A Brief Introduction of checkm8
– an unpatchable iOS jailbreaking method

Shaochen (Henry) ZHONG

Revisited by February 13, 2020 based on Dr. Loui's feedback
EECS 338, Dr. Ronald Loui

Abstract

`checkm8` is a recently released (Sep 27, 2019) jailbreaking tool developed by `axi0mX`. Being a jailbreak method targeting the BootROM (a.k.a `SecureROM`) of iOS devices, it is well-known for its unpatchable nature and strong compatibility across 11 generations of iOS devices. This article is intended to give a brief introduction of `checkm8`, with a proper exposure of knowledge from the very concepts and needs of iOS jailbreaking, to the highlighted characters of `checkm8`, then to provide a relatively more technical walk-through on the hacking mechanism implemented in `checkm8`. In the end, it analyzes the philosophical reason behind this exploitation and provides some potential solutions to prevent it from happening again.

Contents

1 Jailbreaking under the iOS context

Jailbreaking – under the context of iOS – is an act of gaining superuser access of Apple's mobile/tablet operating system(s). Since Apple restricts such superuser privilege by default, a successful jailbreak is typically performed with exploitations of bugs and system design flaws, known as *privilege escalation*.

Throughout the history of iOS jailbreaking, there are various successful – or semi-successful – jailbreak methods; and the capability of each jailbreak method also varies depending on its attack methodology and Apple's patch. However, the main and common goals of all jailbreak methods are the same: to bypass the software installation restrictions imposed by Apple; and to utilize previously unavailable lower level portals (e.g., APIs) to collect data and therefore develop features that are impossible to implement under a non-jailbroken device.

2 `checkm8`: the overview

Like many other releases of jailbreaking tools, `checkm8` is a recently released (Sep 27, 2019) jailbreaking tool developed by `axi0mX` [2]. While each successful jailbreak method has its pros and cons, the highlights of `checkm8` are:

- It is unpatchable by the nature of the BootROM exploitation, meaning that Apple will never be able to release a patch to nullify the tool.
- It is applicable to a vast range of generations of iOS devices, from iPhone 4s (A5 chip) to iPhone 8/X (A11 chip). This means millions of active devices in use today can benefit or be targeted by this jailbreaking method.

2.1 The unpatchable nature

`checkm8` targets the BootROM (a.k.a `SecureROM`) of iOS devices – which is a low-level read-only ROM that being utilized during the booting stage of iOS devices. With the bugs that `checkm8` exploits and proper implementation of Heap Feng-shui[1], you can execute any code at the BootROM level, thus carry out a successful jailbreak.

Due to the read-only nature of BootROM, the booting code is encoded in a hardware manner when a device is manufactured. Thus it is impossible for Apple to release software security patches to alter the code within BootROM, and this results in an unpatchable hack. Also, since a vast amount of iOS devices share BootROMs with similar architecture, the compatibility of a BootROM exploitation can be simultaneously widely applicable and long-standing.

2.2 The lack of persistence

Though with many highlights, one significant limitation of `checkm8` is its lack of persistence. Since to the mechanism of `checkm8` is to exploit the BootROM – which is essentially a type of ROM – a jailbroken device will revert to its unexploited state once a reboot is performed. Thus making `checkm8` only a tethered jailbreak[2]. In a real-life scenario, a solo[3] `checkm8` jailbreak is only possible by turning the phone into DFU (Device Firmware Upgrade) mode and connecting it to a computer (to run scripts that inject custom code).

[1] Details refer to *Section 3.3 iBoot and Heap Feng-shui.*

[2] The term 'tethered jailbreak' is the antonym of 'untethered jailbreak,' where the latter one refers to jailbreak methods which are so powerful that they can exploit a device even after rebooting, and without the aid of a connected computer.

[3] `axi0mX` admits that it is theoretically possible to remotely jailbreaks – thus 'hacks' – a `checkm8` supported device by using a chain of exploits that available and unavailable to the public [4]. However, it is extremely unlikely to happen due to the technical mechanism of `checkm8` and the mindset of malicious attackers. More about it in *Section 2.3 The potential of malicious uses.*

2.3 The potential of malicious uses

2.3.1 The protection of Secure Enclave

One of the major concerns of `checkm8` is the fear of privacy breaches due to the implementation of `checkm8`. According to `axi0mX`, devices with the feature of Secure Enclave[4] will protect your data if the possessor of your phone cannot input the correct credential; as it is stored in a separate system, and `checkm8` is not effective to the secure enclave processor. However, for devices with no Secure Enclave feature, the data within such a device can be easily compromised as there is no separate system protecting it [4].

Note that `axi0mX` also mentions the fact that accessing the data of a device is likely not the only goal for an attacker [4]. Thus although your data can be protected with Secure Enclave, the attacker might be going after something else – e.g., reselling the phone with a "seemingly fresh" system.

2.3.2 Unencourage factors to attackers with malicious intents

In *Section 2.2*, I have introduced the fact that for one to perform a `checkm8` jailbreak, one is required to obtain physical access of the phone and connect to a computer with proper tools installed and scripts available. Thus, `checkm8` cannot be performed remotely by nature. `axi0mX` also emphasizes the idea that it is unlikely that `checkm8` will be mainly used for malicious purposes, as most attackers with malicious intents prefer to stay in the distance – e.g., utilizing phishing emails, webpages, Wi-Fi hotspots, etc. – but not to be physically up-close [4]. Since `checkm8`'s nature of requiring physical access makes close physical contact inevitable, it is likely not preferred by malicious attackers.

3 `checkm8`: the mechanism

3.1 The booting mechanism of iOS

To establish a minimum understanding of `checkm8`, it is required to have a proper explosure to the booting mechanism of iOS devices. It is demonstrated in [Figure 1].

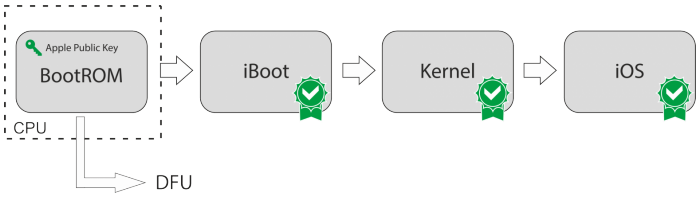

Figure 1: The Secure Boot Chain of iOS [1]

The `Kernel` and `iOS` stages are rather self-explanatory, while the `BootROM` and `iBoot` are not. In short, every stage will perform its task, check for the integrity of the system, and load and prepare to move to the next stage. Thus, `BootROM` is responsible for executing the first several programs of the booting, checking the integrity, and loading necessary data to prepare for moving to the `iBoot` stage. Then `iBoot` does the same for `Kernel` stage. Note that due to the similar in functionalities, `BootROM` shares a significant amount of system and library code with `iBoot` [1].

As introduced in *Section 2.1*, `BootROM` is a read-only memory with code encoded in a hardware manner when a device is manufactured; thus, it has the character of unwritable (and therefore unpatchable). Apple further implements security protocol to lock its memory storage once all executions within the `BootROM` stage are completed [1][3]; thus, it **should** also be unreadable to users.

[4] Secure Enclave is a security feature implemented with a separately booted 4MB flashable AKF processor core called the secure enclave processor (SEP). It performs biometric checks and decrypts the information – thus, the main operating system will not directly read the encryption key stored in Secure Enclave's storage. It is only implemented within devices that have A7 chip or later (with Touch ID or Face ID).

3.2 Exploiting SecureROM under DFU mode

Apple's philosophy is, if a program is unwritable and unreadable, then it is certainly safe. However, checkm8 manages to exploit a design flaw under the Device Firmware Upgrade mode[5] of the devices. For the sake of readability, here is a simplified version[6] of the procedure of DFU mode [1][3][5]:

1. usb_dfu_init() is called, an command-handling interface is registred and a buffer is allocated (hereinafter io_buffer).
2. Send a DFU_DNLOAD request that eventually calls out to the interface code.
3. DFU checks the request package, if the request package is shorter than the allocated io_buffer, the pointer to io_buffer is passed to a global variable.
4. The request package is copied into the io_buffer.
5. DFU stores the io_buffer into a device-specific address for loading the operating system.
6. The USB code will reset all variables (including the global variable containing io_buffer) and ready to handle the new package.
7. Once all packages are handled, a DFU_DONE request will be sent, and DFU will therefore free the io_buffer), and then try to boot device with the system stored at the device-specific address. If the boot is a success then you enter the temporary OS; otherwise, usb_dfu_init() is called again, and the device is back to the first stage.

The bug is embedded in *Step 3* and *Step 7*. As you may send a DFU_DONE request[7] at *Step 3* to bypass *Step 4* to *Step 6*. Altough the io_buffer is freed in *Step 7*, the global variable updated in *Step 3* still points to the supposely freed io_buffer, therefore results a *use-after-free* scenario [3][5].

3.3 iBoot and Heap Feng-shui

checkm8 exploits the *use-after-free* flaw by taking several spots on the heap. As the SecureROM of iOS uses libc version of malloc(), it is possible to induce some critical requests to store at the address of io_buffer which we have access of – this kind of maneuver is known as *heap feng-shui*. Then, by altering the value within io_buffer, we may complete a jailbreak [3].

Note this is an oversimplification of what checkm8 does. checkm8 utilizes the usb_device_io_request structure to construct a callback-chain, and eventually gets the device to execute the exploit's payload stored in io_buffer [1]. checkm8 also relies on the leaked iBoot source code to reverse-engineer the addresses of many functions within BootROM – as the iBoot code is vastly shared with the BootROM [1][3].

4 Summary

In summary, the philosophy of relying on confidentiality to protect system security is a rather unrestful idea [3]. As code and manual can be leaked by human – in this case, the leaking of iBoot is the initial cause of this catastrophic security breach – and the manufacture will not be able to refactor as the code is already in ROM. Instead, a comprehensive testing procedure should be enforced – e.g., fuzz testing – to ensure the security completeness of the program.

However, the overhead of building a testing procedure that is comprehensive enough is considerably huge; meanwhile the creativity of hackers can never be underestimated. Thus, a commonly embraced solution is transparency: e.g., invites hackers to security contests, or even open-sourcing the code for public scrutiny. By doing that, the manufacture may actively refactor its code based on public (e.g., whitehat community) feedback. And most importantly, the objective quality of the product can therefore be improved as it is now "more secured," instead of "secured if kept confidential."

[5] DFU is part of the BootROM. It is designed to restore a device by running a temporary system from USB.
[6] Infomation regarding wLength data length check, stall, leak, no_leak asynchronous execution, and details regarding DFU main/interface code are intentionally left out for simplification [1].
[7] By modifying the USB controller [5].

5 References

[1] a1exdandy. Technical analysis of the checkm8 exploit. https://m.habr.com/en/company/dsec/blog/472762/, 2019.

[2] axi0mX. ipwndfu: open-source jailbreaking tool for many ios devices. https://github.com/axi0mX/ipwndfu, 2019.

[3] Gh0u1L5. A brief analysis of the epic iphone exploit: checkm8. https://zhuanlan.zhihu.com/p/87456653, 2019.

[4] Dan Goodin. Developer of checkm8 explains why idevice jailbreak exploit is a game changer. https://arstechnica.com/information-technology/2019/09/developer-of-checkm8-explains-why-idevice-jailbreak-exploit-is-a-game-changer/, 2019.

[5] littlelailo. Apple bootrom bug: apollo.txt. https://gist.github.com/littlelailo/42c6a11d31877f98531f6d30444f59c4, 2019.