WEN ETA JB? A 2 million dollars problem

Eloi Benoist-Vanderbeken and Fabien Perigaud
eloi.benoist-vanderbeken@synacktiv.com
fabien.perigaud@synacktiv.com

Synacktiv

Abstract. iPhones are generally considered to be the most secure in their category. Indeed, unlike the Android ecosystem, Apple has the advantage of controlling both the hardware and the software part of its phones. As a result, it is possible for them to directly take advantage of the latest advances in hardware as soon as they are available.

This article aims at showing that the $2M bounty from Zerodium for a remote iOS jailbreak might not be overrated.

1 Introduction

Since Apple released its first iPhone back in 2007, there has been a growing interest in the ability to jailbreak it, allowing the user to gain root access to his device and install custom applications, without relying on the official Apple Store.

While first jailbreaks just consisted in flashing patched firmwares through legitimate mechanisms, Apple quickly introduced protections preventing this trick. Jailbreakers then started looking for security vulnerabilities to gain code execution on the device. Most of these vulnerabilities required plugging the iPhone on a computer via USB and booting it in a special mode for the vulnerability to be exploited.

For the sake of simplicity, the website jailbreakme.com appeared shortly after, allowing to jailbreak a vulnerable iPhone by simply visiting a web page through the Safari browser. This website had multiple versions exploiting different vulnerabilities, starting from a buffer overflow in libtiff for iPhone OS 1.x directly allowing a jailbreak, to full exploit chains on iOS 4.x exploiting an initial bug to gain arbitrary code execution, followed by another one that elevates privileges to root (Star [19] and Saffron [18] jailbreaks). The latest jailbreak running through Safari, TotallyNotSpyware, targets 64-bits iOS 10.x devices by chaining a Safari and a kernel exploit.

While these websites have created a huge fan base of iPhone users, they also provided free stable exploit chains which could easily be reused

by malicious parties to gain arbitrary code execution with high privileges on the targeted devices.

Over the years, Apple fixed a bunch of vulnerabilities and added several mitigations to harden its devices, on both the software and hardware parts, to prevent attackers or `jailbreakers` from exploiting the browser, the kernel, and gaining persistence on the `iPhones`. Nowadays, `jailbreaks` are usually simple applications embedding a kernel exploit and patching some userland services to allow unsigned code to run on the device; the whole chain including the browser exploit and the persistence mechanism having become too costly in time to develop.

On the offensive side, vulnerability brokers announce very high prices to acquire such a full exploit chain: Zerodium recently raised the bar to 2 millions dollars [38]!

In this article, we will detail all the barriers an attacker has to defeat to obtain a remote `iOS` jailbreak.

2 Definitions

2.1 iOS Sandbox

The sandbox is the first obstacle faced by an attacker. It is implemented in the kernel and is able to block or allow specific actions and to grant privileges. Each process can have a unique sandbox profile. A sandbox profile is a set of rules that describes how to make the decision to allow/grant each filtered action/privilege. If the action is forbidden, the sandbox profile can choose to report the violation in the logs. Sandbox rules have access to specific variables such as the process entitlements, the *uid*, the arguments passed to the filtered function, the type of the kernel build (debug or release), etc.

Sandbox rules are written in SandBox Profile Language (SBPL), a Scheme-based language, which is compiled into a bytecode interpreted by the kernel. In iOS, the sandbox profiles are stored directly in the kernel in read-only pages. It is technically possible to create other sandbox profiles but this doesn't seem to be used by iOS. The sandbox profile is usually specified by name in the executable entitlement, it could also be provided by the parent process by using the undocumented *posix_spawn* attributes or by using *mac_execve* but this doesn't seem to be used in iOS.

2.2 amfid

On iOS, a binary can either be signed by Apple, which is the case for all classic applications, or have its hash directly embedded in the kernel, which makes it a platform binary. In the first case, part of the signature verification is delegated to `amfid`, a userland daemon.

This makes sense because the signature verification process is very complex: `amfid` doesn't just check whether the signature is cryptographically correct, it also has to check that the certificate is not revoked, that the provisioning profile—if any—is correct, that it has been approved by the user in case of an enterprise profile, etc.

2.3 TrustedBSD MAC Framework

A lot of the security mechanisms in XNU are built around the TrustedBSD *Mandatory Access Control Framework* (abbreviated MACF in the XNU sources and in this article). To quote the TrustedBSD documentation:

> Mandatory access controls extend operating system access control policy by allowing administrators to enforce additional constraints on user and application behavior. The TrustedBSD MAC Framework is a kernel programming interface allowing loadable modules to augment the system security policy in order to implement mandatory access control in a flexible manner.

Before executing sensitive functions, the kernel will call the MACF hooks to let them decide if the current process is authorized to do the operation or not.

2.4 Mach subsystem

The iOS kernel, XNU, is a hybrid kernel built on BSD and Mach. Basically, the Mach subsystem is built around ports receiving messages. A port is an interface to a service provided by a server. The server owns the unique receive right on the port and clients use their send right to send messages to the server. Mach messages can contain payloads of almost arbitrary size and port rights. In userland, a port is identified by a port name which is basically an index that is similar to handles in Windows.

Depending on the context and by metonymy, the word port can design a port name or a port send/receive right. In this article, the word port will be used to refer to a port send right.

A Mach service can be provided by a userland process or by the kernel. In the latter case, the port identifies a kernel object which can be a task

(the Mach representation of a process), a thread, the host, etc. A task port can be used to read and write a process memory and a thread port can be used to suspend, resume a thread and to modify its context. As thread ports can be retrieved via a task port right thanks to the *task_threads* function, having a task port was sufficient to completely take over a process until Apple added some hardening...

2.5 Dyld shared cache

To save on RAM and speed-up the loading and runtime of libraries on `iOS` and `macOS`, system libraries are merged into a big cache named the *dyld shared cache*. The dependencies are resolved offline to eliminate the indirect branches and memory accesses, thus accelerating the code. As the shared cache code is not position independent, it is loaded at the same virtual address in all the processes (like Windows and unlike Linux). This considerably eases the exploitation of privileged services by eliminating all the benefit of the ASLR against code reuse attacks. At least up to `A12`...

2.6 Entitlements

To provide a more granular permission level and more in-depth security, Apple introduced the `entitlement` concept. Signed, applications and executable files can be bound with `entitlements`, which represent different permissions at the application level. An `entitlement` is characterized by a name, such as `com.apple.developer.networking.networkextension`, and by a value that is often a boolean, but can also be another structured type (array, dictionary...).

`Entitlements` are checked by services and the kernel in order to choose whether the requested action is allowed. For example, the `entitlement` `com.apple.private.kernel.global-proc-info` is mandatory to access any `API` that lists information about running processes, `task_for_pid-allow` gives a process the right to get the task of another process, `get-task-allow` is used to authorize debuggers to attach to the process which has this `entitlement`, etc.

Third party developers or entities such as companies can only sign binaries with a limited set of `entitlements`.

3 Initial RCE

3.1 Targetting the web browser

Usually, exploit chaining starts by gaining code execution in the context of a process exposed to the outside world. There are several endpoints running on an iPhone which can be directly targeted in a zero-click or one-click scenario, such as:

— Baseband: this is the component handling all the radio communication. Attack surface is quite huge and it usually lacks the latest software mitigations.
— SMS/MMS on the application side: the applications handling these messages can be reached without user interaction, but exploiting a vulnerability can be quite complicated because of the lack of a scripting environment.
— Documents parsers (PDF, Office files): the applications handling documents usually provide means to manipulate the memory layout but still lack a real scripting environment.
— Web browsers: this is the easiest target to exploit because of the ability to execute arbitrary `JavaScript` code, leading to strong memory manipulation and action chaining primitives.

`Safari` is `iOS` default browser, based on the open-source `WebKit` engine [2]. During the last few years, a huge effort has been performed to introduce new mitigations complicating exploitation, which will be the main topic of this section.

3.2 Gaining read and write primitives

One of the first goals in the process of making an exploit is to gain arbitrary read and write (R/W) primitives to the process memory, in order to achieve effective code execution. Gaining these primitives is usually achieved by abusing one of the objects allowing arbitrary data storage in a backend buffer, such as `JavaScript Array`, `ArrayBuffer` and `TypedArray` objects.

`ArrayBuffers` and `TypedArrays` are the best targets, since they provide methods to store various data types directly in the backing buffer, such as 8, 16 and 32 bits signed and unsigned integers, as well as floats.

On the other hand, `Arrays` can contain mixed content, and thus values are usually stored in their `boxed` form. In `WebKit`, `JavaScript` values are stored as `JSValues`. A `JSValue` is a 64-bits integer, where the upper

16-bits are used as a marker defining the encoded data type. The following types can be encoded in JSValues:

— Integers: the 16 upper bits are set to 1, and a 32-bits integer is stored in the lower 32-bits.
— Pointers: the 16 upper bits are set to 0, and a pointer is stored in the remaining 48 bits, which is sufficient because `iOS` on `AArch64` only addresses 39 bits of memory.
— Floats: any other value for the 16 upper bits. The actual float value is computed by substracting `1 « 48`.

For the sake of optimization, when all the elements contained in an `Array` share the same type, they are stored in their `unboxed` form (e.g. a float is directly stored as a float, and an integer as an integer). Thus, an `Array` containing only floats is also a good target, since you can almost fully control the 64-bits of data written in the backing buffer.

Therefore, gaining arbitrary R/W primitives could be achieved by forging one of these objects in memory, or modifying an existing one (by changing the pointer to their backing buffer by the pointer where we want to read or write data).

3.3 Here comes the Gigacage

To prevent using these objects to gain R/W primitives, `WebKit` introduced a new allocation zone called the `Gigacage`.
For each dangerous object type, a 32 GB zone is allocated, and all the backing buffers are allocated in this zone. Finally, a 32 GB `runway` zone is also allocated, with `PROT_NONE` protection.

At the time of writing this article, there are only two kinds of `Gigacages`:

— `Primitive`: this `Gigacage` contains the `ArrayBuffers` backing buffers and the `WASM` allocations.
— `JSValue`: this one contains the `Butterflies` allocations, which is the structure used as the backing buffer for various objects, such as `Arrays`.

Using a 32 GB size means that if one can corrupt the size (which is stored as an unsigned 32-bits integer) of an object storing the biggest items (up to 8 bytes), the access will still land in one of the zones or in the `runway` zone, ensuring that no harmful data would be corrupted. As the `Primitive` cage can receive arbitrary values, the `runway` zone is allocated just after it.

A third kind of `Gigacage`, `String`, was originally introduced to hold the `String` objects buffers, but removed in May 2018 [10] because of performance issues. This decision has a strong impact on security: `String` objects can now be used to build an arbitrary read primitive. However, as `Strings` are immutable in `JavaScript`, they cannot be used to build a write primitive.

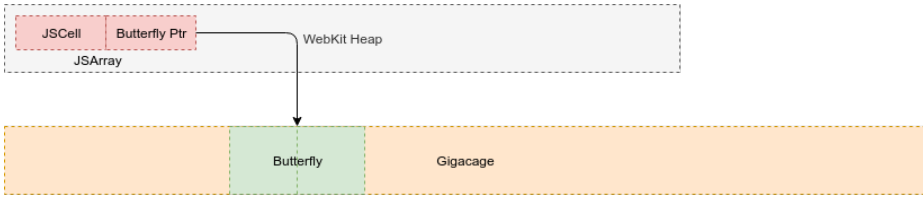

Fig. 1. `Array` object pointing to the `Gigacage`.

Using these new allocation zones would be meaningless if the pointers were simply used as-is when getting or setting data in the backing buffers. In fact, whenever one of these `Gigacage` pointers (named `CagedPtr`) are accessed, its value is masked to make it an offset into a `Gigacage`, and added to the `Gigacage` base pointer corresponding to its kind (see listing 1).

```
void* caged(Kind kind, void* ptr) {
    void* gigacageBasePtr = basePtr(kind);
    if (!gigacageBasePtr)
        return ptr;
    return gigacageBasePtr + (ptr & mask(kind));
}
```

Listing 1. Simplified `caged` function.

Using this access method ensures that, even if the backing buffer pointer is corrupted to point outside of a `Gigacage`, read and write accesses will be performed inside a `Gigacage`.

Attackers wanting an arbitrary write primitive will thus have to find other objects to corrupt, and might end up with a less convenient primitive.

3.4 Shellcode execution

After having gained arbitrary R/W primitives, the next goal usually consists in achieving arbitrary shellcode execution. While this seems to be an easy task, such an execution in an `iOS` process is not just a matter

of allocating RWX memory. Indeed, for every sandboxed process, the XNU kernel forbids mapping RWX memory. Processes needing this feature for JIT purpose, which requires dynamically writing code that will be executed later, must have the specific entitlement dynamic-codesigning. This entitlement allows the process to map RWX memory only once, using the MAP_JIT mmap flag.

Despite this hardening, executing a shellcode in WebKit is just a matter of finding the JIT page location, which is an exported symbol (startOfFixedExecutableMemoryPool), and overwriting a jitted function code with a shellcode. While this can still be done to get shellcode execution on MacOS, iOS introduced new mitigations to prevent direct usage of the JIT page.

Separated WX Heaps. On devices with a SoC prior to A11, a second mapping of the JIT page is performed at a random address. This second mapping is marked as RW, while the first one is marked as RX, and a jitted function is created to transparently write in the RW mapping when trying to copy data in the RX one. Finally, this function is marked as execute-only to prevent getting the RW mapping address by reading its code.

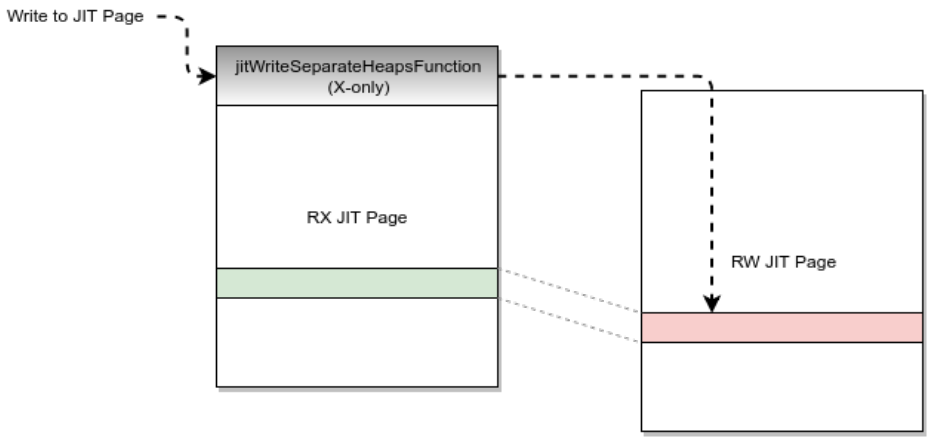

Fig. 2. JIT pages in memory.

With this mitigation in place, writing to the JIT page should be quite painful: there is no way to find the page location to use it with the write primitive. However, the execute-only function created to handle writes in the JIT page is marked as export, allowing an

attacker with a read primitive to find its address by looking at the `jitWriteSeparateHeapsFunction` symbol. Therefore, the only thing left before shellcode execution is finding a small `ROP` chain to call this function with arbitrary parameters and flush the cache, which should not be so difficult given the quantity and size of modules loaded in `Safari`.

`A11` and `A12` SoCs. With iPhones 8 and X came the `A11` SoC introducing new hardware features to harden even more the `JIT` page usage. Indeed, a new system register `S3_4_c15_c2_7` allows dynamically and atomically changing permissions on `RWX` pages in the process. Therefore, `WebKit` uses macros to change permissions to `RW`, performs a `memcpy`, then restores permissions to `RX` (see listing 2).

```
static inline void* performJITMemcpy(void *dst, const void *src,
    size_t n)
{
[...]
    if (useFastPermisionsJITCopy) {
        os_thread_self_restrict_rwx_to_rw();
        memcpy(dst, src, n);
        os_thread_self_restrict_rwx_to_rx();
        return dst;
    }
[...]
}
```

Listing 2. Writing to the `JIT` page on post-`A11` devices.

These macros write magic values present in memory, at addresses `0xfffffc110` and `0xfffffc118` respectively, in the aforementioned system register.

Unlike the function used to write in the `RW` mapping on older devices, the `performJITMemcpy` is not exported and is usually inlined in the function using it. Therefore, there is no easy way to simply call it to perform a write in the `JIT` page.

In order to write in the `JIT` page on A11 devices used in recent exploits [30], one must build a `ROP` chain which jumps in the middle of a higher-level `JIT` function: `JSC::LinkBuffer::copyCompactAndLinkCode`. This requires setting up the right context to prevent crashing and setting the stack cookie to allow the function to correctly reach its end, which is not very generic.

On `A12` devices (`iPhones XS` and `XR`) however, this trick can't be directly used because of the new `PAC` feature, and there are no known working public exploits at the time of writing this article.

3.5 PAC: Pointer Authentication Code

PAC is a feature introduced in ARMv8.3-A to prevent pointers modification or reuse. It basically adds new instructions to sign and verify both instruction and data pointers, either using a context or not, a context being either a fixed arbitrary value or a register content. This theoretically allows signing every pointer before storing them in memory, and verifying them before using them again.

The specification allows two different keys (A and B) for each pointer type ((I)nstruction and (D)ata), as well as a fifth key used to compute a MAC on arbitrary data.

iOS on AArch64 uses a 39-bit addressing scheme, which leaves 24 bits for a signature, because 1 bit is used to keep the pointer origin information (userland or kernel). In practice, in userland, it seems that 16 bits of the pointer are used to store the signature and the 17th stores the value of the original extension bits, leaving the 8 upper bits set to 0.

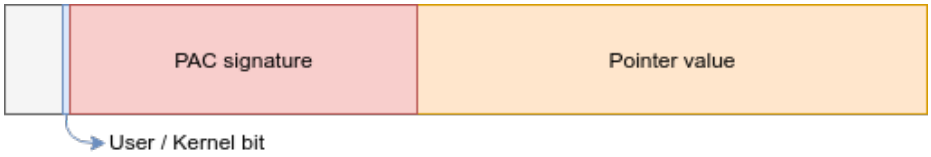

Fig. 3. Signed pointer using PAC.

While the ARMv8.3-A specification indicates that QARMA can be used as the signature algorithm, the one used in iOS is unknown. About the instructions supporting PAC, the following categories exist:

- PAC*: add signature to a pointer.
- AUT*: check and remove signature from a pointer.
- XPAC*: remove signature from a pointer, without checking it.
- RETAA / RETAB: check X30 (register holding return address) using SP as a context and return.
- BR* / BLR*: check signature and branch (or branch and link).
- LDRAA / LDRAB: check signature and load from memory.

Listing 3 shows a use of such instructions in Safari for A12 devices.

- PACIBSP is used to sign X30 with instruction key B and SP value as context.
- BLRAA is used to authentify X8 with instruction key A and X9 as context, then perform a branch and link.
- RETAB is used to authentify X30 with instruction key B and SP value as context, then return.

```
WebCore:__text:189D79A50    PACIBSP
WebCore:__text:189D79A54    STP X29, X30, [SP,#var_10]!
WebCore:__text:189D79A58    MOV X29, SP
WebCore:__text:189D79A5C    BLRAA   X8, X9
WebCore:__text:189D79A60    LDP X29, X30, [SP+0x10+var_10],#0x10
WebCore:__text:189D79A64    RETAB
```

Listing 3. Function using PAC instructions on A12 devices.

As every function saving the link register on the stack now uses PAC instructions, it is no longer possible to build a ROP chain without being able to sign arbitrary pointers. However, PAC usage in WebKit is not fully complete yet: there is a huge use of instruction pointers signing, but it still lacks data pointers signing. Enabling such a feature would make it really harder to corrupt the JavaScript objects backing buffers to gain the initial R/W primitives.

Regarding a potential bypass, the initial Safari release in iOS 12.0 used pointers signed with zero context in JSValues vtables (see listing 4), making them vulnerable to pointers substitution attacks: an attacker could replace a pointer in the vtable with another zero-context signed pointer providing him with a new primitive. This was however hardened in version 12.1.

```
AND             X8, X8, #0xFFFFFFFFFFFFC000
LDR             X8, [X8,#0x3ED8]
LDR             W9, [X0]
LDR             X8, [X8,#0x100]
AND             X9, X9, #0x7FFFFFFF
LDR             X8, [X8,X9,LSL#3]
LDR             X8, [X8,#0x40]
LDR             X8, [X8,#0x48]
BLRAAZ          X8
```

Listing 4. Zero-context signed pointer called in JSValue vtable.

Brandon Azad, from Google Project Zero, presented a weakness in the current PAC implementation allowing him to forge arbitrary zero-context signed pointers given an execution primitive [9]. Although his attack targets the XNU kernel, the same patterns can also be found in userland.

When changing a pointer signature, e.g. from a context-signed pointer to a zero-context signed pointer, the compiler produces the following code:

```
LDR             X10, [X9,#0x30]!
MOV             X11, X9
AUTIA           X10, X11
PACIZA          X10
STR             X10, [X9]
```

Listing 5. Context to Zero-context signed pointer conversion.

When the signature is invalid, the `AUTIA` instruction removes the bad signature from the pointer, and indicates a failure by changing bits 61 and 62, as explained in the `ARMv8 Architecture Reference` [1].

```
if ((PAC<54:bottom_PAC_bit>) == ptr<54:bottom_PAC_bit>) && (PAC
    <63:56> == ptr<63:56>)) then
    result = original_ptr;
else
    error_code = keynumber:NOT(keynumber);
    result = original_ptr<63>:error_code:original_ptr<60:0>;
```

Listing 6. PAC Auth pseudo-code.

When the `PACIZA` instruction is executed on the invalidated pointer, it computes the correct signature for the lower 39-bits of the pointer, and if there is an error in the upper bits, flips bit 54 in the signature.

```
PAC = ComputePAC(ext_ptr, modifier, K<127:64>, K<63:0>);

// Check if the ptr has good extension bits and corrupt the pointer
    authentication code if not;
if !IsZero(ptr<top_bit:bottom_PAC_bit>) && !IsOnes(ptr<top_bit:
    bottom_PAC_bit>) then
    PAC<top_bit-1> = NOT(PAC<top_bit-1>);
```

Listing 7. PAC AddPAC pseudo-code.

Thus, this behavior can be abused to forge zero-context signed pointers, by retrieving the output of a pointer conversion and flipping a single bit!

4 Privilege escalation and Sandbox escape

Once initial RCE is achieved, an attacker still has a lot of work before completely compromising the system. The usual way to achieve this is to :

1. Attack a service to land in a less sandboxed process (optionally).
2. Exploit a vulnerability in the kernel to create a R/W primitive.
3. Use the kernel R/W primitive to gain arbitrary entitlements/privileges.
4. Since `iOS 12`: Find a way to bypass CoreTrust.
5. Patch or replace `amfid` to bypass code signature.

There used to be userland-only `jailbreaks` [3,6] but we will see that Apple made considerable efforts to complicate userland-only `jailbreaks` as much as possible.

4.1 Escaping the sandbox

The sandbox inner workings and the way the bytecode can be decompiled were already described in many papers [15, 20, 26]. In this article we will focus on the previous sandbox bypass, and how the sandbox was hardened by Apple.

The sandbox has two main roles: to reduce the data or the features available to an application but also to reduce the OS and services attack surface. Over the years, Apple expanded the sandbox capabilities by filtering new functions and by applying it to more services, more aggressively.

The kernel ensures that every `non-platform binary` process is sandboxed, by default with the *container* sandbox profile. Since `iOS 10`, in addition to the process specific sandbox profile, every userland process is sandboxed with a default system-wide profile which is evaluated first. That means that even if an attacker manages to achieve arbitrary code execution in a root unsandboxed service, he will still be sandboxed.

The system sandbox is, however, quite permissive and Apple tries to tailor a specific sandbox for as many privileged services as possible. In `iOS 9`, 117 profiles were defined, whereas in `iOS 12`, there are now 193. More and more services which weren't sandboxed now are. For example, after Mark Dowd exploited `AirDrop` [21], Apple added a specific sandbox profile for the associated service, `sharingd`.

Profiles are also more and more restrictive. For example, the `WebKit` rendering process (`com.apple.WebKit.WebContent`) used to be able to connect to the network, because the sandbox was including all the default system sandbox, but also because of compatibility issues with the `AppCache` feature. Now, since changeset 229093 [4], published on February 18th 2018, all network accesses are forbidden.

Apple also hardens its sandbox by adding new `MACF` hooks: `iOS 9` had 305 hooks whereas `iOS 12` now has 24 extra hooks. Some of them cover new functionalities like `skywalk` [28] and `apfs snapshots` [27] via, amongst others, the `skywalk_flow_check_connect` and `mount_check_snapshot_create` hooks. Others were created to block known vulnerabilities such as the process enumeration vulnerability discovered by Stefan Esser [22] which is blocked by the `proc_check_get_cs_info` hook. Still, others, like `socket_check_ioctl`, harden existing functions.

All this hardening means that it becomes more and more necessary to attack a userland service before being able to trigger a kernel vulnerability.

4.2 Userland exploitation mitigations

Services used to be the weakest point of the iOS ecosystem. There is a lot of them and they are often closed source, therefore they are likely to be less reviewed (compared to the kernel). However, even with multiple vulnerabilities, it is not that easy to escape the iOS sandbox...

NX. NX is aggressively deployed in iOS. Except for the WebKit renderer process, processes on iOS can't map RWX pages. There is no /bin/sh or interpreter on the device, so basic ret2libc techniques won't work either. Logical vulnerabilities or code reuse attacks are mandatory to exploit a service.

ASLR. Like all modern OS, iOS has ASLR. However, even on recent 64-bit architectures, it is still possible, under certain conditions, to spray the heap or the page allocator to get data at a known address. This was used in several exploits [7, 14, 31]. To our knowledge, this is still the case on the latest iOS/macOS versions.

PAC. On A12 devices all the platform binaries are compiled with PAC support. However, to this day, Apple doesn't allow developers to push binaries with PAC support to the store. Because process without PAC support cannot use signed pointers, on A12 devices the shared cache is loaded at two different addresses, one with signed pointers, the other without signed pointers. This means that an application on the store or not compiled with ARM64e support will not be able to reuse shared cache addresses or to use PAC instructions to sign pointers. This greatly complicates the exploitation of daemons.

For the moment, the A key used to sign function pointers is shared across all the processes that support PAC. This has been exploited by Ian Beer [13] to forge arbitrary signed pointers to exploit privileged daemons. This will probably change in the near future as signed pointers are already segregated in dedicated sections (__auth_ptr, __auth_stubs and __auth_got) so it would be easy for Apple to just resign those sections for different keys without wasting too much memory. Different keys could then be used for different privilege levels: platform binary/third party, root/mobile, etc.

Before claiming PAC bypass, it is important to keep in mind that this mitigation is still young and that a lot more could be done with it ; most notably by signing data pointers, or multiplying keys.

Mach API. A standard strategy to exploit a service was to force the server to send its task port to the attacker in a Mach message. The received port could then be used to call arbitrary functions under the exploited service identity. Apple was well aware of this method and, in `iOS 10` (not `macOS`), it started to restrict how and by whom a task port could be used. Indeed, only a platform binary could use another platform binary task port. This didn't block a platform binary from exploiting another one, which was a shame as the `WebKit` renderer is actually a platform binary, but it provided a good protection against jailbreaks relying on an application for their initial code execution. However, the protection wasn't sufficient as it was only focused on task ports, it was still possible to manipulate threads via thread ports. Brandon Azad used this weakness to exploit the `ReportCrash` service which gave him the ability to get arbitrary task ports, `ReportCrash` being one of the very few binaries to possess the `task_for_pid-allow` entitlements [8].

Now, Apple blocked this method by filtering both tasks and thread ports so even if a non platform binary task manages to get a privileged task or thread port, it cannot do much with it. This new protection doesn't just mitigate logic vulnerabilities that directly give privileged task or thread ports, it also significantly raises the effort needed to exploit classical memory corruption vulnerabilities in a privileged service, as there are not many options left to the attacker other than to `ROP/JOP`.

4.3 Code signature

One of the goals of an attacker is to be able to run arbitrary code on a device with arbitrary privileges. It is theoretically possible to achieve this in-memory, but it is easier to just have the ability to launch arbitrary executables with arbitrary entitlements and let the system work as intended instead of manipulating a lot of undocumented or unstable kernel objects or `APIs`.

However, before `iOS 12`, all an attacker had to do to completely bypass Apple code signature mechanism was to take control of `amfid` or to impersonate it. Even if the kernel tried to mitigate this by checking the identity of the daemon passed by the kernel in the Mach message trailer, this was easily circumvented by manipulating `amfid` [6, 12].

In `iOS 12` however, Apple decided to remedy it by adding an extra redundant check in kernelland with the new `CoreTrust` kext [27]. This kext will validate the signature, the whole certificate chain, and will ensure that the root CA is one of three hardcoded ones. If the signature is validated by `CoreTrust`, the kernel will continue the classical verification process by

calling amfid. Otherwise, it will directly reject the binary. Moreover, by doubling the checks, CoreTrust decreases the probability of an exploitable bug in the signature verification process.

This means that to bypass Apple code signature, a CoreTrust bypass is mandatory. It could be a vulnerability directly in CoreTrust to bypass the code signature or to directly add the hash of the binary in the platform binary hashes list. An attacker could also just use a valid or stolen developer certificate.

4.4 Kernel protections

PXN/PAN. Privileged eXecute Never (PXN) and Privileged Access Never (PAN) are ARM equivalent to Intel Supervisor Mode Execution Prevention (SMEP) and Supervisor Mode Access Prevention (SMAP). They can be used to prevent the kernel from reading, writing or executing user memory pages. On old 32-bit devices, this technology was emulated by having the userspace unmapped when the kernel was executing, this emulation was removed on early 64-bit devices (from iPhone 5S to iPhone SE) until PAN and PXN were introduced in the iPhone 7.

This complicates the development of a kernel exploit because the attacker cannot use its own address space to store data or code.

Watchtower/KTRR. Watchtower (also called Kernel Patch Protection, KPP) and its successor RoRgn/KTRR are designed to circumvent kernel patches and kernel code injection.

Watchtower was introduced in iOS 9 for all the pre iPhone 7 arm64 devices. Its code runs at the secure monitor privilege level (EL3), the highest privilege level on ARM and periodically checks the state of the kernel. It has been bypassed because it is a passive mechanism and relies on the kernel to be called. All an attacker has to do to bypass it is to make sure that the kernel is clean before calling Watchtower and to find a way to regain code execution after Watchtower execution [23, 37]. Even if the second step, regaining code execution, can be made more and more difficult by Apple, it is just a cat and mouse game that the attacker is sure to win with enough effort. That's why Apple introduced RoRgn/KTRR in the A10 processor, first used in the iPhone 7 on iOS 10.

RoRgn/KTRR (allegedly for Read-only Region/Kernel Text Region Range) are built on hardware extensions made by Apple, probably in the memory controller of their SOC. At each boot and after each processor wakeup, very early in the code, RoRgn and KTTR registers are set and

locked via a MMIO [32]. RoRgn is used to mark a physical region of the memory as read only and KTTR is used to restrict the range of physical region mappable as executable in kernel mode (EL1). Any attempt to write in the RoRgn region or to execute code outside the KTTR region in EL1 will panic the phone. Of course the KTTR range is included in the RoRng one so it's impossible to patch the kernel code. Because KTTR was not correctly reset after a deep sleep, Luca Todesco was able to modify the kernel mapping and thus to bypass KTRR [5, 34]. After Apple patched this flaw, no public exploit has been released that bypass KTRR.

KASLR. Kernel Address Space Layout Randomisation or KASLR was introduced in iOS 6. The basic idea of KASLR is to hide the kernel memory structure to the attacker. As a lot of exploits will need the kernel to access data, either to execute ROP chains or to create fake objects, PAN will force an attacker to place controlled data in the kernel and the ASLR will force him to find a way to leak its address.

In iOS, KASLR is quite weak and it is possible to generically bypass part of it by spraying data in the kernel, as demonstrated by Qixun Zhao [39].

PAC: Pointer Authentication Code. PAC, introduced in section 3.5, is also used in the kernel. This forces the attacker to perform data only exploitation or to leak and reuse authenticated pointers. A technique for forging arbitrary kernel authenticated pointers has been described by Brandon Azad [9] and then patched by Apple. Blocking arbitrary kernel code execution significantly increases the cost of developing a full jailbreak or to attack other security mechanisms like codesigning, SEP, rootfs protection etc. However, from an attacker point of view, having arbitrary kernel code execution is not necessary to get access to all the data stored on a phone, having arbitrary read or write is sufficient. Apple is well aware of that and is trying to force an attacker to get code execution to be able to modify some sensitive kernel values.

4.5 PPL/APRR

PPL, which meaning is unknown, is in direct line with RoRgn/KTRR as it is a hardware protection designed to block an attacker from performing sensitive operations even though he might have gained arbitrary kernel R/W primitives. Instead of permanently blocking write access to a given physical region like RoRgn, PPL is more subtle and only blocks write access to virtual pages unless a specific MSR is set to a specific value

(0x4455445564666477) [29, 35]. Because PAC blocks arbitrary function calls and RoRgn/KTRR block arbitrary kernel code addition or modification, this means an attacker cannot easily write in those protected pages nor abuse the functions called when write access is enabled.

For the moment, PPL is only used to protect physical page mapping and code signing information, but it could also be used to protect other dynamic sensitive data. Nothing stops Apple from creating other specialized physical regions for other purposes.

5 Persistence on the system

Breaking all the iOS protections is hard, doing it after each reboot is even harder. An attacker who wants to achieve this goal will have to circumvent two problems:

— How to bypass code signature.
— How to get its code executed.

5.1 Signature bypass

Legitimate certificates. The easiest way is probably by not bypassing code signature in the first place, by just using a valid certificate. There are two different kinds of certificates in iOS that could be used to sign arbitrary code:

— Development certificates: designed to debug code during application development, they are tied to iPhones via their UDIDs, stored in the application provisioning profile. The UDIDs need to be registered on Apple website, which is obviously problematic for attackers if they don't want Apple to know who they are targeting...
— Enterprise certificates: they are not tied to any device but require a valid enterprise identity. The user also needs to manually trust the profile and an internet connection is required to verify the validity of the profile.

Enterprise certificates were already exploited to achieve persistence [21, 40] and to gain initial code execution by the Pangu Team [33].

Static code signature bypass. Several static code signature bypasses were released in the past, in binaries [24], libraries [25] and shared cache [36]. However, the attack surface is quite small and Apple considerably hardened the code used to parse and load mach-o files.

debugserver manipulation. `debugserver` is the daemon used to debug applications on an `iPhone`. As it needs to put breakpoints, it is able to execute arbitrary code in the binaries it debugs. Only binaries with the `get-task-allow` entitlement can be debugged, without this `entitlement`, the debugger may get a task port but will not be able to modify the code without triggering a code-signing violation. There is one Apple signed binary which has this entitlement to allow developers to debug VPN extensions: `neagent`. This executable has already been exploited in the past by Pangu [36] to gain initial code execution on the `iPhone` and the CIA [17] used multiple clever tricks to launch and manipulate `debugserver` at each boot to force it to execute an arbitrary shellcode.

It is hard for Apple to mitigate this because this is a wanted feature. The different vulnerabilities have been patched by sandboxing `debugserver` so it cannot launch `neagent` by itself and by patching the hole that let the CIA execute arbitrary commands, but `debugserver` will probably always be able to execute arbitrary code in a running `neagent`.

Scripting language. In the past, a JavaScript console, `jsc`, was included in the `JavaScriptCore` framework and could be used to execute arbitrary `JavaScript` code. This was exploited by the CIA [16] and NSO [11] associated with a `jsc` vulnerability to gain arbitrary code execution on reboot on iOS.

5.2 Code execution

`launchd`. The obvious way to get code execution at reboot is to register a new service. Services under `iOS` are managed by `launchd`, the first process launched by the kernel which owns the `PID` 1. Apple made it impossible to register new services by embedding the services list in `launchd` special section `__TEXT.__config` so patching it means invalidating `launchd` signature. Also, to make sure that an attacker does not replace a service executable with one of his own, `launchd` makes sure that the service is a platform binary, a binary which signature is directly stored in the kernel trust cache.

To bypass this, an attacker must rely on platform binaries and find a way to pivot to execute non-platform binaries. For example, the CIA [16] used `dhcpd` configuration files to have `dhcpd` execute arbitrary commands. As the commands were launched by `dhcpd`, the restriction of being a platform binary didn't apply and they could run arbitrary signed executables.

Legitimate methods. Before iOS 10, VOIP applications could be automatically started at each reboot. This could be exploited to gain early code execution on the phone [40]. Now VOIP applications need to use the PushKit API that will only start the application when a notification is received.

Extensions (network extensions, custom keyboards) or MDM applications could also be used to gain code execution after each reboot.

Both solutions however would ask some efforts to hide the installed application to the user.

5.3 Rootfs

To modify services executables or configurations, an attacker often has to modify the rootfs of the device. To do this, an attacker would have to bypass multiple protections put in place by Apple over the years.

At first, no special protection was used on the rootfs and all an attacker had to do was to remount it as a read-write volume. Apple then added some checks in the kernel to block the remount of the rootfs. Those checks were first bypassed by patching the code and then by modifying the rootfs flags in kernel memory before remounting it when KPP/KTRR blocked kernel patches.

In iOS 11.3 Apple moved to APFS, a file system that support snapshots, and restored a base snapshot of the rootfs at each boot. Even if attackers managed to modify the rootfs, it was restored at the next reboot. Moreover, as the snapshot was mounted read-only, trying to remount it in read-write mode would crash the kernel. This was bypassed by deleting the snapshot used at each reboot, by managing to remount the file system as read-write in another directory.

6 Conclusion

There are voices in the jailbreak community claiming that Apple adds protections that would only hinder jailbreakers and not real-world attackers. This is simply not true. Even if attackers don't need a full jailbreak to exfiltrate sensitive data, for example by exploiting a vulnerability in a mobile daemon, they need something similar to persist on a phone or to plant a backdoor. Blocking attackers necessarily means blocking jailbreaks.

Through this article, we showed that Apple takes their devices security very seriously, using an incremental approach to add new mitigations as

soon as a new exploitation method has been made public. While their first implementations of such mitigations might have been incomplete and/or easily bypassable, they have completed them across various minor versions to finally obtain a robust mechanism.

Furthermore, their ability to add custom features directly in their SOCs gives them a leg-up on Android phones constructor, which do not control both the hardware and the software sides.

As a conclusion, it has now become highly costly to build a stable and complete jailbreak for modern iPhones. The new upcoming hardware features, such as Memory Tagging described in ARM v8.5-A, will make vulnerabilities exploitation on such platforms even harder, killing entire vulnerability classes.

References

1. ARMv8 Architecture Reference Manual. https://static.docs.arm.com/ddi0487/da/DDI0487D_a_armv8_arm.pdf.
2. WebKit Github repository. https://github.com/WebKit/webkit.
3. failbreak. https://github.com/grp/amfi_interpose, 2012.
4. Remove network access from the WebContent process sandbox. https://trac.webkit.org/changeset/229093/webkit, 2018.
5. Marwan Anastas. Analyse du contournement de KTRR. https://connect.ed-diamond.com/MISC/MISC-102/Analyse-du-contournement-de-KTRR, 2019.
6. Brandon Azad. CVE-2018-4280: Mach port replacement vulnerability in launchd on iOS 11.2.6 leading to sandbox escape, privilege escalation, and codesigning bypass. https://github.com/bazad/blanket, 2018.
7. Brandon Azad. An introduction to exploiting userspace race conditions on iOS. https://bazad.github.io/2018/11/introduction-userspace-race-conditions-ios/, 2018.
8. Brandon Azad. iOS privilege escalation via crashing. https://bazad.github.io/2018/09/ios-privilege-escalation-via-crashing/, 2018.
9. Brandon Azad. Examining Pointer Authentication on the iPhone XS. https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html, 2019.
10. Saam Barati. Strings should not be allocated in a gigacage. https://github.com/WebKit/webkit/commit/8aa923d836ac6f999ce221a2f275687ffcf54276, 2018.
11. Max Bazaliy, Cris Neckar, Greg Sinclair, and in7egral. Technical Analysis of the Pegasus Exploits on iOS. https://info.lookout.com/rs/051-ESQ-475/images/pegasus-exploits-technical-details.pdf, 2016.
12. Ian Beer. XNU kernel UaF due to lack of locking in set_dp_control_port. https://bugs.chromium.org/p/project-zero/issues/detail?id=965#c2, 2016.
13. Ian Beer. Splitting atoms in XNU. https://googleprojectzero.blogspot.com/2019/04/splitting-atoms-in-xnu.html, 2019.

14. Eloi Benoist-Vanderbeken. macOS: How to gain root with CVE-2018-4193 in < 10s. https://www.synacktiv.com/ressources/OffensiveCon_2019_macOS_how_to_gain_root_with_CVE-2018-4193_in_10s.pdf, 2019.
15. Dionysus Blazakis. The Apple Sandbox. https://www.blackhat.com/html/bh-dc-11/bh-dc-11-archives.html#Blazakis, 2011.
16. CIA. TRICLOPS Summer 2015 - Ottawa. https://wikileaks.org/ciav7p1/cms/page_24969246.html.
17. CIA. PREDUX. https://wikileaks.org/ciav7p1/cms/files/Triclops%202015%20-%20PREDUX.pdf, 2015.
18. comex. Saffron jailbreak. https://www.theiphonewiki.com/wiki/Saffron.
19. comex. Star jailbreak. https://www.theiphonewiki.com/wiki/Star.
20. Razvan Deaconescu, Luke Deshotels, Mihai Bucicoiu, William Enck, Lucas Davi, and Ahmad-Reza Sadeghi. SandBlaster: Reversing the Apple Sandbox.
21. Mark Dowd. Malwairdrop: compromising idevices via airdrop. http://2015.ruxcon.org.au/assets/2015/slides/ruxcon-2016-dowd.pptx, 2015.
22. Stefan Esser. System and Security Info iOS Application. https://sektioneins.de/en/blog/16-05-09-system-and-security-info.html, 2016.
23. Jonathan Levin. Yalu. http://newosxbook.com/forum/files/2_65253536265b91dc0ee02a0b4827de41.
24. Jonathan Levin. 28 Days Later - TaiG 2 (Part the 1st). http://newosxbook.com/articles/28DaysLater.html, 2015.
25. Jonathan Levin. The Annotated (informal) guide to TaiG - Part the 1st. http://newosxbook.com/articles/TaiG.html, 2015.
26. Jonathan Levin. The Apple Sandbox - Deeper into the Quagmire. http://newosxbook.com/articles/hitsb.html, 2016.
27. Jonathan Levin. Darwin 18 (Beta) Changes. http://newosxbook.com/free/security12deltae.pdf, 2018.
28. Jonathan Levin. Darwin Networking. http://newosxbook.com/bonus/vol1ch16.html, 2018.
29. Jonathan Levin. Casa De P(a)P(e)L. http://newosxbook.com/articles/CasaDePPL.html, 2019.
30. Niklasb. Exploit for CVE-2018-4233. https://github.com/phoenhex/files/blob/master/exploits/ios-11.3.1/pwn_i8.js, 2018.
31. Phoenhex. Pwn2Own 2017: UAF in JSC::CachedCall (WebKit). https://phoenhex.re/2017-05-04/pwn2own17-cachedcall-uaf, 2017.
32. Siguza. KTRR. http://siguza.github.io/KTRR/, 2018.
33. Pangu Team. The Userland Exploits of Pangu 8. https://cansecwest.com/slides/2015/CanSecWest2015_Final.pdf, 2015.
34. Luca Todesco. iPhone 7 10.0 / 10.1 KTRR bypass. http://yalu.qwertyoruiop.com/y7.txt, 2018.
35. Luca Todesco. Life as an iOS Attacker. http://iokit.racing/bluehatil.pdf, 2019.
36. Tielei Wang, Hao Xu, and Xiaobo Chen. Pangu 9 Internals. https://www.blackhat.com/docs/us-16/materials/us-16-Wang-Pangu-9-Internals.pdf, 2016.

37. Xerub. Tick (FPU) Tock (IRQ).
https://xerub.github.io/ios/kpp/2017/04/13/tick-tock.html, 2017.
38. Zerodium. We're now paying $2,000,000 for remote iOS jailbreaks.
https://twitter.com/Zerodium/status/1082259805224333312, 2019.
39. Qixun Zhao. IPC Voucher UaF Remote Jailbreak Stage 2 (EN).
http://blogs.360.cn/post/IPC%20Voucher%20UaF%20Remote%20Jailbreak%20Stage%202%20(EN).html, 2019.
40. Min Zheng, Hui Xue, Yulong Zhang, Tao Wei, and John C.S. Lui. Enpublic Apps: Security Threats Using iOS Enterprise and Developer Certificates.
https://www.cse.cuhk.edu.hk/~cslui/PUBLICATION/ASIACCS15.pdf, 2015.