A Trusted UI for the Mobile Web

Bastian Braun, Johannes Koestler, Joachim Posegga, Martin Johns

A Trusted UI for the Mobile Web

Bastian Braun[1], Martin Johns[2], Johannes Koestler[1], and Joachim Posegga[1]

[1] Institute of IT Security and Security Law (ISL), University of Passau, Germany
[2] SAP Research, Karlsruhe, Germany
{bb,jp}@sec.uni-passau.de, martin.johns@sap.com,
koestler@fim.uni-passau.de

Abstract. Modern mobile devices come with first class web browsers that rival their desktop counterparts in power and popularity. However, recent publications point out that mobile browsers are particularly susceptible to attacks on web authentication, such as phishing or clickjacking. We analyze those attacks and find that existing countermeasures from desktop computers can not be easily transfered to the mobile world. The attacks' root cause is a missing trusted UI for security critical requests. Based on this result, we provide our approach, the MobileAuthenticator, that establishes a trusted path to the web application and reliably prohibits the described attacks. With this approach, the user only needs one tool to protect any number of mobile web application accounts. Based on the implementation as an app for iOS and Android respectively, we evaluate the approach and show that the underlying interaction scheme easily integrates into legacy web applications.

1 Introduction

Since the introduction of the original iPhone in 2008, mobile devices are first class citizens in the world of computing. Due to the impressive advances in energy consumption, mobile processor power, and display quality, the majority of the common computing tasks can nowadays be done as easily on a mobile device as on a "real" computer on the desktop.

However, while the computational power of the mobile devices is almost comparable to their desktop counterparts, other key differences, in areas such as screen estate, UI paradigms, or operating system induced limitations, remain for the foreseeable future. These differences have a significant impact on the device's security characteristics: Reduced screen estate results in significant less space for visual security indicators that could help combating phishing attacks [1, 2]. Changed user interaction paradigms allow for different clickjacking variants [3]. Virtual keyboards on mobile devices lead to choosing insecure passwords, due to necessary, uncomfortable context switches between letters, numbers, and special characters [2]. And finally, the current restrictions in mobile operating systems and the lack of an extension model for iOS' mobile browser render most of the currently proposed attack mitigation tools impossible on mobile devices.

As we will explore in Section 2, these limitations especially amplify security threats against mobile web authentication. For this reason, we propose a novel

authorization delegation scheme using a native application, the MobileAuthenticator, that functions as a companion application to the mobile web browser. In this paper, we make the following contributions:

- We analyze how common web authentication attacks, such as phishing or clickjacking, manifest themselves in mobile scenarios and identify a common root cause – the lack of a trusted UI of the browser.
- We propose a novel authorization delegation scheme for mobile web applications that leverages a native companion application. It serves as a trust anchor for the mobile web application's client side through providing the missing trusted UI capabilities.
- We report on a practical implementation of our system as an app for the two currently dominating mobile operating systems, iOS and Android. In this context, we show how the concept can be realized through leveraging the platform-specific facilities for inter-app cooperation.

In the remainder of this paper, we first cover relevant attack classes that target web authentication mechanisms and discuss their specific characteristics in mobile web scenarios (Sec. 2). Then, we present our solution in Sec. 3. In Sec. 4, we document our practical implementations for iOS, Android, and Wordpress. The security and usability properties of our scheme are evaluated in Sec. 5. After revisiting related work in Sec. 6, we conclude the paper in Sec. 7.

2 Security Threats

In the last decade, numerous security problems in the field of web applications have been discovered and documented, among them phishing, cross-site scripting (XSS), cross-site request forgery (CSRF), session fixation, or clickjacking. In this section, we discuss the listed security threats in respect to how they apply to mobile web applications. Furthermore, we explore if previously proposed solutions can be adopted in a mobile environment.

2.1 Threat Classes

In general, mobile web applications are susceptible to the same class of threats as their desktop counterparts. However, it has been shown that several attack types, such as phishing or clickjacking, are harder to solve in the mobile scenario, due to their direct interplay with the available screen estate and web browser chrome [2, 4, 1]. In this section, we list applicable security issues and briefly discuss special aspects of the mobile case.

Phishing: The term *phishing* subsumes all attacks that aim to obtain the user's password via tricking the user to interact with a web resource that claims to be a legitimate part of the targeted web application but in fact is under the control of the attacker. It has been shown [5, 2, 1] that mobile web applications expose a higher level of susceptibility to such attacks, mainly due to the significantly reduced availability of optical indicators, such as browser chrome or SSL indicators.

Clickjacking: A *clickjacking* attacker deludes the user concerning the context and target of her actions to make her click in the attacker's interest. For the mobile case, Rydstedt et al. [4] coin the term *tapjacking* for this attack vector as users do not click but tap on their mobile devices. One of their techniques is zooming elements of the target web page. They found that the hosting (i.e., attacking) page can set a zoom factor overriding the iframe's own scaling. This way, an attacker can include a transparent "Like" or "Tweet" button fitting the entire width of the screen.
CSRF: A *CSRF* attacker inserts a crafted link into some website that makes the browser send a request to the target web application, seemingly on behalf of the user. The mobile case is similar to the desktop scenario with a slight exception: Client-side protection approaches like CsFire [6] do not work because mobile browsers have no or not sufficient extension support.
XSS: The *XSS* attacker is able to inject malicious script code into benign web pages. The code runs in the context of the benign domain and can impersonate the user. There is actually no difference between XSS attacks on mobile browsers and desktop browsers.
Session fixation: During a *session fixation* attack, the attacker sets a session identifier before the user logs in. The attack is successful if the SID is not changed during the login. The attacker's window of opportunity lasts until the user logs out. The mobile case is very similar to the desktop case. However, sessions in some mobile web applications expire later [4], or do not expire at all but only delete the client-side session cookie upon logout [7]. This extends the attacker's control over the user's account.

2.2 On the Infeasibility of Existing Mitigation Approaches in Mobile Web Scenarios

In this section, we discuss several potential solutions to the outlined security problems and show their insufficiency in the realm of mobile web applications.
Client-side SSL authentication: The current generation of – at least Android – smart phones is missing proper tools support for certificate management. Furthermore, the usage of this authentication method only solves a subset of the identified security implications, i.e., all issues that exist in connection with the potential stealing of passwords (i.e., mainly phishing). However, security problems that concern attacker-initiated state changes (e.g. caused by XSS, CSRF, or clickjacking) remain unprotected.
Browser extensions or plugins: A potential approach to overcome shortcomings of web browser-based applications is to include the security mechanism directly into the browser using a browser extension or plugins, such as Silverlight or Flash. However, the web browsers in current smart phones do not support plugins[3], and the only browser offering support for extensions is Firefox Mobile for Android with only a limited number of APIs[4].

[3] The Android platform offered limited support for Flash on a subset of existing devices. Adobe discontinued support by Aug 15, 2012. See `http://adobe.ly/1a1EpPH`.
[4] See `http://mzl.la/1fwQNoX` and `http://bit.ly/1k7NQOE` for details.

Dedicated modified browsers: It is possible to deploy dedicated web browsers to mobile devices, which incorporate enhanced security mechanisms. However, they can not be used within applications that offer an integrated web-view, nor can they be set to serve as the default browser on iOS platforms, thus, excluding roughly half of all users. Finally, developing and maintaining a special browser variant is of high effort and cost, which is also a major roadblock for this potential approach.
Local network-layer helpers: Finally, there are several approaches that rely on local network-layer utilities, such as HTTP proxies. Such tools cannot be deployed to the current generation of mobile devices.

2.3 Root Cause Analysis

Generally speaking, a web application is a reactive system. The web server receives incoming HTTP requests and reacts according to the implemented business logic of the application. A subset of the incoming requests lead to changes in the server-side state while others only retrieve data stored on the server. If received as part of an authenticated session, the first case may represent security sensitive actions on the application data. The handling of such requests requires special attention. Within this paper, we will repeatedly utilize the term *authorized action*.

Definition 1. ***Authorized Action*** *An authorized action is a security sensitive event on the server that is triggered by an incoming authenticated request, meaning that the user authorized the web application to perform the requested action on her behalf.*

Which events have to be considered security sensitive highly depends on the internal logic of the application. Hence, the applicable set of authorized actions has to be determined on a per-application basis. Frequently encountered examples include the login into the application, changing the user's data record, and ordering and purchasing of services or goods. For all such actions, the underlying assumption is that the owner of the credential (password or authenticated SID) is the originator of the triggering event and that the details of the action have not been tampered with by unauthorized third parties. All discussed security issues have in common, that the application's back-end component (i.e., the web server) cannot distinguish authorized actions, which have been conducted intentionally by the user, from authorized actions, that have either been conducted directly by the attacker (e.g., through credentials that have been stolen via phishing or XSS) or have been initiated by the attacker via tricking the user (through clickjacking or CSRF). What web applications are missing is a *trusted path* between the user and the back-end system. The back-end system needs reliable evidence, that the initiated security sensitive actions have indeed been deliberately conducted by the user:

Definition 2. ***Trusted Path*** *An application provides a trusted path, if it can be verified on the server-side that all incoming authorized actions are caused with the user's explicit consent and that their integrity is ensured.*

3 Mobile Authenticator

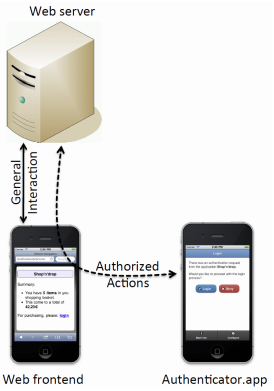

Fig. 1. Solution Overview

The general idea of our approach is to establish a trusted path between the user and the web application in order to protect the user against the attacks given in Sec. 2.1. We implement the approach as an app but we envisage it as an integral feature of mobile operating systems. The mobile application enables the user to communicate securely with the web application's server side using authorized actions that (1) have been explicitly initiated by the user, (2) thus are fully intended by the user, instead of being created without her consent (i.e., through clickjacking or XSS), and (3) have not been tampered with. This way, the security functionality is strongly separated from the web application's browser-based front-end, and hence, the web-specific weaknesses and limitations do not apply anymore. The actual application logic can still be implemented as a cross-platform web application which can be accessed on any web-enabled mobile device. The only part that needs to be implemented as a native application for each mobile platform is the MobileAuthenticator. The MobileAuthenticator itself provides generic security functionality. As a consequence, it can serve as a trusted interface for more than one mobile web application.

3.1 Providing a Trusted Path Through an App

We propose to introduce the MobileAuthenticator as a dedicated system app that serves as a trust anchor for the user in the communication with the web application. It establishes a trusted path between the UI and the application's back-end. However, as extending modern mobile operating systems is out of our scope, we describe the approach as an app that can be installed by the user.

Concept The MobileAuthenticator is a dedicated application that encapsulates the user's credentials and authorization state and that maintains a trust relationship with the web server. Authorized actions are routed through the MobileAuthenticator on behalf of the web application. The mobile web browser never receives, processes, or sends credentials that can be utilized for conducting authorized actions. This way, the MobileAuthenticator serves both as a trusted UI for the mobile web interface as well as a second authentication factor, effectively elevating all supporting web applications to using an implicit two-factor authentication scheme.

Interaction Pattern For most purposes, the interaction between the web browser and the mobile web application remains unchanged. Only in cases, when the user initiates an authorized action, the control flow is routed via the MobileAuthenticator, implementing a challenge/response scheme to capture the user's intend.

1. Using a dedicated interaction bridge between the web browser and the MobileAuthenticator, the authorized action, which is supposed to be triggered, as well as all needed parameters including the server's challenge are passed over to the app.
2. The user explicitly acknowledges the authorized action in the trusted UI of the MobileAuthenticator. This causes the MobileAuthenticator to compute the response to the server's challenge.
3. The MobileAuthenticator passes the control back to the browser including a dedicated credential which allows the triggered authorized action to be conducted.
4. This credential is passed from the web front-end to the server.

Please note: This process is only executed when authorized actions are conducted. For the vast majority of a user's web interaction, the web application remains unchanged (see Sec. 5.3). This also entails, that general authentication tracking is done the regular way, i.e., using HTTP cookies, and that application handling does not change significantly from a user's perspective.

3.2 Components

The overall architecture consists of three main components: The actual MobileAuthenticator that runs on the mobile device and provides the trusted UI, a server-side module that evaluates incoming requests and checks the integrity of the authentication token, and a JavaScript library that is delivered to the browser and takes care of delegation between all participants.

MobileAuthenticator The client-side component, the MobileAuthenticator, maintains a repository of preconfigured authorized actions (see Sec. 3.3) including a human understandable description of each action's impact. Upon receiving a security critical request from the browser, it looks up the respective action's details in its repository, displays the description to the user, and asks for consent. The MobileAuthenticator signs the request using a shared secret (see Sec. 3.3) with the web application, and passes it back to the browser, if the user agreed.

Server-side Module On the web application's server side a counterpart is needed that maintains a trust relationship with the user's MobileAuthenticator instance and implements the challenge/response process to accept incoming authorized actions.

AuthenticationBroker The AuthenticationBroker is a small JavaScript library that provides the necessary interface to the application's web front-end to delegate authorized actions to the MobileAuthenticator for obtaining user consent. Upon receiving the MobileAuthenticator's response, the acknowledged request is routed to the web application for processing. It is evident that the AuthenticationBroker itself is not security critical. This is an important fact because otherwise malicious injected script code might be able to manipulate or disable the AuthenticationBroker and, thus, run an attack. The worst impact of an attack against the AuthenticationBroker, however, is a denial-of-service that prevents authenticated requests from being routed towards the MobileAuthenticator.

3.3 Initial Enrollment on the Mobile Device

Each instance of the MobileAuthenticator that the user wants to use has to be enrolled individually. In this process, the web application's server-side and the application instance initiate a device specific trust context, represented through a shared secret. This enrollment process works as follows:

After account setup, the web application provides the user with a unique URL pointing back to the application, which carries parameters that identify the enrollment process. The user copies this URL to the MobileAuthenticator. The MobileAuthenticator displays the application's domain to ask the user for confirmation. The user confirms by entering her password which is then used by the MobileAuthenticator for authentication. If the initial authentication step terminated successfully, the MobileAuthenticator and the web application compute a shared secret using the Diffie-Hellman key exchange. This secret is not only specific for the user but also for this particular MobileAuthenticator instance. The MobileAuthenticator then discards the user password as it is no longer needed. All further app-to-server interaction uses the shared secret for authentication. As long as this secret is valid, the user will not be required to enter her password again. Finally, the web application supplies a repository of configured authorized actions, including parameters and actionID, and a human understandable description of each request's impact. The MobileAuthenticator is able to maintain several of such (shared key,repository) records and can thus protect all user accounts for compatible web applications on the device.

3.4 User Login

After the MobileAuthenticator and the web application are synchronized, the overall login procedure adheres to the following protocol: The user first accesses the web application's login page in her mobile browser. As the user is not authenticated yet, the server can not utilize user-specific credentials at this step. Instead, it issues a challenge consisting of its `AppID`, the login's `ActionID` and a timestamp (see Sec. 3.7). The challenge is signed using the web application's private key. The respective public key is stored in the MobileAuthenticator during enrollment (see Sec. 3.3). When tapping the login button, the control is delegated by the AuthenticationBroker (see Sec. 3.2) to the MobileAuthenticator. In this step, the server challenge is pushed to the MobileAuthenticator that takes over and asks the user whether she wants to login to this web application. A phishing attack would fail at this point, as the password is never entered to the mobile device for login. If the signature is valid, the MobileAuthenticator compiles the response from the server's challenge, the username, and the device ID and signs it using HMAC with the shared secret, and control is transferred back to the browser. Finally, the AuthenticationBroker sends the signed login request to the web application.

Upon receiving this request, the web server extracts the username and device ID and verifies that the request was indeed signed using the shared secret and, thus, finishes the user's login process. Username and device ID are required to pick the correct shared secret for signing and verification.

3.5 Conducting Authorized Actions

The process for conducting further authorized actions is similar to the login process (see Sec. 3.4). For the login, the MobileAuthenticator witnesses the user's consent and proves the request's integrity and its own authentication by signing the request using the shared secret. The same features are necessary for authorized actions: First, in the browser, the user taps a link or a button requesting an authorized action. The respective request is then relayed to the MobileAuthenticator that obtains the user's consent, signs, and returns the request to the browser. The AuthenticationBroker forwards the request to the web application that checks the signature and performs the requested authorized action.

3.6 Unknown Authorized Actions

During enrollment, the server pushes a list of allowed authorized actions to the MobileAuthenticator. If the web application has been updated since the enrollment of the MobileAuthenticator instance, it can happen that the MobileAuthenticator receives a request for an unknown authorized action. In this case, the MobileAuthenticator updates its local repository by a new list from the web application. This update process can also be triggered in a regular manner or based on push messages. After receiving the updated list from the web server, the MobileAuthenticator verifies that the requested authorized action is indeed contained in the list. If this is not the case, the app rejects the action request.

3.7 Challenge and Response Formats

In this section, we briefly specify the challenge/response formats.
Server Challenge For a given authorized action challenge, the server compiles a tuple consisting of: $CTuple = \{AppID, UserID, ActionID, timestamp\}$. The server HMAC-signs this tuple with the user-specific shared secret to allow the MobileAuthenticator to verify the challenge's authenticity. The values in this tuple have the following meanings:

- *AppID & UserID*: Identifiers of the web application and the user account, to allow the MobileAuthenticator to choose the correct authentication context.
- *ActionID*: Unambiguous identifier of the requested authorized action.
- *timestamp*: To mitigate potential replay attacks, each challenge can be assigned a dedicated lifespan.

The resulting challenge consists of the tuple and the corresponding HMAC signature: $SChallenge = HMAC(CTuple, shared\ secret)$. On the server-side, the challenge is bound to the user's session and, thus, to her session identifier.
Client Response After interacting with the user to capture her explicit consent, the MobileAuthenticator creates the response by assembling the response tuple: $RTuple = \{SChallenge, (Parameter_1), ..., (Parameter_i)\}$. Again, this tuple is HMAC-signed using the shared secret: $CResponse = HMAC(RTuple, shared\ secret)$. The existence and number of the *parameters* depends on the authorized action. For instance, the login procedure requires the username and device ID (see

Sec. 3.4), while the transfer of money in a banking application will most likely include the amount and the receiving account number in the signed response value.

4 Implementation

To practically evaluate the feasibility of our concept, we implemented the solution for the two leading mobile operating systems, iOS and Android. Furthermore, we outfitted the popular CMS Wordpress with server-side support for our system.

4.1 Client-side Implementation

In this section, we point out the platform dependent differences between the implementations for iOS and Android respectively. Our implementation shows that the approach can be put into practice without support by platform providers though we favor an integration into the mobile platforms.

Implementation for iOS On iOS, communication between apps, such as the web browser and the MobileAuthenticator is severely limited. The only - for our purpose - usable channel is leveraging custom URL schemes: An iOS app can register a URL scheme, such as `mobileauth:`, which is registered with the operating system on app installation. When a different app accesses a URL that starts with this custom URL scheme, iOS conducts a context switch and activates the application that has registered the scheme while pushing the calling app into background. The activated app receives the full URL in form of a string for further processing.

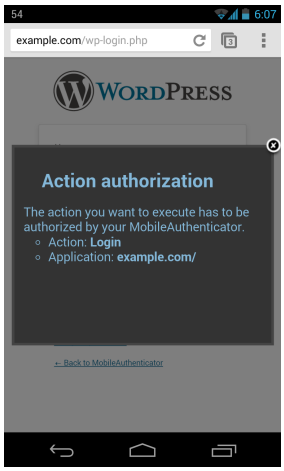

Fig. 2. Triggering an authorized action

We use this mechanism to delegate the authorized action from the web browser to the MobileAuthenticator: The AuthenticationBroker (see Sec. 3.2) compiles a `mobileauth`-URL which carries the server's challenge and the required parameters. Furthermore, the location of the active web document is attached to the URL as the callback URL. Then, the script makes the browser request the compiled `mobileauth` URL via assigning it to `document.location`. This, in turn, causes the operating system to activate the MobileAuthenticator. After user acknowledgment, the MobileAuthenticator calls the callback (`http`-)URL and appends the `CResponse` as a hash identifier. This prevents a page reload in the browser and submits the response to the AuthenticationBroker.

Implementation for Android The MobileAuthenticator provides a background service that is started right after the boot process completed. This service hosts a WebSocket server and is therefore accessible from the device's browser

using the AuthenticationBroker and the HTML5 WebSocket API. The AuthenticationBroker establishes a WebSocket connection to the MobileAuthenticator's background service when it hooks an attempt for an authorized action. It obtains the challenge from the action's HTML meta data and pushes the request together with the challenge to the background service. The background service then launches an activity bringing the MobileAuthenticator to foreground (see Fig. 2). After the user took a decision (either consent or denial, see Fig. 3), the app computes the HMAC on the entire request, including the challenge, appends it and sends the whole string back to the AuthenticationBroker using the established WebSocket connection.

4.2 Server-side Implementation

We implemented server-side components to support the MobileAuthenticator and integrated them into the popular PHP weblog Wordpress as a plug-in. This allows to support legacy web applications without changing the existing codebase. There are three logical components of the plug-in: First, a client administration component manages the enrollment process for new devices, including a device confirmation in the user account, and the revocation of authorized device connections, e.g. because the device was lost or stolen (see Sec. 5). Second, an action verification component issues new challenges and checks incoming requests for valid response tokens. These two components are generic and need no adaption to the particular web application. The last component, however, is application specific. It glues the above components into the legacy code, incorporates the client administration function into the user profile pages, and activates a central request filter that checks if an incoming request targets an authorized action. If so, it forwards the request to the action verification component. The AuthenticationBroker is a JavaScript file that is included with every web page. It is roughly 10kb, is stored in the browser's LocalStorage together with a list of authorized actions, and hooks requests for those actions.

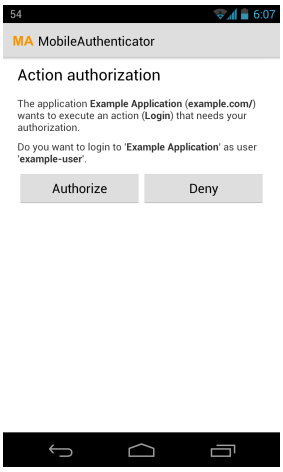

Fig. 3. Obtaining user consent to perform an authorized action

5 Evaluation

We evaluate the MobileAuthenticator with respect to its security and protection properties as well as to its usability.

5.1 Security Evaluation

Phishing: An attack can only succeed if the user enters credentials on a phishing site ignoring the fact that this is not necessary on her device. Expecting a redirect

to the MobileAuthenticator, users become suspicious if their used comfort is missing.

Clickjacking: An attacker can still lure his victim into clicking on links but the target web application then redirects the victim to the MobileAuthenticator where, of course, the attack becomes obvious and the victim does not acknowledge the targeted authorized action.

CSRF: An attack is only detectable for a potential victim if the attacker can forward his payload to the MobileAuthenticator (see Sec. 4). Even if the attacker manages to do so, the victim suddenly faces the MobileAuthenticator asking for permission to perform an authorized action on a different website.

XSS: Injected JavaScript code can perform all actions on the user's behalf. It can raise new authorized actions and redirect the respective requests to the MobileAuthenticator. However, due to the missing shared secret, it can not sign the requests. So, as long as users do not acknowledge unintended actions, no authorized action can be triggered. The only damage an XSS attacker can cause is a denial-of-service by discarding all signed requests and, thus, preventing intended authorized actions.

Session fixation: The attacker can still get access to the user's account. The login step elevates the session cookie to an authorized state granting access to the owner. However, the attacker can not perform authorized actions because he has no access to the shared secret.

No more password entry: Felt and Wagner discussed the fact that mobile keyboards actively discourage the usage of complicated, and thus secure, passwords, as the entry of numbers or special characters require cumbersome context switches [2]. Our scheme obliterates the necessity of entering passwords completely. Hence, the password cannot be stolen, as it is neither stored nor entered again. Moreover, this process allows the usage of arbitrarily complicated application (master) passwords, as the usability drawbacks upon password entry do not apply for our system.

Device-specific credentials: As a matter of fact, mobile devices get lost or stolen from time to time. A thief or finder can use the MobileAuthenticator to log into accounts and conduct authorized actions, once he vanquished the display lock. However, there is built-in protection against this threat: During enrollment (see Sec. 3.3), the MobileAuthenticator and the web application compute a shared secret. The MobileAuthenticator does not store the user password. So, the user only has to revoke the shared secret in her account to prevent any access using the lost device. A thief, in contrast, can not exclude the user as changing the password is not possible without knowing the old password.

5.2 Attacking the MobileAuthenticator

We briefly discuss attacks that might apply directly to our implemented mechanism. The proposed solution as a system app is not susceptible to these attacks.

App Spoofing An attacker may offer a malicious app via the respective platform's market, i.e., Apple's App Store or Google Play. When installed on a user's

device, it could try to obtain user credentials pretending to be the legitimate MobileAuthenticator. The only occasion is the registration of new accounts in the MobileAuthenticator. This, however, is initialized by the user, usually by shortcuts on her home screen. So, as long as this malicious app is not able to replace the legitimate app shortcut with its own, the attacker can not gain confidential knowledge. We want to emphasize that spoofing the legitimate app when the AuthenticationBroker forwards them for signing does not reveal any credentials to the malicious app, because the user only confirms or denies but does not enter anything. The implementation as a system app can register an exclusive protocol scheme such that the registration URL is instantly forwarded to the MobileAuthenticator.

Task Interception There is a task interception attack on Android devices. A malicious app having the necessary permissions (given by the user at installation time) can poll running tasks and display a phishing screen as soon as the target app is started. The user, expecting this screen, would probably enter the credentials. Finally, the malicious app can exit and call the genuine app. This kind of attack is not promising when run on the MobileAuthenticator because the background service is permanently running, thus, revealing no indication for the moment to spoof the MobileAuthenticator screen.

5.3 Usability

Felt et al. phrase crucial criteria for user-friendly interaction with respect to questions and user-based decisions [8]. We generalize and apply their criteria though they study mobile apps and their questions for permissions. In fact, the MobileAuthenticator is similar because it needs a user's decision on the permission to perform an authorized action. We show that the MobileAuthenticator complies with their criteria.

Their first point is to conserve user attention and only ask if the respective question has severe consequences. The MobileAuthenticator only comes into play when such confirmation is necessary. This way, we limit user interaction to the absolute minimum while, in the end, the web application determines the actual authorized actions (see Sec. 2.3).

Second, a usable security mechanism avoids interrupting the user's primary task with explicit security decisions. We achieve this by integrating the user question into the usual workflow. For instance, the MobileAuthenticator can ask the user for consent while presenting an overview of the purchase, including payment information, goods, shipping, etc. The user expects such a final inquiry. So, the integration of the MobileAuthenticator does not interrupt the user's primary task.

Finally, Felt et al. recommend using a trusted UI for *non-revertible*, *severe*, and *user initiated* actions. The authorized actions are generally *not revertible*, which means that the MobileAuthenticator can not let them happen and revert if needed. They are *severe*, meaning that carelessness is not an option and drawing the user's attention is justified. Finally, authorized actions are generally *user initiated*. This is an important point why one can expect the user to confirm her

intent. Other, i.e., implicit, actions can not be confirmed that easily because the user does not know what to decide and why that dialogue popped up.

For instance, a usual shopping workflow and an online banking transaction only require one acknowledgment using the MobileAuthenticator respectively. This acknowledgment can be smoothly embedded in the workflow as a last step being expected by the user anyway. Social networks need to assess their users' risk: publicly posted messages on the one hand are deletable (i.e., revertible), so there is no need for a trusted UI. On the other hand, however, annoying or insulting posts might damage the victim's reputation which is non-revertible and severe. This decision could also be left to each customer weighing her personal or business interests respectively. As a rule of thumb, an acknowledgment step using the MobileAuthenticator is at least necessary when a re-authentication (providing the password again) or second factor authentication (e.g., via Google Authenticator, one-time passcodes, flicker codes) has been in place.

6 Related work

There is no other approach covering the whole range of authentication-based attacks. Existing approaches either protect the login process against phishing [9–11] or target session-based attacks [6, 12–15] Finally, the related body of work includes authentication and authorization protocols in the web [16–18]. GuarDroid [19] aims at establishing a trusted path between the user and the web application using a modified execution platform (firmware). It protects against malicious apps installed on the mobile device and prevents leakage of the user's password. GuarDroid does not require changes of the installed apps nor of the remote web application, however, it can not protect against session-based attacks which still allow a malicious app to impersonate the user towards the web application. GuarDroid causes considerable network latency and requires the user to set, remember, and check a secure passphrase that authenticates the secure login form and delays the system boot process. Finally, the user is responsible to verify the target URL for login requests to prevent phishing attacks, thus, demanding a high level of awareness and increasing the risk that users just click through the dialog. Other existing approaches for trusted paths concerning user login [7] and user actions in authenticated sessions [20] focus on surfing web applications using desktop browsers.

7 Conclusion

In this paper, we presented a web authorization delegation scheme for mobile devices that utilizes a native companion app, the MobileAuthenticator, to realize a trusted UI. For a set of predefined *authorized actions*, our system reliably mitigates state changing effects of currently known user impersonation attacks, such as phishing, CSRF, or clickjacking.

Furthermore, the MobileAuthenticator effectively becomes the user's authentication credential, obliterating the necessity to frequently enter passwords on

the mobile device, thus, correcting the usability drawbacks that are observed when entering secure passwords on mobile keyboards.

The MobileAuthenticator itself is independent from specific characteristics of the protected web application and, thus, can serve as the central trust anchor for many different, independent applications. In consequence, a future integration of such a service on a platform-level into the mobile operating system is a compelling option.

References

1. Amrutkar, C., Traynor, P., van Oorschot, P.C.: Measuring SSL Indicators on Mobile Browsers: Extended Life, or End of the Road? In: ISC. (2012)
2. Felt, A., Wagner, D.: Phishing on Mobile Devices. In: W2SP. (2011)
3. Luo, T., Jin, X., Ananthanarayanan, A., Du, W.: Touchjacking attacks on web in android, ios, and windows phone. In: Foundations and Practice of Security. (2012)
4. Rydstedt, G., Gourdin, B., Bursztein, E., Boneh, D.: Framing Attacks on Smart Phones and Dumb Routers: Tap-jacking and Geo-localization Attacks. In: wOOt. (2010)
5. Niu, Y., Hsu, F., Chen, H.: iPhish: Phishing Vulnerabilities on Consumer Electronics. In: UPSEC. (2008)
6. Ryck, P.D., Desmet, L., Heyman, T., Piessens, F., Joosen, W.: CsFire: Transparent Client-Side Mitigation of Malicious Cross-Domain Requests. In: ESSoS. (2010)
7. Bursztein, E., Soman, C., Boneh, D., Mitchell, J.C.: SessionJuggler: Secure Web Login from an Untrusted Terminal Using Session Hijacking. In: WWW. (2012)
8. Felt, A., Egelman, S., Finifter, M., Akhawe, D., Wagner, D.: How to Ask for Permission. In: HotSec. (2012)
9. Chou, N., Ledesma, R., Teraguchi, Y., Boneh, D., Mitchell, J.C.: Client-side Defense against Web-Based Identity Theft. In: NDSS '04. (2004)
10. Dhamija, R., Tygar, J.: The Battle Against Phishing: Dynamic Security Skins. In: SOUPS. (2005)
11. Balfanz, D., Smetters, D., Upadhyay, M., Barth, A.: TLS Origin-Bound Certificates. [IETF draft], `http://tools.ietf.org/html/draft-balfanz-tls-obc-01`
12. Huang, L.S., Moshchuk, A., Wang, H.J., Schechter, S., Jackson, C.: Clickjacking: Attacks and Defenses. In: 21st USENIX Security Symposium. (August 2012)
13. Jovanovic, N., Kruegel, C., Kirda, E.: Preventing cross site request forgery attacks. In: Securecomm. (2006)
14. Sterne, B., Barth, A.: Content Security Policy. W3C Working Draft, `http://www.w3.org/TR/2011/WD-CSP-20111129/` (November 2012)
15. Johns, M., Braun, B., Schrank, M., Posegga, J.: Reliable Protection Against Session Fixation Attacks. In: ACM SAC. (2011)
16. Mozilla: Persona. [online], `https://developer.mozilla.org/en-US/docs/Mozilla/Persona`, (09/19/13)
17. Lockhart, H., Campbell, B.: SAML V2.0. `https://www.oasis-open.org/committees/download.php/27819/sstc-saml-tech-overview-2.0-cd-02.pdf` (March 2008)
18. Internet2: Shibboleth. [online], `http://shibboleth.net/`
19. Tong, T., Evans, D.: GuarDroid: A Trusted Path for Password Entry. In: Mobile Security Technologies (MoST) 2013. (2013)
20. Braun, B., Kucher, S., Johns, M., Posegga, J.: A User-Level Authentication Scheme to Mitigate Web Session-Based Vulnerabilities. In: TrustBus '12. (2012)