An Apple a Day Keeps the Exploiter Away

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

Synacktiv

Abstract. Three years ago, we presented all the difficulties an attacker has to face when exploiting a state-of-the-art iPhone device. Back in the days, the amount of defense-in-depth was already quite impressive, and a public price for a full exploitation chain was 2M$.

There have since been 3 new major iOS versions and as many generations of iPhones, coming with their new software and hardware mitigation. This article aims at describing how Apple significantly raised the bar for an attacker to be able to gain a privileged access to an up-to-date iPhone 13 (the latest model when writing this article).

1 Introduction

This article is a follow-up of a previous one we presented back in 2019 [7]. In three years, a lot of things have changed. Multiple 0-click vulnerabilities have been discovered and patched in iOS [4,6,8], a bootrom exploit [2] and a takeover of Apple secure processor [12] have been released, Zerodium now pays more for an Android zero click full chain with persistence than for an iOS one...

Is Apple losing the security game? In this paper we will see that this is quite the opposite. In the recent years, Apple actually accelerated the security hardening of their operating systems and phones. They also learned a lot from their vulnerabilities and from the attackers which gave them the ability to eliminate whole classes of vulnerabilities and exploits strategies.

This article does not reintroduce all iOS security mechanisms and it is highly recommended to (re)read the 2019 one before diving into this one.

2 Pointer Authentication Codes

Pointer authentication is supported since the iPhone XS and XR with the A12 SoC and iOS 12. Since then, Pointer Authentication Codes (`PAC`) have been bypassed numerous times, both in userland [8, 9] and

kernelland [3, 9]. In our previous paper [7] we insisted that the technology was still young and that Apple could sign more things and use different keys. It turns out that it's exactly what they did.

The obvious attack against `PAC` pointers is to modify unsigned pointers and to swap pointers that are signed with the same key and context. Apple is well aware of that and made this significantly harder.

2.1 More Signed Pointers

First of all, more and more data pointers are now signed. This includes both sensitive pointers like the sandbox label but also data structures known to be used by attackers to build full exploits. For instance, after multiple jailbreaks and exploits using pipes to build an arbitrary read/write primitive in the kernel [11], Apple started in iOS 14.2 to sign pipes data pointers.

The userland is not left out. For example, after being used by Samuel Groß in his iMessage exploit [8], the `ISA` (as in *this object IS A box/cat/apple*) pointer, a very important pointer at the beginning of every `Objective-C` object, is signed since iOS 14 (but only checked since iOS 14.5... [13]). Some other pointers that weren't correctly signed by the compiler or in assembly sources are now also protected. For example, the function `__chkstk_darwin`, used to check that dynamic stack allocations don't overflow the stack, was not signed in the global offset table.

Probably to simplify pointer test handling, null pointers are not signed. This can be sometimes used to exploit logic flaws. The most famous one is the sandbox label pointer. When this pointer is null, the process is only restricted by the system sandbox, which only restricts access to sensitive APIs (like access to the host special ports) and is permissive by default. To escape the application or WebKit sandbox, patching this pointer with a null was sufficient, even if this pointer was signed. This has been patched by Apple in iOS 15.0 by always checking the signature, even if the pointer is null (see fig. 1 and fig. 2).

2.2 More Diversity

To protect against pointer swapping, Apple tries to use the appropriate key and a specific context for each usage. For example, at the beginning of `PAC` and since iOS 14, every function pointer stored in a structure was signed with a null context. Moreover, all those pointers signed with a null context were all stored in the same section `__DATA_CONST:__auth_ptr`. Now every function pointer field is signed on the fly with a specific context

Fig. 1. PAC sandbox label before iOS 15

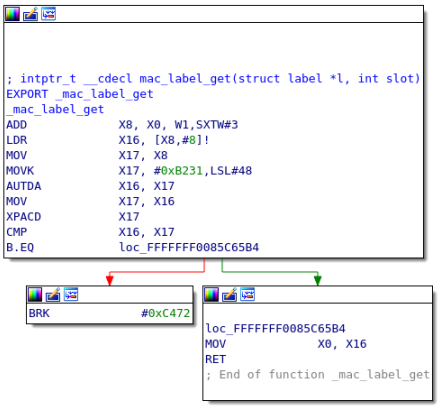

Fig. 2. PAC sandbox label after iOS 15

(but the pointer address is not used, maybe to support structure copy, so it is still possible to swap two instances of the same field, see fig. 3 and fig. 4).

Now the only pointers signed with a null context stored in kernel memory are two pointers on `mig_strncpy_zerofill` and `__chkstk_darwin` and these are in read-only memory. The only other zero context pointers are functions arguments and return value and those are stored in registers that may only be temporarily saved on the stack. This greatly reduces the possibility to swap pointers in memory and the available gadgets.

2.3 More Keys

At the beginning of `PAC`, all processes shared the same A key, used to sign function pointers. So even if all processes already had different B keys, used to sign process specific data like the saved return address, it was still pretty easy to attack another process with a JOP chain (whereas ROP has been killed by `PAC`). Now, processes have a A key that depends on their `Team ID`, which is a unique identifier generated by Apple for every developper account. Daemons and Apple apps don't have any `Team ID` but they can use a specific entitlement (`com.apple.pac.shared_region_id`) to get a different A key nevertheless. Of course, WebKit uses this entitlement so it is now as difficult to attack daemons from WebKit than from any other application (at least from a `PAC` point of view).

```
ADRP            X9, #zsigned_pty_get_ioctl@PAGE
LDR             X9, [X9,#zsigned_pty_get_ioctl@PAGEOFF]
ADRP            X10, #tty_dev_head@PAGE
LDR             X11, [X10,#tty_dev_head@PAGEOFF]
STP             X11, X9, [X8,#0x10]
ADRP            X9, #zsigned_pty_get_name@PAGE
LDR             X9, [X9,#zsigned_pty_get_name@PAGEOFF]
STR             X9, [X8,#(__pty_driver.name - 0xFFFFFFF009418D70)]
```

Fig. 3. function pointer fields before iOS 14, zero-signed pointers are stored in `__DATA_CONST:__auth_ptr`

```
ADR             X16, _pty_get_ioctl
NOP
MOV             X17, #0xB4AC
PACIA           X16, X17
STR             X16, [X8,#(__pty_driver.open - 0xFFFFFFF009A559C8)]
ADR             X16, _pty_get_name
NOP
MOV             X17, #0x707
PACIA           X16, X17
STR             X16, [X8,#(__pty_driver.name - 0xFFFFFFF009A559C8)]
```

Fig. 4. function pointer fields after iOS 14, different fields use different keys and pointers are signed on the fly

2.4 Fewer Weaknesses

Last but not least, Apple fixed several `PAC` bypasses in its code and even killed two bypass classes by adding hardware mitigations and another one with a compiler mitigation.

The idea of `PAC` is that some of the upper bits of a pointer are used to store a signature. When one of the `AUT` instructions is used on a signed pointer, if the signature is valid, a pointer stripped of its signature is returned. This stripped pointer can then be used with classic instructions. If the signature is invalid, the resulting pointer will be invalidated by flipping one of its higher bits, this can be detected by checking this bit or by directly trying to dereference it, which would trigger a exception.

One original weakness of `PAC` was that when an invalid pointer was signed, a single bit of the signature was flipped and it was trivial to get a valid signature from it. The `AUT` then `PAC` combination is frequent as function pointers need to be signed for different contexts. For example, a function pointer passed as an argument to a function will be signed with a null context and if the function sets a structure field with it, it will have to be resigned with the field context. Since the `A14` (first used in the iPhone 12), `EnhancedPAC` (as defined in `Armv8.5`) is implemented, so when an invalid pointer is signed, the signature is discarded (filled with zeroes) and it is not possible to deduce the signature for a valid pointer.

With this protection, it is still possible to bruteforce a valid signature under some circumstances. All the invalid pointers will have one signature and the only valid one will have a different one. With this oracle, it is possible in seconds to minutes (depending on the oracle speed and the signature length) to find a valid signature. Apple again killed this method by forcing the compiler (with the `-fptrauth-auth-traps` option) to add checks after every `AUT` instruction to crash the process (or the kernel) if the pointer passed to the `AUT` instruction is not correctly signed. Moreover, in the `A15` (first used in the iPhone 13), Apple made sure to catch all the invalid signatures by implementing the `Armv8.6-A FPAC` extension that raises an exception when an `AUT` instruction encounters an invalid signature. At last, they also made sure to make this attack less practical by increasing the size of the signature in userland from 16 to 24 bits in iOS 13 (it has always been 24 bits in kernelland).

3 Page Protection Layer

The hardware foundation of the `Page Protection Layer` (`PPL`) is present in Apple `SoCs` since the `iPhone 8` and was already used to protect the `JiT` page, but without `PAC` it was worthless to protect kernel data, so we had to wait until the `A12 SoC` to see `PPL` in the kernel. `PPL` is supposed to protect arbitrary pages against modification by an attacker having an arbitrary read/write in the kernel and even if they are able to bypass `PAC` and execute arbitrary existing code in the kernel.

At first, `PPL` was only used to protect physical page mapping, some structures related to code signing (most notably the dynamic trust cache that contains the hash of all the platform binaries) and to protect platform binaries against code injection (it's a little bit more complex than that but this is the general idea). But since then, and as a lot of Apple security features, `PPL` gained a lot more importance.

3.1 Entitlements and Profiles

Starting from iOS 15.0, `PPL` is used to validate and protect a very important piece of Apple security systems: profiles and entitlements.

Before iOS 12, hooking a userland daemon, `amfid`, was enough to bypass all the signatures and entitlement checks. Apple mitigated that by checking the executable signature in kernel with `CoreTrust` which validates the signature and the certificate chain but doesn't check the entitlements and is not able to check all the certificate details (most

notably its expiration date and revocation status). A simple and effective way to bypass this was to sign the executables with an expired or revoked certificate while still hooking `amfid` to bypass the other checks. Since iOS 15.0 however, the kernel also checks the profile signature and if the entitlements requested match the ones authorized in the profile. If there is no profile, the executable has to be a platform binary or be signed with the Apple `App Store` certificate otherwise it is limited to a very restrictive set of entitlements.

To make sure that the entitlements have not been tampered with after being validated, they are checked and stored in `PPL` and all the pointer chains from the thread to the entitlements are, of course, signed with dedicated `PAC` contexts. Last but not least, the entitlement bytes are also signed with `PAC`.

One could say that an attacker with an arbitrary kernel read/write will nevertheless always find a way to get their hands on the data they need, even without arbitrary entitlements. It is true indeed but it considerably complicates the development of a useful backdoor.

3.2 RO Zones

As if pointer signature was not enough, Apple introduced read-only (`RO`) zones in iOS 15.2. Several sensitive kernel structures are now allocated in specific zones protected by `PPL`. Task credentials, threads exception ports, sandbox profiles, entitlements or other signature-related elements now cannot be written without a `PPL` bypass. That means that it is not possible anymore to just patch the uid of a process to become root or to nullify the sandbox pointer to escape the sandbox.

The design is quite robust and Apple blocked the obvious bypasses. To make sure that `RO` pointers are not swapped, there is a back reference in all `RO` allocations. If the back reference doesn't match the address where the `RO` zone pointer was stored, the kernel panics. Even without this back reference, type confusions are worthless as the zone id is passed to the `PPL` functions and checked against the effective zone id. It is also obviously not possible to replace a `RO` zone pointer with a pointer in a read-write memory or a pointer to another `RO` zone as the zone id is also checked when the `RO` zones are read. The only `PPL` function used to write in the `RO` zones also checks that the address is aligned, that the `memcpy` will not overflow, that the source is either on the stack or in another `RO` zone, so classic memory corruption vulnerabilities do not apply here as well.

4 Kernel Mitigations

4.1 Zones Hardening

`XNU` has a zone allocator. A zone is a set of memory pages used to allocate a certain type of objects with a fixed size. For example, there is a zone for `mach ports`, and making an allocation in this zone using `zalloc` will return a pointer to an allocation of the size of an `ipc_port` structure in a page containing only `ipc_port` structures.

This behavior has a side effect in terms of exploitation: exploiting a use-after-free in a zone was a bit complicated, as an allocation can only be reused by another allocation of the same type. However, it was still possible to re-use a whole page of allocations: the page has to be completely freed, so the kernel garbage collector would potentially reassign it to another zone.

A first mitigation has been introduced in iOS 13.2: for some critical kernel objects which are often abused in exploits, such as `mach ports`, a call to `zone_require` is added in functions manipulating such objects. This function ensures that the object address belongs to the correct zone, so it is no longer possible to craft a fake object in a random allocation, or reuse a freed port address in another allocation. However, there is no check about the address alignment, so it can point in the middle of an allocation.

To further prevent abuse of the garbage collector, zone sequestration has been introduced in iOS 14.0. This mechanism ensures that a page virtual address belonging to a specific zone will never be reused for another zone. This definitely prevents some use-after-free vulnerabilities from being exploited. This mitigation is not enabled on all zones by default. For example, as demonstrated by Ian Beer from Project Zero [5], the `ipc.ports` zone was not sequestrated before iOS 15.2. The reason was that `zone_require` checks were supposed to prevent a misuse of a fake port.

Moreover, across the various iOS versions, the number of zones has been increased, and a bunch of allocations made in the kernel heap now belong to a specific zone.

Speaking of the kernel heap, before iOS 14, all `kalloc` allocations were made in `kalloc.x` zones depending on their size. For example, a `kalloc(1000)` resulted in an allocation in `kalloc.1024`. Starting from iOS 14, the heap has been split in 4 different heaps:

— KHEAP_DEFAULT: for kernel objects which do not belong to a specific zone;

— KHEAP_KEXT: for allocations made by kexts;
— KHEAP_DATA_BUFFERS: for allocations containing only data, no pointers are present in this heap;
— KHEAP_TEMP: for allocations done in scope of a thread (the same thread allocates and frees the pointer). This heap is no longer present in iOS 15 and has been merged with KHEAP_DATA_BUFFERS.

This heap separation widely mitigates the ability to spray controlled data in order to exploit a use-after-free in `kalloc.x` zones, as user-controlled allocations with controlled content are now performed in the `KHEAP_DATA_BUFFERS` zone.

Furthermore, some interesting objects from an attacker point of view, such as `IPC messages`, had their structure change: they were previously allocated in a standard `kalloc.x` zone if the supplied size could not fit in a zone allocation. Starting from iOS 14.2, the userland controlled data is allocated in the `KHEAP_DATA_BUFFERS` heap while the original `ipc_kmsg` structure is always allocated in the dedicated zone.

Finally, a new mitigation has been added in iOS 15.2: `SAD_FENG_SHUI`. Its goal is to randomize zone assignments to one of the 4 general submaps, so that there is no longer a guaranted zone interleave. This prevents some OOB exploits relying on this fact or exploits hardcoding a kernel address targeting a specific zone allocation after having filled the corresponding zone.

4.2 Critical Port Rights Separation

Historically, there was a unique type of task or thread port, allowing to use all APIs indifferently (read/write memory, read/write context, etc.). Starting from iOS 14, there has been a separation in different `flavors`, each one with its own port.

In iOS 15.4, the following flavors are defined for a task:
— TASK_FLAVOR_CONTROL
— TASK_FLAVOR_READ
— TASK_FLAVOR_INSPECT
— TASK_FLAVOR_NAME

Threads have one less flavor:
— THREAD_FLAVOR_CONTROL
— THREAD_FLAVOR_READ
— THREAD_FLAVOR_INSPECT

The previous flavors are listed from the most to the least privileged, and each port gives access to a subset of APIs. For example, using

the `mach_vm_read` API requires a `TASK_FLAVOR_READ` port whereas the `mach_vm_write` API requires a `TASK_FLAVOR_CONTROL` port.

In a fun way, this privilege separation has introduced a regression: before iOS 14.5, the functions used to convert a port to a task object skipped a call to `task_conversion_eval` when the flavor was not `TASK_FLAVOR_CONTROL`. This function ensures that only a platform binary can resolve the task port of another platform binary. This means that, in iOS versions between 14.0 and 14.5, a non-platform binary was able to read the memory space of a platform binary.

iOS 14.5 is also the version where Apple started to set the `self` task port and main thread port as immovable, which means that these ports rights cannot be sent in a mach message. They also introduced port labels, another mechanism to control who can receive sensitive ports (task and thread ports but also userland drivers ports for example). This change mitigates many logical vulnerabilities and exploitation techniques, as it no longer possible for a service to be exploited to send its `self` task port or to legitimately send it to another process.

4.3 Minor Hardening

Since iOS 14, building a tfp0 is a little bit harder. There is an extra check in `convert_port_to_map_with_flavor` which ensures that the map does not directly give access to the kernel pmap.

With the RO zones protecting the cred structure (see subsection 3.2), passing root with an arbitrary kernel read/write primitive became significantly harder. To make that a little bit harder, the kernel was also stripped of two functionalities: the handling of suid binaries and the suid cred port support (a deprecated way to spawn process with arbitrary uids thanks to a port created by a root process with the `com.apple.private.suid_cred` entitlement).

5 WebKit

WebKit security has been widely impacted by all the generic PAC improvements, and there is no public method nowadays to bypass PAC in a WebKit exploit on iOS 15.x.

However, some WebKit-specific mitigations have also been added. As stated in our initial publication [7], a usual WebKit exploit consists of the following steps:

— Trigger a vulnerability to gain `addrof` and `fakeobj` primitives;

— Gain read/write primitives by crafting a fake JavaScript object;
— Gain arbitrary code execution by bypassing the hardware-backed mitigations (APRR and PAC).

To make the second and third steps harder, new mitigations have been added.

5.1 Structure ID Randomization

To be able to craft a fake object, an attacker has to build a `JSCell` structure. This structure has the following representation in memory:

```
struct JSC::JSCell {
    JSC::StructureID m_structureID;
    JSC::IndexingType m_indexingTypeAndMisc;
    JSC::JSType m_type;
    JSC::TypeInfo::InlineTypeFlags m_flags;
    JSC::CellState m_cellState;
};
```

Previously, a `StructureID` was just an index into an array of `Structure` objects. Getting a valid `StructureID` simply was a matter of creating `N` (`N` being greater than 1000) new different objects and picking `N/2` as a valid `StructureID`.

Now, random entropy bits have been added to the `StructureID`. The number of bits has changed across time, but it is not possible to predict them before having gained a `read` primitive.

For now, each time a `Structure` has to be accessed, the index is extracted from the `StructureID`, checked against the `StructureIDTable` size, then the `encodedStructureBits` are xored with the lower bits of the `StructureID` shifted by the pointer size (48-bits), to finally retrieve a pointer to a `Structure`.

The encodings are represented as a comment in `WebKit` sources [1]:

```
1. StructureID is encoded as:
-------------------------------------------------------------
| 1 Nuke Bit | 26 StructureIDTable index bits | 5 entropy bits |
-------------------------------------------------------------
The entropy bits are chosen at random and assigned when a
StructureID is allocated.

2. For each StructureID, the StructureIDTable stores
encodedStructureBits which are encoded from the structure pointer
as such:
-------------------------------------------------------------------
| 11 low index bits | 5 entropy bits | 48 structure pointer bits |
-------------------------------------------------------------------
The entropy bits here are the same 5 bits used in the encoding of
the StructureID for this structure entry in the StructureIDTable.
```

Bypasses of this mitigation have been made public back in 2019 [10]. The idea is quite simple: building a fake object and using it to gain a leak or a read primitive without reaching code referencing the underlying `Structure` to avoid any bad pointer dereference.

5.2 JIT Instructions Signature

A known method to bypass `APRR` was to modify the native code generated by the JIT engine before it is written in the JIT page.

Now, on devices supporting `PAC`, the generated instructions are signed. This is performed by computing a new signature for the block of instructions each time a new instruction is added. Then, when the block has to be written in the JIT page, the signature is verified. On iOS 14.4, this has been improved by keeping a signature for each instruction instead of a unique one for the whole block. Each instruction signature is verified just before the copy in the JIT page to ensure that no modification can occur.

The signature algorithm makes use of the `PACDB` instruction. This implies that, to be able to modify JIT instructions, an attacker needs code execution to be able to execute `PAC` instructions to build the signature.

5.3 PAC Exceptions Termination

Back in 2020, a blog post on Project Zero blog described a way to bypass `PAC` and `APRR` by creating infinite loops in the exception handling code while making another thread try to bruteforce a `PAC` pointer. This would allow the main thread to catch the exception in case of an invalid signature, and resume the thread to try the next value.

This has been mitigated by adding a specific entitlement to the `WebContent` process. This entitlement adds the flag `TF_PAC_EXC_FATAL` to the kernel task. This flag is involved whenever an exception is handled by the kernel: if the flag is present and the exception is related to an invalid `PAC` pointer (either a data pointer, an instruction pointer or a specific `BRK` following a bad pointer authentication), the task will directly exit without reaching the userland exception handlers.

5.4 Sandbox

Since our previous study, the sandbox has been greatly improved:
— Ability to filter unix syscalls;
— Ability to filter mach syscalls;
— Ability to filter any fcntl;

— Ability to filter any ioctl by their code and file name;
— Ability to filter mach messages by their endpoint name and message number;
— Ability to filter IOKit methods by their id;
— Support of different states of a process life.

The `WebContent` sandbox profile is usually where the latest sandbox improvements appear first, as seen on the WebKit github repository:

— Unix syscalls are all denied, except a subset specifically required by the process. This subset is divided between `syscall-unix-only-in-use-during-launch` and `syscall-unix-in-use-after-launch`. When exploiting the `WebContent` process, only the latter subset is available to try to escape the sandbox;
— Mach syscalls are all denied, except a subset specifically required by the process. This subset is divided between `syscall-mach-only-in-use-during-launch` and `syscall-mach-in-use-after-launch`. When exploiting the `WebContent` process, only the latter subset is available to try to escape the sandbox;
— only 4 services are accessible from the sandbox and some of them are restricted (directly via the sandbox or thanks to specific entitlements);
— fnctl are all denied except a subset of less than 20 specifically required by the process;
— ioctl are all denied except 2 and they must be sent to the `/dev/aes_0` device;
— IOKits are all denied.

In iOS 15.4 (latest available version when writing this article), the process life states handling is not implemented yet.

6 Conclusion

While public jailbreaks have still been a thing on iOS 14, there are no known complete jailbreaks for iOS 15 yet (despite public POC giving kernel read/write primitives). Vulnerabilities that were easily exploited a few months ago are now completely useless to an attacker.

Apple does a really good job at finding the root cause of vulnerabilities and at mitigating whole classes of bugs and exploit strategies. Post exploitation is also taken into account, even with powerful primitives, extracting sensitive data and compromising applications is not simple,

which gives Apple more time to fix bugs and gives users time to update their devices. They also do a great job at integrating hardware mitigations: all attackers fear the day when memory tagging will be implemented in their SoC. Moreover, this paper didn't even fully list all the job they did to improve general iPhone security (the rootfs seal to prevent persistence, their custom C compiler to mitigate classic memory corruption in the boot process, the secure storage component used to protect passcodes in recent iPhones, etc.).

However, powerful vulnerabilities can still be used to attack modern phones and even if WebKit is strongly sandboxed, the vast majority of the applications still have access to a huge attack surface in the kernel. Moreover, Linus Henze has proven [9] that powerful logical vulnerabilities can bypass even the latest mitigations.

It may become impossible in the near future to fully compromise an iPhone with a simple web page but it doesn't mean that the game is over.

References

1. Structureid memory representation. https://github.com/WebKit/WebKit/blob/main/Source/JavaScriptCore/runtime/StructureIDTable.h#L140, 2022.
2. axi0mX. Announcement of checkm8. https://twitter.com/axi0mX/status/1177542201670168576, 2019.
3. Brandon Azad. Examining pointer authentication on the iphone xs. https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html, 2019.
4. Ian Beer. An ios zero-click radio proximity exploit odyssey. https://googleprojectzero.blogspot.com/2020/12/an-ios-zero-click-radio-proximity.html, 2020.
5. Ian Beer. Xnu kernel use-after-free in mach_msg. https://bugs.chromium.org/p/project-zero/issues/detail?id=2232, 2022.
6. Bahr Abdul Razzak Noura Al-Jizawi Siena Anstis Kristin Berdan Ron Deibert Bill Marczak, John Scott-Railton. Forcedentry - nso group imessage zero-click exploit captured in the wild. https://citizenlab.ca/2021/09/forcedentry-nso-group-imessage-zero-click-exploit-captured-in-the-wild/, 2020.
7. Fabien Perigaud Eloi Benoist-Vanderbeken. Wen eta jb? a 2 million dollars problem. https://www.sstic.org/2019/presentation/WEN_ETA_JB/, 2019.
8. Samuel Groß. Remote iphone exploitation part 3: From memory corruption to javascript and back – gaining code execution. https://googleprojectzero.blogspot.com/2020/01/remote-iphone-exploitation-part-3.html, 2020.
9. Linus Henze. Fugu14 writeup. https://github.com/LinusHenze/Fugu14/blob/master/Writeup.pdf, 2021.
10. YONG WANG. Thinking outside the jit compiler: Understanding and bypassing structureid randomization with generic and old-school methods. https://i.blackhat.com/eu-19/Thursday/eu-19-Wang-Thinking-

Outside-The-JIT-Compiler-Understanding-And-Bypassing-StructureID-Randomization-With-Generic-And-Old-School-Methods.pdf, 2019.

11. Ned Williamson. Sockpuppet: A walkthrough of a kernel exploit for ios 12.4. https://googleprojectzero.blogspot.com/2019/12/sockpuppet-walkthrough-of-kernel.html, 2019.
12. Hao Xu. Attack secure boot of sep. https://github.com/windknown/presentations/blob/master/Attack_Secure_Boot_of_SEP.pdf, year = 2020.
13. Jundong Xie Zhi Zhou. Hack different: Pwning ios 14 with generation z bugz. https://i.blackhat.com/USA21/Wednesday-Handouts/us-21-Hack-Different-Pwning-IOS-14-With-Generation-Z-Bug-wp.pdf, 2021.