aracne

Classificazione Decimale Dewey:

005.8 (23.) SICUREZZA DEI DATI

GERARDO **IOVANE**, VINCENZO **OLIVA**

RED TEAM CYBERSECURITY LECTURES

CYBER ATTACK ON IOS SYSTEMS: THEORY, ENGINEERING AND ALGORITHMS

aracne

aracne

©

ISBN
979-12-218-2526-8

PRIMA EDIZIONE
ROMA 19 FERBBRAIO 2025

Table of Contents

Part I

1. Introduction

The widespread reliance on smartphones for personal and professional activities has been brought about by the prevalence of mobile technology, which, in turn, has resulted in new and evolving security challenges. As one of the leading mobile platforms, iOS is equipped with a comprehensive security architecture and strong adoption rates, and is therefore made a frequent target for advanced cyber threats. The core research question addressed by this paper involves the analysis of how offensive operations against iOS can be systematically modeled, engineered, and simulated within a red team context—and which methodologies, tools, and payload structures are most effective for achieving persistence, stealth and data exfiltration. As it relates to iOS, offensive cybersecurity is examined in this work, with a focus being given to the intersection of mobile security, threat modeling, and red teaming. To simulate adversary behavior and uncover vulnerabilities before they can be leveraged by hostile actors, offensive security practices, including ethical hacking and penetration testing, are used. By exploring the specific characteristics of iOS security mechanisms, common attacker strategies, and the spectrum of tools and frameworks available, theoretical foundations are connected by the paper to practical implementation. By the increasing complexity of threats directed at mobile devices and the necessity for proactive security measures, the need for such an approach is highlighted. A detailed reference for red teamers, security researchers, and advanced practitioners working in mobile offensive security is provided as the primary objective of this paper. Conceptual overviews are integrated with hands-on examples, and understanding of iOS security, threat actor profiling, and operational red teaming is aimed to be strengthened by the study. For iOS, red team methodology, custom payload engineering, execution of multi-phase offensive operations, and a discussion of ethical and legal aspects relevant to mobile red teaming are included in the scope. For those aiming to improve their expertise in offensive strategies targeting iOS, the analysis is designed to serve as a resource.
Foundational context is combined with in-depth technical modules within the research approach. Red team principles tailored to iOS, including reconnaissance, software architecture, and payload delivery, are covered in the beginning segments. Through practical guidance on

executing offensive operations and the specifics of iOS payload engineering, this theoretical basis is enhanced. Advanced topics such as zero-click delivery, persistence techniques that avoid jailbreaking, remote control, surveillance payloads, data exfiltration, and anti-forensic measures are detailed in the following sections. On offensive iOS security, a rigorous and practical perspective is fostered, and literature review, comparative analysis, and some real-world examples are drawn down on by the work.

A rapidly advancing field is shown by contemporary research in mobile offensive security, with notable studies into threat modeling, malware analysis, and exploitation of vulnerabilities on both personal and enterprise devices. The development of iOS encryption techniques, the emergence of sophisticated spyware like Pegasus, and the documentation of sandbox and application security have expanded the knowledge base. A unified operational framework for offensive security requires the integration of these disparate findings—an aim addressed by this paper's synthesis of current methodologies and engineering practices — despite these advances. A logical progression from foundational concepts to practical applications and reflections is provided by the structure of the work. After establishing the context and core question, iOS security architecture and the threat landscape are examined in the following chapters. Following that red team methodology is detailed, reconnaissance techniques are explored and payload delivery and engineering are analyzed. Post-exploitation practices and anti-forensics are reviewed and, it concludes with a summary of findings and future outlook.

2. Mobile Security Architecture and Threat Landscape

For a basic comprehension of the defense and attack of mobile platforms, especially iOS, understanding the underlying security principles and the security threat landscape is key. This section aims to explore the system's architecture, application security, and attack targets to set the foundation for the defensive side as well as the attack strategy in later chapters.

2.1. OS Security Model Overview

The fundamental security principles that underlie iOS will be explored in this section, with a focus being placed on its application isolation mechanisms, system safeguards, and hardware-enforced protections. The grasp of the platform's resilience against attacks and the vulnerabilities that advanced adversaries may target is considered essential through an understanding of these foundational elements. Critical insights are provided for both offensive simulations and defensive strategies discussed throughout the work, situated within the broader context of mobile security architecture.

2.1.1. Sandbox Architecture and Restrictions

iOS operates on a sandbox structure to isolate applications from each other and from the operating system. Each app has its own container area that restricts an app's access to other app data and to sensitive system components. This drastically minimizes the possibility of system-wide damages if an exploit is successful by confining an attacker to a specific app's container area. This allows the OS to protect system-level services and operations. Schmidt (2015, p. 1) states that this enables developers to compartmentalize applications into separate containers that provide isolation for runtime resources as well as the file system. Shah (2010, p. 4) supports this, mentioning that the iOS application sandboxing system also controls direct system calls to sensitive APIs and devices. Since an attacker cannot access the system resources and the file system directly, they have to communicate between different app containers through sockets or through other apps on the iOS device. This means an adversary has to focus more on inter-container communication mechanisms rather than target file access. The constraints imposed by Apple's sandbox are stringent and must be taken into consideration when building out attack chains on iOS targets. Another Apple mechanism is Mandatory Access Controls (MAC) that prevent the loading of unsigned code or unauthorized injection of libraries. Schmidt (2015, p. 4) mentioned that MAC on iOS is used to disable unauthorized code injection as well as disable unauthorized execution of arbitrary code. On the system architecture layer, the ARM CPU used by iOS devices utilizes hardware-backed No-eXecute (NX) bit. This feature denies code execution from pages such as the stack and the heap (Schmidt, 2015, p. 2). The ARM CPU utilizes NX Bit to harden pages containing user data, and it prevents traditional buffer overflows by preventing code from executing in these pages. This makes memory exploits, such as buffer overflows and ROP (Return-Oriented Programming), highly difficult on iOS. In the past, memory exploits of this type were often the cause of successful sandbox bypasses. With NX Bit enabled, adversaries have to explore more complicated techniques, such as gadgets chaining or malicious use of legitimate control flows. Further enhancing Apple's security design, apps have to be code-signed with cryptographic signatures before they are deployed. This prevents applications with malicious code or other unauthorized applications from running. For example, the sandbox in combination with code signing effectively blocks library injection and code injection (Schmidt, 2015, p. 2). All of the aforementioned security controls and mechanisms enable such an iOS sandbox environment, making it even more difficult to escape.

Despite the many measures taken to prevent iOS sandbox escapes, there have been a number of successful jailbreaks. These jailbreaks illustrate that state-sponsored actors and determined attackers are able to bypass many of the restrictions that Apple's iOS enforces on applications. Wardle (2014, p. 2) and Stewart (2014, p. 1) state that an attacker is able to obtain elevated privileges outside the sandbox by finding an exploitable hole in a trusted system API. An attacker can also bypass sandbox restrictions by exploiting inter-process communications, because these mechanisms operate outside the sandbox. When there is a zero-day vulnerability, an attacker will be able to execute commands from outside of the sandbox container. Successful exploits of Apple security features and mechanisms, such as the ones from the Flashback malware and OSX/Crisis rootkit, bypass many security features that attempt to constrain an app to the sandbox, making them very effective at breaking into these types of apps (Wardle, 2014, p. 7; Stewart, 2014, p. 1). All these exploits of various degrees of sophistication allow an attacker to fully escape the iOS sandbox and execute code outside the app container. Many of these exploits require chaining together lower-severity exploits, illustrating a point on how effective an iOS sandbox is when properly patched. This point is highly relevant to iOS security in red team operations, in that a lack of updates or weak configurations can be targeted to escape the sandbox. Therefore, monitoring iOS systems and timely application of patches is a critical defense against sandbox escapes. As red teams assess iOS sandboxes, they must be creative in targeting vulnerabilities that lie outside the sandbox restrictions themselves and focus on vulnerabilities within device synchronization, cloud infrastructure, and other areas outside the sandbox perimeter. This also means avoiding file system-based exploits, since there is limited access, and focusing on memory exploits that can target specific components of an app running in the sandbox. Therefore, the exploitation of APIs, communication protocols, and other aspects should be validated to be properly used, in terms of validation and integrity of incoming and outgoing data and processes.

Even legitimate digital forensics becomes complicated in iOS devices. Since each application container is isolated from other applications, data stored in one application container can't be directly accessed from another. In iOS, root privileges are required to access data outside the sandbox perimeter (Schmidt, 2015, p. 6), but on non-jailbroken devices, this is impossible unless a serious vulnerability can be exploited. Another challenge is that a number of files on iOS, such as contacts and calendar appointments, are stored in encrypted file containers (Schmidt, 2015, p. 6). These file containers are further hardened by unique AES hardware-backed keys embedded inside each individual device, which renders them unreadable on any other device. If the AES hardware keys and encrypted file containers are combined with the sandbox in iOS devices, the contents in those containers are impossible to access without exploitation to extract the AES hardware keys. These sandbox measures and encrypted files also make digital forensics very challenging, and in order to conduct useful and effective digital forensics on iOS devices, the encryption has to be bypassed using multiple exploitation techniques and processes, such as extracting AES keys to decrypt the containers (Andriotis & Tryfonas, 2015, p. 1). If AES key extraction is not possible, the extraction of relevant data becomes futile due to the encryption.

Despite the complexity of Apple's measures to restrict applications, the number of applications available on Apple App Store increased tremendously. By 2010, the App Store contained over 225,000 applications, which shows the extent of Apple's development ecosystem (Schmidt, 2015, p. 2; Shah, 2010, p. 4). While that is good for mobile app developers, it has expanded the attack surface to thousands of new vulnerabilities since each application is unique with different implementations, configurations, security controls, etc. The threat landscape includes malware, ransomware, spyware, remote access trojans, etc. Apple attempts to minimize this threat through its mandatory app review process and their uniform sandbox implementation, in addition to requiring code signing, to ensure that malicious code isn't added to applications prior to deployment. Apple's countermeasures detect unwanted deviations from normal app behavior. However, this process doesn't eliminate the thousands of vulnerabilities. This also highlights how hard it is to write a payload that executes inside an iOS sandbox. In order for a payload to work inside a sandbox environment, it needs to be a module that doesn't modify the program to a great degree or, even worse, that causes unintended faults in other modules of the program. The payload has to be properly integrated so that it doesn't create a single point of failure within an iOS application's code, causing other pieces to shut down with the payload. The countermeasures and controls used by Apple effectively prevent malicious code from running on iOS without them being detected by these checks.

Compared to iOS, Android's sandbox implementation has major limitations. The variability in hardware and software configurations introduces differences in sandbox features and behavior, and the fragmentation in operating systems contributes a number of vulnerabilities (Kumar, 2022, p. 22). Malware detection using Machine Learning in the Android environment has a high success rate, but due to a lack of uniformity across the platform, many of the research and techniques of Android cannot be used in the same way on iOS. Although Google does not maintain any source codes or implementations of the security countermeasures of Android, malware countermeasures are still a serious issue for both iOS and Android malware analysis. Kumar (2022, p. 22) states that mobile red teams have to be constantly developing and improving their tools and techniques for their campaigns to overcome the sandbox and other countermeasures that attempt to prevent attackers from bypassing these defenses, leading to privilege escalation or information theft from applications. Schmidt (2015, p. 2) emphasizes the importance of continuing research in both the Android and iOS worlds in order to better learn the interactions within sandbox and bypass attempts in one or both platforms. With a larger and growing mobile sandbox ecosystem, with complex mobile applications running in controlled environments, this requires continuous research to find new or even basic bypass methods for sandboxes in order for the red team and pen testing efforts to achieve success and proper simulations for iOS attacks.

2.1.2. Application Security Framework

The iOS application security framework ensures that iOS applications are built, deployed, and executed on devices securely. For this, digital code signing, or simply signing, is used as an integrity mechanism to assure that only verified code from Apple can run on devices. As defined by Schmidt (2015, pp. 1–4) and Shah (2010, p. 4), it helps guarantee the uniformity of the system. It prevents running arbitrary code, and attackers must bypass this restriction in order to continue with the execution of a payload.

For developing applications on iOS, Objective-C is a common language of choice. Due to Objective-C's dynamic runtime nature, techniques like method swizzling, message interception, and other means of memory tampering are possible and can be leveraged by attackers. As explained by Shah (2010, p. 4), if code auditing does not address the risks inherent to Objective-C and other dynamically typed languages, developers are more likely to introduce security vulnerabilities in applications.

Application security often relies on memory management. For example, issues such as dangling pointers or privilege escalation by incorrectly managing object lifetimes are possible, and according to Shah (2010, p. 4) have also been exploited in the wild. The need for better memory management techniques rises as a developer starts using Objective-C, because developers may unknowingly apply unsecure principles while coding.

The code signing mechanism, alongside an application's sandbox, helps limit the impact of such misconfigurations. Due to the requirement to sign the application package and embed certain entitlements, the installation of unauthorized or malicious applications is prevented. As stated by Schmidt (2015, pp. 1–2) and Shah (2010, p. 4), the signed app provides protection against installation of a modified version of it, as long as the target device has not been compromised via system or hardware vulnerabilities. This defense can provide a considerable obstacle for red teams in gaining persistence in iOS applications or performing further actions. App functionality and flexibility versus security is a tradeoff. Shah (2010, p. 4) writes that the increased complexity of an application leads to trade-offs that may hurt the overall security. Also, applications will often remove some checks or relax code-signing settings during debugging. These are only a couple of examples of the constant struggle developers have, while aiming for both functionality and flexibility while also maintaining security.

Applications for iOS run in sandboxes, i.e., isolated environments that are associated with specific user accounts, which are used by the OS to track a certain application (Schmidt, 2015, pp. 1–4). Each application is installed into a distinct sandbox, limiting interaction, and it has its access to APIs restricted based on sandboxing policies. For this reason, inter-application communication is significantly hindered, as well as the installation of malicious apps (Schmidt, 2015, pp. 1–4).

At the application layer, iOS imposes access controls over sensitive device features (Schmidt, 2015, pp. 1–4). To access these features, API limitations must be circumvented using exploits within the operating system, or at the application-level. Furthermore, it ensures that applications cannot execute background processes, or send or receive data without user consent, preventing unauthorized information transfers (Schmidt, 2015, pp. 1–4). This provides some reassurance that, at the application layer, the damage that a vulnerable application can cause is limited. The restriction for background execution allows developers to make sure that no tasks can run silently in the background and affect device security or battery life (Schmidt, 2015, pp. 1–4). Finally, if location services are used, the applications cannot obtain the exact location of the device without user consent (Schmidt, 2015, pp. 1–4). These controls implemented on top of other general security and privacy policies can contribute significantly to an overall secure environment, if correctly implemented in the application and respected at the OS-level (Schmidt, 2015, pp. 1–4).

Sandbox policies and permission models continue to evolve, guided by lessons learned on mobile devices. Schmidt (2015, pp. 1–4) claims that the high incidence of Android malware stems from inconsistent application security policies and permission enforcement due to customization of permissions by Android OEMs (Original Equipment Manufacturers). The fact that this fragmentation is controlled and the uniformity of policies by Apple can explain the lower incidence of malware in iOS devices (Schmidt, 2015, pp. 1–4). For red teams, this policy requires a more streamlined approach to attacking an iOS device.

All of these points of this subsection are considered crucial defenses that Apple implemented, and they help reduce the surface for attacks. They contribute to creating a secure environment on iOS devices and prevent many common attack techniques that have been observed in practice. However, it has also been observed that although the technical aspects of security and protection mechanisms are enforced at multiple levels (operating system, app store), poor coding techniques or human factors contribute to frequent cryptographic protocol vulnerabilities. These vulnerabilities often involve accepting expired certificates or not verifying server certificates (Dananjaya, 2023, p. 2). The OWASP mobile security top ten issues 2014, 2016, and 2018 include many instances of improper platform usage. When implementing cryptographic protocols in apps, developers are responsible for not using flawed configurations. For example, a misconfigured certificate, or implementation issues such as relying on an outdated or insecure library, makes the data in transmission vulnerable (Dananjaya, 2023, p. 2). Furthermore, cryptographic issues in web servers often stem from failing to change default values after initial installations (Dananjaya, 2023, p. 2). For example, allowing SSL 3.0 to be available for connections still poses a risk because it is easily compromised with the POODLE exploit (Dananjaya, 2023, p. 2). However, a change can also create more risks for a system administrator: switching from a weaker to a stronger cipher can increase the computational load, especially during initialization, and slow down connections from machines with weaker hardware (Dananjaya, 2023, p. 2). While they may pose risks, these protocols offer a solid framework for encrypting sensitive data, while still permitting for a simple transition.

As described above, device capabilities in terms of both CPU power and memory size have improved exponentially, leading to increasingly complex payloads of malware (Delac, 2011, p. 2). As an example, the most popular iOS-based device, the iPhone, has its clock rates ranging from 800 MHz to more than 2 GHz (Delac, 2011, p. 2). Regarding the physical memory space, it goes from 256 MB to 4 GB (Delac, 2011, p. 2).

With higher speed and larger RAM, attackers can run complex and resource-demanding attacks. This means that applications also have the ability to handle and store more data. This is a security concern, as malicious apps may send all collected data through network interfaces to an attacker (Delac, 2011, p. 2). In response, mobile application security has to adapt accordingly.

Finally, threat intelligence provides a structured way of evaluating attacks and evaluating the effectiveness of defense strategies for mobile security frameworks such as the iOS application security framework. The MITRE ATT&CK Mobile Matrix catalogs attack techniques with documented instances of their execution. In addition, Zimperium (2023, p. 2) explains that it provides mitigation strategies and countermeasures against threats to mobile operating systems and applications. The effectiveness of an organization's security mitigation strategies may be assessed and tested (Hubbard, 2020, p. 5). It encompasses a wide range of documented attack techniques in a comprehensive list (Zimperium, 2023, p. 2). It maps mobile malware and application threats to both Android and iOS platforms (Hubbard, 2020, p. 5). MITRE ATT&CK's knowledge base is curated from documented reports as well as actual breaches and real incidents. By following the MITRE ATT&CK paradigm, red teams can iteratively test mitigations.

2.1.3. System Protection Mechanisms

iOS system protection mechanisms are the culmination of hardware and software security features. It starts with AES 256 crypto engine embedded in all iOS devices to reduce the load on the CPU during cryptography and, as noted by Teufl et al. (2013, p. 4) and Schmidt (2015, p. 6), decreases system latency. This proprietary AES crypto key, known as the UID key, is created during hardware manufacturing in the embedded security element and is not software accessible. Teufl et al. (2013, p. 4) note that this protects the system from possible key leakage. The UID key is also used as input for other keys, such as 0x89B and 0x835, which are used for encryption on the app and file level respectively to further compartmentalize encrypted data (Teufl et al., 2013, p. 4). As the system's cryptography keys are embedded at the hardware level, physically extracting data from the phone, either by NAND mirroring or chip-off methods, will only reveal ciphertext. Decrypting the ciphertext without the original hardware is, according to Teufl et al. (2013, p. 4), infeasible. One other aspect of Apple's Data Protection system involves the use of password-based key derivation with the PBKDF2 key derivation algorithm. By mixing a user's passcode with the UID key and a cryptographic salt, the system ensures that key derivation takes 80 milliseconds, improving defense against brute force passcode guessing and emphasizing the importance of passcode complexity (Teufl et al., 2013, p. 6). The PBKDF2 algorithm iterates hashing the passcode to add entropy even for simpler passcodes and helps to prevent mass brute force attempts. Yet short and simple passcodes are often used, and should an attacker gain temporary physical access to a device, they could bypass Apple's security and crack a simple password. As Teufl et al. (2013, p. 6) point out, this computational overhead for each brute force passcode attempt provides a challenge not only for the attacker but also legitimate users who need forensic recovery.

The NX Bit on ARM devices is another piece of the hardware component of Apple's security. By marking the stack and heap as non-executable, Apple can prevent buffer overflows or traditional shellcode injection into the device (Schmidt, 2015, p. 4). Combined with ASLR, the increased unpredictability adds difficulty to traditional memory corruption exploitation. The majority of exploits will need to chain multiple vulnerabilities to first bypass ASLR and then the NX Bit. According to Schmidt (2015, p. 4), "Executing traditional memory corruption exploits on such an environment is an extremely hard task." This being said, advanced attackers, such as the actors behind Flashback or OSX/Crisis, have shown that zero-day chain exploits still may work due to architectural shortcomings on Apple hardware and software (Wardle, 2014, p. 2). In addition to hardware level system protection mechanisms, iOS also has some powerful software defenses that help protect the system. Code signing for applications is a hard enforcement policy that prevents any unsigned applications on iOS devices from being able to run. All applications on iOS need to be signed, verified, and approved by Apple to be able to be installed (Shah, 2010, p. 4). By forcing these applications to be approved, Apple prevents the installation of traditional malware. Attackers have difficulty distributing their software by techniques such as drive-by downloads, and users are not able to download arbitrary third-party apps. According to Shah (2010, p. 4), attackers will have to bypass code signing by attempting exploitation at the kernel level. Even if the attacker attempts persistence, rootkits will need to get past Apple's code signing. This forces the attacker, according to Wardle (2014, p. 8), to perform advanced exploitation tactics that make use of a stolen provisioning profile. The authors of a jailbreak for iPhone OS 3.1, Shah (2010, p. 4) note, exploited code signing on the iPhone in addition to using privilege escalation to get their application to run on the phone. After Apple was informed, this vulnerability was quickly fixed in a patch.

This patching procedure is also handled at a hardware level. As Apple controls the hardware for iOS devices, they are able to provide updates to a variety of phones easily. This update patch is sent to all of Apple's supported hardware so that they all have the latest protection from malware authors and any other security exploits found out in the wild by security researchers or Apple themselves. By having all of the devices having the newest software, Wardle (2014, p. 1) remarks that this reduces the window of opportunity for attackers to execute zero-days in the wild. Zhou et al. (2025, p. 2) indicate that zero-day authors need to continuously invent new exploits due to their short shelf life. This hardware enforcement that supports timely patch release is a significant feature that contrasts with the fragmentation on other mobile operating systems, such as Android. Nonetheless, Wardle (2014, p. 1) argues that the weakest link still lies with users, who are slow to update their software even when prompted. Since updating to newer iOS versions, users leave their devices open to the newest exploits, malware, and vulnerabilities to enter in the wild. All of this security research would be ineffective if users do not update their devices to the latest version.

While Apple has made large strides in the advancement of its security, there still remains a challenge of providing this robust security for forensic researchers and the offensive security community. Hardware-backed encryption, combined with application sandboxing and code signing, according to Schmidt (2015, p. 6), causes for security attacks to have multiple layers. This results in the failure of traditional post-mortem forensics and live forensics due to encrypted application data and logs as well as limited inter-process communication on the operating system. These limitations, according to Pontiroli and Martinez (2015, p. 11), give incentive for security attackers to innovate their methodology by utilizing encrypted communication channels, runtime code obfuscation, and anti-forensic analysis measures in an attempt to evade defense. With attackers innovating, the defensive security community is forced to adapt their techniques accordingly by employing security measures that are reactive (Pontiroli & Martinez, 2015, p. 11).

Ultimately, Apple's hardware and software security measures are aimed at limiting attackers to having only a few chances to bypass its defenses, forcing them to spend the time and resources necessary to research and design bypass strategies. With this limitation and Apple's