kASLR

- Stage 3-1: Defeat kASLR
  - Use CVE-2016-4655 to read excess kernel memory
  - Find kASLR slide
- Stage 3-2: Trigger the bug
  - A reference to an object that has been freed is retained in an array
  - This object will later have object->retain() called on it
  - Result is a vtable call on a free object
- Stage 3-3: Replace freed object
  - Object is replaced with controllable data
  - Combined with kASLR leak, can use this to initiate ROP chain in kernel
  - ROP chains installed in kernel which allow arbitrary DWORD writes in kernel and also performing kernel function call

- Protections
  - Kernel Patch Protection (KPP)

- Stage 4-1: Get kernel_task
  - Use kernel ROP gadgets to patch task_for_pid()
- Stage 4-2: Escalate to root
  - Patch setreuid
- Stage 4-3: Remove sandbox policy on attacker’s process
  - Modify mac_policy_list member registered by Sandbox.kext
- Stage 4-4: Disable code signing
  - Set amfi_get_out_of_my_way and cs_enforcement_disable and some global debugging variables
  - Also patch vm_map_enter() and vm_map_protect()
  - Patch csops function
- Stage 4-5: Remount root filesystem with read/write access
  - Patch LightweighVolumeManager's partition array

- Protections
  - Root filesystem is read/write but not really protected from remount w/ TFP0

- Stage 5-1: Install persistence files
  - Replaces a system binary (rtbuddyd) with a trusted Apple binary (jsc - JavaScriptCore command-line tool)
  - Can run a javascript file from the command-line
  - Plant a JavaScript file that will exploit a bug at boot up
  - The jsc binary also has the "run-unsigned-code" entitlement
  - Then re-exploit kernel bug

- Pegasus exploits a bug in the setImpureGetterDelegate() function
  - Type confusion bug
  - Exploitation strategy is similar to the initial browser vulnerability

Part IV: Jailbreaking iOS 12

Jailbreaking iOS 12: Overview

A lot has changed since Pegasus

iOS 10 (September 2016)

- Kernel heap hardening (zone metadata, zone checks)
- Safari JIT dual mapping (A10 X-only)
- Kernel Text Readonly Region (KTRR)
- Apple FileSystem (APFS)

iOS 11 (September 2017)

- Safari JIT improvement (A11 Fast-permission switch)
- Safari GIGACAGE / heap isolation
- APFS snapshot hardening

iOS 12 (September 2018)

- Pointer Authentication (PAC)
- PMAP Protected Layer (PPL)

Jailbreaking iOS 12: Overview

- Many of these mitigations have been talked about at length in the past
  - See Luca Todesco's excellent MOSEC 17 talk: https://papers.put.as/papers/ios/2017/A_Look_at_Modern_iOS_Exploit_Mitigation_Techniques.pdf

- New mitigations with iPhone XS/XR (A12)
  - Pointer Authentication (PAC): Hardware-enforced CFI for user and kernel (and iBoot and SEP)
    - Some discussion: https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html
  - PMAP Protected Layer (PPL): Hardware-enforced "sandboxing" for kernel PMAP layer (governs page protections, code signing)
    - A brief overview here: https://rstforums.com/forum/topic/110506-explaining-apples-page-protection-layer-in-a12-cpus/?tab=comments#comment-668078

Jailbreaking iOS 12: Overview

- Biggest changes
  - Safari exploit mitigation refinements (Stage 1)
  - Safari (and other) sandboxes tightened (Stage 1+2)
  - Kernel exploit mitigation refinements (Stage 3)
  - Runtime integrity mitigations (Stage 4+5)

Jailbreaking iOS 12: Safari Hacking

- Dual mappings (iOS 10) -> Fast-permission switching (iOS 11)
- Cannot directly write to JIT region anymore (in theory)
- Complicates arbitrary execution stage

JIT memory protections (iOS 10, iOS 11)

- Isolates data of similar types to same region of memory
- Greatly complicates numerous UaF bugs and popular TypedArray exploitation techniques
- Many UaF bugs in the DOM are considered unexploitable now – focus is nearly entirely on JavaScript engine bugs

Gigacage (iOS 11)

- Prevents ROP (if required)

PAC (iOS 12/A12)

Jailbreaking iOS 12: Kernel Hardening

- Kernel freeing to wrong zone mitigation
  - Eliminated a bug class
- Zone page metadata heap hardening
  - Prevents some exploitation techniques

Jailbreaking iOS 12: Runtime Integrity

- Remember goal 1 of a jailbreak:
  - Compromise runtime
  - Compromise bootchain

- Pre-iOS 9, runtime integrity was completely undermined with TFP0
  - Arbitrary kernel code could be run
    - Changing page permissions etc via kernel_task
  - Arbitrary usermode code could be run
    - Modify trust cache to run any binary
    - Modify other process-related data structures to inject entitlements / disable codesigning

- Apple is addressing this shortcoming aggressively

Jailbreaking iOS 12: Runtime Integrity

iOS 9 / A9

- Patching kernel code risks kernel panic (KPP)
- Solution: Temporary patching (or bypass)

iOS 10 / A10

- Kernel code cannot be modified (KTRR)
- Solution: Kernel ROP

iOS 12 / A12

- Bug required for Kernel ROP (PAC)
- Can’t easily run unsigned code (PPL)
- Solution: ??

Jailbreaking iOS 12: Runtime Integrity

@pod2g and @kernelpool hopefully know!

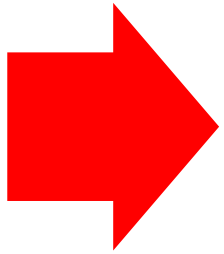
Pegasus Revisited

- Pegasus Revisited: What problems are there?
- Stage 1 (Safari)
  - ~~Gain code execution writing to JIT~~ (BulletProofJIT)
- Stage 3 (Kernel)
  - ~~Overwrite VTABLE~~ (PAC)
  - ~~Install ROP backdoors~~ (PAC)
- Stage 4 (Payload)
  - ~~Patch kernel code~~ (KTRR)
  - ~~Remount Filesystem~~ (APFS)
  - ~~Backdoor usermode processes~~ (PPL)
- Stage 5 (Persistence)
  - jsc (and other developer binaries) removed from trust cache

Jailbreaking iOS 12: New Requirements

iOS 9 Remote Jailbreak

- Safari
- Sandbox escape*
- Kernel
- KPP bypass*
- Persistence



iOS 12/A12 Remote Jailbreak

- Safari
- Sandbox escape*
- Kernel
- Persistence
- BulletProof JIT bypass
- Usermode PAC bypass*
- Kernel PAC bypass*
- Kernel PPL bypass*
- APFS remount bug*

Summary

- iPhone jailbreaking has always been difficult, but the newest A12 mitigations make full compromise of the ecosystem really tough
  - Often confined to ROP
  - Even ROP has been dealt a major blow
  - PAC/PPL further maturing represent a formidable barrier
- Future is likely to increasingly rely on data-only attacks
  - Likely newer mitigations will target this by expanding on data structure integrity mitigations

The End

- Thanks for listening!