Life as an iOS Attacker

~~~qwertyoruiop@tensec~~

~qwertyoruiop@bluehatIL
~~~

whoami

- Luca Todesco aka @qwertyoruiopz
- iOS security researcher at KJC Intl. Research S.R.L.
- iPhone owner and jailbreak enthusiast since iOS 2
- Released a public jailbreak for iOS 10.2
- Gave this talk already in Shenzhen at TenSec
  - But I added a few things this time around
- Can be found at irc.cracksby.kim when the server isn't down

Agenda

- Brief recap of iOS security design
- iOS attacker model
- Typical 1-click exploit
- arm64e changes
  - Pointer Authentication
  - PMAP hardening
- The future of iOS attackers

iOS Security Design

A brief rant

iPhoneOS 1.0

- No ASLR
- No DEP
- Every process ran as root
- No sandboxing
- No code signing
- No real secure boot
    - Image signature validation doesn't happen for flashed images on S5L8900 (iPhone EDGE, iPod Touch, iPhone 3G)
    - Image signatures are not personalized, downgrades allowed

iPhoneOS 1.0

- **No 3rd party apps**
    - Potential of native 3rd party apps was obvious to anyone who owned an iPhone back then
    - The iOS homebrew community was thus born
        - Called “Jailbreaking”
            - AppTapp, JailbreakMe, PwnageTool were the tool of the trade
        - Even a few Apple employees were involved with the jailbreaking community

A wild threat appears!

- iPhoneOS 2.0
    - AppStore is introduced
        - Apps were sandboxed by default
    - App piracy was rampant on desktop OSes
        - Apple tried to limit it with “walled garden” approach
            - Codesigning is introduced
        - AppStore apps shipped with DRM by default (FairPlay)
    - The only real threat at that point was Apple vs. piracy

Cat & Mouse

- A cat and mouse game thus started between the jailbreaking community and Apple
  - Great learning opportunity for Apple on attacker mentality
- Over time this process exposed a significant amount of weaknesses in the iOS security model
  - Attack surface reduction (Bootloader functionality stripping)
  - Exploit mitigations (ASLR & DEP)
  - Vulnerability impact management (downgrade prevention)

Unintended consequences

- Historical security design decisions still to this day carry unintended consequences
  - iOS used to heavily rely on security by obscurity
    - Still does in some aspects (SEP & bootloader images are encrypted)

    - A few images (kernel, rootfs, ramdisks) are now decrypted and auditable by third party researchers
    - Kernel sources are now public
    - Bootloader sources recently leaked

Unintended consequences

- Historical security design decisions still to this day carry unintended consequences
  - Downgrade protection
    - Provides little-to-none security against any threat who isn't some kid trying to decrypt FairPlay'd apps and can't afford to buy a phone for each other version
  - Anti-debugging measures
    - You can't easily play with iOS internals even on your own devices without using jailbreaks

iOS Attacker Model

iOS Attacker Model

- Initially mostly jailbreakers were really in the game
  - These are attackers that are negative for Apple but fundamentally positive for end-users
- Over time, a new breed of attackers appeared
  - Interested in attacking iOS for intelligence gathering, law enforcement and forensic analysis
  - These attackers are mostly neutral for Apple's bottom line but negative for end users

iOS Capabilities

- These advanced attackers are attracted to multiple classes of iOS capabilities
  - Remote
    - 1-Click (Browser + privilege escalation)
    - 0-Click (XSS, System services, WiFi, Bluetooth, Cellular)
  - Physical
    - Evil maid attacks & in-transit device tampering (initial vector + persistence)
    - Data protection attacks for forensic analysis (initial vector + SEP code exec)

The iOS game

- Individual researchers will usually specialize on only one aspect for maximum efficiency and find vulnerabilities suitable to achieve a given capability
- Offensive companies then chain multiple vulnerabilities (either purchased or found in-house) into an exploit chain capable of running code at a privilege level high enough to accomplish task
- Post-exploitation toolkits are developed and maintained to provide customers with data from victim devices
- Apple eventually kills some bug in the chain
  - GOTO 1

The iOS game

- As mitigations creep in, advanced persistent attackers that are on top of the game can easily play catch up
  - Attacker cost on each iteration is marginal
- New players are likely to get overwhelmed
  - Full attacker cost that for others was distributed over several years has to be paid immediately just to get started in the game
- Halvarflake has a good talk about this

Typical 1-click iOS Exploit Chain

- 1-click is the most reliable and likely-to-be long-term viable capability due to complexity of web browsers
    - And can actually be turned into a 0-click chain with a XSS in either some app or system service
- It usually involves the combination of at least one WebKit vulnerability and at least one privilege escalation vulnerability
- WebKit vulnerability must be powerful enough to derive an info leak or an additional info leak vulnerability is required in order to bypass ASLR or leak heap pointers
    - Heap spraying is in theory viable but it’s unlikely for advanced attackers to rely on it as it comes with reliability concerns

Typical 1-click iOS Exploit Chain

- DEP can be bypassed by ROP or by data-only exploitation
  - Data-only is the advanced attacker's choice as it requires minimal maintenance and is generally more powerful
    - Aim is usually to gain read/write primitives
  - ROP can be simpler at times but requires updating exploit code on every version and makes process continuation trickier
    - Necessary with some vulnerabilities
      - My suggestion is to save those for pwn2own and keep looking for better ones to use for your 1-click chain

Typical 1-click iOS Exploit Chain

- Once the ability to perform arbitrary reads, writes and function invocations from WebKit is gained, the next stage is to load shellcode
    - iOS has mandatory code-sign checks for all executable code
        - But exempts an area of reserved memory used by JIT
            - Write your payload in JIT memory and invoke it
                - Back in iOS 9, this was all you needed to do

“Bulletproof” JIT

- Write your payload in JIT memory and invoke it
  - Apple decided to try and harden this back in iOS 10 with a mitigation called “Bulletproof JIT”
    - RWX area was split in two maps, one writeable and one executable
    - Pointer to writeable map is only ever saved in a small stub in executable-only (non-readable) memory
  - Further hardened with silicon changes on A11 and later CPUs
    - RWX area is not split anymore, but not really writeable until a specific system register is set to a certain value
      - Only a special JIT memcpy function touches that register and undoes the change after performing the copy

“Bulletproof” JIT

- Fundamentally Bulletproof JIT forces you to do some ROP or JOP in order to emit arbitrary opcodes
  - So the mitigation itself is useless unless control flow integrity is also there since ROP is trivial
    - Early warning for careful attackers that CFI was coming
      - 2 years lead time, plenty for serious iOS game players to make strategies around it

Typical 1-click iOS Exploit Chain

- Some will just write the second-stage payload as raw shellcode
  - Kinda painful if you ask me.
- Easier to write payloads in C and compile into a dynamic library with the iOS SDK
  - Copy in JIT memory and ask the dynamic linker nicely to load it
    - No intended API to load a library from memory

Typical 1-click iOS Exploit Chain

- No intended API to load a library from memory
    - Main executable is loaded in the address space by kernel
        - Kernel invokes the dynamic linker upon process creation by calling “dyld_start” which takes care of rebasing and linking
            - Nothing stops us from calling this again
- I think this strategy is not optimal due to several issues such as process continuation being tough to achieve
    - Still OK for things like jailbreaks
        - https://jbme.qwertyoruiop.com uses this to load stage2

Typical 1-click iOS Exploit Chain

- I personally ended up just writing my own dynamic linker for Mach-Os in Javascript
  - Works pretty well as far as process continuation and maintainability goes
  - Javascript can truly do everything these days given enough memory corruption

Typical 1-click iOS Exploit Chain

- Usermode privilege escalation can re-use ASLR leak from WebKit as the shared cache load address is only randomized on boot
    - ROP/JOP in order to access privileged resources to either trigger a kernel-mode privilege escalation or capture some data
- Kernel-mode privilege escalation requires a separate kernel info leak (or a vulnerability strong enough to both IL and gain code execution)
    - ROP/JOP/data-only in order to derive read/write primitives

Apple's name for the new A12-specific ABI

arm64e Changes

Control Flow Integrity

Control Flow Integrity

- Writing was on the wall as early mentioned
- Both backwards-edge and forward-edge
- Implemented using ARM8.3 authenticated pointers
- Both kernel-mode and user-mode make use of CFI
- Not fully fine-grained but not coarse-grained either

Pointer Authentication

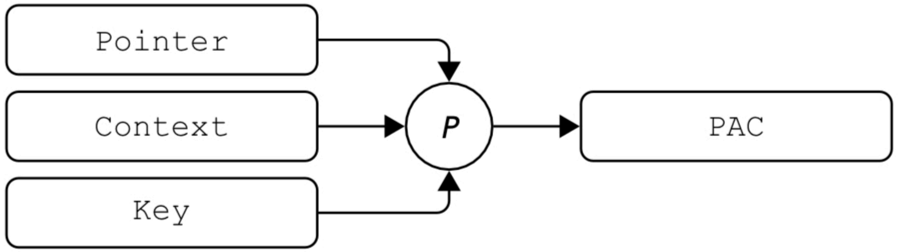
- New instructions to sign and authenticate pointers
  ... against a user-chosen (dynamic) context
  ... e.g. return address is valid for a given stackframe
  ... architecture provides mechanism, not policy
- Uses a Pointer Authentication Code (PAC)
  ... authentication metadata stored within pointer
  ... so no additional space required

Pointer tagging via bits normally unused for virtual addressing

https://events.static.linuxfound.org/sites/events/files/slides/slides_23.pdf

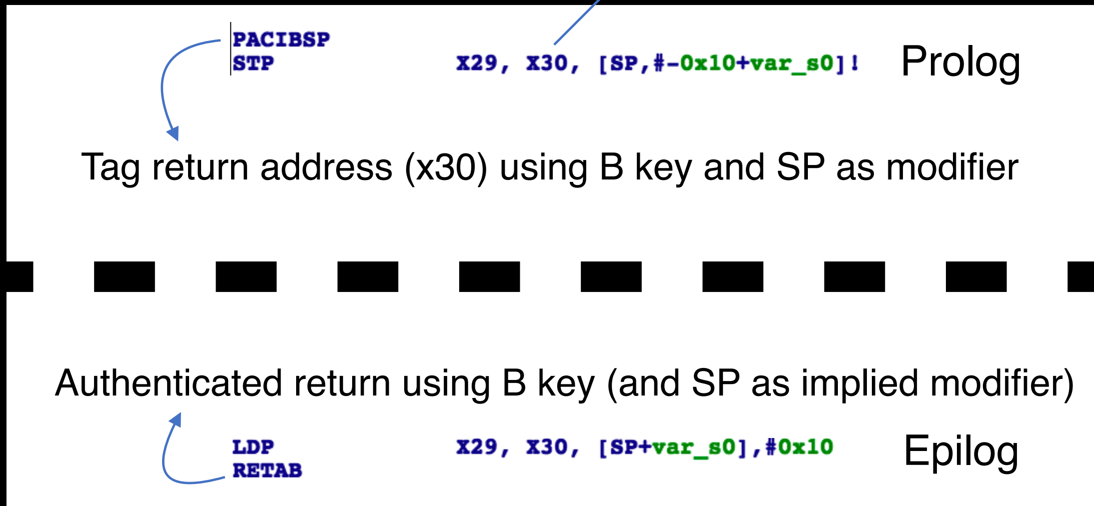
Pointer Authentication

- 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



Backwards Edge CFI Implementation

X30 is the return address in arm64



Somewhat limited opportunities for pointer substitution attacks due to SP-specific signature

Forward Edge CFI (C++ Virtual Call)

X0 = C++ object

```
LDR                 X8, [X0]
AUTDZA   X8
ADD                 X9, X8, #0x5B8
LDR                 X8, [X8,#0x5B8]
MOVK                X9, #0xEA23,LSL#48
MOV                 X1, X2
BLRAA    X8, X9
```

Fetch vtable off C++ object

Authenticate vtable pointer with A data key and zero context

Use pointer to virtual function pointer as context

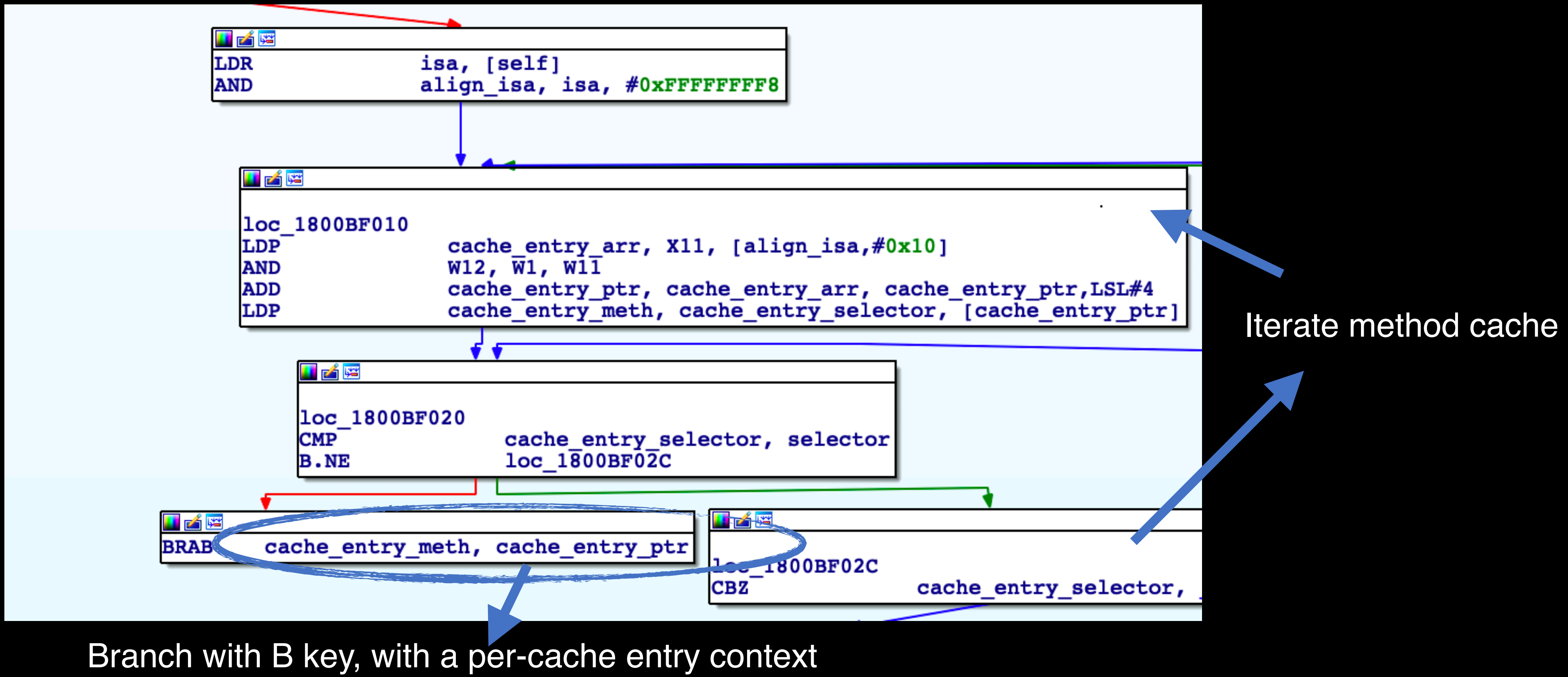
Fetch virtual function pointer

Set bits 64:48 of context to a virtual call specific value

A-key authenticated branch to virtual function pointer using non-zero context

Very limited opportunities for pointer replacement attacks due to vcall specific context

Forward Edge CFI (objc_msgSend cache)



Method pointers are signed with the B key and a per-cache entry context
Might be interesting to play with “creative ideas” here

Forward Edge CFI (C indirect branch)

Initial X8 = Unauthenticated pointer to C struct

```
LDR              X8, [X8,#0x18]
MOV              W5, #1
MOV              W2, #1
MOV              X0, X24
MOV              X3, X21
MOV              X4, #0
BLRAAZ   X8
```

Fetch function pointer

A-key authenticated branch with zero context

Lots of opportunities for pointer replacement attacks

Special CFI cases

- Usermode
  - thread_set_context, pthread_create and other “usual suspects” also require A-key signed pointers
    - However they lack context
      - Intrinsically weak points
- Kernelmode
  - thread_call_* APIs require A-key signed pointers
  - G key is used for pexpert machine context validation

Practical Pointer Authentication Attacks

- Pointer replacement attacks
  - Leak signed pointers (say, with a read anywhere primitive) and use them as long as key and context is the same
- Pointer forgery attacks
  - Signing gadgets are present in executable address space
    - Just need to find one CFI weakness and can use signing gadgets to build ordinary ROP/JOP chains to perform more advanced operations

Non-practical Attacking Pointer Authentication

- Brute Force
  - Given enough fork()s you might very well be able to pull this off
    - Not practical for serious attackers, maybe OK for specific scenarios (jailbreaking?)
- Keyspace attacks
  - It appears in iOS the same 64 bits are repeated twice in order to derive the 128 bit key used for PAC
    - 2^64 operations are still within the realm of possibility
      - Maybe practical for serious enough attackers, but doesn't really scale and I'd figure this can easily be changed to use more bits

Impact of Control Flow Integrity

- Strict enforcement is done on C++ virtual calls by using special context values for pointer signatures
  - Virtual call on free'd element is a very common scenario for WebCore and IOKit UAFs
    - Likely a good chunk of these sorts of bugs are now are unexploitable
      - Advanced enough attackers can always just find better bugs

Miscellaneous information

- Key A is shared across processes, so forward-edge scenarios can still be abused for usermode sandbox bypass
- Not CFI-specific, but JavaScriptCore makes heavy use of pointer authentication (preventing some trivial CFI breakage scenarios)
- As much as it'd be nice to get rid of software stack cookies and use PAC instead, it is intrinsically weaker as the return address is usually the last member of the stack frame, and spilled registers come before that

Impact of Control Flow Integrity

- In our previous 1-click exploit chain scenario, things get really complicated now
  - Being able to use data-only to load arbitrary Mach-Os in JIT seems hard
- However a single valid code path with an unprotected branch is all you need
  - Legacy apps not compiled for arm64e
  - Handwritten assembly that uses indirect branches
  - Forcing JSC’s JIT to emit controlled opcodes might also be viable
    - Emit a non-authenticated BR and invoke a signing gadget

Data-only CFI attack ideas

- Being able to issue arbitrary syscalls from data-only WebKit context seems hard
  - iOS however is designed around message passing
    - Often possible to invoke mach_msg with a controlled message by poking stack frames given arb. R/W (I talked about this strategy at MOSEC 2017)
      - Can reach significant privilege escalation attack surface from mach_msg (both usermode PE and kernelmode PE)
      - It's probably possible to obtain pointer forgery from such a powerful primitive
        - Reach any other codepath then

Real Life Pointer Forgery

@bazad’s attack

Brandon Azad's Pointer Forgery

- In January 2019 Brandon Azad released the first public pointer authentication code forgery attack.
- Released in order to provide a kernel-mode arbitrary code invocation primitive that bypasses CFI for his voucher_swap exploit
- Very elegant trick used
  - Major props to him, it's a really cool one!

Brandon Azad’s Pointer Forgery

- His approach was to look for a signing gadget that can be reached from an already-signed pointer
- However no trivial signing gadgets are present
  - Most legit code paths with PACI* instructions will actually “convert” pointers from one kind of authentication code to another (i.e. different context or different keys)
  - Represented as an AUTI*/PACI* sequence
    - First will authenticate with previous context / key, then sign the result with the new context / key

Brandon Azad’s Pointer Forgery

- Interesting to note that there is no bailout case if AUTI* fails
  - This relies on error propagation from the AUTI* instruction to the PACI* one
  - However, PACI* will perform this error propagation by flipping a single bit *AFTER* applying a valid signature
    - Can just flip back to obtain a real pointer forgery

Brandon Azad's Pointer Forgery

- His attack used sysctl_unregister_oid, which will take a signed pointer and turn it into an AAZ-signed one
- Legitimate part of l2tp_domain_module_stop()
  - For which there is a valid AAZ-signed pointer to that we can invoke manually via a pointer replacement attack
- Once invoked, it is possible to read back the result of PACI*, then correct the authentication code
- You can use this to sign a pointer to an unauthenticated branch in order to further invoke arbitrary things without having to perform this again

Brandon Azad's Pointer Forgery

- Once an arbitrary unauthenticated invocation primitive is derived, it is possible to use JOP gadgets that do not require signed pointers in order to manipulate the state
  - Can use this in order to invoke a PACI* skipping the AUTI* part in order to get a "proper" easier-to-reuse signing gadget

Brandon Azad’s Pointer Forgery

- This attack was fixed by adding a failure case after AUTI* instructions
  - His strategy can still be used as a pointer validity oracle
    - **Failure case is not immediately fatal**
    - Can bruteforce an authentication code for a single unauthenticated branch gadget, and then game’s on again

arm64e Changes

pmap hardening

pmap hardening

- pmap is the code in charge of pagetable housekeeping in iOS
- Responsible for several codesign-related tasks
  - Codesign on iOS is enforced at fault-in time
    - Arbitrary physical writes can bypass codesign by altering already-faulted pages
- AMFI delegated trustcache handling to pmap in iOS 12

pmap hardening

- pmap code that alters pagetables and related codesign-critical routines have been put in it's own code segment
  - Only code in this area is able to alter codesign-critical data such as TrustCaches and pagetables
- Entering this code is done through a special routine called "ppl_dispatch"
  - Invoked by a trampoline that sets a system register in order to signal that we are allowed to access protected memory, and moves system context into protected memory to prevent tampering
- Routines exposed via ppl_dispatch validate input
  - They make sure you pass in real pmap_t pointers for instance

pmap hardening

PPL entry:

```
__text:FFFFFFF008FDFFE8
__text:FFFFFFF008FDFFE8 loc_FFFFFFF008FDFFE8                    ; DATA XREF:   text:FFFFFFF007A03CB8↑o
__text:FFFFFFF008FDFFE8                                         ;   text:FFFFFFF007A03CBC↑o ...
__text:FFFFFFF008FDFFE8                 MOVK            X14, #0x4455,LSL#48
__text:FFFFFFF008FDFFEC                 MOVK            X14, #0x4455,LSL#32
__text:FFFFFFF008FDFFF0                 MOVK            X14, #0x6466,LSL#16
__text:FFFFFFF008FDFFF4                 MOVK            X14, #0x6677
__text:FFFFFFF008FDFFF8                 MSR             #4, c15, c2, #1, X14
__text:FFFFFFF008FDFFFC                 ISB
__text:FFFFFFF008FE0000
```

PPL exit:

```
__text:FFFFFFF008FE7FEC ; ---------------------------------------------------------------------------
__text:FFFFFFF008FE7FEC ; START OF FUNCTION CHUNK FOR sub_FFFFFFF007A03C34
__text:FFFFFFF008FE7FEC
__text:FFFFFFF008FE7FEC loc_FFFFFFF008FE7FEC                    ; CODE XREF: sub FFFFFFF007A03C34+1C8↑j
__text:FFFFFFF008FE7FEC                                         ; sub FFFFFFF007A03E10+1CC↑j ...
__text:FFFFFFF008FE7FEC                 MOVK            X14, #0x4455,LSL#48
__text:FFFFFFF008FE7FF0                 MOVK            X14, #0x4454,LSL#32
__text:FFFFFFF008FE7FF4                 MOVK            X14, #0x6466,LSL#16
__text:FFFFFFF008FE7FF8                 MOVK            X14, #0x6477
__text:FFFFFFF008FE7FFC                 MSR             #4, c15, c2, #1, X14
__text:FFFFFFF008FE8000
```

pmap hardening

PPL entry:

```
__text:FFFFFFF008FDFFE8
__text:FFFFFFF008FDFFE8 loc_FFFFFFF008FDFFE8                      ; DATA XREF:   text:FFFFFFF007A03CB8↑o
__text:FFFFFFF008FDFFE8                                           ;    text:FFFFFFF007A03CBC↑o ...
__text:FFFFFFF008FDFFE8                 MOVK            X14, #0x4455,LSL#48
__text:FFFFFFF008FDFFEC                 MOVK            X14, #0x4455,LSL#32
__text:FFFFFFF008FDFFF0                 MOVK            X14, #0x6466,LSL#16
__text:FFFFFFF008FDFFF4                 MOVK            X14, #0x6677
__text:FFFFFFF008FDFFF8                 MSR             #4, c15, c2, #1, X14
__text:FFFFFFF008FDFFFC                 ISB
__text:FFFFFFF008FE0000
```

Single bit in a system register is flipped in order to signal PPL mode

PPL exit:

```
__text:FFFFFFF008FE7FEC ; ---------------------------------------------------------------------------
__text:FFFFFFF008FE7FEC ; START OF FUNCTION CHUNK FOR sub_FFFFFFF007A03C34
__text:FFFFFFF008FE7FEC
__text:FFFFFFF008FE7FEC loc_FFFFFFF008FE7FEC                      ; CODE XREF: sub_FFFFFFF007A03C34+1C8↑j
__text:FFFFFFF008FE7FEC                                           ; sub_FFFFFFF007A03E10+1CC↑j ...
__text:FFFFFFF008FE7FEC                 MOVK            X14, #0x4455,LSL#48
__text:FFFFFFF008FE7FF0                 MOVK            X14, #0x4454,LSL#32
__text:FFFFFFF008FE7FF4                 MOVK            X14, #0x6466,LSL#16
__text:FFFFFFF008FE7FF8                 MOVK            X14, #0x6477
__text:FFFFFFF008FE7FFC                 MSR             #4, c15, c2, #1, X14
__text:FFFFFFF008FE8000
```

***I'm actually curious as to how/why the pattern 0x4455445n64666477 was chosen**

pmap hardening

- Additionally, exception handling code includes checks in order to let PPL handle them in case we're in PPL mode

```
loc_FFFFFFF007A030F4                          ; CODE XREF: sub_FFFFFFF007A02000+100↑j
                SUB             SP, SP, #0x350
                STP             X0, X1, [SP,#0x350+var_348]
                MRS             X0, #4, c15, c2, #1
                MOVK            X1, #0x4455,LSL#48
                MOVK            X1, #0x4454,LSL#32
                MOVK            X1, #0x6466,LSL#16
                MOVK            X1, #0x6477
                CMP             X0, X1
                B.NE            sub_FFFFFFF007A03E10
                LDP             X0, X1, [SP,#0x350+var_348]
                ADD             SP, SP, #0x350
```

PPL mode exception handler

pmap hardening

- It is important to note that everything is still in normal EL1
- This all relies on custom silicon logic implemented by Apple
  - Pretty cool machinery!
- Design is not perfect
  - But it’s likely a work in progress
    - Writing is on the wall: the bar to play iOS game will keep going up exponentially

pmap hardening

- Mostly significant for jailbreakers still
  - At least unlike KPP/KTRR this time it also affects other attackers rather than just pissing off end users
  - Not strictly necessary to violate codesign to advanced attackers
    - Rootkit attacks are definitively trickier to pull off
    - Naive data exfiltration unaffected
- Given arbitrary function invocation (CFI break), it is possible to invoke ppl_dispatch in order to add things to the TrustCache
  - Can load arbitrary code without violating code sign

The future of iOS attackers

The future of iOS attackers

- At the end of the day, we're fighting a losing battle
  - But some battles are being lost faster than others
- I always have a thought at the back of my head telling me memory corruption is going away eventually
  - So far it's been wrong
- Realistically web browsers have so much complexity it's always going to be possible to pull something off
  - Life is going to be ok for attackers focusing on 1-click
    - 0-click might still survive via XSS, but there's probably so many ways to pull that off

The future of iOS attackers

- Jailbreaks are likely going to fade away
    - From an Apple fanboy's perspective it kinda sucks that we're going to lose one of the few things that allows us to thinker with the platform
        - Breaking iOS is in itself an act of curiosity towards the impressive work done by Apple
- Physical attacks will probably still be viable given enough time
    - Can brute force pointer auth for the initial exploitation step

- Individual researchers doing full chains is still a thing in iOS 12
  - But it’s getting tougher and tougher, and at some point you need to take into account mental health costs
    - Important to strike a good work/life balance
      - Too much life means you fall behind the curve
      - Too much work yields burnout
- Eventually we’ll probably all need a new job
  - Hopefully later rather than sooner

Thanks!

Shout out

- Apple for making iOS exploitation fun again
- Advanced iOS attackers out there
  - Please don't hack my phone
    - I'm actually boring enough that you'd regret it anyway
- Brandon Azad (@bazad) for his excellent work and very good writeup detailing the thought processes needed required in order to experiment with unknown security features
  - This process is often ignored in most writeups, and experimentation is key for mounting attacks against these new security features

Shout out

- People trying to get into the iOS game
  - It’s a fun thing to be involved in, don’t let the complexity let you shy away from it! Once you get the hang of it, it all makes sense
- Nano development team
  - Best text editor