Phishing Attacks on Modern Android

Simone Aonzo, Alessio Merlo, Giulio Tavella
DIBRIS - University of Genoa, Italy
{simone.aonzo,alessio}@dibris.unige.it
me@giuliotavella.info

Yanick Fratantonio
EURECOM, France
yanick.fratantonio@eurecom.fr

ABSTRACT

Modern versions of Android have introduced a number of features in the name of convenience. This paper shows how two of these features, mobile password managers and Instant Apps, can be abused to make phishing attacks that are significantly more practical than existing ones. We have studied the leading password managers for mobile and we uncovered a number of design issues that leave them open to attacks. For example, we show it is possible to trick password managers into auto-suggesting credentials associated with arbitrary attacker-chosen websites. We then show how an attacker can abuse the recently introduced Instant Apps technology to allow a remote attacker to gain full UI control and, by abusing password managers, to implement an end-to-end phishing attack requiring only few user's clicks. We also found that mobile password managers are vulnerable to "hidden fields" attacks, which makes these attacks even more practical and problematic. We conclude this paper by proposing a new secure-by-design API that avoids common errors and we show that the secure implementation of autofill functionality will require a community-wide effort, which this work hopes to inspire.

KEYWORDS

Mobile Security, Phishing, Password Managers, Instant Apps

ACM Reference Format:
Simone Aonzo, Alessio Merlo, Giulio Tavella and Yanick Fratantonio. 2018. Phishing Attacks on Modern Android. In *Proceedings of 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3243734.3243778

1 INTRODUCTION

The role mobile devices have in our lives has been exponentially increasing in the last decade. Recent reports have shown that more than half worldwide website traffic has been generated via mobile devices [37]. Users take advantage of these devices not only to browse websites, but also to access social networks and other online services, such as online banking. Thus, to improve user experience, developers of web services often implement native Android apps, making mobile devices portals to their associated web backends. For example, a vast portion of Facebook accesses in the US is performed via mobile device [12]. According to these reports, this trend is forecasted to only increase in the future, and users are going to perform more and more often one of the most basic security-sensitive action: authenticate to mobile apps backends by inserting their credentials. On the one hand, this shift towards the mobile world pushed Google and platform developers to design new technologies and mechanisms to decrease the friction of these user interactions. On the other hand, unfortunately, the more frequently users will be asked to insert credentials on their mobile devices, the more attackers will find mobile phishing attacks rewarding.

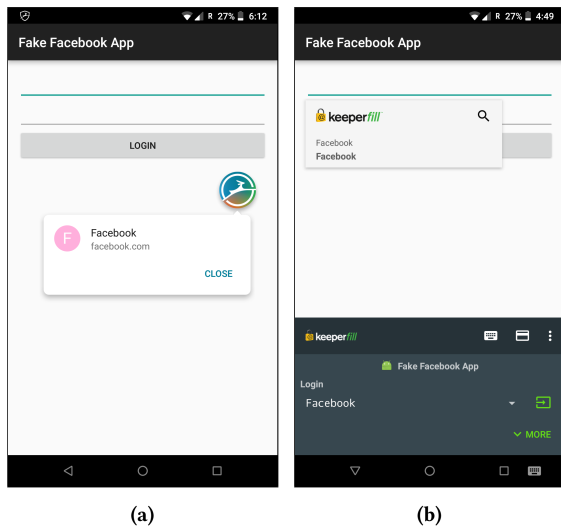
In this paper, we take a look at new features introduced in modern versions of Android, and we show that while they do simplify both users' and developers' lives, their weak design and implementation allow attackers to abuse them, making mobile phishing attacks significantly more practical than what previously thought.

Mobile password managers. The first aspect we look at is the growing popularity of *mobile password managers*. Password managers have been initially developed for the web, and the security community has long praised their many benefits. For example, they provide a practical way for users to use different pseudo-random passwords for each web service they interact with, thus discouraging the use of simple, easy-to-guess, and shared passwords across accounts. In fact, the user has a chance to store her credentials and to associate them to specific websites: when the user later navigates to the same website, the password manager identifies the website through its domain name, and it then suggests (and in some cases automatically fills) the right credentials on behalf of the user.

As a way to support the increasing amount of mobile users, password managers are now also available for mobile devices. Mobile password managers are developed as apps, and they include advanced sync features, which allow suggesting (and filling) website-related credentials to their associated apps.

From a technical standpoint, these password manager apps either need to have support from the Android Framework, or they require modifications to their potential "clients" (e.g., the Facebook app). In fact, Android apps sandboxing mechanism prevents them to interact with external apps programmatically. To date, there are three mechanisms that act as necessary basic blocks to allow for their implementation. The first is the Accessibility Service [19] (a11y, in short): while, in theory, a11y is a mechanism to allow apps to be "accessible" to users with disabilities, it also allows apps to interact with others programmatically, and it thus provides the technical capability needed by password managers to implement their functionality. Since recent works have shown how a11y can be abused [5, 6, 16, 29, 32, 33, 38], Google has recently implemented the Autofill Framework [20], a new component of the Android Framework specifically developed to allow password managers to suggest and autofill credentials to mobile apps (without the need to rely on a11y). The third mechanism is called OpenYOLO,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '18, October 15–19, 2018, Toronto, ON, Canada
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243778



(a) (b)

Figure 1: Android password managers (1a) Dashlane and (1b) Keeper, suggesting Facebook credentials to a fake malicious app.

a recently-proposed protocol for storing and updating credentials for mobile apps [18]. This mechanism is developed by Google and Dashlane, and it follows a different paradigm: it does not affect the Android Framework, but it requires modifications of each "client" (e.g., Facebook) and "server" app (e.g., the password manager).

In this paper, we show that all these three mechanisms are affected by design and implementation issues. At the root of the problems is the need to *bridge the mobile world with the web world*: given an app with a login form, how can a password manager know whether this app is the legitimate Facebook app (and it is thus entitled to access Facebook credentials) or whether this is a malicious app attempting to appear as the legitimate one? How is it possible to know which app is linked to which domain name? The key design issue is that *all these three mechanisms use the app package name as the main abstraction to identify an app*. Password managers thus need to somehow *map* package names to associated websites.

While a technical solution to securely implement such mapping exists, this work shows that the poor design choices of the underlying mechanisms push to the implementation of vulnerable solutions. In particular, we have investigated the four leading third-party mobile password managers app (Keeper [24], Dashlane [2], LastPass [3], 1Password [1]), as well as Google Smart Lock (GSL) [22]: we have found that only GSL is securely implemented. Moreover, we have found that Keeper, Dashlane, and LastPass all implement various (vulnerable) heuristics, each of which misplaces trust in an app package name or other metadata. The net result is that *it is possible for a malicious app to systematically lure these password managers to leak credentials associated with arbitrary attacker-chosen websites*. To make it worse, we note that these attacks also work for websites for which an associated mobile app does not exist. These attacks effectively make mobile phishing more practical: differently than all previous works, the user is not even asked to type her credentials; the user is just asked to allow password managers to autofill the credentials on her behalf.

It is interesting to note how, on the web, password managers do not ease phishing attacks, but quite the opposite. In fact, web password managers check the current website domain name to determine whether to auto-fill (or auto-suggest) credentials: if the domain name does not match the expectations, no credentials are suggested. Thus, an attacker that uses particular Unicode characters to create a *facebook.com*-looking domain name may fool a human, but not a password manager: the malicious domain name will be different from the legitimate one, and the password manager suggestion will not trigger. We thus argue that the mere fact that a mobile password manager is suggesting credentials associated with the target website inherently adds legitimacy to the attack, making it even more effective.

Instant Apps. The second modern feature we explore in this paper is called *Instant Apps*. This technology, implemented by Google, allows users to "try" Android apps at the touch of a click, without the need to fully install the app. Under the hood, the system works by asking developers to upload small portions of their Android app, called Instant Apps, and to associate a URL pattern to it. The developer needs first to prove that she controls the domain name of the URL pattern. This is carried out through a multi-step procedure called App Link Verification [27] which relies on Digital Asset Links [17] protocol (it makes possible to associate an app with a website and vice versa, via verifiable statements). After this deployment step, the user will be able to click on a link (pointing to the specified URL), and, after a one-time confirmation, the Instant App is automatically downloaded and executed on the user's device.

In this paper, we show that this technology, while indeed a very useful Android feature, can make phishing attacks more practical. The key observation is that *Instant Apps provide an attacker the ability to gain full control over the device UI, without the need of installing an app*. In a browser-only phishing scenario, the user would have a chance to notice the green lock and inspect the domain name. However, in an Instant Apps-based attack, the attacker has full capabilities to deceive the user. For example, an attacker could create a full-screen Facebook login view (as the real Facebook app would do). As reported in existing works [4, 8, 9, 36], users cannot distinguish between these. As another example, an attacker could simulate the view of a full browser; as the attacker controls every pixel of the screen, nothing prevents her to show the user a browser-like view with a spoofed *facebook.com* domain name and a green lock: once again, this attack is indistinguishable from a legitimate scenario. As highlighted by several recent works, the key insight is that the UI on mobile devices cannot be trusted, and Instant Apps provide a technical way for an attacker to move from a scenario where she does not fully control it (like a web page somehow constrained by the web browser security mechanisms) to a scenario where she fully does.

End-to-end attack. The combination of flawed mobile password managers and Instant Apps allow attackers to develop and mount mobile phishing attacks that are much more practical than what previously known [8, 9, 16, 36]. In fact, we have found that, although Instant Apps are not "fully installed" apps, 1) password managers currently do not notice the difference, and that 2) their package

name is the same as the associated full app. This means that the package name of the Instant App is attacker-controlled, and that *it is thus possible to trick password managers to auto-fill credentials for an attacker-chosen website even without requiring the installation of an additional app*. This allows an attacker to bootstrap an end-to-end phishing attack by luring the victim into visiting a malicious webpage: such webpage may contain, for example, a fake Facebook-related functionality. Upon clicking on it, the Instant App mechanism is triggered, the attacker can spoof a full-screen Facebook login form, at which point the password manager would offer to automatically fill the credentials on behalf of the victim.

To make things worse, we found that *current password managers fill hidden fields as well*. An attacker could thus create a form with a visible username field but a hidden password field: while the unsuspecting user thinks she is autofilling only the username, her password manager will silently leak her password to the attacker.

To the best of our knowledge, the attacks presented in this paper are the most advanced and practical phishing attack techniques to date. In fact, all existing approaches assume a malicious app installed on the user's device, ask the user to manually insert her credentials (which although not technically problematic, may reduce the attack success rate), or fall back to web-based phishing attacks (that are noticeable at least from the browser bar) [8, 9, 16, 36].

A look to the future. The future of these problems does not look encouraging. The current API has a design that is error-prone and does not force developers to take all necessary steps to avoid severe vulnerabilities. In this paper we discuss the design of a new API, called *getVerifiedDomainNames()*, that *uses domain names as the main abstraction level* (instead of package names, which should not be trusted), and it hides behind a single, central implementation the necessary logic and security steps to establish that a requesting app does have authority over the credentials it is requesting. Internally, this new API relies on an existing technical solution based on Digital Asset Links [17] verification. This solution requires websites owners to publish an "assets" file on their website so that an app-website "link" can be established.[1] This is the same mechanism that Autofill Framework and OpenYOLO *suggest* developers to use: the difference is that our API *forces* them to use it, instead of leaving them open to implement vulnerable solutions—as they did.

Unfortunately, although we believe that this solution is technically sound, the current ecosystem is far from being ready. In fact, the App Link Verification requires collaboration from websites owners, as they would need to upload the appropriate assets file to their website. To determine the readiness of the ecosystem to this mechanism, we first built a dataset of 8,821 domain names extracted from the password managers we have analyzed (given the source of this dataset, these domain names are guaranteed to have at least one login form, otherwise they would not be relevant to password managers). We then checked how many websites already link themselves to an app: to our surprise, only 178 of them currently have an *assetlinks.json* compatible with the proposed solution, which is around 2%. This means that, to date, *password managers developers do not have the necessary information to securely implement their functionality, even if they wanted to*. One may then wonder how Google Smart Lock, which we found to be secure, implements such mapping. We found that, although a technical solution exists, this process is not automatic: according to the official documentation [23], the last step of the process requires developers to manually fill a Google Form [26] to provide the needed information. We conclude that the adoption of a secure mapping cannot be easily addressed by the single actors alone, but it requires a community-wide effort, which this work hopes to inspire.

[1] *Such "assets" file needs to be placed at a specific location: https://domain.name/.well-known/assetlinks.json*

In summary, this paper provides the following contributions:

- We performed the first security analysis of mobile password managers and the three core technologies they rely on: a11y, Autofill Framework, and OpenYOLO; we have uncovered design and implementation issues that allow attackers to trick password managers to leak to malicious apps credentials associated to arbitrary attacker-chosen websites;
- We show how Instant Apps can be abused to gain full UI control and how they can be used to lower the bar for stealthy and practical phishing attacks;
- We present an end-to-end phishing attack that abuses password managers and Instant Apps, and we show that current implementations automatically fill hidden password fields. We believe this to be the most advanced and practical phishing attack to date;
- We propose a new secure-by-design API that moves the abstraction from package names to domain names;
- We provide empirical evidence that the current ecosystem is not ready yet to support secure autofill on mobile devices, and that a community-wide effort is required to address these issues.

2 BACKGROUND

Android mobile apps are compiled and distributed as self-contained files, called APKs. Apps are usually distributed via app stores. One main distribution option is the official Google store, called Play Store. Alternatively, a developer can upload her app to so-called third-party markets. Although very popular in some countries (e.g., China, India), these markets are traditionally perceived as less secure. For this reason, to install apps hosted outside the Play Store, users need to manually enable a security option called *side-loading* (off by default).

Every app needs to define some metadata, the most important being the *package name*, a developer-specified string that acts as the main app identifier. While it is commonly believed that package names are analogous to web domain names for mobile apps, they are actually very different for what concerns security guarantees. In fact, the only constraint is that the package name needs to be unique 1) across the apps published on the Play Store and 2) across the apps installed on a given device. No other security guarantees are provided.

Once installed, apps execution is sandboxed via a number of security mechanisms. Thus, apps have access to private storage, and they cannot interfere with the execution of other apps. Access to security- and privacy-related functionality and information are controlled by the Android permission system (i.e., each app needs to declare the list of permission it needs to work properly), and inter-app communication is implemented via the Intent system.

3 ANDROID PASSWORD MANAGERS

A password manager (PM from now on) is a tool that stores and manages user's credentials like usernames and passwords. PMs aim to suggest to the user the right credentials to insert in login forms, thereby leveraging the same user from the burden of memorizing their sensitive data.

PMs have been originally conceived for the web domain and mostly implemented as browser extensions. They work as follows: the first time a user visits a website and inputs credentials in online forms, the PM stores such credentials on its backend and it maintains the association between the credentials and the domain name. When the user visits the same domain later on, the PM recognizes and verifies the domain, and it suggests the credentials to insert in the corresponding login form.

The increasing popularity of mobile apps acting as wrappers of their corresponding websites (e.g., email providers, online documents, social networks, home banking) has motivated the development of password managers for mobile devices. These are implemented as mobile apps, and they have the capability of helping managing and automatically filling user's credentials in other apps. Modern PM apps and browser extensions also provide advanced sync functionalities between app and website credentials. For example, consider a user opening for the first time the Facebook app, which requires the users credentials: at this point, the PM identifies the app, determines which domain name this app is associated to (i.e., *facebook.com*), and checks whether it has credentials associated to it; if this is the case, it auto-suggests them to the user, who can thus authenticate herself with few clicks, without the need of manually inserting her credentials. Figure 1 shows two examples of password managers auto-suggesting credentials.

From the technical standpoint, filling credentials requires proper mechanisms allowing PMs to access the UI of other apps, thereby bypassing the isolation provided by the sandbox. To this end, modern Android versions offer three mechanisms to support the implementation of PMs apps: *Accessibility Service*, *Autofill Framework*, and *OpenYOLO*.

Accessibility Service. The Accessibility Service, a11y in short, is a framework that allows third-party apps to be accessible to users with disabilities [19]. An app can make use of this framework by requesting the `BIND_ACCESSIBILITY_SERVICE` permission and by implementing a component that, while in the background, receives callbacks by the system when "Accessibility Events" are fired. These events are related to some specific transitions on the user interface, e.g., the focus is changed or a button has been clicked. This service has also access to relevant contextual information, the most important being which app the user is currently interacting with. This last information is made available by means of the package name of the app.

Even if a11y has been developed to assist users with disabilities, app developers have (benignly) abused this framework to implement a variety of different features, one of which is the implementation of password managers. In particular, PMs rely on a11y to determine which app the user is interacting with and whether there are text fields that could be filled with stored credentials; if that is the case, the PM then relies once again on `a11y` to programmatically interact with the target app and automatically fill the credentials fields on behalf of the user.

Unfortunately, while `a11y` is certainly useful, in the past few years there have been a number of research works from the industry and academic communities that show how `a11y` can be abused to perform a number of malicious functionality, from stealing user's personal information to the complete compromise of the device [5, 6, 16, 29, 32, 33, 38]. Due to these threats, Google has developed additional Android features so that apps do not need to have access to such powerful mechanism to implement their functionality. Since password managers are some of the most common and prominent use cases, Google has recently introduced the *Autofill Framework*.

Autofill Framework. The Autofill Framework [20] has been introduced in Android Oreo. This framework offers to password managers apps a technical solution to implement their core functionality without requiring access to `a11y`. In particular, the Autofill Framework allows an app to 1) determine which app the user is interacting with, and 2) fill credential fields programmatically.

The Autofill framework requires the developer to create an app that implements an *Autofill Service*, which allows filling out forms by injecting data directly into the views, such as the `EditText` widgets that store the credentials. To use that, the app needs to require the `BIND_AUTOFILL_SERVICE` permission. Android Oreo has also introduced some new XML attributes to assist password managers: `importantForAutofill`, which specifies whether the view is *autofillable*, `autofillHints`, which is a list of strings that suggests to the service what data to fill the view with, and `autofillType`, which tells the Autofill Service the type of data it expects to receive. Through these attributes, an app implementing an Autofill service is able to detect, classify, and fill form fields according to their types (e.g., username, email address, password). Note that an app that wants to be "compatible" with the Autofill Framework *must* use these XML attributes. Note also that only one Autofill service can be active at the same time (the user can select which one to use through a dedicated setting menu).

At run-time, when the user opens a supported app with a login form, the password manager is able to determine which app the user is interacting with (once again, through its package name) and it can offer the possibility to the user to automatically insert the corresponding credentials on her behalf.

OpenYOLO. OpenYOLO (YOLO stands for "You Only Login Once") is a recently developed protocol, supported by Google partnering with Dashlane, and it is available as an open-source library [18]. OpenYOLO does not require neither `a11y` nor Autofill Framework, but it requires to modify each app that wants to support OpenYOLO-based PMs. This mechanism is constituted by two components: the *client* and the *credential provider* (the server). The client is a component that needs to be embedded in each app that wants to support this protocol (e.g., Facebook). The credential provider, instead, is used within the password manager itself, and it is in charge of providing information to the password manager about the requester app identity. At run-time, the client seamlessly interacts with the credential provider (via the Intent mechanism), which, with the cooperation of the password manager, then returns to the client a set of credentials, if available. The interaction between the two components is depicted in Figure 2.

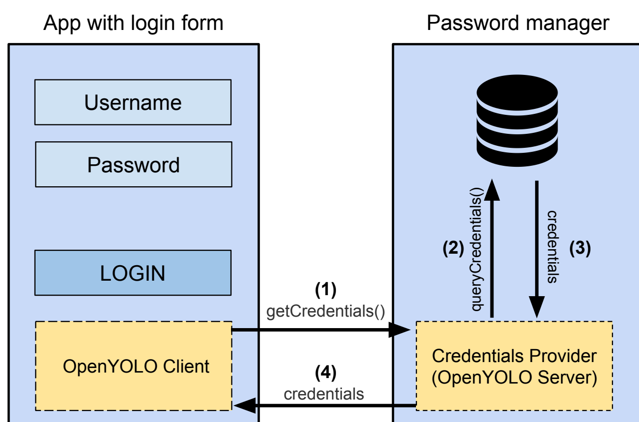


Figure 2: Deployment and workflow of OpenYOLO. We note that the interaction between the client and server is actually implemented via the Intent mechanism.

Note that OpenYOLO only helps PMs to interact with the target app. However, the implementation logic in charge of retrieving the correct credentials is left to the PM developers. In particular, the OpenYOLO credential provider exposes to the password manager the package name and the signature of the app requesting credentials. Once again, the PM is in charge of mapping the given package name to the appropriate domain names and credentials.

The central role of package names. Independently from which mechanism a password manager is relying on, the key information to identify which app the user is interacting with is the app *package name*. Unfortunately, in all these cases, the developers of the PM are left with the responsibility of securely mapping package names and domain names. As we will discuss in the rest of this paper, this design choice has a severe negative impact on the security of password managers and of the entire ecosystem. In fact, while mobile password managers have access to package names (and thus apps), the user's credentials they manage are related to websites. And this begs the question: "*how do mobile password managers actually link apps to their respective websites?*"

4 WEB AND MOBILE APPS WORLDS

The three mechanisms discussed in the previous section allow PMs to feed website-related credentials to the corresponding mobile app counterparts. To work properly, a PM needs 1) to identify the app that requires credentials and 2) to bridge the mobile and the web worlds. Since all the available mechanisms use apps package names as the main abstraction, in order to determine the right credentials to suggest, PMs need to somehow define a mapping between these package names and their corresponding website. We argue that *package names are the wrong abstraction for PMs to work with.* This section discusses the many pitfalls associated with this process, and how it is likely to misplace trust in these package names.

4.1 The Mapping Problem

PMs have access to package names as the key information to identify apps and to determine whether to automatically suggest credentials and for which website. Given a package name, PMs need to bridge the gap between the mobile apps and the web worlds. There is thus the need of *mapping* package names to their associated web domain names.

One of the problems is that package names resemble URLs (e.g., the package name of the official Facebook app is *com.facebook.katana*), thereby suggesting to inexperienced Android developers the same level of trustworthiness of the associated domain name, *facebook.com*. As we will see later in this paper, even developers of leading PMs severely misplace trust in package names, thus affecting the security of PMs and the entire ecosystem by making mobile phishing attacks more practical. We now discuss the main characteristics of domain names, package names, and the relation between them.

Domain names are trusted. In the modern web, domain names can be considered as trusted. With the wide adoption of robust DNS services and HTTPS, users and developers can determine whether they are securely visiting a given URL: the browser would verify the identity of the domain name by means of the PKI and the digital certificates ecosystem. Thus, web PMs do rightfully place trust in domain names. For example, a PM will automatically suggest Facebook's credentials whenever the user browses to *facebook.com*. Notably, PMs do *not* suggest Facebook credentials when the user visits a different domain name.

No authentication of package names. Differently than domain names, there is no authentication of package names. Anybody can create an app with a given package name, and it is possible for an attacker to create an app with the same package name of, for example, the legitimate Facebook app. However, one constraint must always be satisfied: there cannot be two apps with the same package name published on the Google Play Store or installed on the same device. In other words, package names act as unique keys. Note that third-party markets are not as controlled, and it may be possible to publish malicious apps with package names of legitimate apps. However, depending on the specific victim, it may be challenging to lure her to install such malicious apps from third-party stores.

No authority on "sub-packages." In the world of domain names, owners of the *example.com* are in control of sub-domains as well. In the world of package names, instead, this is not the case: the owner of *com.example* package name does not have any control over package names that may appear as "sub-packages." For example, nothing prevents anybody to create an app with package name *com.example.evil*: there is no relation between them. Thus, the sub-domain trustworthiness of the web world does not hold in the mobile counterpart. Unfortunately, as we will discuss later in the paper, this false sense of safety is a key cause of security issues among PMs.

The mapping problem. In the vast majority of cases, credentials are associated to websites, not to mobile apps: in fact, credentials are generally used to authenticate to a web service backend, not to a mobile app. Thus, given an app package name, PMs need to answer the question "*which website is this package name associated*

to?". This is not a trivial question to answer. To make things worse, PMs developers are left to implement their own "solution". Unfortunately, there are many pitfalls in implementing this mechanism, and we found that even leading PMs opted to rely on heuristics to solve this problem. It turns out that *most of these heuristics are vulnerable, and malicious apps can trick PMs to leak credentials associated to arbitrary websites.*

4.2 Attacker Practicality Aspects

From an attacker perspective, there are several aspects that would make a phishing attack more or less practical. In this section, we enumerate some questions related to the attacker capabilities. We will put them in relation to each vulnerable mapping in the next subsection.

Q1) Is the mapping vulnerable? The first question is, of course, about whether the mapping is vulnerable or not. We consider a mapping as *vulnerable* if an attacker can create an app that, although not being the legitimate one, can trick PMs into auto-suggesting credentials associated to a given website.

Q2) Can the legitimate and malicious apps co-exist? One of the most basic attack vectors is for a malicious app to have the same package name as the legitimate one. Since no two apps installed on the same device can have the same package name, this implies that, in this scenario, the legitimate and the malicious app cannot co-exist. This, in turn, implies that an attacker exploiting this package name-colliding technique would need to first lure the user to uninstall the legitimate app before the attack can be performed. Of course, this poses practicality issues. Thus, this question is about: can an attacker bypass this constraint? In other words, to give an example, can an attacker create a malicious app that can co-exist with the legitimate Facebook app and that, when opened, would trick PMs to auto-suggest the legitimate Facebook credentials?

Q3) Can the malicious app be hosted on the Play Store? In the general case, it is more difficult to lure the user to install an app that is not hosted on the Play Store. Thus, one relevant question is: is it possible for an attacker to upload her malicious app to the Play Store? The main constraint for an attacker is that no two apps with the same package name can be hosted on the Play Store at the same time. In other words, this question asks whether an attack requires creating an app with the same package name of an already-existing app on the Play Store. If yes, the only venue for the attacker is to lure the user to install the malicious app from a third-party market (via the side-loading process): although this attack is possible, it is less practical.

Q4) Can the attacker generate tailored suggestions? PMs have the capability to auto-suggest one or more set of credentials. Then, the user can choose one of them and, at the touch of a click, these credentials are automatically filled in the target app. Now, from an attacker perspective, the ideal situation would be to able to write a malicious app such that, for example, the PM would only suggest the credentials of *facebook.com* (or any other domain name chosen by the attacker). A less-ideal scenario is a PM where all the credentials are always suggested: although the user has the possibility to lure her credentials to the malicious app, this attack would be slightly less practical. Thus, the question is: can the attacker have fine-grained control over which and how many credentials are suggested?

Table 1: This table systematizes vulnerable mapping implementations and puts them in relation with attacker practicality aspects.

	Q1	Q2	Q3	Q4
Secure mapping				
Static one-to-one mapping	✓			✓
Static many-to-one mapping	✓	✓		✓
Crowdsourced mapping	✓	✓	✓	✓
Heuristic-based mapping	✓	✓	✓	✓
No mapping (all credentials suggested)	✓	✓	✓	

4.3 Vulnerable Mappings

This section systematizes the different possible implementations of the package names → web domain names mapping. For each of them, we describe how such implementation is vulnerable, to which attacks, and how practical it is with respect to the questions discussed above. The insights presented in this section are systematized in Table 1.

Secure mapping. The safest way to implement a mapping consists in securely verifying whether the developers of the current app have authority over a given domain name: if that is the case, then it is safe to auto-suggest the credentials of such domain name to the current app. One known solution to achieve this mapping is called Digital Asset Links [17] (DAL from now on). From a conceptual point of view, DAL allows for the definition of authentication domain equivalence classes, and it makes it possible to associate an app with a website and vice versa, via verifiable statements. This mechanism works by asking websites owners to publish on their website an "assets" file that contains a list of apps that can be legitimately associated with it. In this case, each app is identified by its package name and by the hash of its legitimate signing key. A third-party can then verify that an app is indeed legitimately linked to a website by checking whether the "assets" include a matching package name and the hash of the signing key.

Static one-to-one mapping. Consider a PM with a static one-to-one mapping, which maps one package name to exactly one domain name, and vice versa. As an example, consider the legitimate Facebook app, whose package name is *com.facebook.katana*, which is usually mapped to the *facebook.com* domain name. This simple mapping technique is vulnerable: in fact, Facebook credentials are suggested to any app whose package name is *com.facebook.katana*, even if the app is not the legitimate one. It would be possible to prevent this vulnerability by checking the certificate that signed the target app, and make sure it is one of the known, trusted one. Unfortunately, maintaining such list of known trusted certificates is a very challenging task. We consider this a vulnerability, but the attack is not very practical: in fact, the malicious app cannot co-exist with the legitimate one.

Static many-to-one mapping. Consider a PM with a mapping that maps n different package names $p_1, p_2, \ldots, p_n$ to the same domain name D. This can happen for different apps belonging to

the same companies: while they are all different apps (and thus they have different package names), they are all associated with the same domain name. This typology of mapping is problematic because it is frequent that the user would install only *one* (or a subset) of these apps. Thus, a malicious app with one of the remaining package names is able to steal the credentials. This attacker is more practical than the previous one because it does not require the attacker to lure the user to uninstall the legitimate app. However, the package names specified in the mapping likely refer to real legitimate apps on the Play Store. This means that the attacker cannot upload her malicious app on the Play Store (because package names need to be unique across the store), and the app needs to be side-loaded.

Crowdsourced mapping. Given the scale of the problem—millions of apps and website to map one with each other—one possibility to create a comprehensive mapping is by means of crowdsourcing. Thus, one approach is the following: consider a user who inserts credentials for a domain D to an app with package name P, and assume that the given PM did not know about this mapping: in such case, a popup can ask the user whether she allows such association to be shared with other users, so that everybody can benefit. If the user allows for it, this new association is sent to the backend, which, depending on the specific implementation, could immediately make this mapping available to all its users, or wait until a number of users higher than a threshold report the exact same association. If an attacker is able to "inject" a new association, then she can mount an attack that is more practical than the two alternatives above. In fact, she could inject a new mapping $p_{attacker} \rightarrow D$ (where $p_{attacker}$ is an arbitrary attacker-chosen package name): in this way, the PM would suggest credentials related to D to the malicious app with $p_{attacker}$ as package name. Since the package name is attacker-chosen, the attacker can choose a package name that does not yet exist, and she can upload the malicious app to the Play Store. Of course, this malicious app can also co-exist with the legitimate one, given the different package name.

Heuristic-based mapping. One last way to implement mapping is through heuristics. For example, one way is to infer which is the appropriate domain name by implementing heuristics on the package name of the app. One other strategy is to rely on some other metadata to take such decisions. From a security perspective, this is the most dangerous scenario. In fact, if such heuristics are implemented in a way that an attacker can game them, the attacker could create a malicious app that "maps" to an arbitrary attacker-chosen target. Also, in this case the attacker may be able to avoid constraints related to the package name of the malicious app, thus avoiding practicality issues.

No mapping. Another alternative for PMs is to not implement any mapping. In this case, the PM would always suggest *all* stored credentials associated with *all* websites. This option is simpler than all other alternatives, but it is not secure, especially when compared to what current web-based PMs do. As an example, consider the LastPass browser extension: in the current version, the extension does not allow a user to insert her Facebook credentials on a website that does not share the *facebook.com* domain name. This is done as a security protection against phishing: even if the domain name graphically looks like *facebook.com* (by, for example, using Unicode character, as it would be the case in advanced phishing attacks), the password will prevent the user to fall for this phishing attack: mobile PMs that do not implement mappings cannot protect from this threat. However, if no mapping is implemented and all credentials are suggested, such protection is not available.

5 CASE STUDIES

We performed the security assessment of the top four third-party leading PM apps (i.e., *Keeper*, *Dashlane*, *LastPass*, and *1Password*), each of which has millions of users around the world. We have also considered the Google Smart Lock, a service integrated with Google Play Services, which currently implements, among many other features, a password manager. In particular, we wanted to study how these PMs address the challenges described in the previous sections, and we were interested in answering questions such as: how does the suggestion system work? How do these apps map apps and package names to their associated websites? Is it possible for a malicious app to trick PMs to provide credentials for arbitrary websites? How difficult is for an attacker to mount such attacks? Moreover, as three out of four PMs include the OpenYOLO library, we assessed the reliability of its implementation.

This section describes the methodology we adopted and the details for each of the PM we have analyzed. Our findings, summarized in Table 2, are worrisome: three of the third-party PMs implement a mapping based on various heuristics that an attacker can easily game. In other words, an attacker can create an app so that the target PM auto-suggests credentials associated with an arbitrary attacker-chosen domain name. Note that, in such cases, an attacker can leak credentials even from websites that do not have an associated mobile app—as long as the attacker can game the auto-suggestion system, the attacker wins.

Last, it is worth noticing that all third-party PMs support both a11y and Autofill Framework (for Android 8+); more precisely, we note that each PMs keep asking for the a11y permission even on Android 8.0 for backward compatibility reasons, as many apps have not modified their layouts yet to include Autofill XML attributes. We have also noticed that from the perspective of a user who sees an app being auto-filled, sometimes the steps to get the credential are slightly different, or there are some graphical differences, between PM relying on a11y or the Autofill Framework. We will discuss them case-by-case; however, we underline that all attacks that we discuss here works independently from the supporting technique.

5.1 Methodology

We developed a three-step methodology to investigate the security of each password manager. These analysis steps are performed using reverse engineering assisted by simple static analysis (e.g., bytecode decompilation) and dynamic analysis (e.g., bytecode instrumentation, network analysis, etc.).

Step 1: Package name as app identifier. The first step is to determine whether a given PM uses the package name of the target app as the *only* information to auto-suggest credentials for a given website. This step is done in the following way: (1) Install the legitimate Facebook app and add the credentials to the PM; (2) Uninstall the Facebook app; (3) Install a malicious app that has the same package name as the Facebook app and contains a login form. This

Table 2: Summary of findings for Keeper (K), Dashlane (D), LastPass (LP), 1Password (1P), and Google Smart Lock (GSL).

	K	D	LP	1P	GSL
Secure mapping					✓
One-to-one mapping	✓	✓	✓		✓
Many-to-one mapping		✓			
Crowdsourced mapping			✓		
Heuristic-based mapping	✓	✓	✓		
No mapping				✓	
Q1) Vulnerable?	✓	✓	✓	✓	
Q2) Can co-exist on device?	✓	✓	✓	✓	
Q3) Can co-exist on Play Store?	✓	✓	✓	✓	
Q4) Targeted suggestion?	✓	✓	✓		

app is written so that the only aspect in common with the legitimate app is the package name, while everything else is intentionally changed; (4) Check whether the PM auto-suggests the real Facebook credentials.

Although this step is straightforward from the conceptual and technical standpoints, it is enough to reveal key information: since in our test we change all the aspects *except* the package name, if the PM provides the correct credentials, it means that the package name is the *only* information used by the PM to identify the requesting app.

Step 2: Mapping extraction. If the first step reveals that the package name is the only aspect that matters, we then proceed to our second step: we aim at determining which specific technique the PM uses to map package names to domain names. This step is performed by a number of black-box tests and by then supporting the findings via manual reverse engineering of the PM.

Step 3: Exploitation. The last step consists in developing techniques to game the system and exploit the peculiarities of a given mapping implementation, if vulnerable. In this scenario, a proof-of-vulnerability consists in an app written so that the PM under analysis is tricked to provide the credentials of an arbitrary attacker-chosen website. In the general case, this app will need to have a carefully crafted package name and, at the very least, a login form. In other cases, it may be required to tweak other additional metadata.

5.2 Keeper

The Keeper app is the most downloaded PM with more than ten million users on Play Store. Keeper supports both a11y and Autofill Framework (on Android 8+), but it does not support OpenYOLO yet. When the user selects a form, it shows an icon with a yellow lock close to the form. When the user clicks on this icon, if the app is recognized, the related credentials are suggested (see Figure 1b). Otherwise, it asks to create a new entry.

Keeper also downloads from its backend a configuration file with a list of known websites (and their names). This file, interestingly, does not contain any reference to known package names. In fact, this list is only used to auto-suggest website names when the user manually inserts a new set of credentials.

Mapping implementation. When the user opens an app that can be auto-filled, Keeper obtains its package name, through a11y or Autofill Framework. Keeper then needs to determine which website is associated with the current package name. To this aim, Keeper builds a *heuristic-based* mapping as follows: it uses the app package name to infer the URL of the app webpage on the Play Store (e.g., when the user opens the Facebook app, whose package name is *com.facebook.katana*, Keeper tries to access the webpage at *https://play.google.com/store/apps/details?gl=us&id=com.facebook.katana*). Then, if the webpage exists, Keeper parses out the domain name of the URL specified in the "app developer website field." This is the domain name that Keeper considers as the rightful owner, and it then stores the package name → domain name association in its internal mapping database. Finally, Keeper auto-suggests the credentials associated with this just-retrieved domain name.

Exploitation. Unfortunately, this mechanism is trivial to exploit for an attacker. In fact, the app developer URL is not validated by the Play Store and it thus cannot be trusted. We were able to create an app (with an arbitrary package name) and to publish it on the Play Store specifying *facebook.com* as the developer's website. In this way, when a user opens our app, the Facebook credentials (and only these credentials) are suggested.

5.3 Dashlane

Dashlane has been installed by more than one million users, and it supports a11y, Autofill Framework, and OpenYOLO. When Dashlane uses a11y, it shows its icon close to the form to fill; when the user clicks on it, the app is recognized and Dashlane suggests the related credentials (see Figure 1a); otherwise it asks to create a new entry. Instead, with the Autofill Framework, it directly shows a window with the suggested credentials or the launcher for creating a new entry, saving one interaction with the user.

Mapping implementation. Dashlane implements the mapping by means of two layers. The first one is a hardcoded mapping package → domain names containing 81 entries. The second layer is a *heuristic-based* mapping that attempts to infer which domain name should be associated to a given package name (this layer is used only if the package name is not contained in the static mapping). Our analysis revealed that such heuristic works in this way: Dashlane first splits the package name in components separated by the dots (e.g., the *aaa.bbb.ccc* is split in the three components *aaa*, *bbb*, and *ccc*). Then, for each component, it checks whether at least *three* of its characters are contained in the "website" field of one (or more) of Dashlane entries. For example, the package name *xxx.face.yyy* triggers an auto-suggestion for *facebook.com* credentials (as well as anything associated with *facts.com*, for example).

Exploitation. The static mapping is rather small and many entries are tied to well-known apps and websites. However, we noticed that such mapping is *many-to-one.* Therefore, there are multiple package names pointing to the same domain name. For example, we found that both *com.etrade.mobilepro.activity* and *com.etrade.tabletapp* point to *www.etrade.com*, the official website of the Etrade online banking platform: the two apps appear to be the smartphone and tablet versions of the same product, respectively.

Consider a user who has installed the smartphone version of the app. An attacker could then exploit the many-to-one mapping by luring the victim to install a malicious app having the package name of the tablet version (that the user did not already install): in this case, the attacker does not need to lure the victim to uninstall the first app (as it would be the case without the many-to-one mapping). We reported this attack for the sake of completeness, but we acknowledge it is affected by practicality issues.

However, the second layer of the mapping is severely vulnerable. In fact, it is sufficient to upload to the Play Store a malicious app whose package name contains three (or more) letters that overlap with the domain name the attacker wants to target; in this case, the malicious app will be auto-filled with the credentials of the victim domain. Furthermore, it is worth noticing that the malicious app can obtain credentials from multiple domains. For instance, we submitted to the Play Store an app with package name *com.lin.uber.face*: when opening this app, Dashlane promptly suggests Linkedin, Uber, and Facebook credentials.

Regarding OpenYOLO, Dashlane is exploitable exactly as a11y/ Autofill Framework, since the selection of credentials relies on the package name, which is parsed as previously described. Therefore, we wrote another malicious app embedding the OpenYOLO client library and we were able to obtain the credentials.

Interestingly, we have noticed that when Dashlane uses Autofill Framework instead of a11y, it performs some additional checks and it is able to determine that our simple proof-of-concept attempting to impersonate Facebook cannot be verified. In this case, a warning is shown to the user. To the best of our understanding, Dashlane employs a hardcoded list of well-known package name and signature pairs, and it checks our app against it. This is a promising step forward in the right direction. However, we found that these checks are easily bypassable. In fact, it is sufficient for a malicious app to *not* be compatible with the Autofill framework (this can be done by not using the new autofill-related XML attributes), and this will be enough to force Dashlane to rely on a11y and the vulnerable implementation.

5.4 LastPass

LastPass has been installed by more than one million users and it supports a11y, Autofill Framework, and OpenYOLO. With a11y, LastPass uses a permanent notification to alert the user if the currently active app has some form to fill; thus, she has to tap the notification to show a popup window with her credential; with the Autofill Framework, the user does not need to tap the notification and she will directly see the pop up window, as in Dashlane. This underlines that the support to OpenYOLO is still immature. However, the current implementation allows the user to select credentials and send them to any unidentified requesting app.

Mapping implementation. LastPass relies on two mappings. The first one is, once again, *heuristic-based*, and it works as follows. Given a package name, e.g., *aaa.bbb.ccc*, LastPass splits it in components separated by the dots (e.g., *aaa*, *bbb*, and *ccc*), and it builds a domain name pattern by using the first two in reversed order (e.g., *bbb.aaa*). LastPass will then suggest to the user all the credentials associated with domain names that end such pattern.

In case an entry does not exist, LastPass allows the user to search among her locally stored credentials and select (in case) one of them, thereby defining a new entry for the mapping. As such entries may be useful for other users worldwide, LastPass allows the user to share them with the community. This sharing step is at the basis of the second mapping, a *crowdsourced mapping*. LastPass downloads this global database at the first installation. At the time of writing, we found 19,273 crowdsourced mapping entries with repeated package names and domains, mostly many-to-one. For instance, we found a mapping between package names *com.tinder* and *com.tinderautoliker2* associated to the web domain *facebook.com*: Tinder is a dating app that needs Facebook credentials to authenticate the user, while TinderAutoLiker is an app available on alternative markets that automates some actions on Tinder services. It is also worth noting that the crowdsourced mapping contains errors, like invalid domains, domains with typos, and IP addresses belonging to local networks.

Exploitation. To exploit the first mapping strategy, the attacker can create an app with a package name beginning with the reverse of the target domain name. For example, we created an app with package name *com.facebook.evil* and we were able to upload it to the Play Store without problems: when the user opens this app, LastPass automatically suggests credentials related to *facebook.com*.

From the conceptual point of view, an attacker could exploit the second mapping as well. In fact, if the attacker is able to inject an arbitrary association, she can directly indicate to LastPass that, for example, her own package name should be associated to, say, *facebook.com*. For the sake of completeness, we tried to share with LastPass an association from one of our package name app to one of our test websites. However, this association did not become public to all users. We assume that LastPass make these “new” associations available to all its users only when a number higher than a threshold suggested them. An attacker could try to create a high number of fake accounts and to automatically share these fake associations. However, we have opted not to do it for ethical reasons. Moreover, an attacker can already game LastPass suggestion mechanism by exploiting the first mapping.

5.5 1Password

1Password has been installed by more than one million users and it supports a11y, Autofill Framework and OpenYOLO. Differently from previously analyzed PMs, 1Password organizes its entries in categories (e.g., credit card, database, driver license, login, wireless router, etc.). We focused on the login category. Once the user selects a form, 1Password behaves differently with respect to the supporting methodology: on Autofill Framework, it shows a small windows bearing the imprint “Autofill with 1Password”. Clicking on it, the user must insert the 1Password master password and search through all its previously saved credentials. With a11y, it directly loads the windows for searching among credentials. Albeit 1Password adopts the OpenYOLO library, the implementation contains just a stub that always returns empty credentials.

Mapping. 1Password does not provide any mapping, but it trivially suggests each stored credential through a searchable list, delegating the choice to the user. In other words, it is possible to autofill any requesting app with any stored credential.

Exploitation. The exploitation of 1Password was straightforward and did not require any further customization of the app. However, this attack is less practical than the other ones as the attacker does not have fine-grained control over the list of credentials that are auto-suggested.

5.6 Google Smart Lock

Google Smart Lock (GSL) is part of Google Play Services for Android. It was created to automatically keep the device locked when the user is not around and unlock it when specific user-defined constraints are met. For instance, the user can choose to have her device unlocked according to the presence of specific wireless connections, trusted locations, or when it recognizes the user's face or voice, or while the user is carrying the device. GSL has been equipped with the PM originally integrated into the Chrome browser. For this reason, GSL also offers a password-saving feature, taking advantage of Autofill Framework (which works just with compatible apps), and a synchronization mechanism with the Chrome desktop browser.

Mapping. We believe that GSL mapping is securely implemented. However, the burden of mapping creation is delegated to the developer who has to provide all the necessary information to Google. In particular, the official documentation describes a multi-step process [23]. From the technical standpoint, this process is based on Digital Asset Links [17], through which an app can be verifiably linked to a website (see Section 4.3, "Secure mapping"). However, this procedure is not fully automated, and developers are requested to fill a Google Form manually and to provide a set of information. We argue that such a process hardly scales, as it is centralized and it requires the manual intervention of the developer. To improve the current approach, Google should push the Digital Asset Links adoption and verify that it is correctly implemented. Moreover, we believe that Google would greatly benefit the community if it could make its current mapping database publicly available.

6 INSTANT APPS FOR FULL UI CONTROL

The attacks presented so far require a malicious app to be installed on the victim's device. This section discusses how this prerequisite can be waived by abusing the recently introduced *Instant Apps*. This technology, implemented by Google, allows users to "try" Android apps at the touch of a click, without the need for a full installation.

This mechanism works in several steps. First, the developer builds an Instant App, a small-but-functional version of her app, and she uploads it to the Play Store. The developer is also asked to associate a URL pattern to it (pointing to a domain name she controls). The idea is that when the user browses to a URL satisfying this pattern, the Android framework starts the process of downloading and running the Instant App associated with it. Of course, for security reasons, the app developer needs to first prove to Google that she controls the target domain name. This is carried out through a multi-step procedure called App Link Verification [27], which relies on Digital Asset Links [17] protocol (this makes possible to associate an app with a website and vice versa, via verifiable statements).

From the developers and users' usability perspective, Instant Apps is a great feature as it significantly lowers the friction for a user to test (and possibly fully install) an app. However, from the security point of view, *Instant Apps provide a venue for attackers to greatly facilitate phishing attacks.*

The key observation is that Instant Apps allow an attacker to move from *web phishing* to *mobile phishing*. Nowadays, web phishing is significantly more challenging than mobile phishing. On the web, the user can clearly see which website she is interacting with: she has the chance to check the domain name, whether the connection is done via HTTPS, and whether there is a valid SSL certificate. In the mobile world, however, there are no such indicators. In fact, as previous works have pointed out [8, 9, 36], there is currently no "green lock" or any space for any trusted indicator: these previous works have shown that a malicious app spoofing the Facebook UI can be made indistinguishable from the legitimate Facebook app—even for a security-savvy user. The key requirement for these pixel-perfect attacks is the ability to control all the pixels on the screen. A website cannot achieve that, but an attacker can use Instant Apps to do just that: gain code execution on the device outside the browser's JavaScript sandbox and gain the ability to fully control the UI (without requesting any permission).

Once the attacker has gained full UI control, there are many possibilities. One first example is that the Instant App could resemble the real Facebook app, which can be made indistinguishable from the legitimate one. A second example would be to *resemble the browser app itself*: as the attacker controls every pixel of the screen, nothing prevents her from showing the user a browser-like view with a spoofed *facebook.com* domain name and a green lock. Once again, this attack can be made indistinguishable from a legitimate scenario.

7 PRACTICAL PHISHING ATTACKS

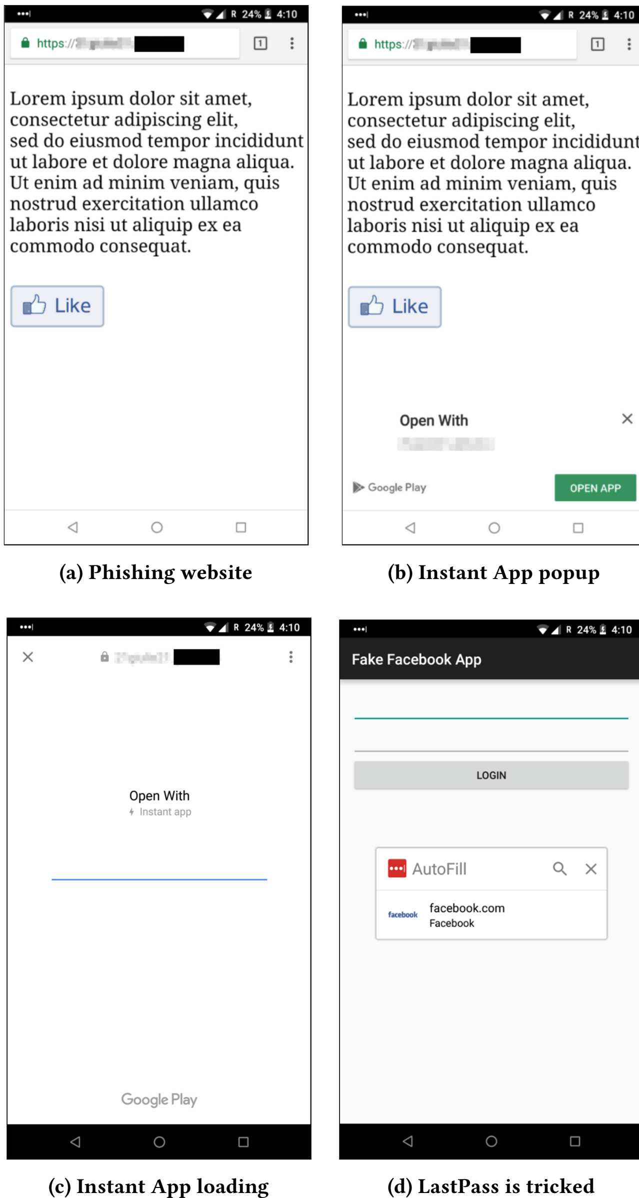
The password managers flaws and Instant Apps "features" we have highlighted thus far are independent of each other. However, we found that for what concerns phishing attacks, these two technologies are, in fact, complementary. In fact, we have shown that password managers can be tricked into revealing users' credentials, but these attacks require a malicious app (with an attacker-chosen package name) to be installed on the victim's phone: Instant Apps can be used to do just that.

We have found that Instant Apps, although they are not fully installed apps, do appear as they were to the Android framework and the components relying on it. The key insight is that even if the Instant App is not fully installed, the app somehow *lives* on the Android device, and its package name, application name, and icon are attacker-controlled (they are, in fact, the same as its associated full app on the Play Store). To make it worse, password managers currently do *not* notice the difference between full and Instant Apps, and they can thus be tricked to leak credentials even to them.

To make things worse, we have found that current password managers autofill hidden fields as well. This yet another "feature" that opens the possibility for a stealthy and practical end-to-end phishing attack, which we describe next.

7.1 End-to-end proof-of-concept

Consider a scenario where the user visits a website showing a spoofed Facebook "like" button, as in Figure 3a. Such button links to an attacker-controlled URL that is associated with her Instant



(a) Phishing website **(b) Instant App popup**

(c) Instant App loading **(d) LastPass is tricked**

Figure 3: Instant Apps phishing attack PoC

App. Once the user clicks on the like button, the Instant Apps mechanism is triggered: the popup asking the user confirmation to start the Instant App is shown, as in Figure 3b. This popup shows the application name and the icon, which, however, are fully attacker-controlled. The reader can see from Figure 3b how it is easy to mislead the user: for this PoC we used "Open With" as the name of the app and a fully white square as the app's icon ("showed" on the left of the application name). Upon the user's click on the "Open app" button, the Instant App is automatically downloaded, while the user is shown for few moments (about one second) the view in Figure 3c. At this point, the malicious Instant App is running on the user's device, as shown in Figure 3d. At this point, since our app was created with a package name following the *com.facebook.** pattern (see Section 5.4), LastPass is tricked to automatically suggest the real Facebook credentials to the user: With a click on the autofill popup, the full credentials are leaked to the attacker.

We note that our app is a clearly "fake" Facebook app, just for clarity sake and for ethical and copyright concerns: as this is a "live" PoC (to test the Instant Apps we needed to publish it to the Play Store), we preferred to avoid having a real spoofed Facebook UI.

Practicality considerations. We have shown how the user can be lured to leak her credentials in just a few clicks. We also note that the click on "Open app" (3b) and the "Loading" view (3c) are only shown the first time. That is, an attacker could make this attack even more practical by luring the user to approve and download the Instant App beforehand and for phishing-unrelated, seemingly innocuous reasons, to then make the transition from "Click to the like button" to "Spoofed Facebook UI" really seamless. We believe this attack strategy significantly lowers the bar, with respect to all known phishing attacks on the web and mobile devices: to the best of our knowledge, this is the first attack that does not assume a malicious app already installed on the phone and that does not even require the user to insert her credentials. These attacks are strictly more practical than all currently known mobile phishing works [8, 9, 16, 36].

7.2 Hidden Password Fields

We have carried out further experiments with the aim of assessing whether current mobile password managers are vulnerable to automatically filling hidden fields. We refer to fields as *hidden* if the field is, for one reason or another, not visible to a user. This is relevant because an attacker could create a form with a username field and a hidden password field: if the victim uses her password manager to autofill this form, her password will be silently leaked to the attacker. This is similar to what previous research has attempted with web-based password managers [**?**]: To the best of our knowledge, we are the first to show that these attacks work with mobile password managers as well. For this work, we considered four different techniques to make a password-related `EditText` seemingly invisible: 1) transparency, 2) small size, 3) same-color background and foreground, and 4) the *invisible* flag.

Transparency. To create a transparent `EditText` in Android, it is possible to set its alpha value accordingly (via the `setAlpha()` API). We note that if the alpha value is set to zero, both the `a11y` and Autofill Service cannot autofill the `EditText` because it is not visible anymore. However, setting an alpha value of 0.01 is enough to keep the field invisible and make the autofill mechanisms work.

Small size. One other venue to make a field invisible to the human eye is to make it very small. We found that password managers autofill password fields even if their size is 1dp × 1dp, independently from whether they are using `a11y` or Autofill Service.

Same-color background and foreground. If the text color is the same of the background color, the field (and its content) will not be visible. This technique works well with `a11y`. However, unexpectedly, it is not enough to trick Autofill Service. In fact, upon autofilling, the Autofill Service would overlay the autofilled fields

with a yellow overlay, thus making the hidden field visible to the user. However, it would be possible for an attacker to create in-app overlays (which do not require additional permissions) to cover this yellow overlay, thus making this artifact not visible to the user.

Invisible flag. It is possible to make a field hidden by setting its visibility to `View.INVISIBLE`. We found that a11y-based password managers do *not* autofill these "invisible" fields, but those ones using Autofill Service do so.

Discussion. We believe these additional techniques make end-to-end phishing attacks even more practical and problematic. While the unsuspecting user will use password managers and instant apps to quickly provide her email address or username, her credentials could be silently leaked to the attacker, with only few clicks. We also note that while some of the above techniques are not working with both a11y and Autofill Service, there is nothing preventing an attacker to *combine* these techniques at her will and adapt given the attack scenario. Finally, we note that these password-stealing attacks are possible only because current password managers implement a vulnerable mapping algorithm: without such vulnerability, no credentials can ever be leaked to non-legitimate apps.

8 A SECURE-BY-DESIGN API

We believe that the attacks presented in this paper are due to design problems of the current mechanisms to support autofill, from a11y, to the more recent Autofill Framework and OpenYOLO. The key design issue is that all these mechanisms use package names as the main abstraction to work with, thus leaving developers of password managers with the daunting task of mapping apps to their associated domain names. Given the number of security issues and misplaced trust assumptions we have identified in leading password managers, we believe third-party developers should not be asked to implement this critical step.

The *getVerifiedDomainNames()* API. We propose a new API that implements a secure-by-design mechanism by using domain names as the only abstraction that password managers need to interact with. Since credentials are created for websites, we argue this is a better abstraction level. In stark difference concerning existing proposals, this API, called *getVerifiedDomainNames()*, would directly provide to password managers a list of domain names that a given app is legitimately associated to. The API internal implementation would then be responsible for performing all the needed security checks. We envision this API to be used following the paradigm of OpenYOLO (as in Figure 2). The main difference is that password managers would directly query for domain names, and not for package names.

Integration and implementation. The request for auto-filling a form follows several steps. First, the client sends an Intent to the password manager to request credentials. Then, the password manager can invoke *getVerifiedDomainNames()*, passing the received Intent as argument. At this point, our API performs a number of steps, whose sequence diagram is depicted in Figure 4. First, it retrieves the sender's package name from the Intent. The package name is used to extract the client's app signing key. Then, *getVerifiedDomainNames()* extracts from the client's manifest file the list of domain names the app claims to have access to (this list should

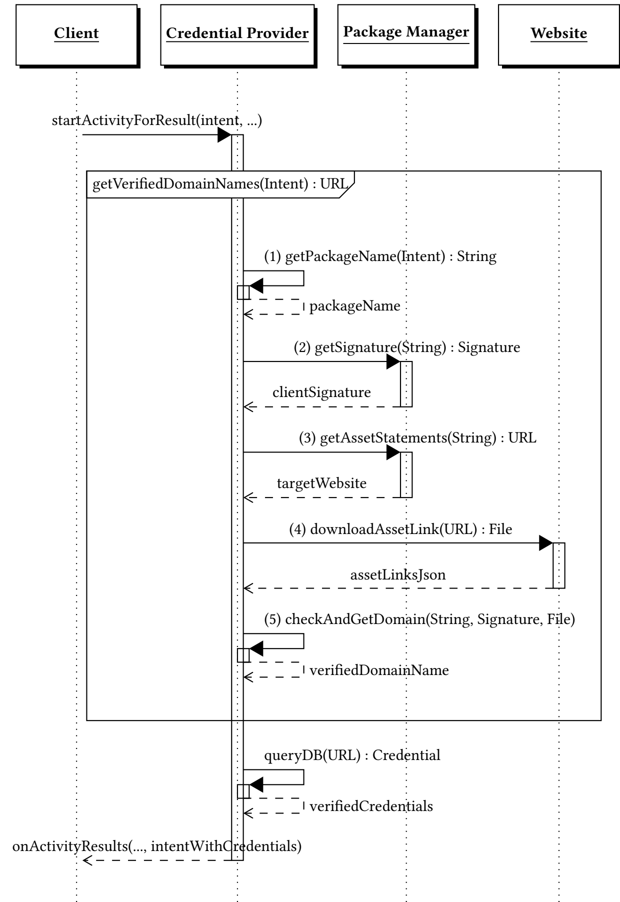


Figure 4: *getVerifiedDomainNames()* API sequence diagram

be specified according to the standard App Link Verification [27] and Digital Asset Links [17] protocols). The API internally downloads, for each of these domain names, the associated DAL file (assetlinks.json) and it verifies that the requesting app (package name + hash of the app signing key) is listed in it. The API includes in its return value to the password manager the list of all domain names that satisfy such security checks. Given these domain names, the PM can then safely query its internal database for associated credentials and send them back to the requesting client.

Avoiding side-channel vulnerabilities. We have noticed that the current OpenYOLO client implementation opens apps to side channel attacks. In particular, the current implementation sends a Broadcast Intent to request credentials from the credential provider, thereby making all other apps aware of such request. A malicious app can use this side-channel to infer that the user is about to login in a specific account: this information can be used for the attacker to know *when* to spawn its spoofed phishing UI [8, 9, 36]. Even if side channels are not required to mount phishing attacks [4], they do make them easier. For this reason, we argue that the communication between the client and the credential provider must remain confidential—not only the content, but even the mere fact that this communication is taking place. To this end, we believe that each

client should have access to a (configurable) list of trusted password managers apps (e.g., Dashlane, LastPass, ...), so that explicit intents can be used instead of broadcast intents. This list could be stored as pairs of package names and hash of signing keys. This is analogous to what browsers do with trusted certificates.

Practicality of adoption. Independently from the API we propose, we were interested in determining how ready the ecosystem is in terms of information required to build a secure app-to-web mapping. Given that the current standard is DAL, we set to analyze the adoption rate by querying a dataset of domain names for their related *assetlinks.json* DAL file. As a dataset, we considered all domain names from all mapping we extracted from the password managers we have inspected. This list is constituted by 8,821 domain names. Note that since they are extracted from password managers, we know that these domain names host at least one page with a login form, thus making them relevant to our analysis.

To our surprise, only 8% (710/8,821) of them host an associated DAL file, and only 2% (178/8,821) specify an Android app in accordance with Google documentation [21]. This low adoption rate is worrisome: password managers would have compatibility problems in securely implementing their solution even if they were fully aware of the problems discussed in this paper. Google Smart Lock has addressed these problems by not relying on a fully automatic technique (developers need to manually fill a Google form) and by supporting app-to-web sync only when a secure mapping exists. We argue that the rest of password managers should follow a similar approach and warn the user about potential problems when a secure app-to-web association cannot be established.

9 RELATED WORK

Phishing is a well-known problem and it has received the attention of the security community for several years. In the realm of mobile devices, there have been a number of works focusing on task hijacking [9, 14, 36], and UI confusion [4, 8]. We built on the insights provided by these works and we have shown how features implemented for convenience can make mobile phishing attacks significantly more practical than what previously thought: we do not assume a malicious app is already running on the victim's device and, for the first time, the user is not even required to type her credentials. Few works also proposed defense mechanisms for mobile phishing [8, 15], which are unfortunately not finding adoption due to the invasive framework modifications they require. Another interesting research direction is the automatic identification of app widgets that contain user's sensitive info [7, 28, 34].

The problem of phishing has also been extensively studied in the browser context [10, 11, 30]. In this context, protection mechanisms are usually implemented in forms of blacklist [25].

Another class of UI-related attacks is tapjacking (also called clickjacking). Some works have shown how an attacker can abuse the overlay system to lure users into unknowingly perform security-sensitive operations [16, 35, 39]. Other works show how accessibility service can be abused to bypass user interaction and perform UI-related attacks [5, 6, 16, 29, 32, 33, 38]. These are very powerful attacks, but they differ from phishing: they are about luring a user to perform a sensitive operation, while phishing focuses on luring them to leak their credentials.

A few recent works have focused on the security analysis of browser password managers [31**?**]. In those works, the authors conduct a security analysis of popular web-based password managers, and some of them were found exploitable, allowing an attacker to leak user credentials. The root-causes of the vulnerabilities were ranging from logic and authorization mistakes to traditional web vulnerabilities like CSRF and XSS. Our work, instead, focuses on *mobile* password managers. We also note that we have not focused on identifying classic implementation bugs, but we aimed at uncovering systemic design issues.

Silver et al. show several attacks aimed at retrieving passwords from in-browser PMs, by exploiting their autofill policies [**?**]; the most powerful attack they uncovered does not require any human intervention and it allows to automatically auto-complete password fields. Several prior works show how combining innocuous visible fields and sensitive invisible fields trigger PMs to autofill, and, consequently, provide sensitive information to the attacker [**? ?**]. This is similar to our experiment with hidden password `EditText` widgets.

For what concern the security of Android password managers, the work by Fahl et al. is one of the few in the area [13]: in this paper, the authors studied 21 popular password managers and show how password managers would somehow push users to "copy" their passwords to their clipboard: this has security implications since the device clipboard can be accessed by any app installed on the user's device. Interestingly, we note that password managers using `a11y` or Autofill Service are not affected by these problems: passwords shared via these "modern" features do not go through the clipboard. However, our paper, unfortunately, shows that even these modern mechanisms are affected by security problems as well.

10 CONCLUSIONS

In this paper, we carried out a security assessment of two recent Android features, originally introduced in the name of convenience. The number of design issues and the variety of vulnerable heuristics that we have identified in leading password managers suggest that the insights offered in this paper are not well-understood by the community. The possibility of abusing Instant apps and hidden fields make these attacks even more problematic and practical. We believe that our proposed API would prevent this class of problems from being introduced and, at the very least, would force password managers developers to critically think about the various challenges. That being said, although a technical solution certainly exists, we believe that password managers developers cannot solve this problem alone, but there is the need of a push from the entire community, which this paper hopes to inspire.

DISCLOSURE AND ACKNOWLEDGMENTS

We have responsibly disclosed our findings to the security teams of the password managers we found vulnerable. We would like to acknowledge their quick and professional handling of the matter. The affected vendors are in the process of deploying countermeasures. We would also like to acknowledge Betty Sebright: despite the passing of time, she and her team are still a significant motivating factor for our research.

REFERENCES

[1] 2018. 1Password. https://1password.com/. (2018).
[2] 2018. Dashlane. https://www.dashlane.com/. (2018).
[3] 2018. LastPass. https://www.lastpass.com/. (2018).
[4] Efthimios Alepis and Constantinos Patsakis. 2017. Trapped by the UI: The Android Case. In *RAID*.
[5] Yair Amit. 2016. 95.4 Percent of All Android Devices Are Susceptible to Accessibility Clickjacking Exploits. https://www.skycure.com/blog/95-4-android-devices-susceptible-accessibility-clickjacking-exploits/. (2016).
[6] Yair Amit. 2016. "Accessibility Clickjacking" — The Next Evolution in Android Malware that Impacts More Than 500 Million Devices. https://www.skycure.com/blog/accessibility-clickjacking/. (2016).
[7] Benjamin Andow, Akhil Acharya, Dengfeng Li, William Enck, Kapil Singh, and Tao Xie. 2017. UiRef: analysis of sensitive user inputs in Android applications. In *WISEC*.
[8] Antonio Bianchi, Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. 2015. What the App is That? Deception and Countermeasures in the Android User Interface. In *Proc. of the IEEE Symposium on Security and Privacy*.
[9] Qi Alfred Chen, Zhiyun Qian, and Z Morley Mao. 2014. Peeking Into Your App Without Actually Seeing It: UI State Inference and Novel Android Attacks. In *Proc. of the USENIX Security Symposium*.
[10] N. Chou, R. Ledesma, Y. Teraguchi, D. Boneh, and J. C. Mitchell. 2004. Client-side defense against web-based identity theft. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS)*.
[11] Rachna Dhamija and J. D. Tygar. 2005. The Battle Against Phishing: Dynamic Security Skins. In *Proceedings of the Symposium On Usable Privacy and Security (SOUPS)*. ACM, New York, NY, USA, 77–88. https://doi.org/10.1145/1073001.1073009
[12] Artyom Dogtiev. 2018. Facebook Revenue and Usage Statistics. http://www.businessofapps.com/data/facebook-statistics/. (2018).
[13] Sascha Fahl, Marian Harbach, Marten Oltrogge, Thomas Muders, and Matthew Smith. 2013. Hey, you, get off of my clipboard. In *International Conference on Financial Cryptography and Data Security*. Springer, 144–161.
[14] Adrienne Porter Felt and David Wagner. 2011. Phishing on Mobile Devices. In *Proc. of IEEE Workshop on Web 2.0 Security & Privacy (W2SP)*.
[15] Earlence Fernandes, Qi Alfred Chen, Justin Paupore, Georg Essl, J Alex Halderman, Z Morley Mao, and Atul Prakash. 2016. Android UI Deception Revisited: Attacks and Defenses. In *Proc. of Financial Cryptography and Data Security (FC)*.
[16] Yanick Fratantonio, Chenxiong Qian, Pak Chung, and Wenke Lee. 2017. Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.
[17] Google. 2017. Digital Asset Links. https://developers.google.com/digital-asset-links/v1/getting-started. (2017).
[18] Google. 2017. OpenYOLO for Android. https://openid.net/specs/openyolo-android-03.html. (2017).
[19] Google. 2018. Accessiblity Service. https://developer.android.com/guide/topics/ui/accessibility/services. (2018).
[20] Google. 2018. Autofill Framework. https://developer.android.com/guide/topics/text/autofill. (2018).
[21] Google. 2018. Enable automatic sign-in across apps and websites. https://developers.google.com/identity/smartlock-passwords/android/associate-apps-and-sites/. (2018).
[22] Google. 2018. Google Smart Lock. https://get.google.com/smartlock/. (2018).
[23] Google. 2018. Google Smart Lock - Associate apps and sites. https://developers.google.com/identity/smartlock-passwords/android/associate-apps-and-sites. (2018).
[24] Google. 2018. Keeper. https://keepersecurity.com/. (2018).
[25] Google. 2018. Safe Browsing. http://www.google.com/transparencyreport/safebrowsing/. (2018).
[26] Google. 2018. Smart Lock for Passwords affiliation form. https://docs.google.com/forms/d/e/1FAIpQLSc3FCn8ccGpgXd1jtLBVRlNJ6EhWQK50hNO5jT_9nuqHI79pg/viewform. (2018).
[27] Google. 2018. Verify Android App Links. https://developer.android.com/training/app-links/verify-site-associations/. (2018).
[28] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. 2015. SUPOR: Precise and Scalable Sensitive User Input Detection for Android Apps. In *USENIX Security Symposium*.
[29] Yeongjin Jang, Chengyu Song, Simon P Chung, Tielei Wang, and Wenke Lee. 2014. A11y Attacks: Exploiting Accessibility in Operating Systems. In *Proc. of the Conference on Computer and Communications Security (CCS)*.
[30] E. Kirda and C. Kruegel. 2005. Protecting users against phishing attacks with AntiPhish. In *Proceedings of the Computer Software and Applications Conference (COMPSAC)*, Vol. 1. 517–524 Vol. 2. https://doi.org/10.1109/COMPSAC.2005.126
[31] Zhiwei Li, Warren He, Devdatta Akhawe, and Dawn Song. 2014. The Emperor's New Password Manager: Security Analysis of Web-based Password Managers.. In *USENIX Security Symposium*. 465–479.
[32] Lookout. 2015. Trojanized adware family abuses accessibility service to install whatever apps it wants. https://blog.lookout.com/blog/2015/11/19/shedun-trojanized-adware/. (2015).
[33] Spandas Lui. 2016. Accessibility Service Helps Malware Bypass Android's Beefed Up Security. http://www.lifehacker.com.au/2016/05/accessibility-service-helps-malware-bypass-androids-beefed-up-security/. (2016).
[34] Yuhong Nan, Min Yang, Zhemin Yang, Shunfan Zhou, Guofei Gu, and Xiaofeng Wang. 2015. UIPicker: User-Input Privacy Identification in Mobile Applications. In *USENIX Security Symposium*.
[35] Marcus Niemietz and Jörg Schwenk. 2012. UI Redressing Attacks on Android devices. *Black Hat Abu Dhabi* (2012).
[36] Chuangang Ren, Yulong Zhang, Hui Xue, Tao Wei, and Peng Liu. 2015. Towards Discovering and Understanding Task Hijacking in Android. In *Proc. of USENIX Security Symposium*.
[37] Statista. 2018. Percentage of all global web pages served to mobile phones from 2009 to 2018. https://www.statista.com/statistics/241462/global-mobile-phone-website-traffic-share/. (2018).
[38] Dinesh Venkatesan. 2016. Malware may abuse AndroidâĂŹs accessibility service to bypass security enhancements. http://www.symantec.com/connect/blogs/malware-may-abuse-android-s-accessibility-service-bypass-security- enhancements. (2016).
[39] Longfei Wu, Benjamin Brandt, Xiaojiang Du, and Bo Ji. 2016. Analysis of clickjacking attacks and an effective defense scheme for Android devices. *2016 IEEE Conference on Communications and Network Security (CNS)* (2016), 55–63.