Activity Spoofing and Its Defense in Android Smartphones

Brett Cooley[1], Haining Wang[1], and Angelos Stavrou[2]

[1] The College of William and Mary, Williamsburg, VA, USA
[2] George Mason University, Fairfax, VA, USA
{brcooley,hnw}@cs.wm.edu, astavrou@gmu.edu

Abstract. Smartphones have become ubiquitous in today's digital world as a mobile platform allowing anytime access to email, social platforms, banking, and shopping. Many providers supply native applications as a method to access their services, allowing users to login directly through a downloadable app. In this paper, we first expose a security vulnerability in the Android framework that allows for third party apps to spoof native app activities, or screens. This can lead to a wide variety of security risks including the capture and silent exfiltration of login credentials and private data. We then compare current defense mechanisms, and introduce the concept of Trusted Activity Chains as a lightweight protection against common spoofing attacks. We develop a proof of concept implementation and evaluate its effectiveness and performance overhead.

1 Introduction

In the recent years, global smartphone sales have grown 73%, while Android phone sales have grown 379% [1]. Due to this massive growth, mobile platforms are being used for a wide variety of services. By accessing these services, end users have become accustomed to authenticating to multiple third parties. This experience can be similar to more traditional models if the services utilize a web-based application. However, more and more services are deploying native applications, which is a departure from the web interfaces more frequently found on a desktop or laptop computing model. Therefore, users are not familiar with the expected results of common tasks like launching an app or navigating between apps when unfamiliar windowing system is in use. These factors lower the barrier of entry for malicious code to fool a user into doing something unintended.

While malware is written to use any and all possible security holes, widespread threats will generally target the most lucrative opportunities with the largest possible user base. Due to the relative unfamiliarity users have for mobile devices and the lack of standardized interface paradigms, taking advantage of this has the potential to affect the vast majority of smartphone users. It also has the potential to be incredibly lucrative, as the possibilities for surreptitious logging of highly personal data like bank account credentials or credit card numbers are both numerous and take little effort. In a compounding effect, there are few to no technical challenges needed to deceive users.

I. Boureanu, P. Owesarski, and S. Vaudenay (Eds.): ACNS 2014, LNCS 8479, pp. 494–512, 2014.

Traditionally, phishing or web spoofing are targeted attacks designed to trick users into revealing sensitive information or credentials to a third party, often without the user's knowledge that something bad has happened. Most phishing attempts do not need to exploit security vulnerabilities in websites, but rely on social engineering, look-alike links, and persuasive or seemingly urgent emails to lure users to release their private information [2]. Many studies have focused on the prevention of phishing, both via automated detection of malicious websites [3–5] and training and informing users [2]. However, all of these studies and tools are focused on web-based phishing, and many of the methods involved are unsuitable for a mobile environment. They are either too specific to be useful in the context of native applications or rely on services which would be too resource intensive and non-trivial to port to mobile devices. All of these roadblocks are due to the difference in the interaction model used by native mobile apps. To fully understand phishing in the context of a native app, we must look at the framework from which native apps are built.

Android is designed with a strong inter-process communication framework, enabling third party apps to integrate both with Android services and other apps. One of the major design goals was to enable component reuse among apps. For example, Android devices with a camera ship with an associated app, allowing third parties to treat the camera as a service, from which they can make simple requests and get back a picture in a variety of forms. At the same time, the user is presented with a familiar camera interface, then returned to the app they were originally interacting with. The Android framework was designed to make this use-case simple to achieve for developers, and interactions with the myriad of hardware devices on a smartphone predictable and easy to manage for users. These goals are made possible by two major portions of the framework, Activities and Intents.

In this paper, we explore how the ability to reuse activities changes user expectations, thus allowing activities to be spoofed by almost any third party app. We show how this can lead to exfiltration of account credentials among other breaches of privacy. We further explore how such an attack can be embedded in a legitimate application, and a proof-of-concept attack is demonstrated. We then discuss different defenses, including simple solutions and more sophisticated approaches already developed. We propose a framework for Trusted Activity Chains, which provide a lightweight, transparent system with a variable level of protection and user interaction. We evaluate the effectiveness of our defense along with its performance overhead. In addition, we discuss how our adaptation can protect against a larger surface of attacks than just credential-focused phishing attacks, and possible areas for future work and analysis.

The remainder of the paper is structured as follows. Section 2 describes the background information on Android activity. Section 3 presents the vulnerability of activity spoofing. Section 4 details different defense mechanisms and our proposed approach. Section 5 evaluates the effectiveness of our solution and its performance overhead. Section 6 surveys related work. Finally, Section 7 concludes the paper with the discussion of future work.

2 Background

Activities can be considered an abstraction for all of the code required to both display and handle events on a single screen. Activities can also serve as the entry point to an app, when properly designated as such. This allows any activity to be reused by an app other than its own as long as the third party app knows the correct message, or intent, to send that activity. The associated framework that enables the aforementioned messages is a custom facility, which Android calls intents. Intents can be constructed in any activity, and are used to either explicitly bring another activity into view or to implicitly request a certain type of activity to handle the attached data. When an activity is requested for use by an intent, it is brought to the foreground and displayed to a user for interaction. Once the user is finished with an activity or the activity in turn sends an intent to yet another activity, the current activity finishes and is stopped. This returns the user to the previous foreground activity or the newly started one. This model is designed around a stack of activities [6], and allows for a fluid exchange of data between apps, and a seamless experience for the user.

Android does provide some methods for limiting the scope of an activity, but they alone do not provide adequate protection for developers or users. The two major methods of protection are permission preconditions and private scoping of activities. Optionally, activities can require apps which send them intents to possess certain permissions, such as the ability to access a user's location. This can be used to ensure correct functionality of an activity, or ensure that some data collected by an app with certain permissions is only passed to other apps with the same or similar permissions. This feature, while helpful, is widely unused [7], and users cannot rely upon apps to safeguard their information to any greater degree than the Android system requires. Even with the use of these permission checks, there are well-known permission escalation attacks [8]. Another privatization method for activities is requiring a custom "key", thereby ensuring that any invoking app knows the exact string required to launch that particular activity. More frequently, they can receive standard system messages, allowing any app which asks for a particular kind of service to possibly be routed to the activity in question. The final, and default option is for an activity to require explicit invocation by name. However, as we will discuss below, this is actually the least effective method of protecting an activity from third party access.

Android does not place any restrictions on what types of apps are allowed to request the execution of activities. By default, without flagging an activity for a certain type of action, activities can only be launched outside of their respective apps via invoking them by their full name including the package in which they reside. However, easy to use tools [9] exist which enable finding the package and full class names of all activities in any published app. This effectively makes securing components the job of the developer, which has been shown to be problematic [7,10,11]. Due to these limitations, a variety of patterns of misuse are available to app developers. We will focus on a particularly dangerous pattern, called *activity spoofing*.

Fig. 1. Facebook's login activity

3 Activity Spoofing

Due to the relative lack or inadequacy of protections for activities, one cannot be assured of which activities belong to which apps. When one app displays an activity which is visually similar or identical to an activity from another app, users will become confused as to which application they are interacting with. This is called activity spoofing. We give an overview of how activity spoofing works, then discuss the properties and magnitude of the attack surface, and finally look into a particular attack vector.

3.1 Overview of Threat Model

Activities consist of many different kinds of information and interactions. We are interested primarily in authentication screens, or other activities that require a user to enter some kind of credentials or private data. These kinds of screens are commonly found in many types of apps, including popular social network apps like Facebook (Figure 1) and Twitter (Figure 2). Activity spoofing normally occurs when a user launches an app, which attempts to display an activity (the *intended activity*), but another activity is immediately launched (the *spoofed activity*) afterwards, which mimics the intended activity in appearance but is not the activity the user is expecting. Activity spoofing which does not have a temporal context can also be executed by an app advertising some kind of interoperability, like the ability to tweet about something from within the app. When the user goes to exercise this functionality, the app is free to launch an activity which mirrors the official app's appearance and feel.

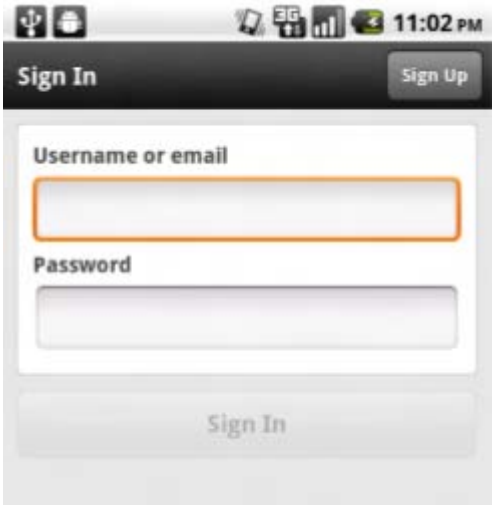

Fig. 2. Twitter's generic login activity, which could further encourage spoofing

When users are presented with the spoofed activity, they assume that it is in fact the intended activity; as it looks identical to the intended activity and is exactly what they expect by launching the intended app. This is primarily because Android operates with only one activity visible at a time. This deprives users of any possible knowledge that there are two activities which mirror each other in appearance. Another enabling factor is that Android does not force activities to actively display what app they belong to. Therefore, users have no way of knowing that the spoofed activity does not belong to the app it appears to originate from. When the user interacts with the spoofed activity, it has full control over what is done with any information the user provides.

If users do enter whatever credentials are required to authenticate to the service which the spoofed activity is masquerading as, such as a login ID or email address and a password or PIN, they are handing the spoofed activity full access to that service. The spoofed activity can capture these values, and is free to exfiltrate them as it pleases. Moreover, the likelihood of these credentials being correct is much higher than more traditional methods of credential collection. This is because users are trying to actively use the service being portrayed, ensuring they are making a good-faith effort to provide current login and password data. Since most apps send credentials to a server to be authenticated, active network I/O is very common after submitting credentials to a login activity. This creates a perfect opportunity to send the data collected by the spoofed activity to an external server associated with the spoofed activity's app. Combining legitimate network traffic with data exfiltration makes detection of the private data leakage more difficult.

To keep users unaware of the data leakage, the spoofed activity can act in a variety of ways after it has captured the entered credentials. It can display an

error message to the users, and exit silently. This has the effect of redirecting the users to the intended activity, which allows them to re-enter their credentials without interference. Another option is to pass the data on to the intended activity and initiate the submission of the data, bypassing the intended activity and displaying the next activity in app. Note that either of the two approaches could raise some suspicions.

By displaying an error message to users, the spoofed activity is returning control to the intended activity's app as quickly as possible, thereby minimizing the chances for detection due to lag in the user-interface or other abnormalities. However, over time, the prevalence of these errors could arouse suspicion. Also, this requires the spoofed activity to closely mimic the intended activity's error dialog. This adds complexity to the spoofing code, which increases the code's footprint. To combat these issues, the app containing the spoofed activity could only launch the spoofed activity a fraction of the times the user attempts to interact with the intended activity. This allows for a more realistic error rate, which users will be more willing to tolerate. If the spoofed activity passes the data on, the expected workflow is uninterrupted. This decreases the likelihood of user suspicion, as the sequence of activities displayed to the user is unchanged by the spoofed activity. However, Android must update the display when the spoofed activity exits, and the ability to programmatically bypass an activity is dependent on the authentication model employed. This delay could potentially draw attention from a trained user. We next describe what common types of interactions are vulnerable to activity spoofing.

3.2 Properties of Vulnerable Apps

Despite Android's unhindered access model, not every activity can be easily spoofed. Activities which display real-time data, or frequently updated streams which aren't publicly available can present real challenges to spoofing. Also, activities which make API-specific request from services over the network typically employ the use of an API key, which would require a spoofed activity to also obtain an API key. While this is significantly more difficult than spoofing a basic activity, there are common weaknesses in this approach which could allow for bypassing the need for an API key [12]. Nevertheless, we will focus on those activities that do not make use of private API's. This covers the large majority of activities, most of which are vulnerable to spoofing, but are also relatively harmless. The harmful activities require user interaction, typically asking users to enter some sort of login credentials or private information. As discussed before, these kinds of activities are prime for spoofing. Depending on the exact methods used for mimicking an application, the spoofed activity could look anywhere from close to an exact replica of the intended activity. In some cases, apps provide very generic login activities which can lead to confusion as to what app is requesting the user's credentials, even without the introduction of spoofing attacks. While many kinds of activities fit these criteria, we now present social networking apps as a particular attack vector.

3.3 Spoofing Social Network Logins

Social networking has been exponentially increasing in the past few years, with the rise in popularity largely due to the sites like Myspace, Facebook, and Twitter. With a website like Facebook having well over 1.2 Billion active users [13], social network users account for one of the largest user-bases available. Combined with the fact that both Facebook and Twitter apps come preinstalled on many Android smartphones, the attack surface available to spoof a social networking application is very large. However, users do not hold access to their social networking accounts as important as access to more important accounts, like bank accounts and online shopping sites. This leads to weaker passwords, and password reuse [14] which furthers the expected value of intercepting this data. As an added benefit, some social networks use the user's email address as the account identifier. By capturing this data along with a valid password, a malicious app would have the ability to access some percentage of users email accounts, due to password reuse as mentioned previously. Many online services allow someone with access to the email address associated with an account to reset the password, thereby gaining access without knowing the chosen password *a priori*. All of these factors combine to make users' social network credentials highly desirable targets. To demonstrate the ease of these kinds of attacks, a proof-of-concept app was developed, and is discussed below.

3.4 Spoofed Activity Attack

In order to demonstrate the ease and effectiveness of spoofing popular applications, we have created a proof-of-concept attack on the Facebook app for Android. A true attacker would implement the activity spoofing code within another seemingly benign app; however, for our purposes, we have simply implemented a stand-alone app. Regardless of how the code is loaded, the app will generally follow a *monitor-capture-exfiltrate* cycle.

Monitoring. In order to spoof an activity, the attacking app must know when the intended activity is being launched. This is trivial if the user is already using the attacking app; but if they aren't, Android provides the ability for apps to launch background tasks called *services*. Our app uses a service which is detached from the app itself, allowing for it to continue to run in the background even after the app's activities are stopped. This service monitors Android's `ActivityManager`, which allows it to launch the spoofed activity as soon as any app requests for the intended activity to be launched.

Capture. Once the intended activity has been launched, our monitor launches the spoofed activity. This ensures it appears after, and therefore, above the intended activity. By utilizing Apktool [9], we have simply copied the relevant design assets and XML files which describe the look and feel of Facebook's app into our own, making the intended and spoofed activities appear identical (Fig. 1 is taken from our proof-of-concept). The users have no possible way of knowing that they are interacting with the spoofed activity, instead of the

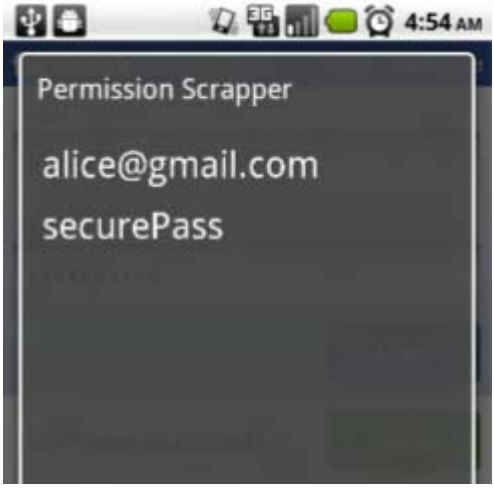

Fig. 3. User's credentials captured by a spoofed activity

intended activity. Once the users attempt to authenticate, the spoofing app will intercept and capture their credentials, unbeknown to them.

Exfiltrate. While our proof of concept simply displays the captured values back to the user (Figure 3), a real application could exfiltrate this data, possibly over the network to a remote server. This presents the most challenging part of the spoofing attack, as tools like TaintDroid [15] exist to track this kind of information flow. However, none of the currently available flow analysis tools would flag this particular information leak, as the user is providing the information willingly, and from their memory or other medium outside the control of the phone. Therefore, any systems-based approach [16–18] wouldn't be able to differentiate this kind of data from valid data captured by the intended app itself. Significantly more intrusive and resource intensive methods would have to be employed for one of these systems to be able to differentiate between spoofed and valid authentication attempts at the activity level.

4 Defenses

While activity spoofing is hard to prevent and easy to implement, there exist a wide range of possible defenses. Any defensive measure must prevent activity spoofing from occurring while allowing benign apps to continue to reuse third party components. In addition, if the defense is to be practical, it must have a very low level of false positives (benevolent apps which are flagged as malicious) and have minimal impact on the user. False negatives (spoofed activities which are not caught) are acceptable as some protection is better than no protection; but false positives will lead to users bypassing any protections offered in order

to simply achieve their desired functionality. Correctly identifying spoofing attempts is difficult because of the plethora of ways to emulate another activities functionality. Below we discuss two fundamental types of preventative measures, including analysis of a current solution [19]. We then offer a refinement on one of our proposed models, which offers a variable level of required user interaction, and protects against a wide range of attacks.

One strategy to defend against phishing is to deploy other apps which monitor or otherwise protect against phishing attempts. This approach has a low barrier to entry, and allows for widespread distribution even to older Android devices which may not receive official system updates. However, app-based solutions will have a limited ability to actively protect the user from other apps, and while monitoring and alerting users to potential threats is worthwhile, a better defense mechanism might be sought when valuable personal information is at risk.

4.1 Secure Phrases

On the Internet, websites have long dealt with phishing and spoofing attacks. One of the most popular methods of combating these attacks is the use of secure phrases and images [21–23]. These images or phrases are set up after successfully authenticating for the first time, and are displayed on the authentication page the subsequent times when users visit the site. This approach requires app developers to modify the apps' implementation. However, most major providers already have the infrastructure in place to offer this service via their web apps. All that would be required is a translation of the feature to work on an Android device. However, since this approach requires minimal work at the Android device, it is also the least effective. While rendering the simple implementation discussed previously ineffective, adding a secure phrase wouldn't prevent more complex spoofing attacks. For example, the spoofing activity could simply implement a transparent frame. If placed in the correct spot on screen, this transparency would give the appearance of displaying the secure phrase or image as if it was being displayed by the spoofing activity itself.[1] On the other hand, this approach introduces zero false positive, and does not restrict activity reuse in any way.

4.2 Spoof Killer

The main idea behind Spoof Killer [19] is to introduce a system interrupt into the login procedure, during which the system verifies that the current context, or the application from which the login request was generated, matches some sort of internal whitelist or certificate list. If the context is safe, the login proceeds, and is otherwise aborted. This system interrupt takes the form of requesting the user activate a pre-defined interrupt key, like the home key on Android phones. This requires users to correctly execute an additional step every time they login

[1] As of Android 4.0, there are built-in measures to guard against such attacks. But by 2014, the top five banking and social network apps have not yet implemented any sort of secondary security measures.

to any app. From a usability perspective this may not be advantageous, as it introduces another step during which a user could fail the login procedure. This may lead users to find the feature cumbersome, and disable it or find an alternate (and thereby more insecure) method of authentication. However, as was outlined in the cited page, when correctly used this feature builds a behavior into users that provides inertia against attacks that try and bypass this mechanism.

From a technical standpoint, requiring a global interrupt for any app authentication must be handled by the app developers, as there is no way for the system to detect every application's version of authentication unless they are using some standardized authentication feature. This could hamper usability as not all applications will implement the system interrupt feature, leading users to either distrust valid applications, or to not form the habit of initiating the interrupt, which weakens the protection provided. Also, by relying on a whitelist, Spoof Killer implicitly places the burden of ensuring an up-to-date whitelist on the end user's system, which may or may not be feasible for all platforms, specifically enterprise infrastructures which largely follow out-of-band update cycles. These concerns lead us to an alternate paradigm for less intrusive protection that requires no prior knowledge of trust.

5 Trusted Activity Chains

We now propose a system-level defense which provides a new framework feature to app developers, called Trusted Activity Chains. We will give an overview of the approach, discuss the components of our framework, potential issues which arise from this model, and the impacts it has on both the usability and security of an Android device.

5.1 Overview

Our framework introduces the concept of sequences of activities which should not be interrupted. App developers should be able to simply annotate a chain of activities with a request that they not be interrupted, and the system should handle the rest. In keeping with the design ideas of the Android framework, at any given time only one activity is displayed in the foreground. The app which owns this activity can therefore be considered the "foreground app". When an activity is brought into the foreground, it requests a lock, signaling that it should not be interrupted. Once this lock is granted by the operating system, the system will begin to monitor any attempts by other apps, background services, or other process to launch a new activity. Depending on the configuration, the system will handle these requests differently than it normally would (e.g. let the activity launch). Once the foreground activity is finished, the next activity in the uninterrupted chain will be launched, or the lock will be given up and the system will revert to standard behavior, allowing any activity to be launched.

5.2 Components

There are a number of interactions that Trusted Activity Chains give rise to. In order for the concept to work, we need to have processes to deal with how an activity acquires the interrupt lock, how it releases the same lock, how the OS manages the lock, and what happens when there is an attempted interruption while the lock is being held. Figure 4 gives a state diagram for the possible states of the lock, which will be referenced below.

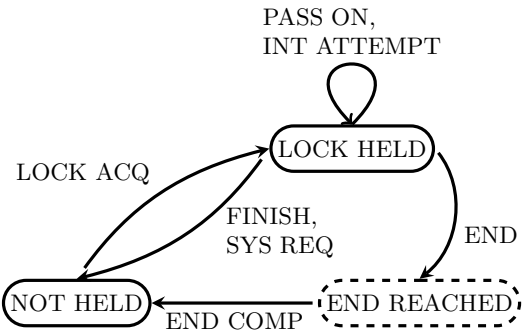

Fig. 4. State Diagram for Lock Life Cycle

Acquisition and Release. To acquire the interrupt lock, an activity simply needs to be marked as requesting the lock. When the `ActivityManager` launches such an activity, it will also check if the lock is free (`NOT HELD`). If so, it assigns the lock to the launching activity (`LOCK ACQ`), and allows it to launch without further interference. The activity now holds the lock (`LOCK HELD`), and may do any number of things with it. If the activity (such as a login screen) wants to assure that the next activity the user is shown upon successful authentication in the app's home screen, then the home screen can be tagged as the end point for the lock (`END`). This allows for the application to ensure that no other activities will be launched before its home screen is displayed (`END REACHED, END COMP`), which protects against post-authentication spoofing, such as a malicious activity launching a cloned version of the intended activity's error dialog for a failed authentication, and asking the user to try again. If the chain of activities which request the lock does not end at the app's home screen, the lock is simply transferred to the next activity (`PASS ON`), and the cycle continues.

Alternatively, the lock-holding activity may simply release the lock when it is paused or removed from the foreground (`FINISH`). This can be caused either by design when the activity is the last one the app wishes to display for the moment; or by the system if it needs to execute an action such as locking the screen or displaying information on an incoming call (`SYS REQ`). Since these activities can only be initiated by the system or the user via hardware buttons, they are safely allowed to override the lock.

Management. The Android framework already handles all activity lifetime services through a central class, the `ActivityManager` [20]. Using this class to simply keep track of which activity, if any, currently has the lock allows for all of the features discussed. The OS must simply pass control back through the `ActivityManager` any time it requests control from activities, so that the lock can correctly be released in these scenarios. This both encapsulates all modifications, and keeps the overhead imposed to a minimum.

Interrupt Handling. While the lock is held, if a third party app attempts to launch an activity, the `ActivityManager` intercepts this request, and can handle it depending upon the configuration of the system. Some options include alerting the user that a third party activity is trying to launch, and notify the user that the third party activity will be launched at the completion of the current app's sequence (`INT ATTEMPT`). While effective, this may be too heavy-handed, and so the OS could allow the third party activity to be launched, but require it identify itself as a third party. The Android framework already includes a mechanism for this by way of title bars. Title bars are an UI element that may optionally be turned off, depending on the theme selected by the app developer. This allows for a visual cue that the current activity is different from the activity that was just in focus.

5.3 Potential Issues

Trusted Activity Chains allow for a very robust defense against many kinds of spoofing and phishing attacks. However, there are some potential drawbacks which could limit the effectiveness of this mechanism. As with any system which employs a locking mechanism, we must handle the possibility of race conditions and deadlocks. Because Trusted Activity Chains require developers to adapt their apps to the framework, we also must address intentional and unintentional misuse by developers, along with issues of adoption.

Race Conditions and Deadlocks. Race conditions could occur when two or more activities are launched, each of which requests the interrupt lock. Due to the nature of Android, the order of activity launches is not precise, but is atomic. That is, one of the activities that is attempting to launch will be first, and will be launched before any of the other activities are picked by the system to begin the launch process. This ensures that one activity will always be able to fully acquire the lock before another activity is allowed to check the status of the lock. Therefore, while the activity receiving the lock may not always be the one expected to receive it, there is never a situation in which two activities simultaneously will have the lock.

Similarly, deadlocks are never an issue, as even in the case where an activity never manually releases the lock, any actionable item the system has to deal with automatically releases the lock. This includes the user pressing the hardware buttons, allowing the user control even in the event of a malfunction.

Developer Misuse. In order for Trusted Activity Chains to protect an activity, the developers of that activity must flag that it requests the lock. This kind of control can of course be misused by developers unfamiliar with the system, as well as opening up a possible attack vector. Fortunately, Trusted Activity Chains are resilient against all forms of misuse. If a developer were to tag an activity as requesting the lock unintentionally, the app will still function, but will simply prevent third party activities from launching. Even if the mistakenly tagged activity requires data from a third party activity to function, because the launch request for the third party activity originates from the activity with the lock, it will be allowed to launch and run as normal.

While inadvertent misuse is a honest scenario to consider when designing a framework, malicious misuse generally is more potentially damaging. However, because our framework is overridden when the system has tasks to handle, a malicious developer is not granted any more power than they already have within the larger Android framework.

Adoption. There are innumerable proposed extensions to Android for the purpose of enhanced security [15–19], all of which have varying levels of security, usability, and overhead. In addition to the vast number of users who are either indifferent or unaware of the security concerns with Android, this plethora of options makes adoption of any one strategy difficult. Furthermore, Trusted Activity Chains require developers to write apps with them in mind in order to be of use.

While these obstacles do exist, Trusted Activity Chains are still useful even without a widespread adoption. We allow for apps using our modifications to interact seamlessly with other apps and the system as a whole. This incremental approach allows for developers to slowly integrate Trusted Activity Chains into their apps at their own pace, which slowly builds momentum. It also relieves the pressure to patch all old apps immediately, as they too can be slowly phased in as time allows.

Trusted Activity Chains can be easily added to old and new applications with very few lines of code. Since the locking and unlocking mechanism does not impair application functionality even when misused, developers can integrate Trusted Activity Chains with confidence that they are not introducing bugs. We also note that by adding support, developers are taking an action which protects their users, which can generate positive press for the developer's apps. It can also call attention to other apps which fail to use this feature. This form of peer pressure can further drive adoption.

5.4 Implications

While providing a high level of protection, our framework could hurt usability. Therefore, we offer a less defensive, but more transparent method of functionality, wherein third party activities which launch during other app's activity chains are required to identify themselves to the user. This would alert the user to the presence of a activity which did not belong to the foreground app. While this

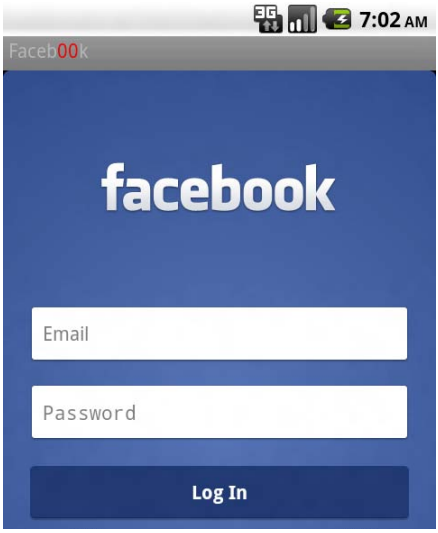

Fig. 5. The spoofed activity with Trusted Activity Chains enabled

does not guarantee that a user won't fall for a spoofed activity, it does allow easy identification of the intended activity in contrast with the spoofed activity. This defense has the added benefit of giving apps more control over how they interact with other portions of the Android system.

We note that this approach, like that of [19] requires developers to actively implement new features in the apps for the defense to function. However, apps which lack our approach do not function incorrectly in its absence, but are merely less secure. This allows for our approach to be implemented without prior widespread adoption as discussed above. Also, our approach is both flexible and extendable, and can be used for non-security related problems as well as protecting against many kinds of malicious behaviors.

6 Evaluation

To validate the efficacy of our proposed defense mechanism, we developed a prototype of Trusted Activity Chains. Then, we evaluated its effectiveness and usability via a case study, as well as performance impact on the Android smartphone. It is important that the proposed Trusted Activity Chains can be easily used by normal users, while its performance overhead is minor for using them. Our experiments were conducted using a Motorola device with a 550MHz ARM A8 processor and 256MB of RAM, running Android 2.2.3. For the purposes of evaluation, our defense mechanism was configured to simply block third-party app execution while a Trusted Activity Chain is underway. This allows for the most accurate performance measurements to be collected.

6.1 Effectiveness and Usability

Obviously, in the mode where third party launches are forbidden during a Trusted Activity Chain, a user will be simply unaware of the attempt assuming that the app developer has correctly annotated their app. So, we have only validated the effectiveness and usability when the mechanism is configured to allow third party apps to launch, but force them to show their title bars. We conducted a case study with a small group of users (<10) who do not use an Android device on a regular basis. We explained the purpose of the study, and told them to use the test device to log on to Facebook, take a photo, and share it. We did not give any information on how Trusted Activity Chains work, nor that there was any modification to the Android device. The Facebook app leverages the stock Android camera app to take photos, which means that it will allow third party code to run when the users attempt to take a picture. We have modified the Facebook app so that this action will occur during a Trusted Activity Chain to simulate developer misuse. We have also left our attack application running, which means the users will be confronted with the spoofed Facebook login activity before getting to the real activity. However, as per our configuration, the spoofed activity will be forced to display a title bar, which will display the app name to which the activity belongs.

During the initial launch, the users correctly identified that the spoofed activity was not the true Facebook login screen, but could not pinpoint the reason beyond that it looked different than what they were used to. This shows that

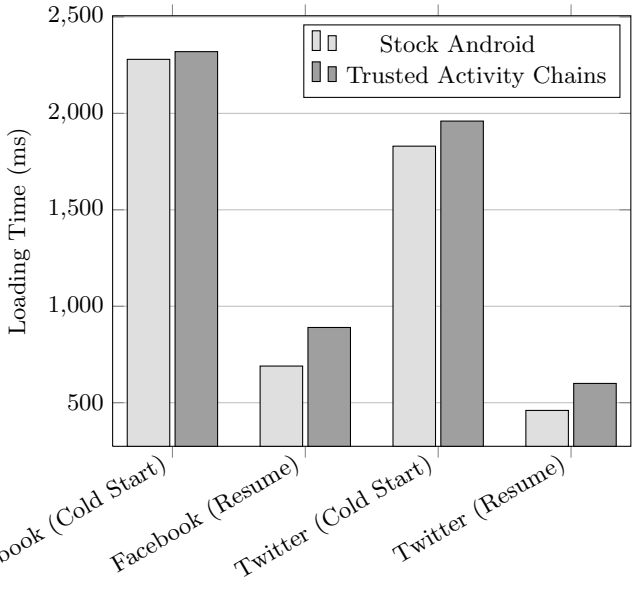

Fig. 6. Average load time for social networking app

even if a malicious application chose a name very similar to the one of the app it was attempting to spoof, users may still be able to discern the difference under our framework. After identifying the fraud, the users were unsure what to do to proceed, but were able to complete a valid login. However, when they attempted to take a picture, they again noticed that the camera app seemed different, and decided to cancel taking the picture. While this outcome is not a failure of our framework but rather due to developer misuse, our future work will consider how to resolve this type of false-positives.

6.2 Performance

In order to measure the performance overhead associated with Trusted Activity Chains, we took measurements of the total time elapsed to launch two social networking apps with and without the defense mechanism enabled. We tested both a cold start of the app along with resuming the app after five minutes of web browsing. We installed our attack application, which monitors for the launch of either app, and attempts to launch its own spoofed version of the login screens. These scenarios were repeated 10 times each, and we took the average time after dropping the highest and lowest times for each.

As we can see in Figure 6, the overhead during a cold start is barely recognizable. This is due to the initialization which must be undertaken for an app to be started for the first time taking much more time than our monitoring. However, on a resume we do see an extra delay. Note that the overall times it takes to resume the app are already rather high at 690ms and 460ms, respectively, under normal conditions. Thus, even though we introduce an extra delay of $\sim$170ms, a user will likely not notice it because the normal transition delays are more than two to three times larger than this additional delay. Furthermore, users expect there to be some delay when moving between apps, as Android's UI shows transition effects when performing a context switch. From these observations, we can conclude that Trusted Activity Chains does not impair normal functionality of an Android device, and can be included without noticeable performance degradation.

7 Related Work

Previous work has identified flaws with both the Android IPC model [7] and the permission model [11]. This work identifies similar possible attack models, including Intent spoofing and activities external to the launcher's app returning information to the launcher. We take this a step further by silently monitoring for valid launches of activities and inject spoofed versions of these activities. A similar attack was presented in [24], however no publications have resulted.

Felt and Wagner [25] detailed numerous categories of phishing attacks levied at mobile devices, including spoofing an activities login screen. This work builds on their result to show an embedded approach, as well as revealing some of the technical features which enable this class of attacks. Further, we provide a robust defense mechanism which could protect against many of their listed attacks.

Russello *et al.* [17] built on TaintDroid [15] to provide information flow tracking as well as a fine-grained label system which allows users to allow or deny access to information on an per-app basis. Their work requires no action on the developer's behalf, but does require users to specify what data should be labeled. Users are generally less aware of application and data security concerns than developers, and this strategy might be too technical for many users.

Quire [18] offers call chain tracking to prevent confused deputy attacks [26] along with a signature scheme which allows intents to be signed and verified. This approach is limited to protecting well-meaning services from being duped by other less benevolent apps, but does not stop information leakage like what is possible via activity spoofing. More recently, ScreenPass [27] secures user passwords on touchscreen by providing a trusted password-entry user interface and defends against spoofing via optical character recognition.

Like Spoof Killer [19], we demonstrate a defense against spoofing attacks. However, our modification requires no whitelists, and protects users from a broad class of attacks as opposed to only protecting against login phishing attacks. Similar to the cited method, our approach does require developer involvement.

8 Conclusion

In this paper, we have presented a simple but dangerous exploit of the Android framework that allows arbitrary applications to spoof sensitive activities of other applications in order to collect private data without users' knowledge. This can be accomplished because of an inability to identify what app a given activity belongs to. We have developed a simple method for constructing spoof activities and integrating them into stand-alone or existing code. The danger of this exploit lies in the simplicity of engineering a spoofed activity for any service that provides a native app for Android, and the ease of collecting the harvested data from user devices. We have then described possible defenses to this class of attacks, considering some existing methods before introducing Trusted Activity Chains. We have demonstrated that Trusted Activity Chains are robust and provide an easy way to optimize for security and usability. We have also shown that they have minor impact on performance. We have further discussed the details of how Trusted Activity Chains support spoofing prevention without the pitfalls generally associated with locking mechanisms.

Our future work includes expanding the proposed defense mechanism, along with a large scale usability study to further demonstrate both the severity of the exploit and the effectiveness of our defense framework. The requirement of developer action to secure users' devices is less than ideal, and we would like to develop an automated method to detect which activities need to be protected and which do not. We also would like to implement a full version of our defense, and integrate it into the Android open source project.

References

[1] Canalys. Press Release 2011/081. Android takes almost 50% share of worldwide smartphone market (August 1, 2011), http://www.canalys.com/newsroom/android-takes-almost-50-share-worldwide-smart-phone-market

[2] Sheng, S., Magnien, B., Kumaraguru, P., Acquisti, A., Cranor, L.F., Hong, J., Nunge, E.: Anti-Phishing Phil: The Design and Evaluation of a Game That Teaches People Not to Fall for Phish. In: Symposium On Usable Privacy and Security, pp. 88–99 (2007)

[3] Abu-Nimeh, S., Nappa, D., Wang, X., Nair, S.: A Comparison of Machine Learning Techniques for Phishing Detection. In: APWG eCrime Researchers Summit, Pittsburgh, PA, pp. 60–69 (2007)

[4] Bian, K., Park, J., Hsiao, M.S., Bélanger, F., Hiller, J.: Evaluation of Online Resources in Assisting Phishing Detection. In: 9th IEEE International Symposium on Applications and the Internet, Bellevue, WA, pp. 30–36 (2009)

[5] Xiang, G., Hong, J., Rose, C.P., Cranor, L.F.: CANTINA+: A Feature-rich Machine Learning Framework for Detecting Phishing Web Sites. ACM Trans. on Inf. and Syst. Security 14(21) (2011)

[6] Android Dev Guide. Tasks and Back Stack (August 28, 2011), http://developer.android.com/guide/topics/fundamentals/tasks-and-back-stack.html

[7] Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing Inter-Application Communication in Android. In: ACM MobiSys, Washington, D.C., pp. 239–252 (2011)

[8] Davi, L., Dmitrienko, A., Sadeghi, A.-R., Winandy, M.: Privilege escalation attacks on android. In: Burmester, M., Tsudik, G., Magliveras, S., Ilić, I. (eds.) ISC 2010. LNCS, vol. 6531, pp. 346–360. Springer, Heidelberg (2011)

[9] Brut.alll. Apktool (May 15, 2011), http://code.google.com/p/android-apktool

[10] Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A Study of Android Application Security. In: USENIX Security, San Francisco, CA (2011)

[11] Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D., et al.: Android Permissions Demystified. In: ACM Conference on Computer and Communication Security, Chicago, IL, pp. 627–638 (2011)

[12] Farrel, S.: API Keys to the Kingdom. IEEE Internet Computing 13(5), 91–96 (2009)

[13] Facebook. Facebook Fact Sheet (March 31, 2012), http://newsroom.fb.com/content/default.aspx?NewsAreaId=22

[14] Ives, B., Walsh, K.R., Schneider, H.: The Domino Effect of Password Reuse. C. ACM 47(4), 75–78 (2004)

[15] Enck, W., Gilbert, P., Chun, B., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: An Information-Flow Tracking System for Real-time Privacy Monitoring on Smartphones. In: USENIX Operation Systems Design and Implementation, Vancouver, B.C (2010)

[16] Beresford, A.R., Rice, A., Skehin, N., Sohan, R.: MockDroid: Trading privacy for application functionality on smartphones. In: 12th Workshop on Mobile Computing Systems and Applications, Phoenix, AZ, pp. 49–54 (2011)

[17] Russello, G., Crispo, B., Fernandes, E., Zhuniarovich, Y.: YAASE: Yet Another Android Security Extension. In: 3rd Conference on Privacy, Security, Risk, and Trust (PASSAT), Boston, MA, pp. 1033–1040 (2011)

[18] Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., Wallach, D.S.: Quire: Lightweight Provenance for Smartphone Operating Systems. In: USENIX Security, San Francisco, CA (2011)
[19] Jakobsson, M., Leddy, W.: Spoof Killer (May 21, 2011), http://www.spoofkiller.com
[20] Android API Reference. ActivityManager (Mar 13, 2012), http://developer.android.com/reference/android/app/ActivityManager.html
[21] Dhamija, R., Tygar, J.: The battle against phishing: Dynamic security skins. In: Proceedings of the Symposium on Usable Privacy and Security (SOUPS), pp. 77–88. ACM (2005)
[22] Whalen, T., Inkpen, K.M.: Gathering evidence: use of visual security cues in web browsers. In: Proceedings of 2005 Graphics Interface (GI), pp. 137–144. Canadian Human-Computer Communications Society (2005)
[23] Schecter, S., Dhamija, R., Ozment, A., Fischer, I.: The emperor's new security indicators: An evaluation of website authentication and the effect of role playing on usability studies. In: Proceedings of IEEE Symposium on Security and Privacy (S&P), pp. 51–65 (2007)
[24] Hassell, R.: Hacking Androids for Profit (August 31, 2011), http://conference.hitb.org/hitbsecconf2011kul/?page_id=1740
[25] Felt, A.P., Wagner, D.: Phishing on Mobile Devices. In: Web 2.0 Security and Privacy, Oakland, CA (2011)
[26] Hardy, N.: The Confused Deputy. ACM Operating Systems Review 22(4), 36–38 (1988)
[27] Liu, D., Cuervo, E., Pistol, V., Scudellari, R., Cox, L.: ScreenPass: Secure Password Entry on Touchscreen Devices. In: Proceedings of ACM MobiSys 2013, Taipei, Taiwan (June 2013)