

# A Look at Modern iOS Exploit Mitigation Techniques

MOSEC 2017

Luca Todesco @ qwertyoruiopz



# whoami

- security researcher by hobby and trade
- contributed to several public iOS jailbreaks
- make private jailbreaks in my spare time
  - both iOS and PS4 on latest version :)
- i got some german guy on twitter super angry
- i was featured in a mixtape or two



# Typical iOS Exploit Chain

- Entry point
  - WebKit
- Privilege Escalation
  - Kernel
    - Sandbox escape may be required to trigger



# prehistory of iOS security

- iPhone os 1.0: everything ran as root and without code sign or sandbox
- Encrypted OS images (security by obscurity)
- iPhone OS 2 introduces code sign and sandbox
  - To this day the biggest contributors to iOS's security is sandbox
- iOS 5: usermode ASLR
- iOS 6: kernel mode ASLR + kernel address space isolation
- iOS 7: let's casually forget about isolation in the transition to 64 bit(???) - iPhone 7 fixed this
- Strongest defense mechanism is the widespread use of sandboxing



- It is important to note the Apple's security fame mainly comes from their (somewhat) strong use of sandboxing
  - However the rest of the industry is catching up
- Apple's strategy used to be hitting post-exploitation
  - Gaining remote code execution on iOS was just as easy as finding some WebCore UaF and applying generic exploitation strategies
- Escalating privileges on the other hand is quite tricky from a sandboxed context
  - Many would use chains of bugs to first escape sandbox then LPE to kernel



# Moder iOS Security

- Prevalent defense strategy is *\*still\** go against post-exploitation
  - Hardening the big targets: kernel and webkit
    - Kernel Patch Protection and Bulletproof JIT
- Some exploit technique mitigation strategies also got implemented
  - Some heap hardening got rid of the 'zone confusion' /wrong kfree attack: possibly single biggest iOS 10 security win
- Reality is: it appears Apple does not actually proactively push exploit mitigations; they merely wait for Ian Beer to show his new nifty exploit strategies then make minor changes to make them slightly less useful
  - Not the best strategy



# Moder iOS Security

- Increase attacker cost
- Cheapest defense strategy is to go against post-exploitation
  - Some exploit technique mitigation strategies also got implemented, but they are rather ineffective
    - Some are working out OK, e.g. at last, mapping NULL is disallowed!
- Aim for the "usual suspects": kernel and webkit
  - Defending the entirety of user mode is a lost battle to begin with
    - Sandbox is getting tightened nonetheless



# Code Integrity

- Kernel
  - Kernel Patch Protection
    - Enforced with TrustZone / EL3 pre-i7 (WatchTower)
    - Hardware enforced on iPhone 7 (incorrectly called AMCC, rumored to actually to called "SiDP")
- WebKit
  - JIT memory is not writeable directly since iOS 10
    - "Bulletproof JIT"
    - Separate RW map from RX one and hardcode RW address in fuction living in executable-only memory
    - Still one function call with 3 arguments away (JOP/ROP)



# The cold, hard truth

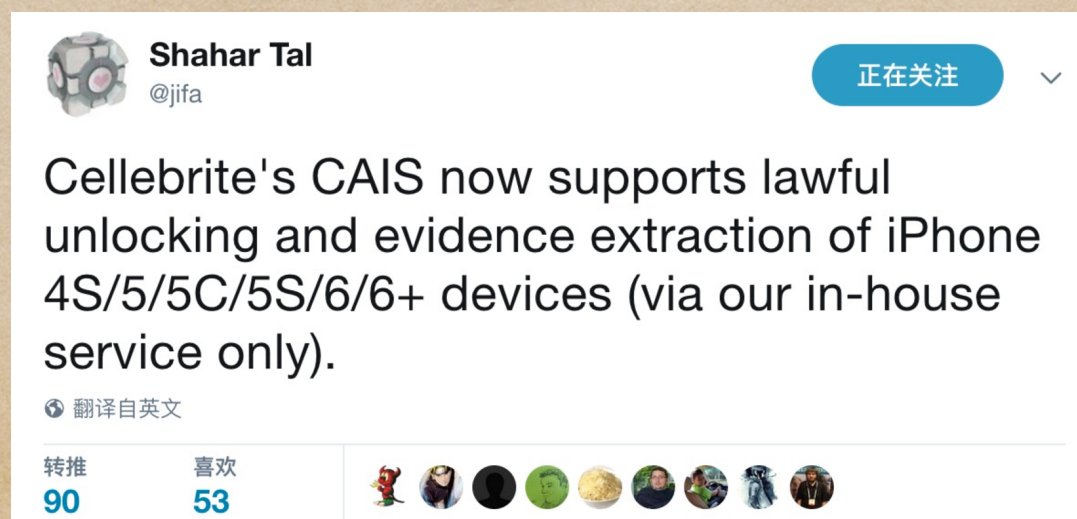
- All these security strategies are ultimately useless
  - Kernel post-exploitation is largely unneeded
    - Only attackers affected are jailbreakers
      - Malware only cares about things userland has access to anyway
        - Define data only `task_for_pid` equivalent (comex's `host_get_special_port` patch for instance)
          - Alter usermode processes
        - Alter physical memory directly




# The cold, hard truth

- All these security strategies are ultimately useless
  - "Bulletproof JIT" is probably the best mitigation out of the bunch
    - Even if currently useless, it will make more sense than KPP in the future
  - Secure Enclave is probably the best design decision ever
    - Attacker cost skyrocketed

• But:




 **Shahar Tal**  
@jjifa 正在关注

Cellebrite's CAIS now supports lawful unlocking and evidence extraction of iPhone 4S/5/5C/5S/6/6+ devices (via our in-house service only).

翻译自英文

转推 90 喜欢 53





# Attacks

WatchTower



# WatchTower

- "Hypervisor" of sorts running in EL3
- Trap IRQS and writes to CPACR\_EL1
- CPACR\_EL1 gets periodically set in EL3 mode to disable floating point execution
- Kernel then re-enables it trapping into hypervisor
  - Const (\_\_\_Text / \_\_\_DATA\_CONST) segment are integrity checked
  - Pagetables also checked
  - A bunch of system registers are also checked
  - TTBR1\_EL1 among others



# TOCTOU

- Altering TTBR1\_EL1 allows us to alter the virtual layout
  - The thing that is actually "used"
- We can remap checked pages in non-checked area
- Including the page containing the write to CPACR\_EL1 (the integrity check trigger)
  - Hook it and branch into shellcode
    - Fix up TTBR1\_EL1 back to valid value, enter hypervisor to our magic value
    - shellcode can be in original pagetables at executable
    - Only PTEs of checked virtual addresses are enforced



# Setting TTBR1\_EL1

- Rewrite first level PTEs, and remap page containing CPACR\_EL1 access
  - Can now hook CPACR\_EL1
- Call gadget to set TTBR1\_EL1
- Works for a few milliseconds
- Reverts back to original pagetables



# Setting TTBR1\_EL1

- A reason for TTBR1\_EL1 to be altered would be reset, since it resets as 0
- Cores actually reset fairly frequently
  - Two causes
    - Idle sleep
      - Used for saving power when CPU is idle
    - Deep sleep
      - Triggered when phone screen has been off for ~30 seconds and no AC power
      - Winocm's kloader hooked into the deep sleep wakeup handler to re-enter boot loader



# Setting TTBR1\_EL1

- After reset, core jumps in RVBAR\_EL1
  - MMU is not operating (everything RWX and addressed as phys)
  - Branch is taken to initialization routine
    - TTBR1\_EL1 is set mirroring the page containing this routine to let code run when enabling translation
    - TTBR1\_EL1 is then set, and a pc relative branch jumps by  $gVirtBase - gPhysBase$ 
      - Jump from "physical" PC to virtual counterpart
  - execution is then resumed



# Setting TTBR1\_EL1

- After reset, core jumps in RVBAR\_EL1
  - MMU is not operating (everything RWX and addressed as phys)
  - Branch is taken to initialization routine
    - Branch target not validated pre-i7
      - on i7, this is privileged code (SiDP not enable yet)
  - Can override to branch to shellcode



# Setting TTBR1\_EL1

- After reset, core jumps in RVBAR\_EL1
  - Branch is taken to initialization routine
    - Branch target not validated pre-i7
- Once in shellcode, overwrite return address of initialization routine
  - Can now execute code after TTBR1\_EL1 is set, but before execution is resumed
  - Change TTBR1\_EL1
  - Now completely bypass kernel integrity check
  - Implemented in Yalu102



# Attacks

iPhone 7 memory protection



# i7 memory protection/"AMCC"/"SIDP"

- Got rid of the hypervisor
  - They actually got rid of EL3 altogether (iBoot is EL1!)
  - KPP performance hit maybe ?
- Code integrity is now enforced by hardware
  - Find a design flaw and it will never be patchable in theory...
    - Doesn't hold well in practice



# i7 memory protection/"AMCC"/"SIDP"

- The security guarantees are different than KPP's
  - You are guaranteed to be unable to write to a range of memory
    - Write-once system registers; access will trigger illegal instruction fault if already written to
  - You are guaranteed that EL1 instruction fetches outside the protected areas will fail
- Guarantees on system register state
  - TBR1\_EL1? :D



# i7 memory protection/"AMCC"/"SIDP"

- You can actually set TTBR1\_EL1 just fine
  - Sort of. More on this later
- Do copy-on-write again
- Cannot however insert new code



# "Data Only" kernel patching

- Previous so-called "Data Only" patches (i.e. pangu9, pangu9.3.3) only attacked sections that were left unprotected by hypervisor
  - Patches weren't data only, shell code was used
  - Security guarantees of i7 memory protection disallow this
- We need truly data only patches
- At most, we can ROP



# Patching strategy

- Function calls from a kext to main kernel functions will pass thru the global offset table
  - Overwrite pointers to redefine functions
- MAC function pointer tables will be used to enforce policies
  - Overwrite with NULL to skip the check
    - Used in Pangu9



# AMFI

- Two patches required
- Bypass codesign
- Control csflags
- MobileSubstrate



# AMFI

- Redefining GOT functions to subvert logic
  - memcmp -> return 0
    - CDHash will always be considered part of platform binary TrustCache
- Redefine a function called early on by the exec hook
  - ROP and return back to the function skipping prolog and function call to avoid recursion
  - Run code after function returns but before control is passed back to caller
    - Can use ROP to alter csflags



# Sandbox

- Need to disable enforcement
  - Single MAC hooks were nulled
    - Pangu9 also did this, but used shellcode to partially disable enforcement for security concerns
- PE\_I\_can\_has\_debugger -> return 1
  - Need to run platform binaries from /var as root



# LightweightVolumeManager

- `_mapForIO` will fail on write to root partition
  - A flag on the partition object will mark it as locked
    - Partition object in heap, remove flag
      - Used in Pangu9
- Additional check since 9.3
  - If writing on root partition and `PE_I_can_has_kernel_configuration` returns 0, fail before even checking the flag
    - Redirect function entry in GOT to return 1 gadget



# mac\_mount

- Will prevent remounting the root partition as R/W
- Pangu9 raced the hypervisor to do this patch
  - New security guarantees prevent this strategy
    - Partition object in heap, remove flag
      - Used in Pangu9
- Checks a flag in the vnode you're trying to mount on
  - If vnode is marked as being root, fail
    - Remove flag, remount reapply -> **bypass check**



# Changing TTBR1\_EL1

- So far, it seems to all be quite easy
- But we can assuming that changing TTBR1\_EL1 is easy
  - In practice, it is
  - In theory, it shouldn't be
- System registers are not protected by new security guarantees
  - Apple obviously knew so tried to do something to mitigate this



# Changing TTBR1\_EL1

- Practice is, from my ROP chain running just after SIDP is enabled, i can still call these and change TTBR1\_EL1
  - Unsure why. Either a off-by-one or a cache issue



# Hijacking code execution on reset

- Code execution on reset has to be achieved somehow
  - The method used on 64-bit devices on yalu102 cannot be used here
- From WatchTower slides:
  - TTBR1\_EL1 is then set, and a pc relative branch jumps by  $gVirtBase - gPhysBase$ 
    - Jump from "physical" PC to virtual counterpart
  - In theory, it shouldn't be
- $gVirtBase$  and  $gPhysBase$  are overwriteable
  - Change  $gVirtBase$  to  $gPhysBase - branch\ target + gadget$
- Remap secure area page containing a pc-relative ref out of page bounds and a indirect branch based on dereferences from it into TTBR0\_EL1 (writeable memory)



# ARM64 ROP

- Can now control the program counter and one register
  - ROP
- ROP is actually quite tricky herre
  - No debugging whatsoever
    - Changing gVirtBase and gPhysBase will corrupt the interrupt handler base
- Our chain must be atomic since multiple cores may run it at the same time, and chain has to be able to get re-run without issues at unpredictable times
- Stack pivoting on a64 can be difficult if you're not creative enough



# ARM64 JOP

- We start with JOP
  - Take a function with a indirect branch based on a dereference from 1st argument close to prolog
    - C++ does this a lot
  - Take a gadget the dereferences  $X0 = *(X0+off)$  then does an indirect branch
    - Can now build a sort of "linked list" that lets me call the function prolog repeatedly



# ARM64 JROP

- We call a function prolog multiple times, but never return from out indirect branch
  - Every time we call it we push a bunch of registers to stack
    - We push many redundant stack frames
  - Set  $x0 = SP + 8$ ,  $x1 = \text{controlled imm}$ ,  $x2 = \text{controlled imm}$
  - jump into memcpy skipping the prolog
  - Overwrite stack frames (except one set, kept for clean return) creating an artificial stack overflow
  - We can now run ROP, then call matching function epilog to pop fake frame back to register
    - Fully atomic, reusable, and function calling convention compliant ROP chain



# ARM64 ROP Tips

- Many gadget candidates will not pop from stack
  - e.g. MSR TTBR1\_EL1, X0; RET
- Will continue recursing on themselves since they're the last x30 value
  - Need to use JOP shim
    - BLR xN; LDP x29, x30, [SP], #16; RET etc.



# ARM64 ROP Tips

- Many gadget candidates will not ret at all
  - Many gadget candidates will have an indirect branch to X8
    - e.g. `mov x0, x29; blr x8`
- Set X8 to `LDP x29, x30, [SP], #16; RET`
  - These are now ROP gadgets
    - (there's plenty on iOS kernels!)



# **Future Attacks**

Outlining strategies to bypass a theoretical future CFI implementation



# Assumptions

- Assuming CFI is coarse-grained
  - No typing enforcement
- Assuming ARMv8.3 “Authenticated Pointers” are used
- Assuming ability to leak authenticated pointers for valid functions



# My magic 8ball is better than yours

- We can speculate that Apple will add control flow integrity in one or two years at most
  - Bulletproof JIT is an early warning for attackers :)
- Let's attack Bulletproof JIT without ROP
  - Writing to JIT is out of CFI security guarantees scope (e.g. do it and your arbitrary code will happily run without any CFI interference)
  - Since we don't have CFI yet all of this is purely theoretical
  - Assuming R/W primitives as starting point



# Attacking Bulletproof JIT

- The strength of Bulletproof JIT is that the writable map for JIT is at an unknown place in the address space
  - Bulletproof JIT is an early warning for attackers :)
- Multiple avenues to figure the address out
  - Mach APIS
  - Brute force
    - We know where executable area is, so we can predict contents
    - Look at every page



# Address Space Oracles

- My previous WebKit exploits have always used the 'dyld\_start' technique to re-execute the dynamic linker on a new memory-mapped Mach-O
  - Allows in-memory execution of unsigned code by relying on JIT
- Need to find dyld base in the easiest possible way
  - Using libdyld was possible but had to parse shared cache and am lazy
- Used shell code in order to do many write() syscalls on a pipe, starting from unslid dyld base, until return value is non-error
  - Using copyin() as read access oracle to determine whether page is mapped or not
  - Essentially what we need to do to bypass bulletproof JIT
  - However writeable map is more randomized than dyld



# Search Space

- All AArch64 iOS devices will have a 0x4000 pagesize in usermode
- Address space usually starts at 0x100000000 due to `__PAGEZERO`
  - We're in the webkit renderer, a platform binary, so `__PAGEZERO` will be 'standard'
- Had never seen it reach higher than 0x200000000 on iOS 9
  - But I did in 10
- Worst case nr. of operations:~524288
  - We can do better than that.



# Reducing the search space

- JIT region is 32MB on AArch64
- We can do 1 read every 32MB and be guaranteed to eventually hit the JIT region mapping
  - $32\text{MB} = 0x20000000$  \*
- Just 256 tries worst-case-scenario needed to scan  $0x1000000000$ - $0x3000000000$



# Finding writeable map

- Exclude executable area from valid search space
- For each mapping found, calculate size
  - Candidates have to be 32 MB minus one page in size
    - First page of JIT region not present in writeable map: this contains the execute-only special memcpy
- Then look at contents to validate that this is indeed the right map
  - Can now write arbitrary opcodes
    - Bulletproof JIT bypassed



# CFI-easy Address Space Oracle

- We must find a function pointer we can arbitrarily replace
  - And it must be used to perform a function call with a controlled argument at an arbitrary point in time
- Additionally we must find a syscall that will copyin/out from/to it's first argument and return success / failure
- And we must be able to detect success / failure conditions
- The write() syscall was used in a non-CFI environment to do this by using simple J/ROP to control multiple arguments; under CFI we need to have the least possible arguments to simplify exploitation



# CFI-easy Address Space Oracle

- Let's look at syscalls.master and look out for user\_addr\_t
  - First candidate in the list is 'chdir'
    - Should work just fine as long as we are able to get an authenticated pointer for it
- We can detect failure by looking at it' errno
  - EFAULT means X0 points to an unreadable address
- All we need now is a pointer to replace for our one-controlled-argument function call
  - Place to look for: C++ objects, GOT, function pointers in \_\_DATA, stack



# CFI-easy Address Space Oracle

- Finding a function call with a controlled argument reachable from JS should not be too difficult but it can be time consuming
  - I didn't have the time to find one in time for this talk, but I'm sure there's plenty around
- A way to simplify this might be to play with stack frames
  - Call some native function that allows reentrance, then upon reentering alter stack frame of caller with r/w primitive
    - Change local variables to control values that wouldn't otherwise be controllable with javascript



# One More Thing

Mach and CFI



# Mach and CFI

- Sending a Mach message gives a lot of control over your process, the kernel and other processes too
- The `mach_msg` function is used for this
  - '`mach_msg`' requires multiple arguments
    - With a function call with single controlled argument primitive it'd be tricky to call it
      - Fortunately, `mach_msg_send` and `mach_msg_receive` just take one argument and will put together the arguments based on data read from that argument



# Mach and CFI

- Another interesting attack relying on Mach involves the functioning of `mach_msg_server`
  - Many apps and daemons are built around this
    - And if they aren't, there's a good chance some frameworks they use will create threads that use this
  - Will receive a message, call a callback, then send a reply
    - Reply is `vm_allocated` at beginning of function and sent after callback returns
      - Reply can be overwritten and upon callback return a controlled mach message can be sent



# Mach and CFI

- mach\_task\_self can usually be easily guessed / hardcoded
- Sending a mach message to the task self port can do a lot!
- Thread context to directly control registers
  - Leak address space layout to bypass bulletproof JIT
  - Reorder address space
    - Collect one authenticated pointer for a given address, then put another page in there



# Takeaways

- Sandboxing and in general post-exploitation mitigation strategies is what made Apple security great
  - Greatness has been Stagnating
  - Apple security needs to become great again to stay ahead of the curve



All fusing is beautiful.  
End the impossible standards.  
All iPhones deserve JTAG1



**Thank you!**