XUlink: XCode plugin for deep link verification and Android Security Analysis for deep links

Authors: Prasad Mate, Akshay Mendki, Niraj Kadam
Department of Computer Science, UGA, {pbm29988, am68638, nvk13361}@uga.edu
Instructor: Dr. Kyu Hyun Lee, UGA

***Abstract*—The World wide web (www) is a vast collection of text document pages which are accessible over the internet using URL. Deep links are nothing but a naive way in which World Wide Web directs its user to exact location of the any document on web. It makes use of the URL to point to the location of the document and load it in any browser in user's system. It has, in a way, evolved as a standard in organizing and displaying the web content using different protocols.**

The deep linking for mobile applications make use of URI instead, to point to exact location in a mobile application. It is still in a nascent phase and there exist a few vulnerabilities which cause easy hijacking of these deep links. The problems persist mainly due to improper implementation of deep links during mobile application development as well as because they are improperly handled by the mobile operating systems.

Our work in this paper focus on analyzing inappropriate implementations and putting a check on developer's side. We present an XCode extension "XUlink" which checks on various levels the integrity of the application before it is published for use in the app-store. Our work also demonstrates the over-permissibility problem that persist with Android OS and suggests practical measures to address the issues. It also demonstrates need for standardization in this evolving deep link technology, by successfully depicting attack scenario on deep links of famous applications of Facebook and Twitter on Android (Android Nougat and Oreo versions) devices.

***Index Terms*— deep link, universal links, applinks, app-intents, https, www, over-permissibility, XUlink, scheme URL, app link assistant**

I. Introduction

THE smartphones these days are providing a very handy tools for its user to access the enormous content over the web and perform necessary actions. May it be surfing the content of the social media or doing online shopping or banking or having tutor or courses, mobile applications find ways to enhance the users' experience by giving them powerful tools at their fingertips. For instance, user can click contact us hyperlink on the webpage which directs it to the telephone application or clicking reach us hyperlink directs the user to the Google Maps, pinpointing the destination location.
It was hard to have such a customized flow between the applications before the advent of the concept of deep linking. Deep linking provides efficient web-to-mobile interface using URIs to access the content and functions in applications [1]. The concept of deep linking exists from 2008 and is derived from the age old concept of web hyperlinking which redirects the user to a specific section in a specific webpage. For example, opening URL https://en.wikipedia.org/wiki/Mobile_deep_linking#Passing_search_data_via_deep_linking in user's browser will specifically deep link the '*Passing_search_data_via_deep_linking*' for the https://en.wikipedia.org/wiki/Mobile_deep_linking instance.

Threat

If In similar way, mobile deep links, help user jump through different applications and open exact URI within the application by inducing the activity within the application. However, there are serious vulnerabilities associated with scheme URLs [2, 3, 4], link-hijacking being a significant one. Any application was able to access the universally exposed deep link within the application, thus obfuscating the actual face of the link to serve malicious intents. For instance, application would create a facade of Facebook login page using Facebook deep link and thus steal user data. However, even though Android and iOS prompt users before launching the applications, there are many cases in which user falls victim to such attacks. Hence there is a need to devise a measure to restrict the malicious user of deep link.

In 2013, Android developed a mechanism known as Intent URL [6]. Intent URL defines how deep links should be called by websites. Instead of calling URL (like fb://profile), the protocol defines specifying destination app identifier in parameter. In 2015, iOS and Android introduced AppLinks. It restricts developers to use http and https schemes only. It also make use of app-to-link-association to verify the applinks with the webhosts. For example, mobile OS verifies registered applink (say https://facebook.com/profile) by contacting corresponding web host (facebook.com) for verification. This restricts other applications from claiming the link.

Even though these measures are being introduced, the applications providing backward compatibility allow the use of old mechanisms of deep links (scheme urls) which are

1. Unverified applink: The website or webhost which does not support deep linking.

2. Preferences: Android user can set his preference by selecting Always option on prompt for invoking application via deep link.

vulnerable. These problems range due to different reasons, like inappropriate developer habits, server misconfiguration, inconvenient protocols used, or simply due to legacy applications [our paper].

Our work presents a survey of these user habits and developing a plugin for developer to enforce correct use of deep link in the application. Along with this, we also demonstrate how Android also suffers from the problem of over-permissibility discussed briefly in section 2.4. Android OS grants excessive permissions to the **unverified AppLinks**[1]. These unverified applinks (HTTP/HTTPS) can be readily intercepted to steal information. The malicious app in face of genuine app can gain permissions from the user, since android allows user to set **preference**[2]. We also demonstrate how lack of standardization in implementation of deep link on webhost can lead to serious threats to application users [section 2.4.1].

II. Mobile Deep Links

Universal links are http links that open in your iOS application instead of in your web browser (safari). They are called App links on Android OS. An older version of these are known as Scheme URLs which registered themselves against a custom URL to open other applications directly on the operating system without any authentication mechanism.[9]

1. Implementation technique in iOS

The newly developed universal/app links help us bring the web and the native mobile operating systems closer. They have also allowed us to index a specific location in various applications on search engines and even in mails. They have also helped remove various security vulnerabilities and authenticate our applications against the website/server it register's itself with.

Following are the steps one needs to follow to correctly implement universal links on iOS:

i. apple-app-site-association (AASA) JSON file␟This file is used to establish connection between your application and website's domain/server. It mainly contains appID and paths where *appID* includes a string which uniquely identifies your application based on it's teamID and bubble identifier while path contains all the paths you which to index/link to.

ii. If you are using shared web credentials [10] you must also sign your AASA file

iii. Uploading the AASA file
You must upload the created AASA file in step 1 to your domain in either its root directory or in a .well-known subdirectory placed right inside your root directory
https://yourdomain.com/apple-app-site-association
https://yourdomain.com/.well-known/apple-app-site-association

iv. You should also make sure that your server has a valid secure certificate and its running on an https protocol and the content-type of the AASA file is set to application/json

v. Add entitlement in application by enabling the app. capabilities in the project. Also add all your domain addresses under associated domains ahead of the given applinks: format.

vi. Finally add the code that would handle the triggering of universal link within your application inside the AppDelegate file by implementing the application (_:continue:restorationHandler:) [11] method.

vii. Do not forget to test your implemented universal link mechanism via mail or notes application.

2. Implementation technique in Android

Deep links are very popular among the android applications. Various applications which are active on both the platforms of Android and Web use deep links intensively to connect their web links with Android applications.

To connect the URLs with the application in Android; user first needs to add the URL as intent-filter of the activity to which it's mapping inside the Android Manifest file. By doing so, the URL is mapped to the activity and whenever the URL is triggered in the Android Phone the activity is loaded. Developer can add his own logic in activity.java file if he wants to use the data present in the URL in the form of arguments of the URL.

Recently, Android also published a tool, 'App Links assistant' [12] which helps developer follow all the steps required for successful implementation of deep links. Though the implementation of deep links looks straightforward, there are various security vulnerabilities that Android did not handle and might be exploited by the attackers.

Implementation steps.

1. Identify URL and activity to be mapped.
2. Add URL as intent-filter in the activity tag of the activity identified inside AndroidManifest.xml file.
3. Add logic in the activity.java file

Create assetLinks.json file and host on the servers.

III. Our Work

(Github : https://github.com/mendkiakshay/xUlinks)

1. iOS

Implementing deep link on iOS can be a tedious and long process which will need developers to follow a long checklist. Even after which the implementation can be flawed if the developer skips or misinterpret even a single step of the process. Once the developer is confident of their implementation; they can use our tool (application) to verify if they have missed or wrongly done any of the steps. xULink will indicate the wrongly done steps after you provide it with the location of the directory of your project.

xULink's implementation is pretty straight forward and it checks the following steps within your project:

1. Associated domain is turned on within your Xcode's project capabilities

2. The domain where the apple-app-site-association (AASA) file is kept is a valid Domain Name System (DNS)

3. Server does not return error codes greater than 400
4. Appropriate content-type is set and received for the AASA file.

Working:

To check if the developer has turn on the switch to use the application capability we follow the project's directory and check for the existence of the .entitlement file. If the file is present it means that at least one of the application capability is turned on. If it is present we parse the entire file which is in XML format and key for the key com.apple.developer.associated-domains which specifically identifies the implementation of associated domain capability. Once the key is present, we fetch the list of applinks (domains) entered to verify against the iOS application and check if it is a valid domain name by making an http request call to the server. We also intend to check if the file is present at the correct location on the server, that is either on the main domain specified or in a .well-known subdirectory present on the domain. We also check if any of the specified domains are returning any error codes greater than 400 where would help us detect any missing URL. Using the networking call we made previously, we also check if the AASA file returns an expected content-type of application/octet-stream and if it doesn't we show a red symbol indicating the incorrect configuration of the it. xULink also checks if the JSON is correctly formatted in the AASA file and notifies us about the status of the same.

We have also made an Xcode extension which is a source code editor which detects the usage of SchemeURIs and adds a soft warning below the usage in your Xcode project.

Security Risk:

One security risk which we could think of while implementing the application was the editing of team identifier and/or the bundle identifier used to identify the application in the AASA file.

To tackle that, we tried to fetch the team and bundle identifier from the available project directory but thanks to the security measures taken by Apple in their development environment it cannot be exposed outside or infect even in the project's *plist* file. Hence we would be adding 2 inputs which would take the team identifier and the bundle identifier explicitly from the user and check those against the AASA file present on the server.

2. Android Security Analysis

Earlier Android used to handle deep links with the help of custom URLs called as scheme links. Scheme links had a custom scheme instead of http or https. Scheme URLs had the biggest problem of link hijacking; where in any app could use any scheme URLs, resulting in malicious applications mapping with genuine URLs. When user install both genuine and malicious application, upon triggering of deep link, Android gives a pop-up or chooser and lets user select which application (malicious or genuine) to open.

To avoid this problem, Android introduced App links, which mandatorily used HTTP/HTTPS as its scheme, it also introduced assetLinks.json file which has SHA256 fingerprints along with namespace and package name. When a user installs the application, the assetLinks.json file is verified from the server and if the configuration is correct it lets deep linking work. However, this feature is active only when developer sets *autoverify* tag in manifest file as on. The attacker in this case can use any genuine link and map it with its malicious app. In this case when the same URL is used by two or more applications, Android will give the chooser again and will let user take the decision.

The situation worsens when there are URLs on the web of an existing web application that are not mapped by its owner to any activity of its Android app. In this case an attacker might use such links and map them to the malicious application. When these hidden web links are triggered, the malicious application is opened by default, without giving a chooser as there is no other app that can handle the triggered link.

On top of the existing issues, Android tends to give excessive permissions to the applications managed by the user; which in turn create more security problem as user may not be aware of these problems. Consider a scenario in which a deep link is mapped by an application (malicious) which is also *browsable*, hence can be opened by the application as well as browser. In this case, when the deep link is triggered user gets a chooser to decide which application to use to open the link. If user selects the malicious app, and clicks on 'Always' use button, Android gives permission to all the deep links that are present in the malicious apps manifest file. The manifest file may contain the deep links that belong to a genuine app. When such links will trigger now, the Android will always open the malicious application by default, as it gave permission to all the links present in manifest file. These problems of over-permissibility and links hijacking in app-links are demonstrated in the experiment.

Along with the security analysis of Android, we also made a helper tool; a console application in java that suggests user to not use Scheme URLs as they are vulnerable to hijacking. Though Android tells its developers to use App Links instead of Scheme URLs, there is legacy code of many applications that still uses scheme URLs, and Android still supports it.

The console application parses through the AndroidManifest.XML file and filters out all the links that do not have schemes as HTTPS and suggests user to use HTTPS as a secure channel.

IV. Experiments and Observations

To experiment the Android vulnerabilities we created two android applications;

1. Taitwale Android Application (TaitwaleImaging.com is a genuine website, we took permission of the owners of this website before conducting the experiment)
2. Attackers Application; this is a dummy Android application to simulate the adversary model.

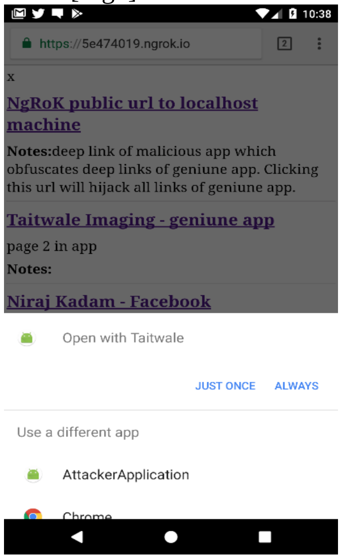
Below is the how different URLs are mapped to different activities inside the applications.

URL ID	URL	Application	Activity
dl1	https://5e474019.ngrok.io/depth/1/	Attacker	Activity 2
dl2	https://www.facebook.com/niirajkadam	Attacker	Activity 3
dl3	https://www.twitter.com/niirajkadam	Attacker	Activity 3
dl4	http://taitwaleimaging.com/	Attacker	Activity 4 (Blank Activity)
dl5	http://taitwaleimaging.com/	Taitwale	Activity 2
dl6	http://taitwaleimaging.com/about-us/	Attacker	Activity 4

Below are the cases that we tested in the experiment.

Case 1: Non-malicious activity in the attacker's application. The weblink, dl1 of attacker's website is mapped with attacker's activity. Upon triggering dl1, activity 2 of attacker application is opened. This is a non-malicious activity as attacker is not hijacking any other applications URL.

Case 2: Link hijacking by the attacker application. The genuine link of Taitwale application, dl4 is mapped by both Taitwale and Attacker application. In this case when dl4 is triggered; a chooser is displayed which lets user select the required application. [Fig1]


https://5e474019.ngrok.io

x

NgRoK public url to localhost machine

Notes:deep link of malicious app which obfuscates deep links of geniune app. Clicking this url will hijack all links of geniune app.

Taitwale Imaging - geniune app

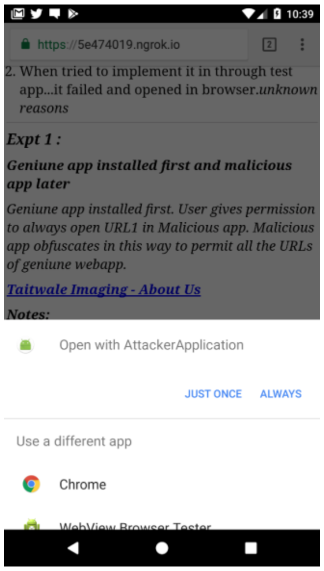
page 2 in app

Notes:

Niraj Kadam - Facebook

Fig 1.

Case 3: Hidden links of genuine application hijacked. The web-link dl6 which is a genuine URL of Taitwale web application, but was not mapped by Taitwale android application; was hijacked by Attacker application. Upon triggering this link Attacker application is opened without showing Taitwale application in the chooser. [Fig2]


https://5e474019.ngrok.io

2. When tried to implement it in through test app...it failed and opened in browser.*unknown reasons*

Expt 1 :

Geniune app installed first and malicious app later

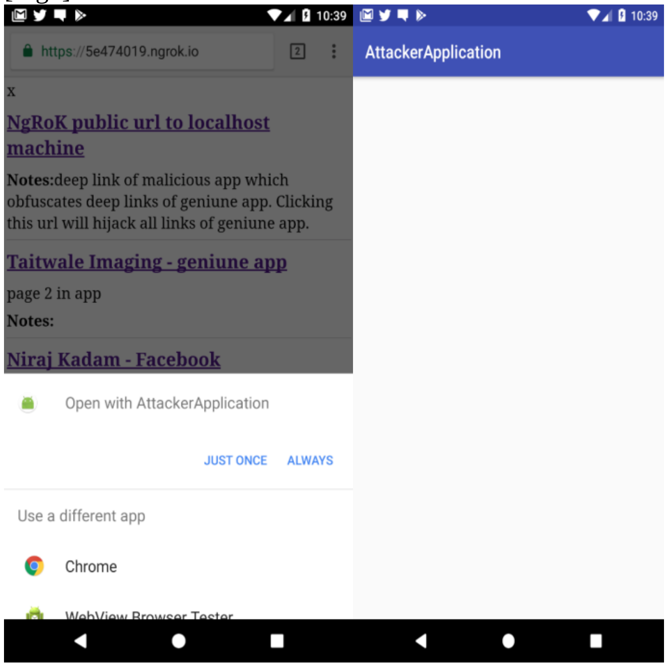
Geniune app installed first. User gives permission to always open URL1 in Malicious app. Malicious app obfuscates in this way to permit all the URLs of geniune webapp.

Taitwale Imaging - About Us

Notes:

Fig 2

Case 4: Over-permissibility problem. The dl1 URL was opened with attacker application with clicking on 'ALWAYS' button in the chooser. Upon doing so, the URL dl4 was opened with attacker application by default without showing the chooser to select 'Taitwale' application. [Fig3]


https://5e474019.ngrok.io

x

NgRoK public url to localhost machine

Notes:deep link of malicious app which obfuscates deep links of geniune app. Clicking this url will hijack all links of geniune app.

Taitwale Imaging - geniune app

page 2 in app

Notes:

Niraj Kadam - Facebook

Fig 3

Case 5: Web links of Facebook mapped by Attacker application. Facebook profile page link, dl2 is mapped by Attacker application. Upon triggering the link.

V. Conclusion and Future Works

In this work, we have identified and analyzed the known vulnerabilities of deep links in Android and iOS, the most popular mobile operating systems. We have successfully demonstrated that even with more secure applinks, the lack of proper implementation and misconfigurations at the server end leaves the applinks vulnerable to attack. Hence, we have developed XCode extension and console application for ensure the probable mistakes at the developer's end. In our work, we have also showed the over-permissibility issue of Android, playing with famous applications like Facebook and Twitter. Our observation show that Facebook incorporate different mechanism for deep link implementation by adding a payload in the link, dynamically generated by the host. This leaves the original deep links open to be attacked by the attacker.

There had been previous study of vulnerabilities of deep links in inter-app communications in Android and iOS [5, 8, 7, 4]. The fundamental issue according to our deduction was the lack of standardization enforcement in practice of deep linking. We also analyzed a strange pattern of use of web URI in android SDK code window. The Android application requires exact URI of the online asset that needs to be accessed. However, it behaved strangely for use of prefix www. It would work for prefix www for some websites (like Twitter) but would not work for other (like Facebook). Also the use of forward slash (/) towards the end of the URI behaves strangely. We verified the same on different versions of Android, and concluded that it may be because of some configurational patterns at the server side.

The domain of deep link needs to be explored more for its security as various applications are being built to different effect to exploit deep link feature. We suggest a protocol to be developed to perform a handshake between the app and the webhost to verify the app and provide permission for the use of the deep link. To avoid link hijacking, all the URLs of any web application should be mapped with its Android counter-part. By doing so, even if attacker uses the same URL, Android will give a chooser, to select genuine or malicious application. This does not solve the problem completely, only mitigates it by giving rights to the end-user. It might happen that developer may not have an activity for that specific URL; in such case a blank activity can be created which has the website URL and can redirect the user to the browser.

References

[1] TANZIRUL AZIM, ORIANA RIVA, S. N. "uLink: Enabling user-defined deep linking to app content." *In Proc. of Mobisys (2016).*

[2] XING, L., BAI, X., LI, T., WANG, X., CHEN, K., LIAO, X., HU, S.-M., AND HAN, X. "Cracking app isolation on apple: Unauthorized cross-app resource access on MAC OS X and iOS." *In Proc. of CCS (2015).*

[3] CHEN, E. Y., PEI, Y., CHEN, S., TIAN, Y.,KOTCHER, R., AND TAGUE, P. "Oauth demystified for mobile application developers." *In Proc. of CCS (2014).*

[4] CHIN, E., FELT, A. P., GREENWOOD, K., AND WAGNER, D. "Analyzing inter-application communication in Android." *In Proc. of MobiSys (2011).*

[5] Fang Liu, Chun Wang, Andres Pico, Danfeng Yao, and Gang Wang, Virginia Tech, "Measuring the Insecurity of Mobile Deep Links of Android." *USENIX 26th security.*

[6] Android Intents with Chrome. *https://developer.chrome.com/multidevice/android/intents*.

[7] ELISH, K. O., YAO, D., AND RYDER, B. G. "On the need of precise inter-app ICC classification for detecting Android malware collusions." *In Proc. of MoST (2015).*

[8] WANG, R., XING, L., WANG, X., AND CHEN, S. "Unauthorized origin crossing on mobile platforms: Threats and mitigation." *In Proc. of CCS (2013).*

[9] Guide to Universal links http://samwize.com/2017/11/10/guide-to-universal-links/

[10] Shared web credentials https://developer.apple.com/documentation/security/shared_web_credentials

[11] NSUserActivity Documentation https://developer.apple.com/documentation/uikit/uiapplicationdelegate/1623072-application

[12] App Link Assistant https://developer.android.com/studio/write/app-link-indexing.html