Defense Against Mobile Phishing Attack

Jie Hou, Qi Yang
{jiehou, yangqi}@umich.edu

ABSTRACT

We developed a web-based phishing attack targeting the iOS mobile platform, specifically exploiting the full-screen touch-based user interface. Our demonstration attack requires no special access or modification to the device and mimics the native application GUI to pass casual observation. We then developed a proof of concept defense against this and other similar spoofing phishing attacks. Our system logs user keystrokes and warns the user when potentially sensitive information is about to be entered while running an untrusted application. Our defense system works for malicious website in a mobile web browser as well as all third-party applications we tested.

MOTIVATION

The proliferation of smart mobile phones in the last decade has brought increasing computing power to the general public. These mobile devices that have seen more personal and more frequent usage than traditional PCs. While they become more capable at tasks traditionally fulfilled by PCs, having a desktop-class web browser and the ability to easily install and run third party applications also enable the traditional social-engineering phishing attack to be mounted, as well as new types of phishing attacks exploiting the unique fullscreen touch-based user interfaces. Given that smart mobile devices are used in greater frequency and in increasingly personal context such as social networking, private photography or direction-finding, attacks to these devices can pose a greater threat to a user's privacy and data integrity.

BACKGROUND

Previous surveys have identified the types of phishing attack that can be effectively mounted on mobile devices (Felt, 2011). The frequent use of password protected services (social networks, shopping sites, app and media store) has conditioned users to enter their credentials by reflex when prompted on computers, and this is no different for mobile devices. Most phishing attacks work by spoofing or intercepting the application or website receiving the user credential (see Table 1 for a list of such attacks from Felt, 2011). The impersonation can be easily done and hard to detect, since most of the smart mobile devices (iOS, Android devices) use a large touch screen as the primary interface, with few physical buttons that are exclusively used for interacting with the locked-down operating system.

For example, the home button on iOS devices always triggers an operating system action, such as returning to the home screen launcher or opening task switching interface, and it cannot be used by a third-party application. In addition, native applications can control the entire display and they frequently do, there is no exclusive part of the screen reserved for the operating system. Web browser on iOS is also vulnerable to framing and tap-jacking attacks by malicious websites (Bursztein, 2010) spoofing other sites by modifying the viewport and showing fake address bars. When a website is saved as a bookmark on the device's home screen, the attacker can also hide browser navigation and address bar entirely. As a result the user may never know with certainty what application is currently running, other than trusting the action of launching an application or website. With rare exceptions such as the volume button, user interact exclusively with third party applications using touch screen. The lack of third-party accessible physical control such as pressing ctrl+alt+del on PC also means there is no guarantee that the normal user input on the touchscreen is received only by the trusted source, such as the official App Store, and not an impostor.

These characteristics of the user interface leaves attacker to mount more effective phishing attacks than previously possible on the desktop, by spoofing native applications or websites. To remove the app/website-launching action that can notify the user of currently running app, the attacker can take advantage of the frequent control-transfer or task-switching action in mobile OS usage. One common example is when user tap on a link to a song or an item in the iTunes/App Store, on a website while running Mobile Safari, the control is transferred to iTunes or App Store, while a app switching animation (Figure 1 shows an example

Legitimate Behavior	Prevalence	Attack Technique	Accuracy
Mobile Sender → Mobile Target			
Social sharing, upgrades, game credits	Very Common	Fake mobile login screen	Perfect
Social sharing, upgrades, game credits	Very Common	Task interception	Perfect
Social sharing, upgrades, game credits	Very Common	Scheme squatting	Low
Mobile Sender → Web Target			
Embedded login pages	Common	Keylogging	Perfect
Opening a target in the browser	Very Uncommon	URL bar hiding	High
Opening a target in the browser	Very Uncommon	URL bar Spoofing	High
Opening a target in the browser	Very Uncommon	Fake browser	High
Embedded HTTP page links to HTTPS login	Very Uncommon	Active network attack	Perfect
App sends user to HTTP page in browser that links to HTTPS login	Uncommon	Active network attack + URL bar spoofing	High
Web Sender → Mobile Target			
Link to mobile e-mail or Twitter	Common	Web site spoofs mobile app	High
Link to mobile e-mail or Twitter	Common	Task interception	Perfect
Link to mobile e-mail or Twitter	Common	Scheme squatting	Low
Web Sender → Web Target			
Payment via PayPal or Google Checkout	Common	Hide the URL bar	High
Payment via PayPal or Google Checkout	Common	Spoof the URL bar	High
User follows link from HTTP to HTTPS	Very Common	Active network attack + URL bar spoofing	High

Table 1. List of potential phishing attacks, Prevalence indicates the frequency of the normal actions that the attack is spoofing. Accuracy indicates how easy it is for the user to detect the deception. (Felt, 2011)

of the task switching animation) is briefly shown, and user is taken to the linked item in the Store app. Replicating such switching animation can create the illusion that the control is transferred to a trusted application.

In this paper we implemented a demonstration of such an attack, and then developed a defense system against the attack.

ATTACK IMPLEMENTATION

In our attack demonstration, we implemented the website-to-native-app task-switching spoofing (In Table 1, Web site spoofs mobile app). We picked iOS as the target since it is trivial to implement the task-switching on Android as it has no such transition animation when switching between applications, and because iOS is allegedly more secure (Nachenberg, 2011). If the attack can be effectively mounted on iOS, in theory it should be no more difficult if not easier to do so on Android given the similar capability of the WebKit browser. We chose to implement a web-based spoof instead of building a native app, since it would be trivial to write a native app that spoofs a legitimate app but it would be difficult to get past the iOS App Store review process.

Our attack (Figure 2) pretends to be a normal music shopping site with links to the official iTunes Store. Taps on the site will bring up a dialog asking for confirmation to open the iTunes Store, reinforcing the illusion. After user confirmation, the fake transition animation is activated and a iTunes Store screenshot is shown, and then a dialog prompting the user to log in to the iTunes Store is shown. If the user logged in previously, the discrepancy of being asked to log in again is more likely to be attributed, by unsuspecting users, to software unreliability than a suspicious event. After the user is duped into entering their credential, a real browser alert view is shown to reveal the attack to the user. A live demo[1] is available for iPhone or iPod Touch.

To implement the attack, we used a combination of HTML, Javascript, WebKit animations to reproduce the native look and animations of iOS. For simplicity we did not implement elements of either the source website and the target iTunes Store Application such as

[1] http://www-personal.umich.edu/~yangqi/pivot/pivot.html (save to home screen first)

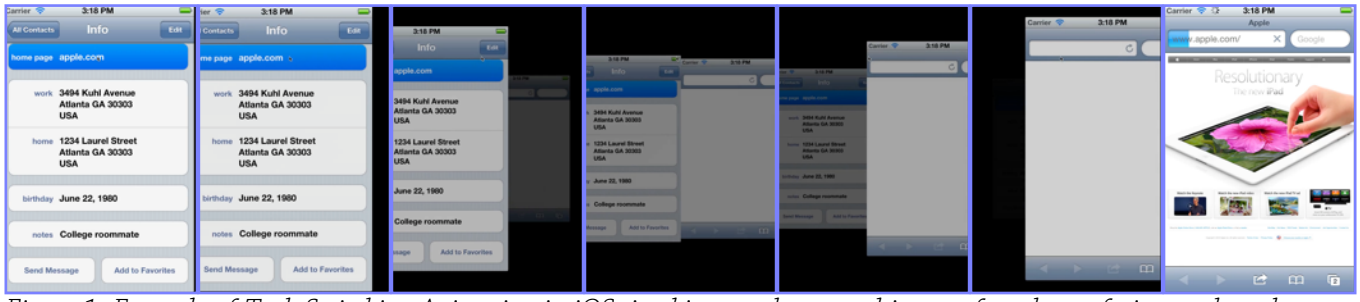


Figure 1. Example of Task-Switching Animation in iOS, in this case the control is transferred to safari.app when the user opens a link in contact.app

buttons or scroll views, since these elements can be done normally as mobile versions of many popular websites do (e.g. Vimeo.com or Apple's iPhone User Guide[2], when viewed on iOS). To hide Mobile Safari's address bar and navigation elements, we saved the site to the device's home screen, a common action for the user when a website is frequently used. This allows us to use the HTML meta tag `apple-mobile-web-app-capable`[3] and `viewport` to view the site in fullscreen mode and disable zooming.

The alert dialogs are created using images reconstructed from screenshots of the real alert dialog and CSS effects such as soft shadow and opacity. The differences can only be discerned if one compares with the real dialog side by side (see Figure 3 for a decomposition of the images used to construct the popup dialog). The animation of the dialogs "pop" into view is done using WebKit keyframe animation[4], at the same time the background shadow, which is also a transparent image, fades in. Once the user confirms the switch by tapping on the "OK" button in the initial dialog, the `<div>` element containing the first application image and dialog is animated out of the view, by spinning slightly outwards, while the element containing image of the iTunes Store is faded and animated in view, mimicking the real app-switching animation. This animation is also implemented using key-framed WebKit 3D transformation.

After arriving in the fake iTunes Store, any tap will trigger a similar dialog asking for user to login, with focus assigned to the username field to bring up browser's built-in keyboard. The text fields are monitored for changes so that as soon as user type in their credential, even without hitting log in or press "enter" the attacker already has their passwords. A final alert view is shown when user press login to reveal that the attack has succeeded. A truly malicious attacker can at this point switch to the real iTunes Store, and the user can continue their original purchase or download unaware of the attack, only attributing the possible redundant login and switching as software or network glitches.

During the attack, it is possible for the user to detect the deception by doing a number of unusual things, such as opening the app-switching interface and examining the previously ran apps, scrolling vertically at the login prompt, or having a authentic login prompt to compare to side by side (Besides other subtle differences, the keyboard used in iTunes Store has a black background instead of the grey one in Safari, but neither look out of place or suspicious). We believe most users are conditioned to immediately login and not examine the interface closely for discrepancies, that our replicated interface would pass as authentic by most iOS users when casually observed.

The seriousness of the threat is also heightened by the low friction of adding frequently used websites to the home screen, which allows the website have greater control of the screen estate. Simple social engineering such as offering popular song suggestions on iTunes Store may be enough to fool user into doing so.

---

[2] http://www.vimeo.com/m/ and http://help.apple.com/iphone/4/interface/

[3]https://developer.apple.com/library/safari/#documentation/AppleApplications/Reference/SafariHTMLRef/Articles/MetaTags.html

[4] http://www.webkit.org/blog/324/css-animation-2

Figure 2. Our demonstration attack fakes the App Switching animation to iTunes Store then asks for user's Apple ID, for which user often have their credit card on file to make media and application purchases

DEFENSE OPTIONS

The lack of currently running application indicator is one of the primary reason that our attack and other similar spoofing attack can be mounted with relative ease. One obvious way of addressing this shortcoming of the mobile OS is to add a persistent indicator of the current running application. Knowing what application is running will hopefully make user more aware if a real task-switching happened or not.

However, both iOS and Android allow third-party applications to display fullscreen, so there is no persistent area of the screen that is exclusively used by the OS. The limited screen space on phones also makes it a challenge to balance the screen usage and obviousness.

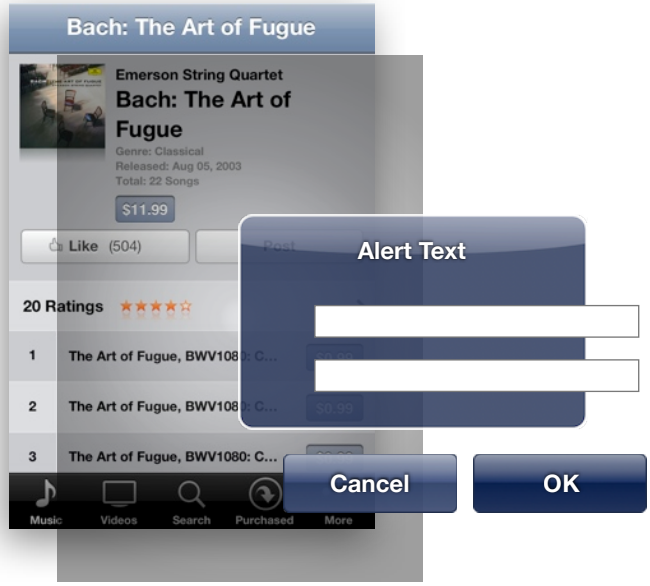


Figure 3. A break down of the image elements of the fake popup dialog

Too big an indicator will take valuable screen space, and when too small the user may fail to notice changes. Even if such a persistent indicator is added, the user still need to be taught to recognize real task-switching, and know which application can be trusted and which can not, which may not be a practical.

We also considered adding an external hardware screen that is exclusively used for displaying currently running application. This may alleviate the screen space issue and requires no modification to existing third party applications, but adding extra bulk to the device that already have a tight engineering constraint on size and power consumption. Furthermore, it does not solve the problem of teaching the user to recognize which applications should be trusted and which should not.

The one option we considered and chose to implement is to actively monitor the user's actions and warn the user if a potentially risky action is about to be committed. Since our attack and the majority of the attack proposed by (Felt 2011) focus on user credentials, our defense monitors user's keystrokes and intervenes when user is about to enter their credential in an untrusted application.

DEFENSE IMPLEMENTATION

To build an effective defense, we need to change OS behavior to monitor user activity and respond. Since we do not have access to the source code of iOS, we cannot modify the operating system from source. Instead, we used dynamic code injection and substitution.

To this end, we used MobileSubstrate[5], which is a runtime tool that can be installed on jailbroken iOS, and it can substitute the original function calls with customized implementations. We implemented customized versions of system functions, and compiled it into a dynamic library, which is loaded by MobileSubstrate runtime to substitute the official library via code injection. This enables us to hook any system functions.

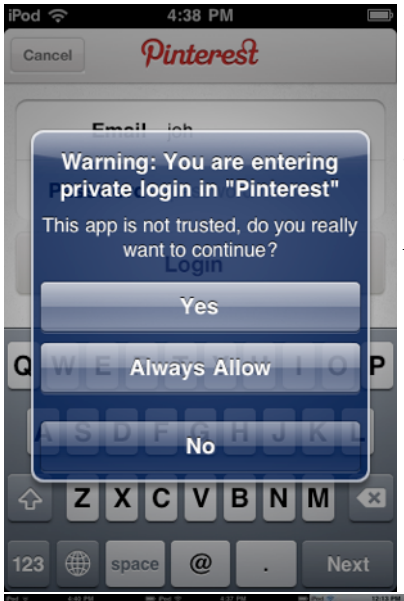
For the defense system, we implemented the following techniques: a keylogger to capture all user input using the system keyboard; an alert system to interrupt the user during potentially risky operations and respond to user's decisions; a framework for setting trust policy and whitelisting trusted applications that user thinks are safe, and intercept the suspicious applications. We will discuss our implementation for theses techniques separately.

1. Keylogger

When capturing the system keyboard input, we have to understand how the on screen touch keyboard works in iOS. When the user touch the screen, a `GSEvent` (GraphicsServices Event) is generated, and registered delegate object can receive and process theses events. For the virtual keyboard, `UIKeyboard` class renders the keyboard view, while `UIKeyboardImpl` class handles key input event. We hooked the method `handleKeyEvent` in `UIKeyboardImpl`, which receives an `GSEvent` and processes it. After hooking `handleKeyEvent`, we deconstructed the data structure of the `GSEvent` object to find the keycode of the keystroke (which is an `UniChar(unsigned short)` value), and save the keycode in our global text input buffer, then call the original implementation of `handleKeyEvent` method which triggers the normal text input. The text buffer has a limited size, but it is large enough for us to see whether it contains the protected user ID or other sensitive information.

2. Alert System

As the user types, we monitor the text input buffer for any match of user ID, if the currently running application is not trusted or whitelisted, an alert is shown to the user. A list of known user ID to be protected is saved for this purpose, as well as a list of trusted applications and a modifiable whitelist. The text matching



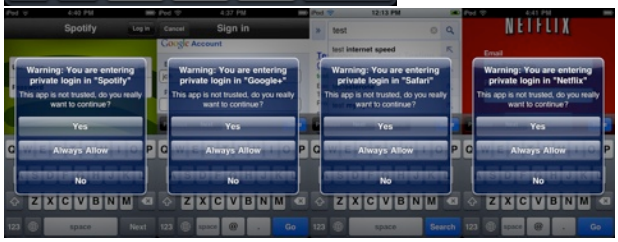


Figure 4. Examples of the alert view shown to user after entering a protected user ID in popular third party applications, we used iOS's standard UIAlertView. (Apps shown are Pinterest, Spotify, Google+, Safari, Netflix)

can be easily extended by regular expression to detect other sensitive information such as credit card numbers.

We chose user ID to monitor instead of entire login since by the time user types in the password the attacker already have the user input, whether or not the user press "Login" or "Enter" key.

A standard iOS `UIAlertView` object is used for alert (see Figure 4 for an example), which displays a prompt that shows the name of the current application that the user is interacting with, and informs the user that a protected user ID input has been detected, prompting the user to make a choice between continuing logging in, or not.

Since the `UIAlertView` object is shown using a different thread than the main thread, its callback function after user responds have no direct access to the `UIKeyboardImpl` object. However, to intercept the user keyboard input (for example, block "enter" key stroke on the keyboard from triggering the login action), we

[5] http://iphonedevwiki.net/index.php/MobileSubstrate

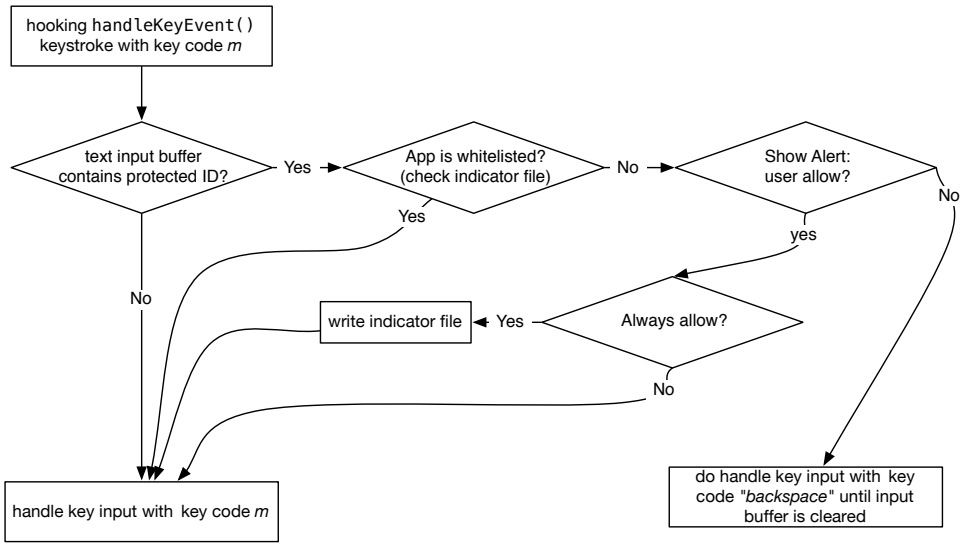


Figure 5. Decision flowchart of our defense system

need the ability to programmatically send and block the keyboard input. This is done by saving a pointer to the `UIKeyboardImpl` object and a function pointer to `handleKeyEvent`, and a memory buffer for holding the function call arguments. We can then easily emulate any keystroke we want, and release the intercepted key stroke at anytime. For example, when the user presses "Yes" in the alert prompt, we release the intercepted key stroke; when user presses "No", instead of releasing the key input, we can emulate "backspace" key strokes to clear the input field to delete the user's login.

3. Policy Whitelist

To avoid needlessly alerting user on every input of user names, which may train user to ignore the warning by reflex, we need finer grained notion of what applications or sites to trust. Our implementation includes the background support for policy settings that can be customized by the user.

We designated all official iOS applications as trusted except the web browser, and for now labelled all third-party applications from the App Store as untrusted for demonstration. Similar to a firewall, when an untrusted application is opened initially, our system will monitor the user's key input and alert user of potential risk. If the user trusts a third party application, they can add the app to the whitelist on an app-by-app basis by selecting "Always Allow" when prompted. An application in the whitelist is considered safe, and will not be monitored by our system anymore.

To indicate whether an application is in the whitelist, we place a indicator file with a magic number inside the application's file-system sandbox. Every time an application is launched, our system checks the existence of the indicator file to see if the application is trusted or not.

EVALUATION

Due to the length of IRB approval process and time constrain, we were not able to conduct a human subject study to evaluate the effectiveness of our defense in terms of usability. Instead, we tested our system against iOS web browser and 15 of the most popular apps on app store and found that it functions effectively with all of them. Our system is able to detect keystrokes, show alert dialog and detect the name of the applications. Since no website can modify keyboard input function of iOS browser, and we believe that popular third-party applications are a representative of how a native attack may be built using the same official API and frameworks, if it past through the App Store review, our defense system should be effective against in most malicious website and native applications.

CIRCUMVENTING DEFENSE

Our system is effective as long as the target application uses iOS's built-in keyboard, if a malicious site or application implements its own keyboard it can bypass the system calls we are hooking to detect keystrokes. However, it is not trivial for a native application to do so, and even more difficult for a web site to implement through Javascript and images, since the system keyboard view is consisted of separate views and animations for each key, and has a high requirement on latency that it might not be easy to replicate fully (it might not be possible to achieve the same kind of per-

formance required for keystroke entry using Javascript).

FIRST PARTY DEFENSE

The fact that our defense requires a jailbroken device means that it is unsuitable for actual usage on iOS (even though there are significant number of jailbroken devices in use, the process of jailbreaking and installing MobileSubstrate plugins is not trivial for the average user, and by its nature the devices are inherently less secure after jailbreaking, and then there is the hurdle of gaining user's trust to acquire their user IDs). It may be useful for actual usage for Android devices. We believe that while our defense implementation serves as a usable proof of concept, a real effective defense, following the same model, have to be implemented by the first party, the OS vendors.

In addition to having access to the App Store catalog and review process to set the appropriate allowance policy for the alert view, OS vendors such as Apple already have the users' trust and ID information (Apple ID), and possibly their other credentials such as Twitter, email, or calendar logins as well. A third party trying to implement such a defense system, even without the technical hurdle of having to make users jailbreak, will be hard-pressed to gain the same level of trust from the users.

CONCLUSION - FUTURE WORKS

We implemented a web-to-native spoofing phishing attack for iOS, and developed a method for defending against such attacks by essentially implementing a keylogger and monitor any user input that resembles their known user ID. Our system interrupts the user then allows them to decide to continue or not. The system also have code supporting policy settings for whitelisting applications. Although the defense can still be circumvented by spoofing the actual keyboard input UI, we believe, as a proof of concept, it is still effective for most web-based malicious spoofing phishing targeting user credentials, and can be beneficial if OS vendors decide to implement it.

Possible areas to explore for future work would be a human subject study on the effectiveness of our defense, compare to one using persistent active application indicator. The code injection/substitution work can also be used to implement a full-fledged password manager for mobile OS.

REFERENCE

Felt, AP, and et al. 2011. "Phishing on Mobile Devices." Presentation at W2SP: Web.

Bursztein, GRBGE. 2010. "Framing Attacks on Smart Phones and Dumb Routers: Tap-Jacking and Geo-Localization Attacks." Usenix.org.

Nachenberg, Carey. 2011. A Window Into Mobile Device Security. Symantec Tech Report.

APPENDIX

The attack demo can be found at:

```
http://www-personal.umich.edu/~yangqi/pivot/pivot.html
```

An iPhone or iPod Touch running at least iOS 4 is supported, the website need to be saved to the home screen then opened.

The implementation source of the defense and set up instructions can be found at:

```
http://www-personal.umich.edu/~yangqi/pivot/keylogger.tar.gz
```

A jailbroken iOS device is required. We developed the defense on iOS 4.2.1, but it should work on iOS 3 as well.

A list of native applications we tested our defense are:

Facebook	Foursquare
Twitter	Google+
Instagram	Netflix
Pandora	Words with Friends
Pinterest	eBay
Skype	Spotify
Groupon	Kindle
LinkedIn	

They are taken from the most popular application list in the App store. Our defense works for all of them.