Contents at a glance

Part I: Defensive Techniques and Technologies

The missing documentation for Apple's proprietary security mechanisms

---

Part II: E pur si rompe

A detailed exploration of vulnerabilities and their exploits

B

Darwin 18 (Beta) Changes

V1.6 of this work is being updated to reflect the numerous security changes introduced by Apple in Darwin 18 (MacOS 14/[iOS/TvOS] 12/WatchOS 5). These changes, primarily in code signing and its enforcement, are still in beta at this point (June 2018) and therefore subject to change. From initial examination, however, it is quite clear where Apple is going with them. This appendix seeks to provide a list of the changes visible from analyzing the binaries. This list is by no means comprehensive, and cannot be made so until the sources of XNU 49xx and higher are published by Apple, in or after September 2018.

Mandatory Access Control (MACF)

A new MACF Policy, AppleSystemPolicy (`com.apple.SystemPolicy`), is now in use in MacOS. The policy (identified as 'ASP') hooks `mac_proc_notify_exec_complete` (new in this version), and the mmap(2) hook. It makes upcalls to /usr/libexec/syspolicyd over `HOST_SYSPOLICYD_PORT` (i.e. host special port #29). The daemon (discussed in Chapter 6), now also provides MIG subsystem 18600, with two messages. The messages are used for `notify_32bit_exec` and `notify_32bit_mmap`. The daemon is likely responsible for popping up an alert, as Apple has indicated that MacOS 14 is the last version to support 32-bit binaries. Its database (/var/db/SystemPolicy) is still surprisingly unrestricted by SIP as of beta 2.

MACF Hooks

In addition to `mac_proc_notify_exec_complete` discussed above, `mac_vnode_check_trigger_resolve` is also defined, and is greedily claimed by the `Sandbox.kext`. Triggers are discussed in Volume II (in the chapter dealing with VFS).

Code Signing

Version 0x20500

One of Apple's touted enhancements is the extension of SIP to third-party applications in MacOS. This feature (discussed in WWDC's sessions). This is presently opt-in, and requires signing with two new features: Specifying a runtime version of 10.14.0 or greater (which XCode manages automatically with `-mmacosx-version-min`) and using the new version 0x20500 (i.e. v2.5) signatures. This adds a new flag to the signature (runtime, or 0x10000).

GateKeeper (MacOS)

Application Notary

XCode 10 offers a new "App Notary" feature. As explained in WWDC 2018 session 702 (which also highlights most of the other MacOS 14 changes), the feature submits Developer-ID signed apps, dmgs or .pkgs to Apple, and subjects them to an automated testing process which is meant to detect malware and (possibly later) ensure other forms of policy compliance. The result of this process is a "ticket", which may be left standalone or "stapled" to its item.

When launched from the UI, GateKeeper detects notarized bundles and - at present - allows their opening through a GUI notice. Apple has made it clear that its future plans are to allow only notarized applications, though this might not happen until MacOS 16.

AMFI

- Code signing enforcement can now be controlled on two levels: process and system. This applies to the kernel variables and their corresponding `vm.* sysctl(2)` MIBs. `cs_enforcement_enable` thus now becomes `cs[_process/system]_enforcement_enable`. There is also a new call `cs_executable_wire`.
- The iOS `rwx` restrictions are introduced into MacOS, with specific checks to prevent write and execute permissions from being possible concurrently, unless the process is entitled. Library validation (restricting loaded objects to Apple's own or same team identifier) is also hardened. Several entitlements are introduced for this purpose:

com.apple.security.cs..	Used for
allow-jit	Enable JIT code genertion
allow-unsigned-executable-memory	Enable executable mapping sans signature
disable-executable-page-protection	Neuter code signing checks for process
disable-library-validation	Allow dylibs with different team IDs

- Debugging protection, which was limited to Apple's processes, is now extended to the masses. In order to enable debugging features, once again entitlements are used:

com.apple.security.cs..	Used for
get-task-allow	Willingly give up own task port (debugee)
debugger	Marks own process as debugger
allow-dyld-environment-variables	Force `dyld` to pass variables to signed process

CoreTrust (iOS12)

iOS 12 (beta 2, at the time of writing) introduces another kext, with the bundle identifier `com.apple.kext.CoreTrust`, to support AMFI's kernel operations. CoreTrust's purpose is to thwart the common technique of "fake-signing" (known to jailbreakers as "`ldid -S`" or "`jtool --sign`"), which is often used to deploy arbitrary binaries to a jailbroken device. In this method (shown in the experiment on page 71), a code signature with an empty CMS blob is generated. Because it is not an ad-hoc signature, AMFI passes the blob to `amfid`, but the latter at this point has been compromised by the jailbreak.

iOS 12's AMFI therefore validates a non-empty CMS blob, and then subjects the signature to CoreTrust's evaluation. CT runs several checks against hardcoded certificates, whose strings can be spotted with `jtool --str`, and contents with `-d __TEXT.__const` (looking for the "30 82" DER marker). Stuffing these certificates in `__TEXT.__const` ensures that they benefit from KPP/AMCC protection and cannot be tampered with. CT may further validate the signature policy (in certificate extension fields), and only if the evaluation is successful, does the normal flow (i.e. passing to `amfid`) ensue. This means that, although the daemon might still be compromised, the attack vector is lessened, as binaries would still be required to possess a signature from an Apple CA (root and/or iPhone Certification), with the daemon only relying to `online-auth-agent`.

CoreTrust will likely prove a pain to jailbreakers, but its impact on APTs is dubious, at best. Such targeted malware operates in process, using a privilege escalation and/or sandbox escape to obtain unfettered code execution. Because it already possesses (or exploits an app with) a valid code signature, CoreTrust will play no role in preventing its payload from running and compromising the device data.

SandBox

The iOS ContainerManager (see Chapter 8) makes its MacOS debut. At the time of writing (beta 2), it is unclear how it will be used.

Privacy

TCC is extended to protect not just XPC APIs, but all access to resources - including directly. A new set of entitlements is defined:

com.apple.security.	Used for
`device.[audio-input|camera]`	Video/Audio device access
`personal-information.*`	Access `location`, `addressbook`, `calendars` and `photos-library`
`automation.apple-events`	Allow sending of Apple Events

APFS Snapshot mount (iOS 11.3)

In an effort to harden the root filesystem protections against remounting, Apple has started to use a snapshot mount for the root filesystem, rather than a standard mount. Using `mount(1)` reveals that `/` is mounted over `com.apple.os.update-`*`GUID`*`@/dev/disk0s1s1`. A snapshot mount is a very clever idea for a read-write mount (as it allows reverting to the base snapshot in case of corruption), but in this case the reasoning is likely different. As the snapshot is mounted read-only, the driver does not permit new writes to it, and panics the kernel (complaining "you must have an extent covering the alloced size"). As discussed in Volume II, an extent is a grouping of logical blocks (or parts thereof) where file data is kept.

Nonetheless, this has been bypassed by Xiaolong Bai and Min (Spark) Zheng. In a Weibo blog post they detail their method, which specifically overcomes two hurdles:

- **XNU checks for attempts to remount an already mounted block device:** Bai and Zheng seek to create another mount - this time directly on the block device - but the root vnode's `v_specinfo->si_flags` (as discussed in Volume II) include `SI_MOUNTEDON`, so that the `mount(2)` system call would return `-EBUSY`. This, in itself, is an integrity rather than security precaution. The duo bypasses it by neutering the flags altogether, which enables the mount.
- **APFS.kext is coerced into believing this new mount is not a snapshot:** by copying the the APFS private mount data pointer over from the secondary mount. This pointer (the `mnt_data` field of the `struct mount` in the vnode's `v_mount` field, incidentally at offset 0x8f8) holds filesystem driver private data. When copied from the secondary mount's vnode over the root vnode, it successfully enables new extents to be created and avoids the panic.

While fairly detailed, Bai and Zheng's article nonetheless omits a fine point - The APFS.kext will compare the `v_mount` from every vnode it processes to a field stashed in its private data. Because those vnodes are technically on the root filesystem mount (/) and not the secondary filesystem mount, a mismatch will be detected. This will not cause a panic, but will still fail vnode data access. The kernel log output is inundated with "vp has different mp than fs System" messages from `apfs_jhash_getvnode_stream`. Using `jtool` to disassemble around this message reveals the specifics:

```
morpheus@Zephyr(~)$ jtool2  -d /tmp/com.apple.filesystems.apfs.kext  |
                    grep -B13 -A10 different
Disassembling from file offset 0x24000, Address 0xfffffff00680d000
     ..4c488  BL      _vnode_mount        ; 0xfffffff00688f674
     ..4c490  LDR     X8, [X22, #416]     ; R8 = *(R22 + 416) = (private->v_mount)
     ..4c494  CMP     X0, X8, ...         ;
-------------------------------------------------------------------------------
if (vnode_mount(vp) != private->v_mount) {
-------------------------------------------------------------------------------
+----..4c498  B.EQ    0xfffffff00684c4cc  ;
|    ..4c49c  MOV     X0, X23             ; X0 = X23 (= vp)
|    ..4c4a0  BL      _vnode_put          ; 0xfffffff00688f68c
|    ..4c4a4  LDR     X8, [X22, #192]     ; R8 = *(R22 + 192) = (private->fsName)
|    ..4c4a8  STR     X8, [SP, #16]       ; *(SP + 0x10) = 0x100000cfeedfacf
|    ..4c4ac  ADRP    X8, 2093870         ; R8 = 0xfffffff005b7a000
|    ..4c4b0  ADD     X8, X8, #2439       ; R8 = "apfs_jhash_getvnode_internal";
|    ..4c4b4  MOVZ    W9, 0x143           ; R9 = 0x143
|    ..4c4b8  STP     X8, X9, [SP, #0]    ; *(SP + 0x0) = R8, R9
|    ..4c4bc  ADRP    X0, 2093870         ; ->R0 = 0xfffffff005b7a000
|    ..4c4c0  ADD     X0, X0, #2400       ; "%s:%d: vp has different mp than fs %s\r"
|    ..4c4c4  BL      _printf  ; 0xfffffff0068218e0
-------------------------------------------------------------------------------
_printf("apfs_jhash_getvnode_internal:1323: vp has different mp than fs %s\n",
         private->fsName);
-------------------------------------------------------------------------------
| +--..4c4c8  B       0xfffffff00684c4ec
-------------------------------------------------------------------------------
 } else {
-------------------------------------------------------------------------------

+-+->..4c4cc  MOV     X0, X23             ; X0 = X23 (= vp)
  |  ..4c4d0  BL      _vnode_fsnode       ; 0xfffffff00688f584
  |  ..4c4d4  MOV     X20, X0             ; X20 = X0 = 0x0
  |     ...
```

Note the check (in 0xffffffff00684c498) comparing the result of `vnode_mount` with a value from `[X22, #416]`. The former function (defined in XNU's bsd/vfs/kpi_vfs.c) merely returns `vp->v_mount`, and the latter is a value in the private data. Xnooping around reveals that it matches the secondary mount's vnode. The value therefore needs to be overwritten to the original root node's `v_mount`, to allow `vnode_fsnode()` to be called and retrieve the `vp->v_data`.

While Bai and Zheng's method works, there are two finer points still left to address:

- **APFS reverts to the initial filesystem snapshot on boot:** Meaning that changes to the root filesystem will still fail to persist across reboot. This can be trivially fixed by creating a new snapshot, and renaming it to match the initial snapshot name (i.e. `com.apple.os.update-`*`GUID`*`@/dev/disk0s1s1`). The process (detailed by Uamng Raghuvanshi in a blog post) is straightforward using `libsystem_kernel`'s `fs_snapshot_[create/rename](2)` wrappers over the `fs_snapshot` (#519) system call. Although the system call normally requires an entitlement, at this point the jailbreak would have kernel credentials. Example source code of a fake-entitled binary to selectively snapshot the system can be found in the QiLin download page.
- **The secondary mount method tends to be unstable** and can lead to a panic (kernel data abort) on the copied mount data pointer if the mount is unmounted. A rigorous method to bypass would involve the re-creation, rather than duplication of the private APFS mount data. With the data format being entirely undocumented, however, this is quite challenging. Still, for developer-oriented jailbreaks, this solution proves sufficient.

The QiLin toolkit (revision 6 and later) contains an implementation of Spark's method, with minor enhancements. These are transparently called through QiLin's `remountRootFS(void)`. LiberiOS and LiberTV, both using the toolkit, thus also now support this method and are compatible with iOS 11.3.1 and earlier.