TU WIEN Informatics

Android vs. iOS

Security of Mobile Deep Links

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Magdalena Steinböck, BSc
Matrikelnummer 01426254

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assistant Prof. Dipl.-Ing. Dr.techn. Martina Lindorfer, BSc

Wien, 19. April 2022

Magdalena Steinböck

Martina Lindorfer

TU WIEN Informatics

Android vs. iOS

Security of Mobile Deep Links

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Software Engineering and Internet Computing

by

Magdalena Steinböck, BSc
Registration Number 01426254

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dipl.-Ing. Dr.techn. Martina Lindorfer, BSc

Vienna, 19th April, 2022

Magdalena Steinböck

Martina Lindorfer

Erklärung zur Verfassung der Arbeit

Magdalena Steinböck, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 19. April 2022

Magdalena Steinböck

Danksagung

Allen voran bedanke ich mich bei meiner Betreuerin Assistant Prof. Martina Lindorfer für die Möglichkeit, an dieser Diplomarbeit zu arbeiten, für ihre durchgehende Unterstützung und hilfreichen Anregungen. Weiters bedanke ich mich bei Jakob Bleier für die Bereitstellung und Vorbereitung des iOS-Datensets.

Ich möchte mich auch bei meinen Eltern bedanken, die mir mein Studium ermöglicht haben und bei meinen Freunden, für ihre mentale Unterstützung.

Kurzfassung

Bridge the Gap bezeichnet einen Trend, Smartphone Apps von einem Web-Browser aus zu starten. Das wird erreicht durch sogenannte *Deep Links*, die es ermöglichen, direkt zu spezifischen Inhalten einer App zu navigieren. Die dadurch gewonnene Fusion zwischen Web und nativen Apps hat jedoch auch neue Angriffsvektoren geschaffen. Für das Open Source Betriebssystem Android gibt es bereits zahlreiche Untersuchungen, die zeigen, dass Deep Links aufgrund ihrer Anfälligkeit für Bedrohungen wie beispielsweise Hijacking ein Sicherheitsrisiko darstellen können. Apples proprietäres Betriebssystem iOS verfügt über eine ähnliche Umsetzung von Deep Linking-Mechanismen, allerdings wurde bisher nur wenig Literatur dazu veröffentlicht – vermutlich aufgrund des fehlenden Source Codes.

In dieser Arbeit untersuchen wir die Sicherheit von Deep Links auf iOS. Zunächst präsentieren wir bekannte Angriffsszenarien für Android, sowohl in Bezug auf *Custom Schemes*, als auch auf *App Links*. Anschließend untersuchen wir deren Anwendbarkeit auf Deep Linking-Mechanismen von iOS. Dafür entwickeln wir für jedes Szenario vulnerable Apps, die besagte Sicherheitsprobleme implementieren und analysieren, ob diese von einem Angreifer potentiell ausgenutzt werden können und mit welchen Folgen. Die Resultate stellen wir jeweils mit der zugehörigen Situation auf Android gegenüber. Abschließend, um die tatsächlichen Auswirkungen von Schwachstellen in Deep Links auf reale Endnutzer darzustellen, untersuchen wir die Verbreitung von Deep Links anhand eines Datensets von über 11.000 iOS-Apps aus dem offiziellen Apple-App Store.

Abstract

Bridge the Gap is a trend that aims to allow web browsers to start smartphone apps on a mobile device. This is achieved by so-called *Deep Links*, which enable direct linking to specific in-app resources. However, the resulting fusion of the web and native apps also introduces new attack vectors. There are numerous studies on security and privacy concerns of Deep Links on the open-source operating system Android, showing that these are prone to threats such as hijacking. The proprietary operating system iOS has a similar implementation of deep linking mechanisms to Android. However, there are not many publications on this matter, possibly due to the unavailability of iOS' source code.

In this thesis, we investigate the security of mobile Deep Links. First, we present known attack scenarios for Android with regards to *Custom Schemes* and *App Links*. Then, we consider the applicability of these attack vectors to deep linking mechanisms on iOS. Therefore, we develop vulnerable apps implementing discussed security issues, analyze whether an attacker could abuse them, and what security and privacy implications this has. Next, we compare our results to the corresponding mechanisms and security concerns of Android. Finally, to gain an insight into the actual security implications of the presented attack vectors, we analyze the distribution of Deep Links in the wild, based on a dataset containing over 11,000 iOS apps from the official Apple App Store.

Contents

CHAPTER 1

Introduction

1.1 Motivation

With the popularity of mobile devices and their usage all over the world, both Android and iOS have become well known operating systems. Android had a market share of 71.83%, whereas iOS had a market share of 27.41% at the beginning of 2021 [50]. On both systems, programs - so-called applications (apps) – are isolated, i.e., they hold a unique directory and run in individual processes with restricted access to outside resources to protect the system from malicious actions. However, these constraints prevent apps from interacting with each other by default. Thus, to make them more versatile, both platforms enable the otherwise sandboxed apps to communicate with other apps through well defined inter-app communication mechanisms.

One mechanism is Deep Links, which allows the developer to define URIs to specified locations within an app. Thereby, apps can register a list of actions that they are capable of and provide these functionalities to other programs installed on the system, i.e., that certain links are opened in the app defining them. This also enables web-to-app communication, as apps can be launched directly from a link clicked in the browser. Initially, this functionality was implemented through Scheme URLs (Android) [33] or Custom URL Schemes (iOS) [9]. They allowed users to directly open an installed app, via clicking a link starting with the scheme corresponding to the app. However, these mechanisms were shown to be flawed regarding security and privacy in the past, enabling hijacking attacks, as they perform no verification to substantiate that a scheme is unique, i.e., no two apps register the same scheme [19, 45, 53].

Therefore, both platforms introduced new techniques to allow apps to handle defined actions: App Links (Android) [31] and Universal Links (iOS) [16] which work similarly. Their usage of HTTP(S) URLs and verification of the ownership thereof should provide more security, as - in theory - no app should be able to register the verified URLs of

another one. Nevertheless, this has to be properly configured, as misconfigurations can again lead to hijacking attacks [45, 9].

While this topic has been extensively researched in Android, there are not many publications on iOS. Especially the newer mechanism of Universal Links has as of now been very much unexplored. However, this does not mean that iOS does not have security flaws, as in 2019 alone 156 CVEs were found according to CVE Details[1] [21]. This thesis aims to explore the security of deep linking mechanisms in iOS and further determine whether it is more secure in this regard than Android.

1.2 Contributions and Limitations

In this thesis, we discuss the security mechanisms of mobile Deep Links on the platforms Android and iOS. Emphasis is laid on determining whether iOS offers more protection regarding deep linking procedures than Android. Therefore, we research what attacks on Deep Links can be performed on each system. As for Android comprehensive literature already exists, we build upon existing research. To evaluate the security of iOS, we will use a special Security Research Device [5] provided by Apple, as well as a jailbroken device. We analyze how Deep Links are established and registered on the device, including how iOS selects candidates to open registered links. Then, we choose a subset of vulnerabilities that we transform into Proof-of-Concept (PoC) attack scenarios on iOS. Further, we perform static and dynamic analysis of iOS services to gain an insight on internal procedures of iOS responsible for handling Universal Links. Additionally, we evaluate a real-world iOS app-dataset on Custom URL Schemes and Universal Links. We consider the popularity of both deep linking mechanisms in addition to misconfigurations and other obstacles.

In this thesis, we are going to answer the following research questions:

RQ1 Which attacks currently exist on Deep Links on Android and iOS?

RQ2 Are threat models for Deep Links on Android also applicable to iOS?

RQ3 Does one system provide more elaborated security mechanisms for Deep Links than the other?

Due to the short lifecycle of mobile operating system versions, we restrict the scope of this thesis to Android 11 and iOS 14, released in October 2020 and September 2020 respectively. As iOS is a highly complex operating system, we limit our static and dynamic analysis of iOS with regard to Universal Links to functions involved in opening them. However, we do not aim to fully reverse engineer this process due to the limited scope of this thesis. The dataset we use for our evaluation of real-world usage of Deep Links was provided by Tang et al. [52]. Due to size restrictions, they provided us with a relatively small set of 11,622 iOS apps. Sadly, parts of the binary seem to be corrupt for some apps. Thus, our analysis of Universal Links defined in the dataset may not be

[1] CVE Details: https://www.cvedetails.com/

exhaustive, as we had to extract them from the app's binary. As of writing, we could not retrieve a newer dataset for our analysis, as extracting iOS apps from the official Apple AppStore is not trivial.

1.3 Overview

The following chapter of this thesis discusses the state of the art of Deep Links in the mobile operating systems (OS) Android and iOS, including the underlying concepts they are based on as well as threats on Android, which are related to Deep Links, but are not within the scope of this thesis. In Chapter 3 we discuss several known attack vectors on mobile Deep Links, including their differences between both platforms based on existing literature. Chapter 4 describes our approach to exploring the effectiveness of selected attack vectors known from Android on iOS, as well as our corresponding findings. We also show different scenarios, including proof-of-concept (PoC) implementations of vulnerable applications. Furthermore, we discuss the security and privacy implications of each attack vector. In Chapter 5, we analyze the usage of deep linking mechanisms in a sample of different iOS apps. We discuss several aspects, such as misconfigurations, collisions and relations for both, Custom URL Schemes and Universal Links. Chapter 6 discusses potential future work. Lastly, Chapter 7 concludes this thesis with a final comparison and summary of Deep Links on Android and iOS.

CHAPTER 2

State of the Art

In this chapter, we discuss different mechanisms of Deep Links that are currently implemented in Android and iOS. Moreover, we describe how they can be established for an application (app) on both systems. Furthermore, in Section 2.3, we present attack vectors on Android, which are loosely related to Deep Links but are not further investigated in this thesis.

2.1 Deep Links in Android

On Android there are two possibilities to establish Deep Links within apps, which are Scheme URLs, following a defined scheme, and App Links, which use ordinary HTTP(S) links. Both of them operate on the principle of Intents and Intent Filters. Each of these mechanisms is discussed in the following paragraphs.

Intent Filters

Intents are generally used to export functionality to other apps on Android. To which Intents an app is able to respond to, is declared in its manifest-file as an Intent Filter [33]. On app installation, the Android OS adds the declared Intent Filters to an internal collection of all installed Intents [31]. There are two types of Intents on Android: explicit and implicit [33]. If an implicit Intent is used, the OS chooses suitable applications which are capable of performing the action. If there are more than one, the system prompts the user to decide which one should be started. An explicit Intent, on the other hand, contains the component name of the assignee and the OS starts the specified component. If the Intent Filter is matched with an Intent, then the app is eligible for responding to the given Intent. If an app does not provide any Intent Filters, it can only be invoked by an explicit Intent [33]. Khan et al. [43] illustrate these mechanisms in detail and compare them to those used in iOS.

```
<activity>
    ...
  <intent-filter android:label="@string/string-displayed">
        <action android:name="android.intent.action.VIEW"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <category android:name="android.intent.category.BROWSABLE"/>
        <data android:scheme="example" android:host="myhost"/>
  </intent-filter>
    ...
</activity>
```

Figure 2.1: Example of an Intent Filter entry in the manifest.

```
<intent-filter>
    ...
    <data android:scheme="myscheme" android:host="myhost"/>
    <data android:scheme="https" android:host="otherhost.com"/>
</intent-filter>
```

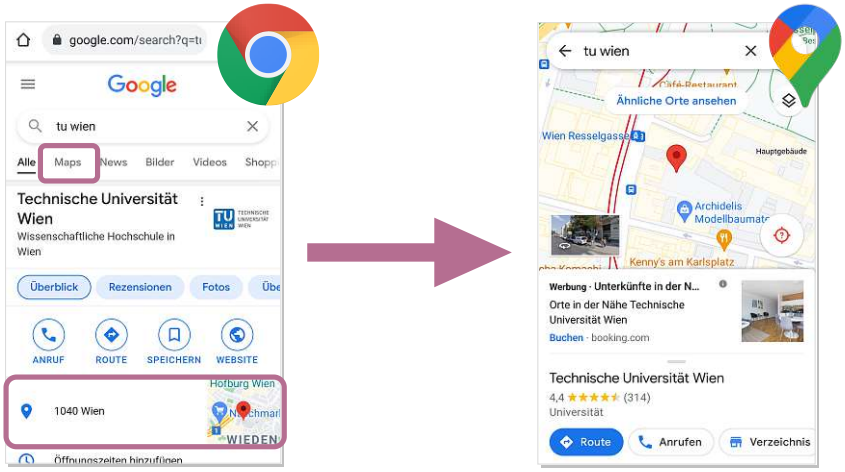
Figure 2.2: Example of an Intent Filter declaring two data elements.

According to the official Android documentation [30], each Intent Filter representing a Deep Link must declare the action `View` as well as the categories `DEFAULT` and `BROWSABLE`. `DEFAULT` allows the activity to receive implicit Intents. Otherwise, only explicit Intents could start the app. `BROWSABLE` is needed to allow the browser to start the app when the user clicks on a Deep Link. The data section specifies the details of the Deep Link, i.e., it defines the supported scheme, host, and paths. Fig. 2.1 shows an example of an Intent Filter specifying a Deep Link, where exactly one specific link of the form `example://myhost/mypath` is defined. Without the path-attribute, the app would support any path starting with `example://myhost`.

However, as mentioned in the Android documentation [25], the resolution of data-elements has unexpected behaviour. While it is possible to declare more than one data-element, they are not treated as self-contained URLs but as an extension to each other. Fig. 2.2 shows an example of an Intent Filter declaring two data-elements. With such a configuration, not only links of the form `myscheme://myhost` and `https://otherhost.com` are directed to the application, but also `https://myhost` as well as `myscheme://otherhost.com`, which might result in unexpected behaviour of the application if not handled properly. This problem can be prevented by specifying a separate Intent Filter for each distinct host.

Scheme URLs

Before Android 6.0, Scheme URLs in combination with Intent Filters were the standard procedure to take users to a specific content inside an app. In the case of Scheme URLs, the scheme can be freely chosen by the developers, as for example `comgooglemaps://` was used as a scheme by Google Maps. Fig. 2.3 shows an example thereof. The Deep Link to the Google Maps application is embedded in the highlighted field of the mobile Google Search site, as shown in Fig. 2.3(a). If a user clicks this link on Android, where



(a) (b)

Figure 2.3: Example of a Deep Link to Google Maps embedded into Google Search. (a) shows the embedded Deep Link in the mobile version of the Google Search site. (b) shows the opened Google Maps app after the user clicked the link in (a). (Google Chrome and Google Maps icon by Google)

the Google Maps app is installed, the system will automatically open the app and display the correct location, as can be seen in Fig. 2.3(b).

The general process of how Android decides which app should open a Scheme URL, as stated in the official documentation [30, 37], is depicted in Fig. 2.4. Once the user clicks on a link, the system determines a capable app previously set as a preferred app by the user. If no app is set as a preference, then Android tries to find a non-preferred app capable of handling the link, which it then starts. In some cases, there might be several apps registering the same scheme. Then, the user is presented with a disambiguation dialogue to choose which app should handle the link. There, the user can decide whether to open the link in the selected application once or set a preference, so that links of this scheme will always be opened in the chosen app.

However, this type of Deep Link is known to be vulnerable to different types of attacks, as shown by Chin et al. [19]. They explore several different attack vectors of Intent-based inter-app-communication, such as unauthorized Intent receipt, where an Intent is opened by an app which is not intended to do so, or Intent spoofing, where a malicious Intent is sent to a victim application to trigger specific behaviour. As developers are not forced to switch to the newer and more secure App Links, Liu et al. [45] found that Scheme URLs are still widely in use. They further confirm that apps using Scheme URLs are still vulnerable to link hijacking even in later Android versions, as the same URL can be declared by multiple apps and the ownership is never verified.

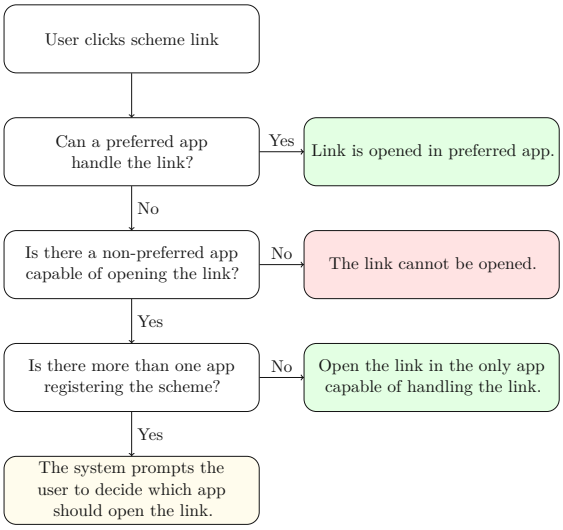


Figure 2.4: Process of Android to determine which app should open a Scheme URL. Modeled on the description of the Android developer documentation [30, 37].

```
<activity>
    ...
    <intent-filter android:label="@string/label-string-displayed-when-presented-to-user"
    android:autoVerify="true">
        <action android:name="android.intent.action.VIEW"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <category android:name="android.intent.category.BROWSABLE"/>
        <data android:scheme="http" android:host="myhost.com"/>
        <data android:scheme="https"/>
    </intent-filter>
    ...
</activity>
```

Figure 2.5: Example of an Intent Filter declaring an App Link.

App Links

App Links were first introduced in Android 6.0. This mechanism uses HTTP(S) URLs to open a link in an app as the default application. If the app is not installed, the link is opened in the browser instead. The procedure for registering App Links is similar to the process of declaring Scheme URLs, but instead of specifying a custom scheme, Deep Links can only use HTTP or HTTPS as schemes. Therefore, developers have to declare Intent Filters for the website's URIs in the manifest-file of the app [31]. Fig. 2.5 shows an example of an Intent Filter declaration for an App Link. The supported Deep Links are of the form `http://myhost.com` as well as `https://myhost.com`. Further, the attribute `android:autoVerify=true` is set, allowing the Android OS to automatically verify the declared Deep Links. Without this attribute present, no verification is performed by the system. Then, the system will open stated links in the browser instead of directly opening them in the app [37].

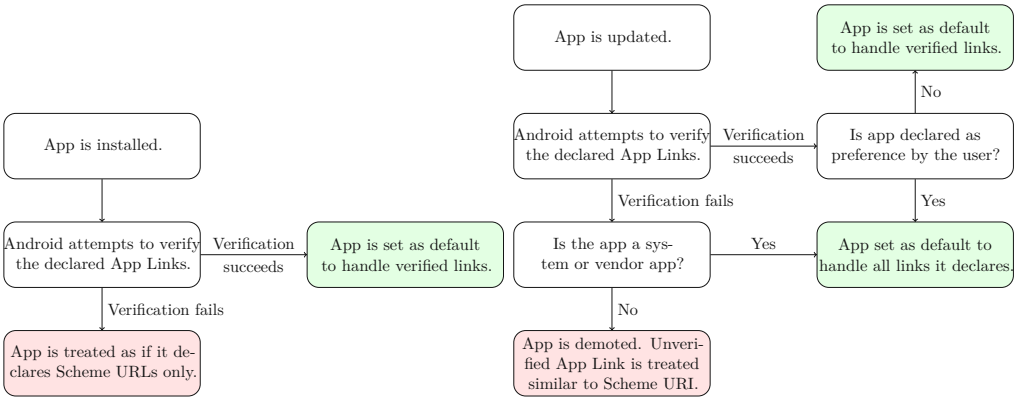
```
[
 ...
 {
  "relation": ["delegate_permission/common.handle_all_urls"],
  "target": {
    "namespace": "android_app",
    "package_name": "com.google.android.apps.maps",
    "sha256_cert_fingerprints":
    ["19:75:B2:F1:71:77:BC:89:A5:DF:F3:1F:9E:64:A6:CA:E2:81:A5:3D:C1:D1:D5:9B:1D:14:7F:E1:C8:2A:FA
        ↪ :00",
     "F0:FD:6C:5B:41:0F:25:CB:25:C3:B5:33:46:C8:97:2F:AE:30:F8:EE:74:11:DF:91:04:80:AD:6B:2D:60:DB
        ↪ :83",
     "3D:7A:12:23:01:9A:A3:9D:9E:A0:E3:43:6A:B7:C0:89:6B:FB:4F:B6:79:F4:DE:5F:E7:C2:3F:32:6C:8F
        ↪ :99:4A"]
  }
}]
```

Figure 2.6: Example of a Digital Asset Links (DAL) file as served by `https://www.google.com/.well-known/assetlinks.json`.[1]

Additionally, verification is required to ensure that the app is associated with the domain specified in the Deep Link. The verification of the association between app and website is performed by Android at the time of app installation and after each app update once. Therefore, as specified by the Android documentation [37], a Digital Asset Links (DAL) file named `assetlinks.json` must be hosted on the website in the directory `/.well-known` via HTTPS. Each distinct subdomain must host its individual file. However, each DAL file can hold entries for several apps. Fig. 2.6 describes the structure of an entry. The "target"-declaration specifies details of the app, such as the package name and the sha256-fingerprint of the app's certificate.

The actual verification is performed by the Android OS, as it downloads the DAL file and compares its data to the one declared inside the app [37]. Fig. 2.7 shows a simplified version of this process we reverse-engineered from the Android source code (version 11.0.0_r40) [26], where 2.7a shows the verification steps performed at installation, whereas 2.7b shows the verification performed at an update. During the installation process, first, the system requests the file from each host specified in the app's manifest. If the DAL file can be retrieved for each domain, the system compares the relation, the package name of the DAL(s) and the fingerprint with the respective data of the app. When all data matches, the verification state of the app is set to `INTENT_FILTER_DOMAIN_VERIFICATION_STATUS_ALWAYS`, in such a way that verified links will be opened by the app declaring them by default.

[1] source: extracted from `https://www.google.com/.well-known/assetlinks.json`, visited: August, 17th 2021



(a) Installation process (b) Update process

Figure 2.7: Process of Android to verify an app's declared App Links [26].

If the DAL file cannot be fetched for at least one host, or verification of the data fails, the app is marked as unverified, and the verification state is set to `INTENT_FILTER_DOMAIN_VERIFICATION_STATUS_ASK`, s.t. the user will always be presented with a disambiguation dialogue, to select whether to open the declared links in the app or in a browser. As this procedure is performed only once at the time of installation, the only chance to pass verification is through the next update. As shown in Fig. 2.7b, if an update triggers the validation process and at least one host does not pass the verification, an already validated app can be demoted. System apps, however, which stay present even after a factory reset, cannot be demoted, even if the verification of their declared App Links fails.

While using verification should make the process secure in theory, the proper application of the verification process seems to be challenging. Liu et al. [45] show that out of over 160,000 tested applications from the official Google Play Store, only around 2.2% pass the verification process due to misconfigurations. They further state that such errors can lead to hijacking attacks. However, even properly configured App Links are prone to such attacks since the introduction of Instant Apps, as discovered by Tang et al. [51]. They proposed three novel techniques to hijack App Links using a malicious Instant App, as these are preferred by the system.

2.2 Deep Links in iOS

Apple's iOS has similar deep linking mechanisms to Android. Custom URL Schemes are the old standard which should be replaced by Universal Links. Both are described in the following paragraphs. However, in contrast to Android, scientific literature on this topic is scarce.

```
...
<key>CFBundleURLTypes</key>
        <array>
                <dict>
                        <key>CFBundleTypeRole</key>
                        <string>Editor</string>
                        <key>CFBundleURLSchemes</key>
                        <array>
                                <string>myscheme</string>
                        </array>
                </dict>
        </array>
...
```

Figure 2.8: Definition of URL Types in `Info.plist`.

Custom URL Schemes

As described by Khan et al. [43], URL Schemes are used as a way of inter-app communication. Each app can define its own custom scheme, which works in a similar way to explicit Intents in Android. Also, developers can define handlers, which specify how a given URL query should be parsed [43].

When using XCode, a custom scheme can be created simply by adding a new "URL Type" at the "Info" tab of the target settings. Thereby, an entry is added to the `Info.plist` file of the project for `URL types` with sub-key `URL Schemes`, containing an item with a string value of the entered scheme, as shown in Fig. 2.8. Further, the documentation [9] describes that it is possible to add an identifier to the scheme, so that it is distinguishable from those of other apps, as well as a Role, expressing whether the app defines the scheme ("Editor") or not ("Viewer"). The handling of an incoming scheme URL request is performed within the app, depending on the life cycle model of the application, which can be either SwiftUI App (Swift UI) or UIKit App Delegate (Swift UI, Storyboard).

However, Khan et al. [43], as well as the official documentation by Apple [9], state that these custom schemes pose an attack vector. Xing et al. [53] describe scheme hijacking, where a malicious app defines the same scheme as another app. They state that iOS binds the app which was installed last on the system to the scheme, allowing the malicious app - if installed last - to steal data of a user. Thus, Xing et al. [53] developed an app abusing this property to obtain a Facebook authentication token for Pinterest. Further, they were able to publish the app in the official Apple App Store.

Universal Links

Universal Links were introduced in iOS 9. They are similar to Android App Links, as the OS redirects HTTP(S) URLs to a registered app if it is installed on the system. Otherwise it opens the link in the browser. iOS also verifies Universal Links of an app using a JSON-file on the publisher's website named apple-app-site-association file [4]. Fig. 2.9 displays an example of such an associate file. When using XCode, Universal Links can be registered to a project by adding the "Associated Domains" capability under the "Signing & Capabilities" tab of the target settings. This adds an entitlements

```
{
    "applinks": {
        "details": [
        {
            "appID": "<application identifier prefix / team id>.<bundle identifier>",
            "paths": ["*"]
        }
        ]
    }
}
```

Figure 2.9: Example of an apple-app-site-association file.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
            "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>com.apple.developer.associated-domains</key>
    <array>
        <string>applinks:myApp.com</string>
    </array>
</dict>
</plist>
```

Figure 2.10: Definition of an Associate Domain in `<app-name>.entitlement`.

file with the key “Associated Domains” and the given domain as value, as shown in Fig. 2.10. However, to register a Universal Link for an app, a provisioning profile with the feature “Associated Domains” enabled is needed, which requires a paid developer account. Universal Links also allow communication between apps including the sending of data according to the official documentation [14]. However, it also includes a warning to properly verify URL parameters, as these could potentially present an attack vector.

Universal Links have not been as well researched as Android App Links, which have several working examples on how to hijack them [45, 51, 19]. Liu et al. [45] present a superficial analysis of Apple association files. They propose that these are also prone to misconfigurations, however, an app can only open the links it has verified. Liu et al. [45] also state that while iOS makes security guarantees, whether they hold still has to be analyzed further. This thesis aims to analyze Universal Links in depth, and check whether attacks on App Links are applicable to these as well.

2.3 Related Threats on Android

In this chapter, we briefly explain related attacks on Android, which are based on the Intent-system but are not abusing Deep Links directly. First, we discuss different types of Intent Spoofing, where an attacker sends Intents with malicious payloads to a vulnerable victim. Next, we describe Intent Hijacking, where an adversary app seeks to intercept implicit Intents. Last, we shortly present how Malware Collusion can be achieved out on Android.

2.3.1 Intent Spoofing

As described by Chin et al. [19], Intent Spoofing is an attack where a malicious application sends an Intent to a victim application which does not expect to receive one, i.e., the victim contains a component which is unintentionally made public. This can cause the triggering of actions within the victim app, potentially leading to data leakage or crashes. We can distinguish between three types of Intent Spoofing attacks [19, 44], which are Malicious Activity Launch, Malicious Broadcast Injection and Malicious Service Launch.

Activity Launch

On Android, developers can export small components (activities) publicly so that other applications may use them. However, Chin et al. [19] describe that these exported activities can be a target of malicious actions, similar to cross-site request forgery (CSRF). Activities may be exposed accidentally. Then, a launch via an intent containing a malicious payload can cause unwanted behaviour of the victim. We discuss this threat model in greater detail in the section dedicated to CSRF of Chapter 3, as it can be applied to iOS.

Broadcast Injection

According to Chin et al. [19], Malicious Broadcast Injection targets Broadcast Receivers (BRs) which do not validate received data sufficiently. If such a vulnerable BR awaits system-broadcasts, but does not perform proper validation to verify that the received message was indeed sent by the OS, a malicious app could trigger the BR to carry out actions only the OS is permitted to request.

Service Launch

A publicly exported service allows other apps to start it and bind themselves to it, enabling bound applications to interact with the service [36]. However, unintentionally exporting a service can have serious security implications. Chin et al. [19] state that the possibility to start services from another component, which have been unintentionally exposed, cannot only break the Singleton-pattern, but more importantly, inputs to the service are controlled by the attacker, potentially leading to information leakage as well as execution of unauthorized or privileged tasks.

2.3.2 Intent Hijacking

Intent Hijacking [44], or also called Unintended Intent Receipt [19], is a problem involving implicit Intents. By design, the issuer of an implicit Intent has no way of knowing which application is chosen to receive the Intent, as this can depend on both, the Android system as well as the user's decision and preferences. Thus, as explained by Chin et al. [19], a malicious application could declare an Intent Filter containing all possible

Intents, potentially intercepting any Intent which is not protected by a permission. This cannot only lead to data leakage but also to phishing and control-flow-attacks [19, 49].

Broadcast Theft

Android defines three different types of broadcasts [28], which are ordered, normal, and local broadcasts. Ordered broadcasts are sent to one receiver at a time following a defined priority, based on declarations in the receivers Intent filter, allowing receivers to read previous results. Normal broadcasts send the message to all recipients at the same time while local broadcasts only send it to receivers within the sender's app. Before API level 21 (Android 5.0), there was also the broadcast type sticky [29], meaning that the Intent containing the broadcast stays available for some time after being sent. It was deprecated due to security concerns [29], since such messages could be read by any application declaring a corresponding Intent Filter and were even re-distributed to new receivers for a certain time frame, as Chin et al. [19] explained. However, Chin et al. [19] also found security issues involving ordered broadcasts, as these enable active denial-of-service attacks. Due to the broadcast being sent in an orderly manner, a malicious app could simply stop the propagation thereof. They further state that it is possible for the attacker to manipulate the content of the Intent, potentially even endangering the sender of the broadcast, since the sending component receives the result after the broadcast has been propagated to all recipients.

Service Hijacking

An app can send an implicit Intent to trigger an action to be performed by a service. If there is more than one service capable of performing the requested operation, the system selects the candidate at random, as observed by Chin et al. [19]. They further explain that a malicious service could be selected to respond to the Intent, potentially enabling false-response attacks, or even endangering the caller application, if callbacks are used. Due to this security threat, since Android 5.0 [33], it is no longer possible to bind a service to an app via an implicit Intent. Instead it is recommended [33] to use explicit Intents exclusively when communicating with a service, and not declaring Intent Filters when implementing a new service.

2.3.3 Malware Collusion

According to Liu et al. [44], malicious apps can work together (collude) to perform malicious tasks. This can enable attacks, which could not be carried out by a single app alone, e.g., several apps request one permission each instead of one app requesting all permissions, tricking the user into believing there is no harm in granting the single permission.

On Android, collusion can be performed by using the same sharedUserID among the malicious apps, as stated by Liu et al. [44]. The sharedUserID-property can be defined in the manifest [34], so that several apps of the same developer can share a Linux user ID. Thus, the apps can communicate using secure interfaces for data-access, without exposing them to the outside [34]. However, while still available to use, this mechanism was deprecated in Android 10 [34].

CHAPTER 3

Attacks on Deep Links (RQ1)

In this chapter, we explain different attacks on Deep Links. They abuse flaws of the overall approach on how Deep Links are established on mobile operating systems. Therefore, even if they have been discovered only on Android, such attacks are candidates to apply to iOS as well.

3.1 Scheme Collision

Neither Android nor iOS, provide protection mechanisms for URL Schemes to guarantee that they are registered by only one app. In contrast to App Links and Universal Links, the OS does not verify whether the declared schemes belong to an app by design. Liu et al. [45] describe that there are predefined system schemes, such as `tel://`, allowing multiple apps to share corresponding functionality. However, an app cannot define a unique scheme, which other apps are prohibited to register for themselves. This enables hijacking, as any app could declare itself a candidate for opening custom schemes of other apps, as shown in Fig. 3.1.

3.1.1 Threat Model on Android

According to Khan et al. [43], a malicious app can register a benign app's scheme by declaring an Intent Filter, to intercept implicit Intents it was not supposed to receive with the purpose of phishing and data leakage. Chin et al. [19] classify this attack as a special case of Intent Hijacking, called Activity Hijacking. However, this attack is not guaranteed to be successful, as the user decides which application should be started in case of a scheme collision. Yet, Khan et al. [43] argue, that a malicious app will most likely masquerade itself as a benign app to trick the user into choosing it. That is possible, as developers can set an icon and name, which will be displayed to the user for each Intent Filter declared in the manifest [32].

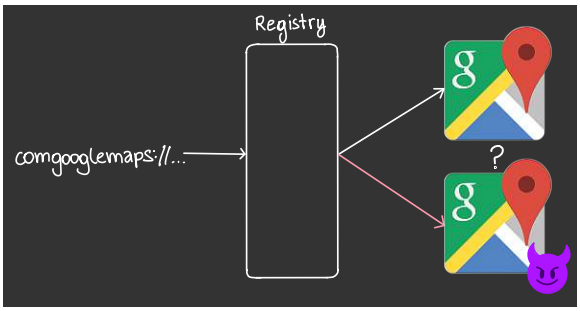


Figure 3.1: Schematic representation of Scheme Collision. Here, a malicious app disguises itself as the Google Maps app, registering the scheme URL `comgooglemaps://` (Google Maps icon by Google).

3.1.2 Threat Model on iOS

Khan et al. [43] describe that, while the internal procedure for inter-app communication differs and iOS places more restrictions on apps, the threat model for Scheme Collision on iOS is fundamentally the same as for Android. A malicious app can register the URL schemes of a benign app by requesting it in its `.plist` file. However, according to Xing et al. [53], the system will assign the last-installed candidate to handle the Deep Link.

3.2 Permission Re-Delegation

As described by Felt et al. [24], permission re-delegation is a type of "Confused Deputy"-attack [41], where the application exposing resources acts as the deputy, being abused by the requester, which has the intention to gain permission that were not granted by the user. The deputy can export functionality, both on purpose and accidentally [24]. However, this allows requesters to perform actions they would otherwise need permission for, and which have to be explicitly granted by the user. That way, the requester can circumvent the user's choice, potentially causing harm to the system, which is depicted in Fig. 3.2.

3.2.1 Threat Model (Android & iOS)

By providing inter-application communication, mobile operating systems allow apps to delegate actions to other apps. However, an app that publicly exposes resources can pose a potential attack vector for permission re-delegation if it holds permissions granted by a user. Other applications can abuse these exported components to execute privileged tasks on behalf of the victim.

A crucial example of such vulnerability was found in the Skype app for iOS. According to a blog post by Nitesh Dhanjani in 2011 [23], Skype accepted the Deep Link `skype://<number>?call`, which resulted in an immediate Skype-call to the given telephone number. In comparison, opening the `tel:<number>` scheme triggered a

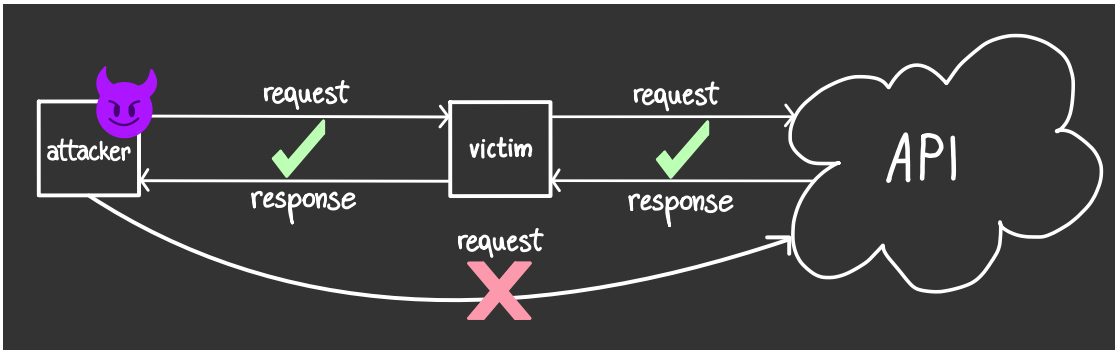


Figure 3.2: Schematic representation of Permission Re-Delegation. Here, a malicious app (attacker) abuses an app which has a privileged permission granted (victim) to execute API calls the malicious app has no permission for. Graphic based on Felt et al. [24], p.2.

system dialogue to confirm or cancel the action to call the given number [23]. Thus, by abusing the exposed Deep Link of Skype, an attacker could trigger arbitrary phone calls executed by the victim's Skype account [23].

Demissie et al. [22] identify an especially critical version of permission re-delegation, named Android Wicked Delegation (AWiDe), describing the abuse of delegation to execute a privileged task, such as an API call, using attacker-controlled data as input, without performing proper input-validation. They further state that this can be achieved by using explicit Intents, as these are not validated directly by the Android framework, in contrast to implicit Intents, but rely on a manual inspection performed by the Intent-receiver. As an example, they proposed an app providing URL expansion, but do not validate the input received from the sender. Thus, a malicious sender could potentially download Ascii-encoded Malware, which is embedded in the expanded URL, without having any permission granted. Therefore, permission re-delegation attacks can go unnoticed by the user, since they might not suspect such malicious behaviour from an app requesting zero permissions.

3.3 Cross Site Request Forgery

It is possible to abuse mobile Deep Links for Cross Site Request Forgery (CSRF) attacks. CSRF allows the performance of actions on behalf of the user, but without the user's consent and knowledge. According to OWASP [47], these actions can range from sending a malicious request to executing state changes, such as changing the user's password. Performing a CSRF attack on mobile devices requires a vulnerable app that does not perform validation on the parameters of the Deep Links it receives, e.g., opening any link given in an Intent in a WebView, or allowing actions to be carried out by Deep Links.

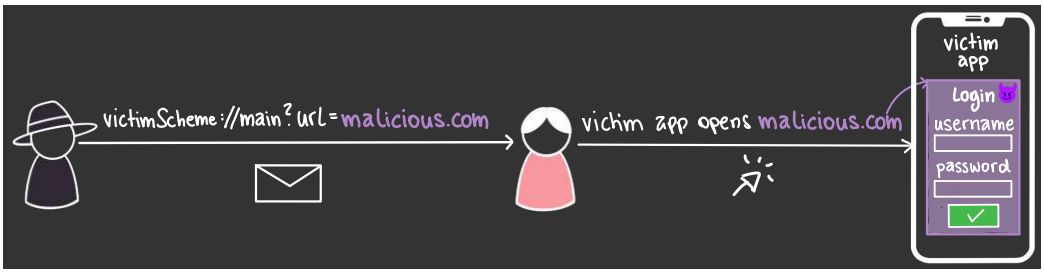


Figure 3.3: Schematic representation of WebView Injection. The attacker sends a malicious deep link to the victim, containing a URL to `malicious.com`, which is then displayed inside the victim app, to perform a phishing attack.

3.3.1 Threat Model – WebView Injection (Android & iOS)

Performing a CSRF attack on mobile devices requires a vulnerable app that does not perform validation on the parameters of the Deep Links it receives, e.g., This threat model is the same for both, Android and iOS. A vulnerable app does not correctly validate the parameters of a Deep Link, instead requesting the provided URL directly within a view within the app's user interface (UI), called WebView.

A real-life example of such a vulnerability can be found in GrabTaxi's Android and iOS app [38], where an attacker could open arbitrary links in the app's WebView. Furthermore, in 2019 researchers of CheckPoint [18] found that the popular TikTok app had a similar vulnerability. It was possible to send an Intent containing the parameter `url`, triggering the app to render the given URL in a WebView, sending the cookie of the currently logged-in user. Haupert et al. [42] even found this type of attack in the N26 banking application. A Deep Link triggered a popup rendering a provided URL within the app's UI, unrecognizable as foreign content, enabling phishing attacks.

3.3.2 Threat Model – Malicious API Usage (Android & iOS)

Further, an attacker could trigger an event by using a Deep Link, without any user-interaction, if a vulnerable app allows actions to be carried out by Deep Links. The Periscope app, for example, made the action to follow a user available through a Deep Link, thus allowing an attacker to force the currently signed-in user to follow an arbitrary user. This vulnerability was present in both the Android [39] and iOS [40] apps. Fig. 3.4 shows a similar situation. While this specific case might not be severe, the capabilities of such an attack should not be underestimated.

3.3.3 Threat Model – Malicious Activity Launch (Android)

Chin et al. [19] classify Malicious Activity Launch as a type of CSRF. Similarly to Malicious Service Launch, it describes the abuse of unintentionally exported activities through Intent Filters. Chin et al. [19] further define three types of Malicious Activity Launch. First, Victim Corruption [19], where the application state of the victim app is

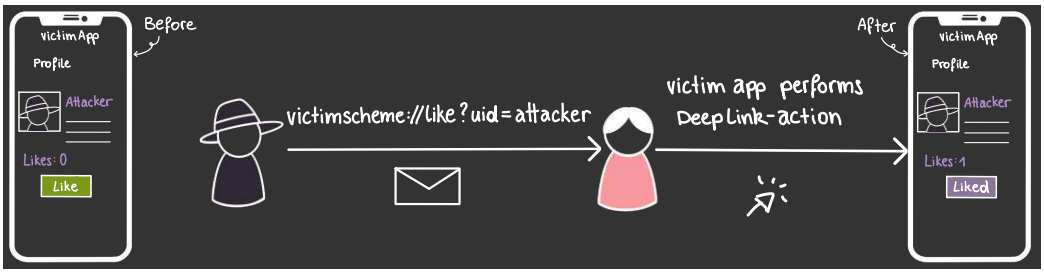


Figure 3.4: Schematic representation of Malicious API Usage. The attacker sends a malicious deep link to the victim, such that the victim app executes the like action on the profile of the attacker without the users interaction.

modified. Second, Phishing [19] can be performed by launching certain activities of other apps to trick the user into performing actions, e.g., logins. Last, the unintended launch of activities could cause Information Leakage [19], as the started activity might return data to the issuing application.

3.4 Attacks on App Links

App Links were introduced to Android as the successor of URL Schemes, requiring verification in order to prevent scheme collisions. However, Liu et al. [45] found that hijacking App Links is still possible. Apps failing the verification process of the OS can still be opened via their App Link, as the user is prompted to decide whether to open an unverified link in the respective app or the browser [45]. Further, a user can even set apps with unverified App Links as the preference for a link when prompted, such that the preferred app will subsequently open this link without further asking the user. However, as Liu et al. [45] discovered, not only the current link will be opened in the preferred app by default, but all links the app declares, even those failing verification. That enables malicious behaviour such as phishing and hijacking attacks.

3.4.1 Threat Model on Android using Instant Apps

With the introduction of Google Play Instant to Android in 2018 [35], so-called instant apps were added to the Google Play app market, with the intention of acting as a trial for the full version of an app. In particular, they are smaller versions of an app, with at most 15 MB in size, and can run on devices without installation. They allow the usage of Scheme URLs and App Links with a few additional requirements, i.e., all intent filters must provide both HTTP and HTTPS, and for each domain only a single instant app can be registered [8].

Tang et al. [51] discovered three attack vectors concerning instant apps and App Links. All of them are based on a malicious instant app, which incorporates a phishing module in its otherwise harmless content. Furthermore, it registers a URL of a benign victim as an App Link, which aims to hijack the link and launch the phishing module instead

of the victim app, despite failing verification of the App Link. The first attack Tang et al. [51] present is named Link Hijacking with Smart Text Selection (STS), which abuses the recommendations of STS when selecting text containing the victim URL to trick the user into opening the malicious instant app.

The second attack is similar to the first, but without abusing STS. Therefore, it is called Link Hijacking without STS [51]. Here, when the user tries to open the victim URL, the system ranks possible candidates to open the link, giving priority to instant apps. Thus, the malicious instant app could easily hijack any Deep Link defined by any non-instant app.

The last attack presented by Tang et al. [51] is Instant App Hijacking, where a malicious instant app hijacks the Deep Link of another instant app. They discovered that when the user clicks on a URL, which is registered by more than one instant app, the Android OS ranks the candidates based on their package names alphabetically descending and chooses the first sorting result to open the link [51]. This procedure happens without any further user interaction [51]. Thus, choosing a package name starting with a's will result in a high possibility of being chosen as the opener.

3.5 Summary

As discussed in the previous sections, there are multiple threat models considering mobile Deep Linking mechanisms on Android and iOS. Table 3.1 summarizes the introduced attack types, including their relevance for each platform. As it can be seen, only a few threat models are Android-specific, as they rely on features that are restricted on iOS, e.g., accessing components within an app or delegation. Furthermore, while other threat models apply to both systems, their implementation differs significantly. For example, for a scheme collision to be abused on iOS, the system must be tricked, while on Android, only the user must be persuaded to open a colliding link in a malicious app. As attack vectors on iOS are not as widely covered in literature as those on Android, we explore a subset of discussed threat models on Deep Links on iOS in Chapter 4. More precisely, we analyze Scheme Collision, Permission Re-Delegation, WebView Injection and Hijacking Universal Links. Further, we demonstrate Proof-of-Concept (PoC) exploits for scenarios that are feasible to implement.

Overview Attack Vectors		
Attack Type	**Android**	**iOS**
Scheme Collision	✓	✓
Permission Re-Delegation	✓	✓
Android Wicked Delegation	✓	×
CSRF	✓	✓
WebView Injection	✓	✓
Malicious API Usage	✓	✓
Malicious Activity Launch	✓	×
Hijacking App Links / Universal Links	✓	?

Table 3.1: Overview of attack vectors on Deep Links on Android compared to iOS.

CHAPTER 4

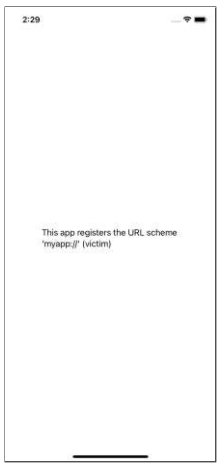
Exploring Attack Vectors of Deep Links on iOS (RQ2, RQ3)

In this chapter, we explore the applicability of the in Chapter 3 discussed attack vectors of Android on iOS. First, in Section 4.1, we analyse different scenarios of how Scheme Collision can be abused. The following section describes our findings on how Permission Re-Delegation can be achieved on iOS. In Section 4.3 we discuss the possibilities of Cross-Site Request Forgery (CSRF) by abusing a vulnerable Deep Link to a WKWebView and SFSafariViewController. Last, Chapter 4.4 explains our findings concerning the relevance of attack vectors of Android's App Links for iOS' Universal Links.

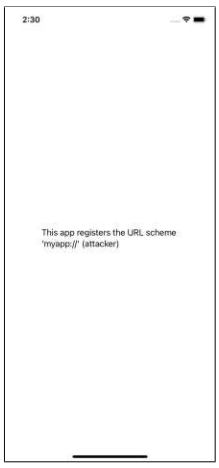
4.1 Methodology

To evaluate the applicability of the attack vectors on Deep Links introduced in Chapter 3, we developed Proof-of-Concept (PoC) iOS apps incorporating respective mechanisms and vulnerabilities. Further, we discuss how malicious actors can abuse these apps and which implications regarding security and privacy for the system and user emerge.

We built our PoCs using XCode 13.2.1 in Swift on a MacMini running macOS Monterey version 12.1. For the evaluation, we used real iPhone devices and the integrated XCode Simulator on iOS version 14.3. Our physical testbed includes a jailbroken iPhone 6S on iOS 14.0.1 and checkra1n beta 0.12.4 and a Security Research Device (SRD) iPhone 11 on iOS 14.3. To gain a better understanding of the underlying processes of the validation of Universal Links, we used Frida [2]. However, due to system restrictions on the SRD, at the time of testing, it was not yet possible to let Frida inspect the respective processes on the device. Thus, we installed Frida onto the jailbroken device to hook into system processes of interest.



(a) Victim app `a.myapp`



(b) Attacker app `b.myapp`

Figure 4.1: Screenshots of Scheme Collision test apps of Scenario 1

4.2 Scheme Collision

To examine how Scheme Collision can be achieved and behaves on iOS, we developed two simple apps, shown in Fig. 4.1. Each registers the same Custom URL Scheme `myapp://`. Clicking any link starting with the custom scheme results in opening one of the two apps. On iOS, only one app can be the opener for a Custom URL Scheme at a time, which the system chooses autonomously. Thus, there is no disambiguation dialogue as in Android, so the user does not play a role in selecting an appropriate app to handle the URL. However, if an app opens a scheme for the first time, the user must confirm the action by selecting "Confirm" on the dialogue shown in Fig. 4.2. Choosing "Cancel" stops the opening of the link but does not provide another app candidate. Further, the confirmation dialogue spawns each time the user clicks the scheme link until they choose "Confirm".

To determine which of the two apps is set as the opener for the URL scheme by the system and under which conditions, we prepared two scenarios:

- Scenario 1: Both apps have the same display name, but different bundle identifiers
- Scenario 2: The apps have different display names and different bundle identifiers

The display name corresponds to the string shown to the user as the app's name, e.g., at the App Store or on the home screen, whereas the bundle identifier uniquely identifies an app within the App Store and on the device. Consequently, multiple apps may share the same display name, but if they share the exact same bundle identifier, they are treated as the same app. Thus, when installing an app, another app with the same bundle identifier will be removed from the system.

Within the above scenarios, apps were installed in both possible orders to verify whether the installation order impacts the opening process of the OS. For testing, we used the

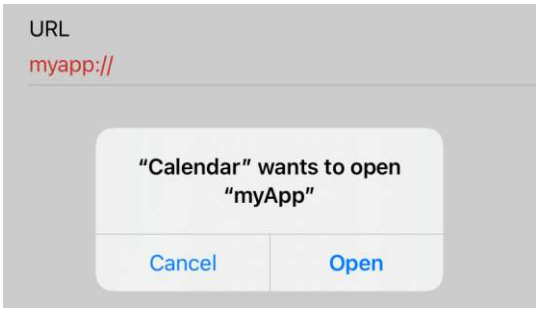


Figure 4.2: The dialogue shown to the user to confirm that an app may open a scheme URL.

XCode Simulator as well as a Security Research Device (iPhone 11), running iOS 14.3. However, installing the developed apps onto the physical device requires them to be signed. Therefore, we had to register a unique bundle identifier for each app to generate valid signing certificates. Thus, we prepended "m.steinb" to all bundle identifiers used in the scenarios.

4.2.1 Scenario 1 – Same Display Name, different Bundle Identifiers

Both apps share the display name `myApp`. The bundle identifier of the first app is `a` and of the second `b`. In the first step, we installed `a.myApp` before `b.myApp` on the device. Screenshots of both apps can be seen in Fig. 4.1. After both apps were installed, the opener of the scheme was set as `b.myApp` by the system, as shown in Fig. 4.3. Next, the device was reset, and this time we installed `b.myApp` before `a.myApp`. Again, the opener was set to `b.myApp`. Thus, we conclude that the order of installation does not determine which app can open a Custom Scheme URL on iOS 14.3.

During testing, we further discovered that if `b.myApp` is removed from the system, `a.myApp` is set as the opener. Then, iOS asks the user for confirmation again. However, if `b.myApp` is uninstalled and all its data removed from the device, then reinstalled, the OS does not show a confirmation dialogue to open links of the form `myapp://` in `b.myApp` if no other opener was confirmed by the user. This behaviour can be observed in both scenarios, suggesting that the system does not clear the data of the opener-app related to its Deep Link.

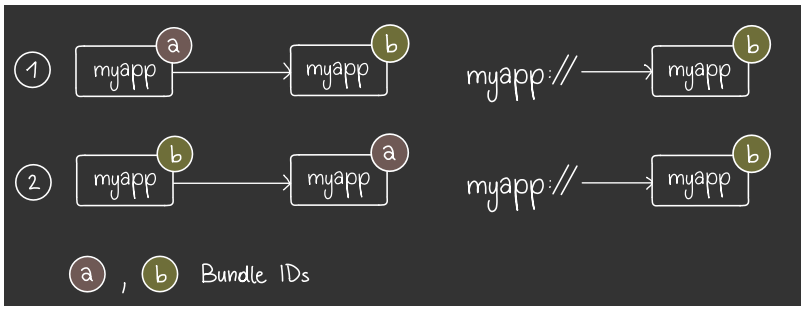


Figure 4.3: Test results for Scheme Collision of two apps with different bundle identifiers but the same display name which register the same scheme `myapp://`.

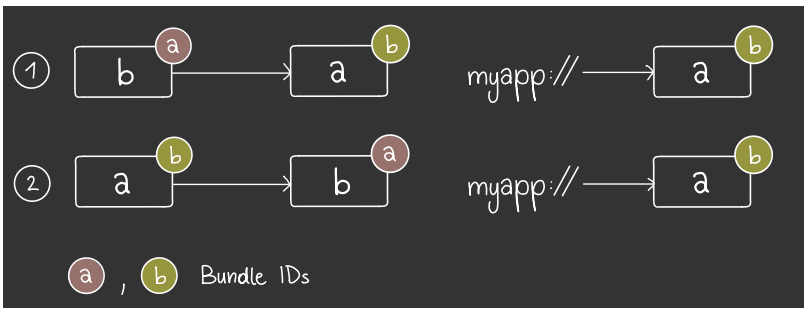


Figure 4.4: Test results for Scheme Collision of two apps with different bundle identifiers and display names which register the same scheme `myapp://`.

4.2.2 Scenario 2 – Different Display Names, different Bundle Identifiers

In this scenario, we defined two test cases to determine whether the bundle identifier or the display name influence whether an app is chosen as the opener of a scheme URL. First, we renamed the apps from scenario 1 to `a.b` and `b.a`. Then, we installed both apps onto the device in both possible orders and checked, which app will open the link `myapp://main`. The results can be seen in Fig. 4.4. In both situations, iOS chose b.a as the opener. Thus, we conclude that only the bundle identifier influences the selection procedure of the OS.

For the second case, we developed three apps. The first two apps are named `myApp` and `myScheme` and register the corresponding schemes `myapp://` and `myscheme://`. The third app's display name is `myAppScheme` and it registers both URLs. Fig. 4.5 shows the results of the experiment. In all cases, the app with the alphabetically last bundle identifier was able to open the app links. Again, we tested each bundle identifier rotation with different installation orders, yet it did not affect the result. Fig. 4.6 summarizes our findings on the opener selection procedure of iOS for Custom URL Schemes as a graph.

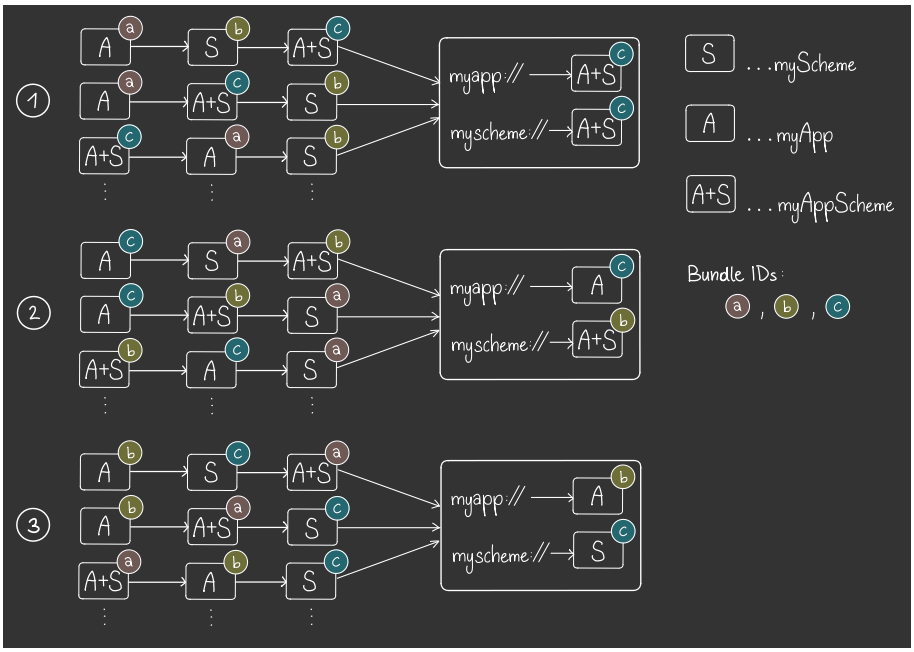


Figure 4.5: Test results for Scheme Collision of three apps with different bundle identifiers and display names.

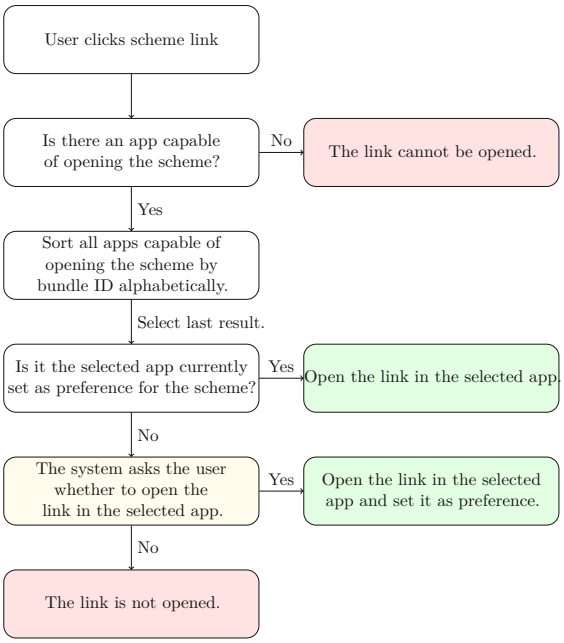


Figure 4.6: Process of iOS to determine which app should open a Custom URL Scheme, based on our findings.

4.2.3 Security and Privacy Implications

As we showed in the previous sections, if multiple apps apply for the same Deep Link, iOS chooses the opener among all candidates autonomously. While Android hands the responsibility to the user, iOS does not involve the user in the selection process. Instead, the system resolves collisions by first ordering all candidates based on their bundle identifier and then selecting the alphabetically last one. Hence, while iOS is responsible for choosing a suitable opener, it is intransparent to the user in which app a link might open. Malicious apps can abuse this behaviour to trick the system into opening them instead of the benign app, as attackers can obtain bundle identifiers of apps via static analysis. Then, by choosing a bundle identifier that is alphabetically lower than the victim's and other possible candidates', the malicious app will be the system's preferred opener, receiving the registered Deep Links instead.

Furthermore, Custom URL Schemes may contain sensitive information such as session tokens in plain text. Then, an interception by a malicious app poses a critical threat to the user. Therefore, Custom URL Schemes should not contain unencrypted sensitive information and should never be trusted to be received by the intended recipient.

Another significant difference to Android is that iOS does not allow users to set a preferred app for opening Custom URL Schemes. Instead, iOS re-evaluates the chosen opener with every newly installed app which registers an already claimed scheme. On the other hand, Android lets users set preferred apps for opening Deep Links. However, this may lead to malicious apps trying to get selected as a preference by the user to eliminate competition.

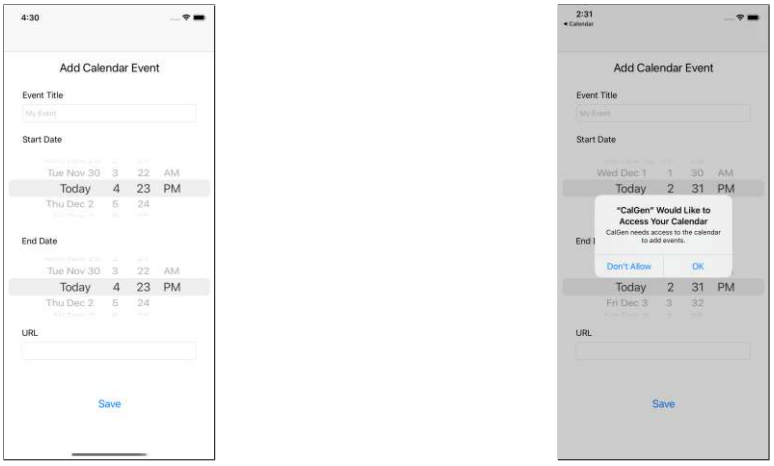
4.3 Permission Re-Delegation

As the implementation of inter-app communication mechanisms is fundamentally different between Android and iOS, only a subset of attack vectors of the prior remains in the latter. In particular, due to the unidirectional nature of communication via Deep Links in iOS, the receiver cannot directly return a response to the requester. Instead, by providing a completion handler to the open method [11], the requesting app can inquire to be informed by the OS whether opening the Deep Link was successful.

4.3.1 Scenario – Calendar permission

We created a PoC to show that a subset of Permission Re-Delegation attacks is still possible on iOS. Hence, we implemented an application called CalGen that is capable of adding events to the iOS Calendar. Fig. 4.7a displays a screenshot of the app's interface – a simple form to generate a calendar event. The app requires the permission to add an event to the calendar from the user, which it correctly requests and monitors, as shown in Fig. 4.7b.

Further, CalGen registers the Custom URL Scheme `calgen://`, which supports creating calendar entries programmatically. The idea behind this is not offensive but seeks to



(a) CalGen UI (b) CalGen permission request

Figure 4.7: Screenshots of the Permission Re-Delegation test app CalGen.

give the user an option to add a new event simply by clicking a link. Yet, a malicious actor could easily abuse this option to add arbitrary calendar events to the victim's calendar, including links to perform phishing attacks and more, without permission to do so granted by the user. Fig. 4.8 visualizes the corresponding scenario.

4.3.2 Security and Privacy Implications

On iOS, apps can provide functionality via Custom URL Schemes. However, it presents a potential risk to the system if such exported actions access protected resources or hold permissions. If the receiver does not correctly verify the caller, any other app could trigger an execution. Verification must be performed manually within the receiving app's code, as iOS does not provide automated mechanisms to restrict access to exported actions via Custom URL Scheme yet, e.g., ensuring that the requester needs to hold the same permission as the provider. Therefore, developers should refrain from exporting functionality via Deep Link, as exported actions can potentially be triggered from the web and other apps, thus posing a security and privacy risk to user and system data. Also, as Felt et al. [24] describe, permissions on iOS belong to the category time-of-use. That means, users do not have to grant third-party apps permanent permissions. Instead, they can also authorize apps access to permissions once or on a time-base, e.g., only when the user uses the app. Thus, the user can reduce the risk of permission re-delegation if they only grant them temporarily.

Furthermore, as we showed in our scenario, it may be intransparent to the user that an app exports a functionality via Deep Link. Then, if another app abuses this behaviour, the user might not even notice it, as only the victim app launches after clicking the link. In our scenario, the user does not see directly that the malicious app generated a calendar entry on behalf of the victim. Hence, if the receiver does not display what

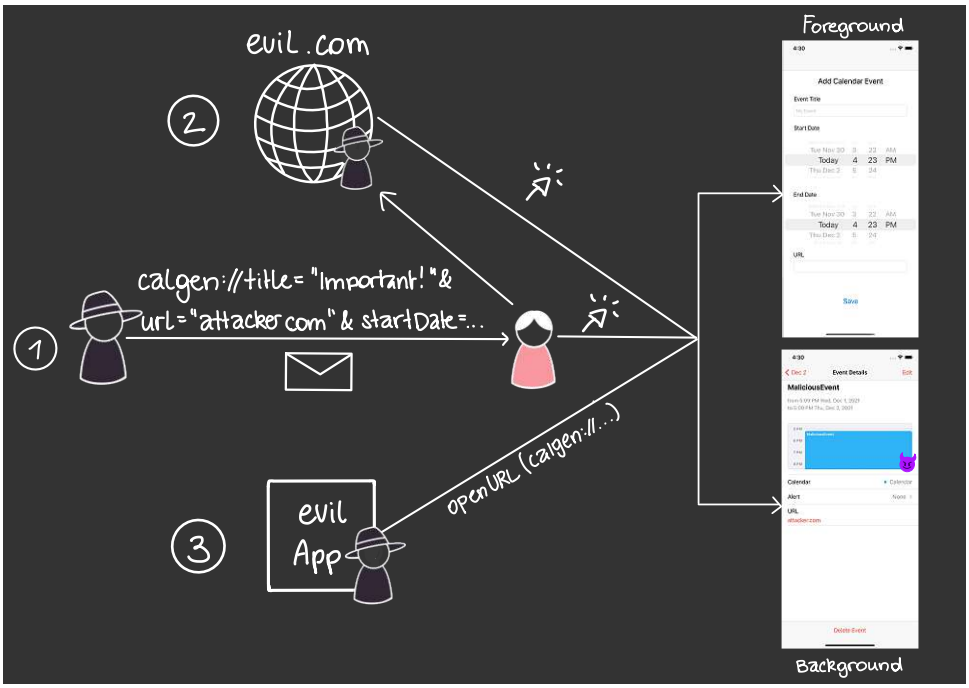


Figure 4.8: Three attack vectors for Permission Re-Delegation of the CalGen victim app. (1) The attacker sends the user a malicious Deep Link directly. (2) The victim clicks a link on a attacker-controlled website, triggering the malicious behaviour. (3) A malicious app installed on the users device can trigger the malicious behaviour by opening the Deep Link via the `openURL` method.

it is performing in the background, the user has no way of telling whether the app was launched legitimately or with malicious intent.

4.4 Cross Site Request Forgery – WebView Injection

In this section, we explore possible attack vectors regarding WebViews that are accessible via Deep Links. Similar to Android, iOS supports two different mechanisms for rendering external content within the app. First of which is `SFSafariViewController` [12]. Equivalent to Custom Tabs on Android, it is essentially a tab of the Safari Browser displayed singly. The second method of rendering web content from an app is making use of `WKWebView` [17]. It acts as an in-app browser, which can be seamlessly embedded within a view of the UI of an app, supporting navigation delegation.

4.4.1 Scenario 1 – WKWebView

Inspired by the findings of Haupert et al. concerning a WebView Injection in the N26 banking app on Android [42], we implemented a vulnerable iOS application that renders

(a) WKWebView

(b) JS is enabled in WKWebView

(c) SFSafariViewController

(d) JS is enabled in SFSafariViewController

Figure 4.9: Different methods to display web content within an iOS App. JavaScript (JS) is enabled by default in both versions.

any given URL provided by a Deep Link of the form `webviewapp://main?url=<URL>` in a WKWebView within the app. Fig. 4.9a presents a screenshot of the app's UI including the embedded WebView. The app further exposes a function to the WebView, to display a received message as a toast within the app.

As the app can render any given URL, an attacker could easily perform phishing attacks by presenting the user with a website that is styled like the app. Moreover, there is no address bar present within the WebView. Features such as an address or navigation bar have to be specifically implemented by the app. Thus, if these features are not present, it is not transparent to the user which webpage is currently displayed – the WebView is indistinguishable from regular app content. Further, during the evaluation, we found that JavaScript is enabled by default for the WKWebView, which can be seen in Fig. 4.9b.

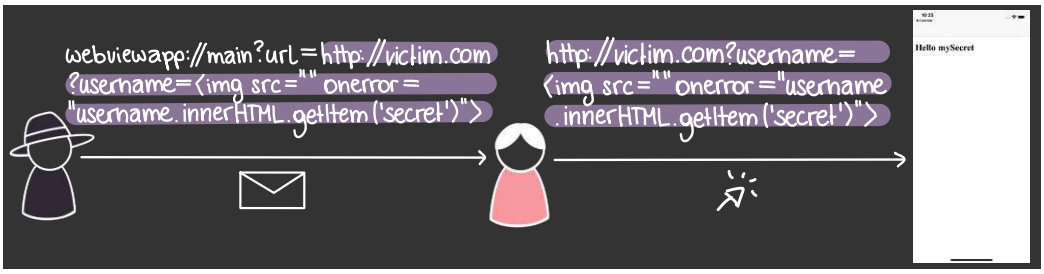


Figure 4.10: Procedure of a CSRF and XSS within a WKWebView.

```
<script>
    window.webkit.messageHandlers.notify.postMessage(
        'You have won! Go to evil.com to claim your price!'
    );
</script>
```

Figure 4.11: JavaScript embedded in the attacker's website, to trigger the `notify` function of the WebView app.

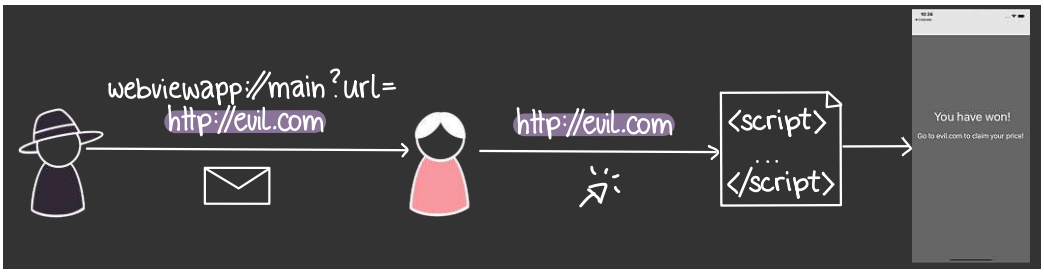


Figure 4.12: Procedure of triggering an exposed API function within a WKWebView.

Thus, an attacker could even perform a cross-site scripting (XSS) attack. However, it should be noted that, by default, WKWebView does not provide implementations for all JavaScript APIs. For example, the JavaScript `alert` function will not trigger the usual alert window, since the behaviour should be specified by the hosting app, not the embedded WebView.

We developed a PoC containing a website, where the query parameter `username` is vulnerable to XSS. Moreover, the webpage stores a secret in the local storage of the browser. This is for simplification and should demonstrate that the injected JavaScript has access to secrets stored in the browser session. By abusing the vulnerability of the username parameter, the attacker can send the user a malicious Deep Link of the form shown in Fig. 4.10, which is possible as JavaScript is enabled in the WKWebView.

Another vulnerability present in the WebView app is its exposed API function to receive a message from within the WKWebView and display it as toast. By embedding a JavaScript similar to Fig. 4.11 in a malicious website, an attacker can display phishing messages inside the app merely by sending the user a Deep Link, as shown in Fig. 4.12.

Figure 4.13: Procedure of a CSRF and XSS within a SFSafariViewController.

4.4.2 Scenario 2 – SFSafariViewController

We implemented another app, similar to the WebView app but using an SFSafariViewController instead of a WKWebView. It registers a Deep Link of the scheme `safariviewapp://`, which allows opening URLs within view that looks like a Safari tab. Fig. 4.9c shows a screenshot of said application. What we can see immediately when comparing both views is an address and a navigation bar. Further, in Fig. 4.9d in the top left corner exists an option to navigate back to the application that triggered the opening of the view controller. That is due to SFSafariViewController acting as a completed web browser, essentially being a Safari tab, and therefore using the same implementations for JavaScript APIs as the regular Safari browser. Further, it prohibits direct access of any user data or action by the app [12].

Fig. 4.9d shows that SFSafariViewController too enables JavaScript per default. Thus, we could use the XSS-vulnerable website of our prior experiment to trigger an alert. As it can be seen in Fig. 4.13, the scenario is almost identical to those for WKWebViews. However, we did not have to access an exposed API to trigger the alert, as the Safari JavaScript API already provides such functionality. In the end, the secret is displayed in a standard Safari alert window.

4.4.3 Security and Privacy Implications

As we showed in the previous sections, displaying any web link in either WKWebView or SFSafariViewController via Deep Link may entail unwanted consequences. Table 4.1 gives an overview of security and privacy considerations we will discuss in the following sections.

When an app displays web content in a WKWebView, users cannot distinguish between the WebView and the actual app content. Then, it is possible to perform stealthy phishing attacks if the WebView is used to render content given in a Deep Link, e.g., a login window that integrates well with the overall UI of the app to steal login information. On the other hand, SFSafariViewController differentiates web and app content by showing an address bar and looks similar to Safari, making it more straightforward for users to notice suspicious behaviour.

	WKWebView	SFSafariViewController
Address Bar visible	×	✓
JavaScript enabled by default	✓	✓
Access App Components	✓	×

Table 4.1: Overview of security and privacy concerns of WKWebView and SFSafariViewController.

Furthermore, JavaScript is enabled by default for both WKWebView and SFSafariViewController. Developers must explicitly disable it when creating one of these views. Allowing JavaScript when rendering any given link constructs a probable attack surface for CSRF and XSS attacks. Then, as WKWebView can interact with app components, an attacker might access an application's API.

4.5 Exploration of Universal Links

In this section, we analyze two scenarios regarding Universal Links on iOS. We take inspiration from threat models on Android, where unverified links can be opened by either setting the app as a preference for either this or any other App Link or manually selecting it as an opener among multiple candidates. Further, we describe relevant processes involved in the retrieval and validation of apple-app-site-association files. Last, we discuss the security and privacy implications of the investigated scenarios.

The first scenario we evaluate contains two apps associated with the same domain. However, only one passes the verification. We investigate whether the unvalidated app can open the link in a similar way as on Android, i.e., by being chosen by the user among all candidates. As manually setting an opener as preference is not possible on iOS 14, we also consider the case where the unverified app is set as the opener for a Custom URL Scheme already.

In the second scenario, we investigate two candidates associated with the same domain. More precisely, we examine what app is chosen as the opener for links of the respective URLs if both pass verification. First, the apple-app-site-association file assigns both applications the same path. In the second case, the candidates are responsible for different paths on the same domain.

4.5.1 Static and Dynamic Analysis

To gain a better understanding of the involved processes, we analyzed the logs of the device. Since iOS 14 [13], the Apple Content Delivery Network (CDN) fetches, caches and distributes apple-app-site-association files for verification purposes to reduce traffic to

```
Message: GET https://app-site-association.cdn-apple.com/a/v1/example-domain.com HTTP/1.1
Accept-Language: en-us
Accept: */*
Accept-Encoding: gzip, deflate, br
Apple-Proxied-Domain-Name: example-domain.com
```

Figure 4.14: GET request from the `swcd` to the CDN.

the hosting websites, as these files are requested and verified upon each installation and update of an app. Responsible for querying the CDN is a service daemon called `swcd` (shared web credentials daemon) that runs on the device. Fig. 4.14 shows the GET-request the service sends to the CDN for the domain `example.com`. Upon reinstallation of an application with a Universal Link set, the `swcd` logs a JSON containing the app link as well as the verification result. That indicates the existence of a dedicated database that stores claimed Universal Links and their verification status.

We used Frida [2] to dynamically analyze what information on Universal Links iOS stores on the phone. Therefore, we first inspected the `swcd` service on our jailbroken device using the command `frida -U swcd`, which instruments said process. There, we listed all included modules of the binary. We statically analyzed frameworks of interest manually by disassembling them with Hopper [20]. We discovered that the `SharedWebCredentials.framework` is included. Thus, we used the command `frida-trace -U -m "*[*SWC* *]" swcd` to log all called function once a Universal Link is clicked. Thereby, we found over 2,100 function calls.

The `swcutil` binary appears to be responsible for adding, editing and storing Universal Link data in a database. Also, it seems to be accountable to find potential candidates for opening a Universal Link. To get data from this database, we further hooked the function `enumerateEntries:matchingService:domain:appID:block:` of the class `SWCDatabase` in the `/usr/libexec/swcd` binary with Frida and logged the `enumerateEntries` input parameter. Indeed, we were able to read JSON entries from the database, as Fig. 4.15 shows. While we could not confirm the location of the database file, `/private/var/mobile/Containers/Data/InternalDaemon/4C58F208-8116-4B4E-AD07-16D816CB18D3/Library/Caches/swc.db` likely contains related data. The `file` command showed that the latter is no ordinary database but a binary plist file format. By renaming the file to `swcd.plist`, XCode could open it and displays its content. The file contains all bundle IDs we also found in the entries-JSON. However, the other fields are either labelled differently or are not easily readable.

We found that when an app switch is triggered by a Universal Link the swcd-daemon calls the function `-[_SWCServiceSpecifier initWithCoder]`. To understand what code is executed from there, we attached a Frida Stalker [1] to all subsequent function calls. We logged each function the Stalker found upon entering. Frida provides the function pointers of the called function, which we could use to retrieve its debugging symbols. The resulting output contained multiple frameworks and libraries, and over

```
[enumerateEntries:{(
   { s = applinks, a = com.apple.Passbook, d = wallet.apple.com, ua = unspecified, sa = approved,
       ↪ patternCount = 7 },
   { s = applinks, a = 0000000000.com.apple.icq, d = icq.icloud.com, ua = unspecified,  sa =
       ↪ approved, patternCount = 1 },
   { s = webcredentials, a = com.apple.AuthKitUIService, d = apple.com, ua = unspecified, sa =
       ↪ approved },
   { s = applinks, a = com.apple.Health, d = *.health.apple.com, ua = unspecified, sa = approved },
   { s = applinks, a = com.apple.Maps, d = guides.apple.com, ua = unspecified, sa = approved,
       ↪ patternCount = 4 },
   { s = applinks, a = com.apple.Maps, d = collections.apple.com, ua = unspecified, sa = approved,
       ↪ patternCount = 1 },
   { s = applinks, a = com.apple.Maps, d = maps.apple.com, ua = unspecified, sa = approved,
       ↪ patternCount = 4 },
   { s = activitycontinuation, a = ZL6BUSYGB3.com.apple.news, d = apple.news, ua = unspecified, sa
       ↪ = approved },
   { s = applinks, a = ZL6BUSYGB3.com.apple.news, d = apple.news, ua = unspecified, sa = approved,
       ↪ patternCount = 1 },
   { s = activitycontinuation, a = ZL6BUSYGB3.com.apple.news, d = news.apple.com, ua = unspecified,
       ↪  sa = approved },
   { s = applinks, a = ZL6BUSYGB3.com.apple.news, d = news.apple.com, ua = unspecified, sa =
       ↪ approved, patternCount = 1 },
   { s = applinks, a = com.apple.shortcuts, d = *.workflow.is, ua = unspecified, sa = approved,
       ↪ patternCount = 1 },
   { s = applinks, a = com.apple.shortcuts, d = *.icloud.com, ua = unspecified, sa = approved,
       ↪ patternCount = 1 },
   { s = applinks, a = <Team-ID>.<bundle-ID>.C, d = <our domain>, ua = unspecified, sa = denied },
   { s = applinks, a = <Team-ID>.<bundle-ID>.B, d = <our domain>, ua = unspecified, sa = approved,
       ↪ patternCount = 1 },
   { s = applinks, a = <Team-ID>.<bundle-ID>.A, d = <our domain>, ua = unspecified,  sa = approved,
       ↪  patternCount = 1 }
)}
```

Figure 4.15: JSON entries of Universal Links we intercepted using Frida.

1,500 corresponding function calls. However, some procedures of previous investigations we knew are called within this process were not present. Hence, we suspect that the traces of Frida Stalker were not exhaustive.

We found three prominent frameworks that are involved in the process of opening a Universal Link. Foremost is the SharedWebCredentials framework, which seems to implement the handling of the Universal Link resolution. Furthermore, the Foundation and CoreFoundation frameworks are mainly responsible for helper functions, memory management and inter-process communication (XPC). Due to the scope of this thesis, we did not further investigate these traces, as our goal is not to fully reverse engineer the Universal Link resolution. Hence, this aspect of the research is left to future work.

4.5.2 Scenario 1 – Unverified Universal Link

In this scenario we consider three cases:

- Case 1: two apps define the same Universal Link and no Custom URL Scheme
- Case 2: two apps define the same Universal Link, and the unverified app also defines a Custom URL Scheme
- Case 3: an app defines two different Universal Links but successfully verifies only one of them

```
{
    "applinks": {
        "details": [
        {
          "appID": "<team-ID>.<bundle-ID>.A",
          "paths": ["*"]
        }
      ]
    }
  }
```

Figure 4.16: The apple-app-site-association file used for Scenario 1.

To test for applicability of the threat model discussed in Chapter 3.4, we developed two PoC apps, app A and app B. Both register the same Universal Link in their entitlements file. We defined them directly in XCode by adding an Associated Domain capability to prevent misconfiguration. Further, we hosted an apple-app-site-association file displayed in Fig. 4.16 at our testing domain under `/.well-known`. However, only app A is associated with the domain in the apple-app-site-association file. Hence, we expect the verification of app B to fail. For Case 3, we added a new Universal Link to app A, which does not pass the verification.

Case 1 – no Custom URL Scheme

When we tested the described scenario on both our jailbroken iPhone 6S (iOS 14.0.1) and the XCode Simulator (iPhone 11, iOS 14.1), we found that only app A could open links to our testing domain. If app A was not present on the device, Safari opened the link instead. Installation order and presence of the other candidate did not affect the result.

Case 2 – the unverified app has a Custom URL Scheme

For this case, we gave app B the Custom URL Scheme `myscheme://`. We did not alter the apple-app-site-association file, and its Universal Link thus did not verify. With this modification, we repeated the experiment from case 1. App B can now open links starting with `myscheme://`, but the opener for the Universal Link is still app A. Hence, the consequence is the same as in case 1.

Case 3 – verified and unverified Universal Links

We added a new Universal Link to a domain we do not own to app A. Therefore, it had one Universal Link where the verification succeeds and one where it fails. We found that app A could still open the verified Deep Link but not the unverified one. Hence, failing the verification for one Universal Link does not invalidate all of which an app registers.

4.5.3 Scenario 2 – Verified Universal Links

We further investigated the behaviour of iOS when multiple apps can correctly verify a Universal Link for the same domain. Again, we differentiate between two cases:

```
{
    "applinks": {
        "details": [
        {
            "appIDs": [
                "<team-ID>.<bundle-ID>.A",
                "<team-ID>.<bundle-ID>.B"
            ],
            "paths": ["*"]
        }
        ]
    }
}
```

(a) one common `appIDs` array

```
{
    "applinks": {
        "details": [
        {
            "appID": "<team-ID>.<bundle-ID>.A",
            "paths": ["*"]
        },
        {
            "appID": "<team-ID>.<bundle-ID>.B",
            "paths": ["*"]
        }
        ]
    }
}
```

(b) two separate `appID` entries

Figure 4.17: Two versions of apple-app-site-association files as used for Scenario 2 case 1.

- Case 1: both apps are entitled to open any link to the same website.
- Case 2: the apps register distinct paths of the verified associated domain.

For both cases, we reused the apps from Scenario 1. For case 1, we extended the apple-app-site-association file hosted on our test website such that iOS can correctly verify the Universal Link of app B. Then, for case 2, we changed the entries of the apple-app-site-association file to distinct paths. We tested both cases on the Simulator and the physical devices with the same outcome. Further, we evaluated both scenarios with the two different possibilities to declare apple-app-site-association files that are shown in Fig. 4.17. Both produced identical results.

Case 1 – Path Wildcard *

As Fig. 4.17 shows, to allow verification of app B's Universal Link to succeed, we added a path entry containing the wildcard `*` to the hosted apple-app-site-association file. Thus, both apps are allowed to open any link of the associated domain. We did not alter the applications themselves. First, we installed the apps in the order of app A before app B. There, iOS selected app A as the opener as long as it was installed on the system. When we uninstalled app A, app B opened the link instead. When we installed app B before app A, the result was identical.

We changed the order of the apps in the apple-app-site-association file and repeated the experiment to gain more information on the reasoning behind the opener selection. Then, the selected opener was app B. That suggests that the app declared sooner in the apple-app-site-association file is the preferred opener candidate.

During testing, we discovered another interesting behaviour. Apple allows developers to append the query parameter `?mode=developer` to an `applink` declaration in the app's entitlements [6] to allow the device to circumvent the CDN and fetch the apple-app-site-association file directly from the webserver. That is useful for testing purposes, as the caches of the CDN are only updated periodically in an undefined timeframe. When we used this additional option on app B, the app we installed first on the system could

```
{
    "applinks": {
        "details": [
        {
            "appIDs": [ "<team-ID>.<bundle-ID>.A" ],
            "paths": ["/a/*"]
        },
        {
            "appIDs": [ "<team-ID>.<bundle-ID>.B" ],
            "paths": ["/b/*"]
        }
        ]
    }
}
```

Figure 4.18: The apple-app-site-association file used for Scenario 2 case 2.

open the link. Then, the opener was re-evaluated on every re-install or update through XCode, i.e., if we re-installed the active opener, the other candidate became the opener instead.

Case 2 – Different Paths `/a/*` and `/b/*`

We changed the paths to be distinct, as Fig. 4.18 shows, to examine the behaviour of iOS when two apps register different paths of the same domain. Regardless of the installation order and declaration order in the apple-app-site-association file, links starting with `<domain>/a` resulted in app A and those beginning with `<domain>/b` in app B. However, all other paths, including the root level domain, could only be opened by Safari.

4.5.4 Security and Privacy Implications

Compared to Android, the approach of iOS is stricter. While Android allows users to choose unverified apps as the opener, iOS does not involve the user in the selection process. Instead, the system determines one candidate responsible for opening the verified associated domain and specified paths thereof, comparable to a preferred app on Android. Further, iOS offers no disambiguation dialogue when multiple apps register the same path of a domain. We found that in this case, the system selects the first corresponding entry of the apple-app-site-association file. Thus, the attack vector for Android regarding the hijacking of App Links through unvalidated apps does not apply to iOS. The stricter approach of iOS offers enhanced security but less transparency and control compared to Android.

4.6 Disclosure

We reported these potential security concerns to Apple. A representative of the Apple Product Security replied the following: “(...) I have received your report, and we are investigating. (...).” However, we had not received further information or response at the time of writing.

CHAPTER 5

Analyzing Deep Links in the Wild

To emphasize the relevance and practical impact of our research, we examined the use of mobile Deep Links in the wild by evaluating a representative sample of iOS apps provided by Tang et al. [52]. In the following paragraphs, we show the findings of our evaluation. First, we discuss the dataset our results are based on. Next, we present our assessment on the distribution of Custom URL Schemes, including misconfigurations and invalid schemes. Then, we analyze occurring collisions and demonstrate our classification thereof, based on methodology from Liu et al. [45]. In the process, we also discuss collisions caused by related apps. Last, we show our findings on the usage of Universal Links.

5.1 Dataset

Our sample was kindly provided by Tang et al. [52], by randomly selecting 11,622 apps from their dataset of over 1,200,000 apps that were downloaded from the official Apple App Store in 2018. The entire dataset is available on their collaborative platform.[1] For replicability, we published the hashes of all apps in our sample on GitHub.[2] The dataset is structured as follows: For each app, the sample contains an archive and the extracted data, which includes the `Info.plist` and the app's Mach-O binary file. To analyze Custom URL Schemes, we parsed the related sections of each app's `Info.plist` file.

Measuring the usage of Universal Links requires more effort, as associated domains are not included in the app's `Info.plist` file. Instead, they are declared in the `<app-name>.entitlements` file, which is an XML file that describes an app's capabilities and becomes compiled into the binary [10]. Hence, we had to extract the entitlements of each app in our sample. So, we applied the methodology of the Mobile Security Testing Guide by OWASP [48]. Therefore, we created a list of all Mach-O

[1] Janus collaborative platform: https://www.appscan.io/
[2] GitLab Repository: https://gitlab.secpriv.tuwien.ac.at/secpriv/systemsec/ios-deep-links

binaries and then we searched for corresponding XML files in each binary using radare2 [3]. However, it cannot be guaranteed that the extraction tools reliably found every included entitlements file. Thus, our data on Universal Links in our sample might not be exhaustive.

5.2 Custom URL Schemes

We found that out of the 11,622 apps from the dataset, 4,840 (~41.6%) register at least one valid Custom URL Scheme, defined in their respective `Info.plist` file. In total, we were able to parse 14,979 schemes from the data, including duplicates. Along with valid Deep Links, we found several misconfigurations, e.g., empty schemes or incomplete declarations. Thus, we removed empty values and schemes declared by the same app multiple times, leaving 14,506 valid schemes. Fig. 5.1 displays the cumulative distribution function (CDF) of apps in the context of the number of valid Custom URL Schemes they declare. The CDF displays the distribution of the number of declared schemes per app in percent. Considering only samples with at least one scheme, more than half register one or two schemes. The 99th-percentile lies at 10 schemes. Thus, 99% of our samples have at most 10 schemes. Considering our entire dataset, the 99th-percentile lies at 9, hence 99% of all samples declare at most 9 schemes. There are two interesting outliers. First, the app with the bundle ID `com.quanmincai.caipiao` declares 53 different weixin[3] schemes (`wx<app-id>://`). The other app has the bundle identifier `com.soul.sword2` and registers 44 schemes, some of which are from other services, such as `taobao://`, `googlechrome://` or `weixin://`. The average number of registered schemes is 1.24, including apps declaring no scheme, and 2.98 considering only apps that declare at least one scheme.

5.2.1 Misconfigurations

Out of 14,979 parsed Custom URL Schemes, we found 358 misconfigurations in the related sections of the corresponding `Info.plist` files. These do not cause collisions by themselves but could lead to unexpected behaviour, especially when XML tags are incorrectly placed or missing. Fig. 5.2 illustrates the total number of incorrectly configured schemes, the number of individual apps containing them, as well as the number of duplicates caused by apps incorporating the same misconfiguration multiple times. The most common misconfiguration is an empty `CFBundleURLScheme` key, as shown in Fig. 5.3a, used by 133 sample apps, a total number of 151 times. The next most frequently occurring misconfiguration is an invalid scheme within the `CFBundleURLSchemes` array (Fig. 5.3b), with 106 occurrences within 95 different apps. Out of these, 104 times the string-tag is empty, once it holds a single whitespace character (`\s`), and lastly once a newline character (`\n`) followed by 10 whitespace characters is used as a scheme-string. Furthermore, 40 apps are missing the `CFBundleURLScheme` key (Fig. 5.3c), and 34 apps declare an empty `CFBundleURLTypes` array (Fig. 5.3d). Additionally, we found

[3]Weixin | WeChat, https://weixin.qq.com/

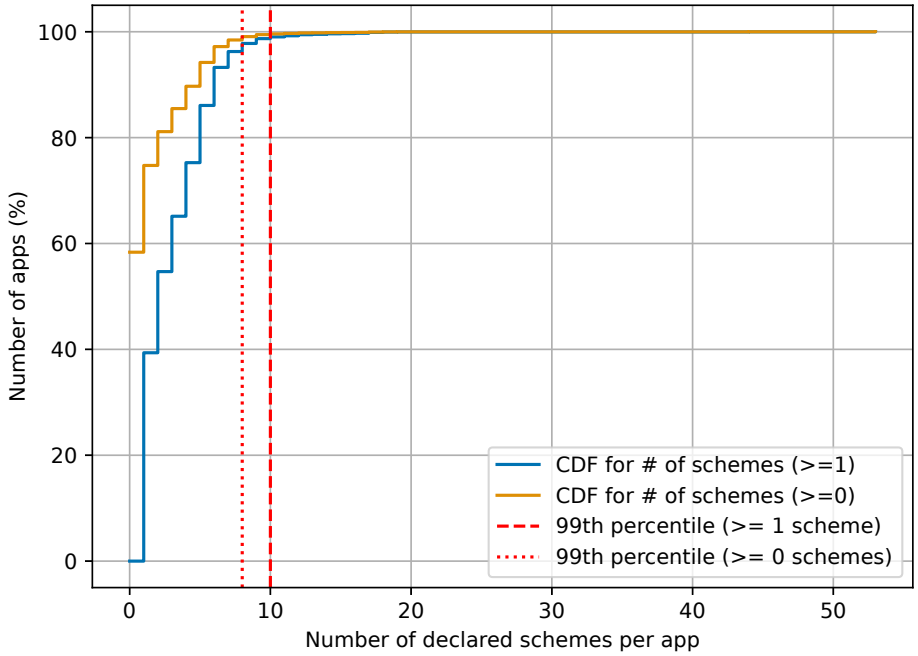


Figure 5.1: CDF of the number of declared Custom URL Schemes per app for apps that declare at least one scheme (blue), and for all apps from our dataset (orange).

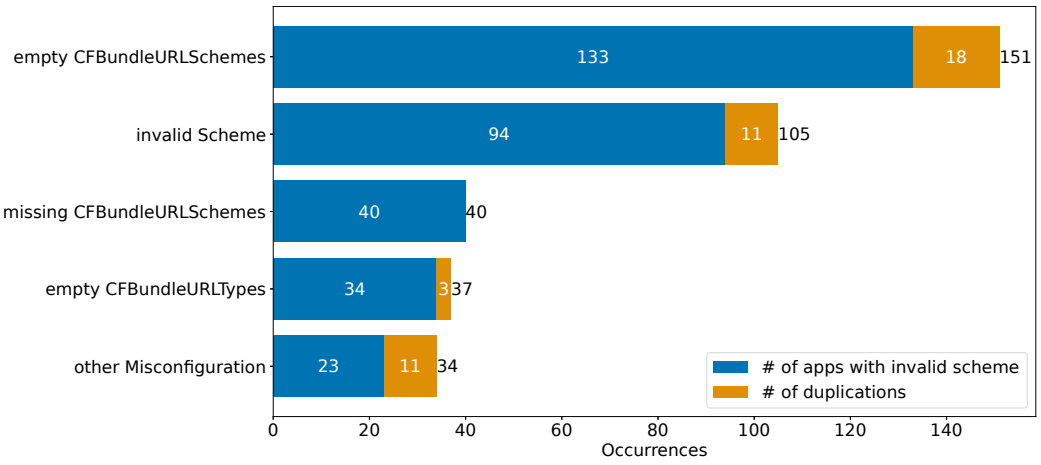


Figure 5.2: Found types of invalid Custom URL Schemes and misconfigurations along with occurrences. The bars show the number of distinct apps per misconfiguration, as well as the number of occurrences caused by apps encorporating them multiple times.

```
<key>CFBundleURLTypes</key>
<array>
    <dict>
    <key>CFBundleURLSchemes</key>
        <array/>
    </dict>
    ...
</array>
```

(a) Empty `CFBundleURLScheme`

```
<key>CFBundleURLTypes</key>
<array>
    <dict>
    ...
    <key>CFBundleURLSchemes</key>
    <array>
        <string> </string>
    </array>
    </dict>
    ...
</array>
```

(b) Invalid Scheme

```
<key>CFBundleURLTypes</key>
<array>
    <dict>
    <key>CFBundleURLName</key>
    <string></string>
    </dict>
    ...
</array>
```

(c) Missing `CFBundleURLScheme`

```
<key>CFBundleURLTypes</key>
<array>
    <dict/>
</array>
```

(d) Empty `CFBundleURLTypes`

```
<key>CFBundleURLTypes</key>
<array>
    <dict>
     <key>NSAllowsArbitraryLoads</key>
    <true/>
    </dict>
    ...
</array>
```

(e) Incorrectly placed `key`-element `NSAllowsArbitraryLoads`

```
<key>CFBundleURLTypes</key>
<array>
    <dict>
    ...
    <key>CFBundleURLSchemes</key>
    <array>
        ...
    </array>
    <key>Item 0</key>
</dict>
...
</array>
```

(f) Invalid `key`-element `Item-0`

Figure 5.3: Excerpts of found invalid Custom URL Schemes and misconfigurations.

34 further misconfigurations, labelled as “other Misconfigurations” in Fig. 5.2, such as incorrectly placed key-values (Fig. 5.3e), invalid key-tags (Fig. 5.3f) or simply typos.

5.2.2 Scheme Collisions

Furthermore, we discovered a significant number of 745 unique scheme identifiers used by more than one app, thus forming a collision. For categorizing collisions, we use the methodology for Android provided by Liu et al. [45], which we discuss in the following paragraph, as it applies to iOS as well. However, the definition of a collision must be adapted, as iOS does not allow a similarly fine-grained definition of URL schemes as Android does. On Android, it is possible to specify paths within URIs, in such a way

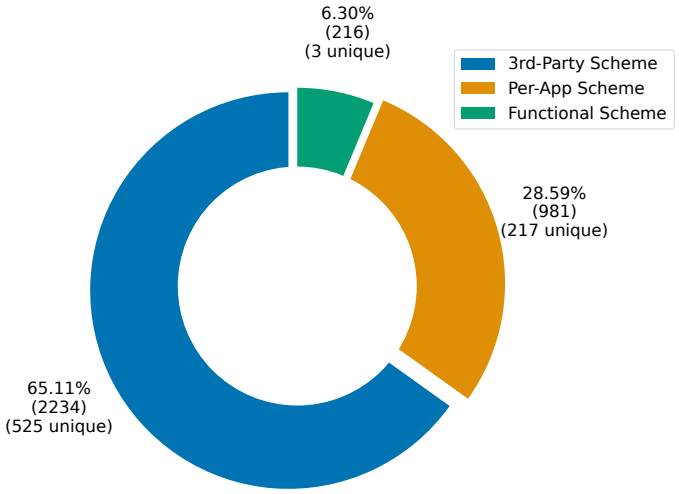


Figure 5.4: Visualization of classification results of observed scheme collisions based on the methodology of Liu et al. [45].

that only those which are explicitly declared are registered. This is currently not possible on iOS, as it only supports the declaration of schemes in the `Info.plist` file. The remainder of the URI, e.g., the path or parameters, has to be handled in the app's code. Hence, we call it scheme collision when a scheme is declared by more than one app, not considering invalid schemes or duplicate declarations of an app. In total, out of the 14,979 Custom Scheme URLs from our sample, 3,471 (~23.2%) are colliding.

Liu et al. [45] define three categories of collisions:

- "functional scheme", intended to be used by multiple apps to provide functionality such as `sms://`.
- "per-app scheme", which should be unique and employed by only one app.
- "third-party scheme", which are primarily used for authentication callbacks by third-party SDKs, such as `wx<app-id>://`, `QQ<app-id>://`, `tencent<app-id>://`, `wb<app-id>://` or `fb<app-id>://`.

We found that almost two thirds (65.11%, 2,234) of all collisions of valid schemes in our sample are caused by "third-party scheme". These are used by third-party SDKs, e.g., FacebookSDK [46], to provide functionality, such as sharing or logins. Generally speaking, they offer a simple way of enabling single sign-on (SSO) on mobile devices. First, the app calls the authenticator service to let the user perform the login. Then, to switch back to the app again, it is invoked by the authenticator via its Custom URL Scheme, which is registered at the SDK-provider of the form `<sdk-provider-prefix><app-id>://`, e.g., `fb107704292745179://` [46]. We parsed schemes of the category "functional scheme" manually. We found three different members: `prefs://`, `Prefs://` and

`App-Prefs://`, which form roughly 6% (216) of all collisions. By registering these private system schemes, apps could open a subpage of the system settings. However, on iOS, there are not many schemes falling into this first category. Moreover, Apple generally does not permit usage of the system settings scheme, as it is not intended for public use, thus being undocumented and may change anytime. An exception to this rule are keyboard-extensions, which are permitted to use the scheme to open the corresponding settings [15]. Other usages may lead to removal from the iOS App Store [15]. The remaining 981 (29.59%) colliding Custom URL Schemes we categorized as "per-app schemes".

To further evaluate the occurring scheme collisions, we determined how many are caused by related apps. We defined apps as related if the first two levels of the bundle identifiers (IDs) [7] match, as the rest is reserved for the app name because the bundle ID of an app must be unique and is suggested to represent a reversed URL of the form `<domain-name>.<company-name>.<app-name>`. We used the top and second-level domain of each app's bundle ID to group apps that are likely related per collision, e.g., `com.mycompany.app1` and `com.mycompany.app2` are considered as associated. Note that we excluded definitions of the same scheme multiple times by the same app from the dataset. We found that out of 745 collisions, 415 (47.4%) are caused by related apps defining the same scheme. These still classify as collisions, as they have the same effect: exactly one app can open the link. The candidate is selected by iOS, depending on the alphabetically lower bundle ID. As the first- and second-level domain of the bundle ID of related apps is identical, only the lowest-level - which in most cases corresponds to the app's name - is considered.

Fig. 5.5 illustrates colliding Custom URL Schemes with equal to or more than 20 collisions. We grouped occurrences based on the corresponding bundle IDs of the apps defining them. The right section of the bar indicates the number of collisions caused by related apps. The left section of the bar displays the number of collisions caused by unrelated apps. In total, we found 21 schemes with at least 20 collisions. The most commonly used Custom URL Scheme is `prefs`, which is claimed 198 times by 173 apps with distinct bundle IDs. The second most employed scheme `wx2b9b4d2442102a89` is used 65 times, out of which 58 occurrences are ascribable to related apps.

5.2.3 Discussion

Our findings show that Custom URL Schemes are widely established. We found that only around 2.4% of all schemes defined in our sample had misconfigurations, which in most cases only affected the misconfigured Custom URL Scheme itself. This suggests that Custom URL Schemes are not hard to configure. However, we observed a significant amount of scheme collisions. Our classification further shows that almost two-thirds of which are caused by third-party schemes. Thus, mainly those used for authentication purposes are colliding. That can be especially critical if the authentication provider sends unencrypted information via the Custom URL Scheme to an app, as there might be another candidate which the Deep Link is resolved to. However, the by far most

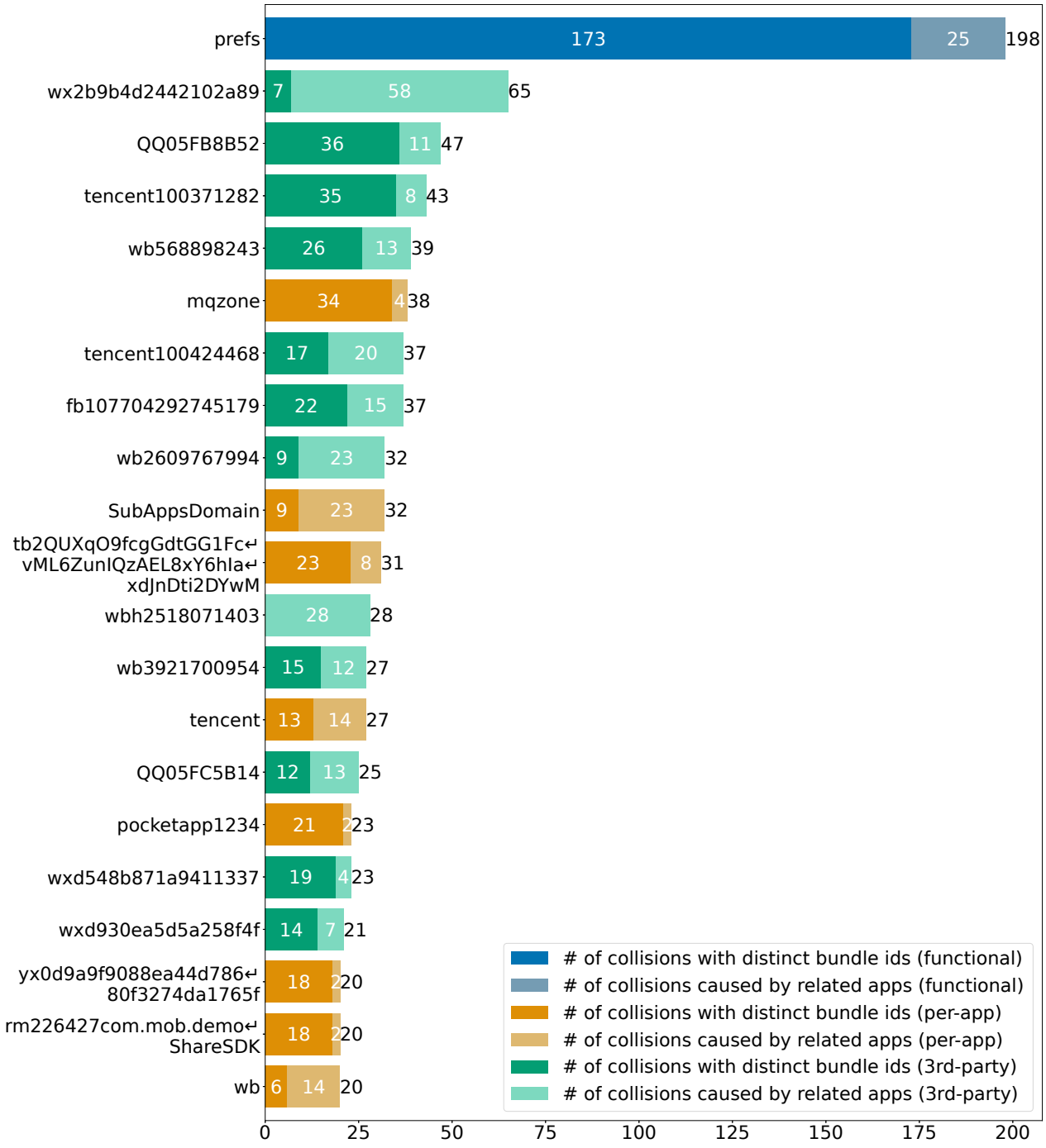


Figure 5.5: Observed collisions with over 20 occurrences, separated into the number of distinct bundle identifiers and the number of related bundle identifiers (related apps) causing them.

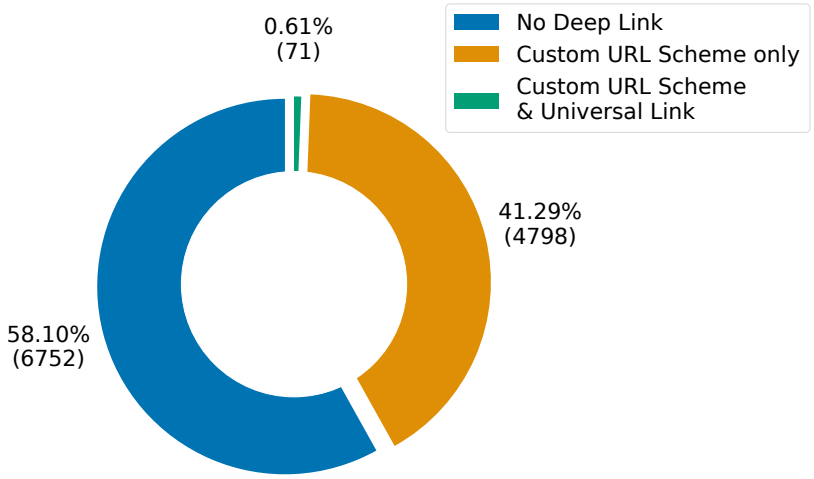


Figure 5.6: Deep Linking mechanisms we observed in our sample.

used scheme is `prefs://`, which accounts for around 26.6% of all collisions. As Apple now prohibits the usage of this scheme for all non-keyboard-extensions, newer apps may not register it anymore. Further, we found that almost half of the collisions are from related (same vendor) apps. That can be due to developers recycling the `Info.plist` from their other applications or publishing a new version while still keeping the old app accessible. However, these collisions are treated the same way as other collisions, i.e., if users were to install multiple apps of the same vendor, the Custom URL Schemes would not work as intended. Then, only a single app – the one with the alphabetically lowest app name – would resolve all shared schemes.

Comparison to Android

Liu et al. [45] analyzed over 164.000 apps listed on the official Google Play Store in 2014 and 2016, respectively. While our sample is much smaller, it is still possible to discover trends, such as how likely collisions appear. Hence, in the following paragraph, we compare our findings on iOS regarding Custom URL Schemes to those of Liu et al. [45] on Android. Out of their App2016 dataset, around 12.3% of apps define scheme URLs. Compared to the App2014 dataset, this accounts for an increase of almost 100%. While our sample is considerably smaller, the increased popularity of Deep Links is observable. Regarding collisions, Liu et al. [45] measured 697 colliding schemes, 397 when excluding apps of the same developer. Out of these, they found 30 unique functional schemes, 197 third-party schemes and 149 per-app schemes. Since iOS generally does not offer public Custom URL Schemes that can be overwritten to replace system apps, we only discovered 3 distinct ones. However, we found a significantly higher number of unique third-party schemes (525) and slightly more distinct per-app schemes (217) in our dataset from 2018, which is only around 7% of the size of App2016.

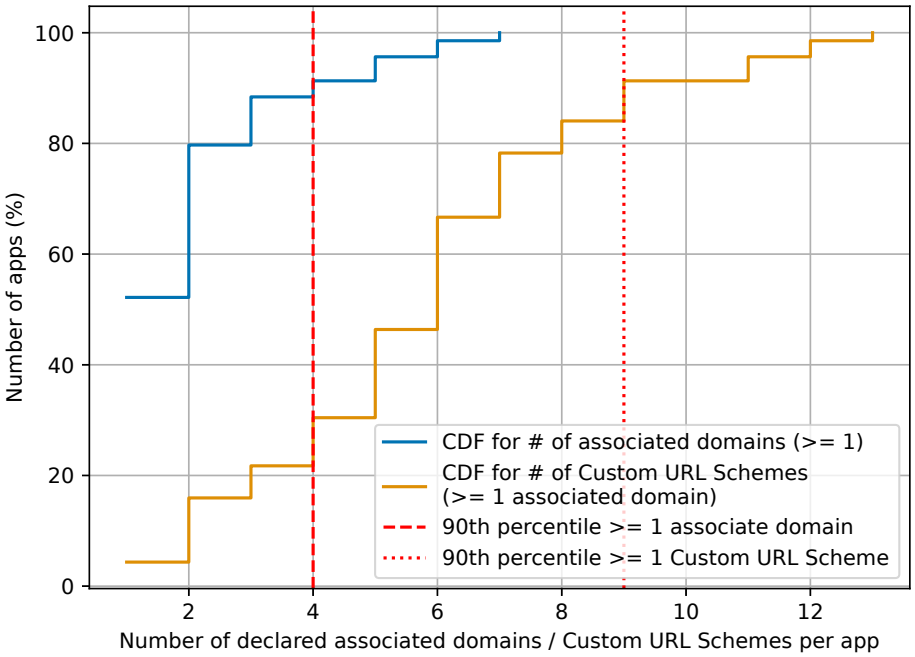


Figure 5.7: CDF of the number of declared associated domains per app for apps that declare at least one compared to their amount of declared Custom URL Schemes.

5.3 Universal Links

Out of 11,622 apps, we found that 2,563 (~22.1%) include an entitlements file, but only 71 (~0.6%) also declare a Universal Link. The low number of Universal Links may be due to corrupt data in our sample. Hence, our analysis thereof may not be exhaustive. Fig. 5.6 displays the distribution of discovered Deep Linking mechanisms. All apps that declared one or more Universal Links also declared at least one Custom URL Scheme. In total, we were able to parse 132 valid associated domains. Fig. 5.7 illustrates the CDF of apps in the context of the number of associated domains they declare, excluding those with less than one, compared to the same apps' number of Custom URL Scheme. It shows that more than 50% of the apps containing at least one associated domain entry only reference a single one, hence on average, these apps declare a single domain, while the average amount of Custom URL Schemes is around 5.8. The 90th-percentile lies at 4. Thus, less than 10% declare more than 5 associated domains. However, the 90th-percentile of the corresponding Custom URL Schemes lies at 9. Thus, apps from our sample mostly define more Custom URL Schemes than associated domains.

Associated Domain	Occurrences
s.mlinks.cc	8
lkme.cc	5
www.lkme.cc	5
palfish.ipalfish.com	2
s2.mlinks.cc	2
wan68.com	2
www.wan68.com	2
www.17dmj.com	2

Table 5.1: Associated domains used by multiple apps in our sample.

5.3.1 Misconfigurations

Out of 134 parsed associated domains, only two can be considered a misconfiguration. In both cases, the associated-domain element in the entitlements file was empty. As the data originated in 2018, we did not analyze other misconfigurations such as incorrectly hosted or invalid apple-app-site-association files. It may lead to wrong assumptions when comparing associated domain entries of these apps with apple-app-site-association files from today, as publishers may update values, domain names, etc.

5.3.2 Collisions

As associated domain entries in entitlements files only contain the domain, but not the specific paths apps may open, the apple-app-site-association file is required to evaluate collisions. Hence, due to the same reasons discussed in Misconfigurations, we did not perform further analysis. However, we discovered that out of 132 associated domains, only 112 were distinct. We found 8 domains that are used by more than one app, shown in Table 5.1. Out of these, 3 are used by related apps (palfish.ipalfish.com, wan68.com, www.wan68.com), while the remaining 5 are providers for managing Deep Links such as mlinks.cc.

5.3.3 Discussion

While we did not discover many Universal Links in our sample, all of the apps that declared them also claimed at least one Custom URL Scheme. In total, 71 apps were associated with 112 distinct domains. The collisions we found were either providers for

resolving Universal Links or the same publisher. However, collisions among Universal Links do not have the same effects as for Custom URL Schemes. It is intended that apps from the same publisher also use the same associated domain, which hosts the apple-app-site-association file that specifies paths each app is allowed to open. Since not every publisher may have its own Web server, some providers offer to manage the hosting of the website and the apple-app-site-association file for multiple apps. Lastly, as our sample is from 2018, we did not analyze misconfigurations in apple-app-site-association files as some domains and apps did not exist anymore. Hence, research with more recent data may be part of future work.

CHAPTER 6

Future Work

For this thesis, we limited our analysis to Android 11 and iOS 14.1. However, recently Android 12 [27] was released, which significantly changes the functionality of App Links. If the OS cannot verify an App Link, the connected app cannot open it anymore. Thus, the behaviour of App Links is now similar to Universal Links. Hence, these procedures are now better comparable, and previous threat models on Android regarding App Links may no longer apply. In the future, we could analyze the newly introduced mechanisms on Android to evaluate which attack vectors are no longer relevant. Besides, a thorough analysis of the new handling of App Links on Android could provide new insights for Universal Links on iOS regarding security and privacy aspects.

For our analysis of Universal Links, we investigated functions of the Shared Web Credentials framework. We were able to reverse-engineer a partial function-call-graph for the opening of a Universal Link. Future work could dig deeper into that subject and reconstruct a full call-graph, not limited to the Shared Web Credentials Daemon (swcd). Furthermore, it could perform a more in-depth dynamic analysis of the procedure for opening, installing and removing applications with Deep Links. The dynamic analysis could also help identify storage locations of known schemes for deep linking and how the system selects the opener on later iOS versions.

Our investigation of threat models on iOS is not exhaustive, as we selected a subset of known impactful attacks on Android. Consequently, we could analyze even more threat models in the future. In addition, future work could develop tools to automatically test iOS apps for vulnerabilities such as CSRF and Permission Re-delegation. Additionally, we limited our data analysis to not include iOS's App Clips, which are the counterpart to Instant Apps on Android. Hence, it could be the subject of further investigation of how App Clips resolve Universal Links and if known attack vectors from Android apply, e.g., as discovered by Tang et al. [51].

In our data analysis, we studied the distribution of Custom URL Schemes and Universal Links in the wild. However, the size of our sample is relatively small. Also, it predominantly consists of apps from the Chinese localization of the iOS AppStore. Hence, future work could analyze a larger, more diverse dataset. Furthermore, as our data originated in 2018, verifying apple-app-site-association files is unreliable, as apps and websites may have changed by now. Future work could analyze a more current dataset and verify their associated sites to gain information on the current distribution of Universal Links and their validity.

Almost 60% of all colliding Custom URL Schemes we found in our sample were third-party schemes. Also, during the manual inspection of plist files, we encountered readable app secrets for such SDKs. Therefore, we could evaluate the security of popular third-party login SDKs regarding Scheme Collision and whether that enables additional attack vectors. As we found misconfigurations in both, plist and entitlements files, future work could also aim to develop a linter to prevent common mistakes.

CHAPTER 7

Conclusion

Mobile platforms are an important market due to their immense popularity in the past couple of years. Hence, publishers want to make their content available to mobile devices. Therefore, Android and iOS introduced deep linking mechanisms to interconnect existing websites and mobile applications. Both operating systems share the same underlying ideas considering Deep Links, whereas their implementations strongly differ.

The old method to link to mobile content is by creating a custom scheme for an app. On iOS, this is called Custom URL Scheme, while Android calls it Scheme URL. Android does not only allow apps to register their scheme but also provides system schemes, e.g., `sms://`. An app may override them to present itself as an alternative to other or pre-installed system apps. However, while this feature enables high customizability for the user, overriding any scheme also gives opportunities for malicious actions. On Android, if multiple apps register the same custom scheme, the user is prompted to decide which candidate may open corresponding links. Thereby, users can also set a preference, such that these links always open in the chosen app. While letting a potentially uninformed user determine the opener can pose a significant threat to their privacy and the system's security, it is leastways transparent that there are other candidates for the scheme. On iOS, on the other hand, when there are multiple candidates for a Custom URL Scheme, the system selects the opener autonomously based on the apps' bundle identifier, as we found. While this approach does not transfer the responsibility to the users, it is unclear to them that there are multiple options for opening the link. Furthermore, the system-chosen app on iOS acts like a user-set preference on Android. As long as it is unchanged, and no new app with an alphabetically lower bundle identifier is installed, it always gets to open the link.

However, the implementations of both systems that enable apps to register any custom scheme can be abused. On Android, an app's appearance on the disambiguation dialogue is customizable. Hence, a malicious app can deceive users into choosing it instead of a benign one. On iOS, a malicious app has to fool the system, which – as of iOS 14 – is

achievable by having the alphabetically lowest bundle identifier among all candidates – a trait that may not be easy to maintain and may change upon every installation of a new app. On Android, several attack vectors abusing Scheme URLs and the underlying intent system exist. In this thesis, we considered Scheme Collision, Permission Re-delegation and CSRF. We analyzed their applicability to iOS. Our findings show that these attack models can pose a threat to iOS. The main difference between Android's and iOS' approaches when handling Scheme Collisions is that Android leaves the selection of the opener to the user. While this decision certainly has its flaws, it reveals to the user that multiple apps claim a scheme. Permission Re-delegation may be a legitimate use-case, yet it could enable malicious actions on both systems. Lastly, custom schemes combined with Web Views or exposed APIs pose a potential attack vector for CSRF on Android and iOS without proper handling. However, both mobile systems delegate the responsibility for appropriate input validation to the developers.

Due to the security flaws of schemes, both mobile operating systems introduced a new mechanism that enforces the verification of the ownership of claimed links: App Links (Android) and Universal Links (iOS). Again, both follow the same fundamental idea that there should be some verification that an app can only claim association with websites the publisher affiliates to. However, the implementation thereof significantly differs among both operating systems. On Android, passing the verification to open declared App Links is not enforced. Yet, if merely a single App Link cannot be verified, the app and all further of its App Links are considered unverified. If the user attempts to open an App Link of an unverified app, the system spawns a disambiguation dialogue. There, it offers the user all candidates capable of handling the link, namely either associated apps or Web browsers. However, the disambiguation dialogue also contains unverified candidates. On iOS, Universal Links can only open in an app if it passes verification, i.e., the app can open verified links only and is not considered a candidate for unverified ones. However, when multiple apps successfully register the same associated domain, and the apple-app-site-association file allows both to open any path, one candidate exclusively opens the Universal Link, as we encountered. The opener is determined based on the order the association file lists the apps, where the first candidate is selected. As of iOS 14, despite having multiple verified apps for a domain, the chosen opener is permanently the same as long as the association file is not updated. Furthermore, the user has no say in the selection thereof. Thus, similar to Custom URL Schemes, how many apps successfully verify the same Universal Link is not transparent for users.

Concluding, custom schemes are flawed on both systems. However, the implementation on iOS is stricter, as the user cannot circumvent its decision. The same applies to Universal Links compared to App Links on Android 11. While Android allows the user to open links manually in unverified apps, iOS does not involve the user. Therefore, developers are required to configure Universal Links correctly, enhancing security and privacy. As of version 12, Android changed its implementation so that users can no longer open unverified App Links, following Apple's approach. As a result, Android's App Links may be equally secure compared to Universal Links. However, this requires further analysis.

List of Figures

List of Tables

Bibliography

[1] Frida Stalker. (Accessed: 2022-02-21). [Online]. Available: https://frida.re/docs/stalker/

[2] Frida • a world-class dynamic instrumentation framework. (Accessed: 2021-06-07). [Online]. Available: https://frida.re/

[3] radare. (Accessed: 2022-02-27). [Online]. Available: https://rada.re/n/radare2.html

[4] Apple. Allowing apps and websites to link to your content | apple developer documentation. (Accessed: 2021-03-23). [Online]. Available: https://developer.apple.com/documentation/xcode/allowing_apps_and_websites_to_link_to_your_content

[5] ——. Apple security research device program. (Accessed: 2021-05-21). [Online]. Available: https://developer.apple.com/programs/security-research-device/

[6] ——. Associated domains entitlement | apple developer documentation. (Accessed: 2022-03-07). [Online]. Available: https://developer.apple.com/documentation/bundleresources/entitlements/com_apple_developer_associated-domains

[7] ——. Bundle IDs | apple developer documentation. (Accessed: 2022-01-21). [Online]. Available: https://developer.apple.com/documentation/appstoreconnectapi/bundle_ids

[8] ——. Create app links for instant apps. (Accessed: 2021-11-07). [Online]. Available: https://developer.android.com/training/app-links/instant-app-links

[9] ——. Defining a custom URL scheme for your app | apple developer documentation. (Accessed: 2021-03-23). [Online]. Available: https://developer.apple.com/documentation/xcode/allowing_apps_and_websites_to_link_to_your_content/defining_a_custom_url_scheme_for_your_app

[10] ——. Entitlements | apple developer documentation. (Accessed: 2022-02-27). [Online]. Available: https://developer.apple.com/documentation/bundleresources/entitlements

[11] ——. open(_:options:completionHandler:) | apple developer documentation. (Accessed: 2021-11-29). [Online]. Available: https://developer.apple.com/documentation/uikit/uiapplication/1648685-open

[12] ——. SFSafariViewController | apple developer documentation. (Accessed: 2021-11-29). [Online]. Available: https://developer.apple.com/documentation/safariservices/sfsafariviewcontroller

[13] ——. Supporting associated domains | apple developer documentation. (Accessed: 2022-03-14). [Online]. Available: https://developer.apple.com/documentation/xcode/supporting-associated-domains#Add-the-Associated-Domain-File-to-Your-Website

[14] ——. Supporting universal links in your app | apple developer documentation. (Accessed: 2021-03-23). [Online]. Available: https://developer.apple.com/documentation/xcode/allowing_apps_and_websites_to_link_to_your_content/supporting_universal_links_in_your_app

[15] ——. Technical Q&A QA1924: Opening keyboard settings from a keyboard extension. (Accessed: 2022-01-21). [Online]. Available: https://developer.apple.com/library/archive/qa/qa1924/_index.html

[16] ——. Universal links - apple developer. (Accessed: 2021-03-23). [Online]. Available: https://developer.apple.com/ios/universal-links/

[17] ——. WKWebView | apple developer documentation. (Accessed: 2021-11-29). [Online]. Available: https://developer.apple.com/documentation/webkit/wkwebview

[18] A. Boxiner, E. Vaknin, A. Volodin, D. Barda, and R. Zaikin. Tik or tok? is TikTok secure enough? (Accessed: 2021-09-10). [Online]. Available: https://research.checkpoint.com/2020/tik-or-tok-is-tiktok-secure-enough/

[19] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*, ser. MobiSys '11. Association for Computing Machinery, 2011, pp. 239–252. [Online]. Available: https://doi.org/10.1145/1999995.2000018

[20] Cryptic Apps. Hopper. (Accessed: 2021-06-07). [Online]. Available: https://www.hopperapp.com/

[21] CVE Details. Apple iPhone OS: CVE security vulnerabilities, versions and detailed reports. (Accessed: 2021-04-23). [Online]. Available: https://www.cvedetails.com/product/15556/Apple-Iphone-Os.html?vendor_id=49

[22] B. F. Demissie, D. Ghio, M. Ceccato, and A. Avancini, "Identifying android inter app communication vulnerabilities using static and dynamic analysis," in

Proceedings of the International Conference on Mobile Software Engineering and Systems, ser. MOBILESoft '16. Association for Computing Machinery, 2016, pp. 255–266. [Online]. Available: https://doi.org/10.1145/2897073.2897082

[23] N. Dhanjani. SANS institute | insecure handling of URL schemes in apple's iOS. (Accessed: 2021-11-30). [Online]. Available: https://www.sans.org/blog/insecure-handling-of-url-schemes-in-apples-ios/

[24] A. P. Felt, "Permission re-delegation: Attacks and defenses," in *20th USENIX Security Symposium (USENIX Security 11)*. San Francisco, CA: USENIX Association, Aug. 2011. [Online]. Available: https://www.usenix.org/conference/usenixsecurity11/permission-re-delegation-attacks-and-defenses

[25] Google. Allowing other apps to start your activity. (Accessed: 2021-08-16). [Online]. Available: https://developer.android.com/training/basics/intents/filters

[26] ——. Android code search. (Accessed: 2021-08-16). [Online]. Available: https://cs.android.com/android

[27] ——. Behavior changes: Apps targeting android 12. (Accessed: 2022-03-13). [Online]. Available: https://developer.android.com/about/versions/12/behavior-changes-12

[28] ——. Broadcasts overview. (Accessed: 2021-08-24). [Online]. Available: https://developer.android.com/guide/components/broadcasts

[29] ——. Context. (Accessed: 2021-09-07). [Online]. Available: https://developer.android.com/reference/android/content/Context

[30] ——. Create deep links to app content. (Accessed: 2021-08-16). [Online]. Available: https://developer.android.com/training/app-links/deep-linking

[31] ——. Handling android app links. (Accessed: 2021-03-23). [Online]. Available: https://developer.android.com/training/app-links

[32] ——. <intent-filter>. (Accessed: 2021-09-09). [Online]. Available: https://developer.android.com/guide/topics/manifest/intent-filter-element

[33] ——. Intents and intent filters. (Accessed: 2021-04-21). [Online]. Available: https://developer.android.com/guide/components/intents-filters

[34] ——. <manifest>. (Accessed: 2021-09-07). [Online]. Available: https://developer.android.com/guide/topics/manifest/manifest-element

[35] ——. Overview of google play instant | android developers. (Accessed: 2021-11-07). [Online]. Available: https://developer.android.com/topic/google-play-instant/overview

[36] ——. Services overview. (Accessed: 2021-08-23). [Online]. Available: https://developer.android.com/guide/components/services

[37] ——. Verify android app links. (Accessed: 2021-04-21). [Online]. Available: https://developer.android.com/training/app-links/verify-site-associations

[38] HackerOne. Grab disclosed on HackerOne: [grab android/iOS] insecure deeplink... (Accessed: 2021-09-08). [Online]. Available: https://hackerone.com/reports/401793

[39] ——. Twitter disclosed on HackerOne: Periscope android app deeplink... (Accessed: 2021-09-08). [Online]. Available: https://hackerone.com/reports/583987

[40] ——. Twitter disclosed on HackerOne: Periscope iOS app CSRF in follow... (Accessed: 2021-09-08). [Online]. Available: https://hackerone.com/reports/805073

[41] N. Hardy, "The confused deputy: (or why capabilities might have been invented)," *ACM SIGOPS Operating Systems Review*, vol. 22, no. 4, pp. 36–38, 1988. [Online]. Available: https://doi.org/10.1145/54289.871709

[42] V. Haupert, D. Maier, and T. Müller, "Paying the price for disruption: How a FinTech allowed account takeover," in *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium*, 2017, pp. 1–10.

[43] A. Khan, S. Lee, and J. Wang, "Differences in inter-app communication between android and iOS systems," p. 7, 2018, (Accessed: 2021-03-23). [Online]. Available: https://aimunkhan.com/papers/Khan_Lee_Wang_IAC.pdf

[44] F. Liu, H. Cai, G. Wang, D. Yao, K. O. Elish, and B. G. Ryder, "MR-droid: A scalable and prioritized analysis of inter-app communication risks," in *2017 IEEE Security and Privacy Workshops (SPW)*, 2017, pp. 189–198.

[45] F. Liu, C. Wang, A. Pico, D. Yao, and G. Wang, "Measuring the insecurity of mobile deep links of android," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 953–969. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/liu

[46] Meta, Facebook. Get started - iOS SDK. (Accessed: 2022-01-21). [Online]. Available: https://developers.facebook.com/docs/ios/getting-started

[47] OWASP. Cross site request forgery (CSRF) | OWASP foundation. (Accessed: 2021-09-08). [Online]. Available: https://owasp.org/www-community/attacks/csrf

[48] ——. Mobile security testing guide | iOS platform APIs. (Accessed: 2022-02-17). [Online]. Available: https://mobile-security.gitbook.io/mobile-security-testing-guide/ios-testing-guide/0x06h-testing-platform-interaction#static-analysis

[49] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu, "Towards discovering and understanding task hijacking in android," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 945–959. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ren-chuangang

[50] StatCounter. Mobile operating system market share worldwide. (Accessed: 2021-04-23). [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide

[51] Y. Tang, Y. Sui, H. Wang, X. Luo, H. Zhou, and Z. Xu, "All your app links are belong to us: understanding the threats of instant apps based attacks," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. Association for Computing Machinery, 2020, pp. 914–926. [Online]. Available: https://doi.org/10.1145/3368089.3409702

[52] Z. Tang, K. Tang, M. Xue, Y. Tian, S. Chen, M. Ikram, T. Wang, and H. Zhu, "iOS, your OS, everybody's OS: Vetting and analyzing network services of iOS applications," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2415–2432. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/tang

[53] L. Xing, X. Bai, T. Li, X. Wang, K. Chen, X. Liao, S.-M. Hu, and X. Han, "Cracking app isolation on apple: Unauthorized cross-app resource access on MAC OS~x and iOS," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. Association for Computing Machinery, 2015, pp. 31–43. [Online]. Available: https://doi.org/10.1145/2810103.2813609