In finishing: A UI phishing attack to exploit the vulnerability of inotify in Android smartphones

Woo Hyun AHN\(^{(a)}\), Sanghyeon PARK\(^{(a)}\), Jaewon OH\(^{(a,b)}\), Nonmembers, and Seung-Ho LIM\(^{(a)}\), Member

SUMMARY In Android OS, we discover that a notification service called inotify is a new side-channel allowing malware to identify file accesses associated with the display of a security-relevant UI screen. This paper proposes a phishing attack that detects victim UI screens by their file accesses in applications and steals private information.

key words: security, Android OS, malware, inotify, phishing attack

1. Introduction

With the growth of Android smartphones, smartphones have become more vulnerable to phishing attacks that steal private information. An Android application (or app) comprises Activities\(^{(1)}\), each of which is a functionality unit with a set of user interface (UI) screens (hereafter, screens), where each screen has a set of UI elements such as buttons, textboxes, and UI layouts. A UI phishing attack needs to detect the execution of a screen (e.g., login screen) for successful attacks. Such detection can be achieved using the system information that acts as a side-channel.

Chen\(^{(2)}\) considers an Activity as a unit of a UI attack, which detects an Activity by exploiting a probability model based on the CPU, network, and memory usage. However, this scheme cannot detect a specific screen among multiple screens composing an Activity and, moreover, can fail to detect an Activity in an app that has a complex connection structure among Activities. In Screenmilker\(^{(3)}\), a malicious screenshot app detects a soft keyboard pop-up by the memory usage to capture a user’s keystrokes. However, this attack is possible only in those smartphones in which the screenshot app has been installed using the ADB debugging tool. Jana\(^{(4)}\) detects a login screen in a web browser, but not in a general app, by the memory usage and infers user inputs by exploiting CPU scheduling statistics.

We have discovered the fact that a file access notification service called inotify\(^{(5)}\) can be used as a side-channel to detect a specific screen in an Activity. This service generates an access event (or event) when a file to be monitored is accessed. Such generation allows malware to collect a set of events (i.e., an access pattern) that are associated with the execution of a specific screen. Our investigation into popular apps reveals that security-relevant screens are highly probable to generate unique access patterns. Thus, if a victim screen has a unique access pattern, inotify gives malware a good opportunity to monitor the access pattern and accurately detect the execution of the victim screen.

This paper proposes an inotify UI phishing attack termed as finishing to demonstrate how malware steals private information from a security-relevant screen by utilizing its unique access pattern. Our experiments show that finishing is effective in stealthily peeking at payment credentials in popular apps, especially credit card scanning apps. Some finishing attacks are shown in video demos at https://sites.google.com/site/finishing.

2. Background

In Android, clicking a button on a screen of an Activity leads to a transition to either another screen of the Activity or a screen of another Activity. Figure 1 (a) shows the transitions among three screens A, B, and C in a shopping app, Wish, each of which comprises one Activity. A user is transferred to either PayPal (B) or credit card (C) payment screen when clicking each of two buttons on A. If some malware could detect the transition to any of the two screens, it would mount a phising attack against the detected screen. To avoid such an attack, Android does not provide any API that allows identifying the currently displayed screen.

Inotify allows an unprivileged app to monitor accesses to all the types of files including system files. The app is notified of events OPEN, ACCESS, or CLOSE_WRITE when any app opens, reads, or closes a monitored file, respectively. Figure 1 (b) shows events occurring during the transition from screen A to B or C shown in Fig. 1 (a). The transition to B generates events of accesses to font files (i.e., .ttf files) and a built-in keyboard apk file (i.e., SamsungIME.apk). A collection of (SamsungIME.apk, ACCESS) and (Roboto-Bold.ttf, CLOSE_WRITE) events becomes a unique access pattern of B because the access pattern does not occur on the transition to any other screens in the app. The transition to C produces a unique access pat-
tern, i.e., a collection of (\text{SamsungIME.apk}, ACCESS) and (\text{Rotobo-Regular.ttf}, OPEN) events. Events are generated during the transition to a screen but not after the transition completion, so that \(B\) and \(C\) can be detected by their access patterns before being displayed.

3. Attack Design

Inishing requires an attacker to choose a victim screen and find out its unique access pattern by tracing a victim app offline. In a mobile device, inishing malware (or malware) identifies the occurrence of the access pattern by using inotify to attack the victim screen. The malware is assumed to run in the background without a root privilege, similarly to the previous phishing attacks [2], [4] in Android smartphones. A typical inishing attack consists of the following operations. First, the malware informs inotify of files specified in the access pattern and waits until the victim app generates an event on the files. Upon an event occurrence, the access pattern of the victim screen is compared with a specified in the access pattern and waits until the victim app generates an event on the file. The execution of a screen produces a bag (or multiset) of events that occur on the transition to the screen. The reason for choosing a bag is that we consider how many times an event occurs within one transition. A screen \(u_j\) can have \(n\) different bags \(b_1^j, b_2^j, \ldots, b_n^j\) when being run \(n\) times. \(p\) is defined as \(b_1^j \cap b_2^j \cap \ldots \cap b_n^j\) (where \(k\) is the number of runs of \(u_k\)) because \(p\) should consistently occur on every transition to \(u_k\). \(p\) is unique if it is not included in the bags of any other screen \(u_j\) (i.e., \(p \nsubseteq b_j^i\) when \(1 \leq i \leq n\)). Such a uniqueness is commonly observed in victim screens of many popular apps shown in Sect. 5.

In a mobile device, malware detects access pattern \(p\) of victim screen \(u_k\) by repeatedly running all the screens offline in a victim app. The execution of a screen produces a bag (or multiset) of events that occur on the transition to the screen. Therefore, for choosing a bag \(p\) is that we consider how many times an event occurs within one transition. A screen \(u_j\) can have \(n\) different bags \(b_1^j, b_2^j, \ldots, b_n^j\) when being run \(n\) times. \(p\) is defined as \(b_1^j \cap b_2^j \cap \ldots \cap b_n^j\) (where \(k\) is the number of runs of \(u_k\)) because \(p\) should consistently occur on every transition to \(u_k\). \(p\) is unique if it is not included in the bags of any other screen \(u_j\) (i.e., \(p \nsubseteq b_j^i\) when \(1 \leq i \leq n\)). Such a uniqueness is commonly observed in victim screens of many popular apps shown in Sect. 5.

In a mobile device, malware detects access pattern \(p\) of victim screen \(u_k\) by repeatedly running all the screens offline in a victim app. The execution of a screen produces a bag (or multiset) of events that occur on the transition to the screen. Therefore, for choosing a bag \(p\) is that we consider how many times an event occurs within one transition. A screen \(u_j\) can have \(n\) different bags \(b_1^j, b_2^j, \ldots, b_n^j\) when being run \(n\) times. \(p\) is defined as \(b_1^j \cap b_2^j \cap \ldots \cap b_n^j\) (where \(k\) is the number of runs of \(u_k\)) because \(p\) should consistently occur on every transition to \(u_k\). \(p\) is unique if it is not included in the bags of any other screen \(u_j\) (i.e., \(p \nsubseteq b_j^i\) when \(1 \leq i \leq n\)). Such a uniqueness is commonly observed in victim screens of many popular apps shown in Sect. 5.

In a mobile device, malware detects access pattern \(p\) of victim screen \(u_k\) by repeatedly running all the screens offline in a victim app. The execution of a screen produces a bag (or multiset) of events that occur on the transition to the screen. Therefore, for choosing a bag \(p\) is that we consider how many times an event occurs within one transition. A screen \(u_j\) can have \(n\) different bags \(b_1^j, b_2^j, \ldots, b_n^j\) when being run \(n\) times. \(p\) is defined as \(b_1^j \cap b_2^j \cap \ldots \cap b_n^j\) (where \(k\) is the number of runs of \(u_k\)) because \(p\) should consistently occur on every transition to \(u_k\). \(p\) is unique if it is not included in the bags of any other screen \(u_j\) (i.e., \(p \nsubseteq b_j^i\) when \(1 \leq i \leq n\)). Such a uniqueness is commonly observed in victim screens of many popular apps shown in Sect. 5.
4. Attacking Popular Apps

Inishing steals a user’s credentials from the PayPal payment screen $B$ in Fig. 1 (a) to show its applicability to a popular app. This screen is a good example of a victim screen because the PayPal is one of the most used online payment systems. The offline observation in Fig. 1 (b) reveals that the transition to $B$ generates a unique access pattern, i.e., a bag of ($SamsungIME.apk$, ACCESS) and ($Roboto-Bold.ttf$, CLOSE_NOWRITE) events, through which malware detects $B$ to pop up a phishing screen.

The phishing screen consists of transparent textboxes (or text fields) with the same sizes and positions as the email and password textboxes in screen $B$. The transparent textboxes are implemented using an always-on-top scheme, which makes the window manager create the two transparent textboxes on the current top screen without making a new Activity and places them over the original textboxes, similarly to the previous research [7]. The scheme allows the transparent textboxes to intercept user inputs that otherwise would be delivered to the original textboxes. The malware sends the stolen inputs to a remote attacker and shows a fraudulent message (e.g., “network error”) so as to hide the attack. On the other hand, the malware may pop up a fake screen [2, 8] identical to $B$, but the fake screen should be displayed earlier than $B$ so that a user cannot suspect the attack. In contrast, the transparent textboxes are less probable to cause user suspicion even if they appear later.

Figure 2 (a) shows that inishing can detect credit card registration screen $R$ among three screens composing an Activity in the Amazon shopping app. Each of the screens is an instance of WebView class that is used to display Web contents in an app. The detection is different from that of Chen [2] that cannot detect a specific screen in an Activity. Figure 2 (b) shows the offline investigation into access patterns occurring on the transitions among the screens. Since $R$ has the same access pattern as that of another screen, malware detects $R$ by two consecutive access patterns of $R$ and its predecessor screen $Q$, which are different from those of any other predecessor–successor pairs of screens. The malware then pops up a phishing screen with transparent textboxes to intercept the card information.

We now show how inishing attacks the vulnerability of an app that scans a credit card through a camera, an example of which is the Uber car-sharing app (Fig. 3). Clicking a scanning button on screen $X$ transfers a user to screen $Y$, which scans a credit card after auto-focusing the camera to insert its information into textboxes in $X$. Malware takes a new picture of the credit card when $Y$ completes scanning it.

An offline observation discloses that the transition to $Y$ causes an access to a specific library file. Android stores a library file of an app in the directory /data/app-lib/app-package-name during the app installation and dynamically loads the file on demand. Uber has several library files including libcardioRecognizer.so, which is consistently opened on every transition to $Y$ but not to any other screens. Such a behavior enables the file access to be a unique access pattern of $Y$.

Malware obtains a photo image of a credit card from the detected screen $Y$ as follows. Uber calls open() to obtain a camera when entering $Y$. The opening prevents the malware from obtaining the camera because it cannot be obtained by multiple apps simultaneously. Thus, the malware calls open() periodically (e.g., every 0.1s) until the app releases the camera, which is then obtained upon its release to take a picture of the credit card in the background.

5. Attack Evaluation

We evaluated the effectiveness, applicability, and stealthi-
ness of our attack on popular apps: shopping apps (Amazon, Wish, eBay), a car-sharing app (Uber), a Korean banking app (Wooribank), a Korean digital wallet (Paynow), and travel booking apps (Orbitz, Ostrovok.ru). The evaluation was conducted on a Samsung Galaxy S3 smartphone installed with the Android 4.2 Jelly Bean operating system.

Table 1 shows several characteristics of the popular apps: the total numbers of Activities, screens, victim screens detected by unique access patterns, transitions among screens, and library files in the directory /data/app-package-name in an app. The victim screens deal with critical private information (i.e., login, payment, bank account) whose stealing can cause economic losses to users. We also measure the average number of events occurring on a transition in an app. Paynow has much more events than other apps do, because our attack monitors events on its apk file that is frequently accessed. In contrast, Wooribank generates the extremely few events because the app accesses its monitored files (see Table 2) mainly during the transitions to the victim screens.

Table 2 shows the access patterns used to detect the victim screens (shown in our demo videos described in Sect. 1) in the following apps: payment screens (Amazon, Wish, eBay, Uber) and a login screen (Wooribank). In eBay, the victim screen is detected using a sequence of access patterns of its own and its predecessor, similarly to Amazon. Each of the victim screens is included in an Activity composed of 11, 64, 3, 5, and 5 screens in Amazon, Wish, eBay, Uber, and Wooribank, respectively. Such multiple screens in an Activity make Chen [2] ineffective in detecting a screen in the Activity because the scheme considers an Activity as a unit of a UI detection, differently from our attack.

Our attack pops up phishing screens consisting of transparent textboxes over the victim screens in the apps except Wooribank, in which the victim screen does not use a built-in keyboard of the device, but its own keyboard. Such a keyboard forces us to exploit Activity hijacking [2], [8] that pops up the same fake login screen as the victim screen. The victim screen is detected using an access pattern to an authorization certificate file (i.e., signCert.der) and its directory, which are stored in an external storage that other apps can access. Therefore, malware can access a user's bank account information by stealing his/her certificate file and login credentials.

Table 3 shows how effectively inishing attacks credit card scanning apps. The transition to a scanning screen generates an OPEN access to a library file (i.e., libcardioRecognizer.so) in the apps, in which the access is a unique access pattern of the screen. The apps load the same library file in their package-name directories upon the transition to their scanning screens, because the apps use a software package of the same credit card scanning company (i.e., card.io).

Our attack is also made on the 10 most popular shopping apps (i.e., coocha, noshomeshop, gmarket, gshop, Ifmall, Igfashionmall, home&shopping, coopang, gshoptv, ditto) in the Korean Google Play store to evaluate its applicability. Two common screens are chosen as victim screens in the apps because they use the same mobile payment sys-
Table 5  Unique file access patterns on LG G2 smartphone

| Apps      | Unique file access patterns                                                                 |
|-----------|---------------------------------------------------------------------------------------------|
| Amazon    | Not found                                                                                    |
| Wish      | Sequence of two access patterns: (1) Predecessor screen: | /data/app/com.contextlogic.wish-1/base.apk, ACCESS) executed 69 times | /system/fonts/LGSmartGothicKR-Bold.ttf, OPEN) |
|           | (2) Victim screen: (/data/app/com.contextlogic.wish-1/base.apk, ACCESS) executed 25 times    |
| eBay      | (/system/fonts/Roboto-Bold.ttf, OPEN) executed 2 times                                       |
|           | (/system/fonts/LGSmartGothicKR-Bold.ttf, OPEN)                                               |
| Uber      | (/data/app/com.uber-cab-1/base.apk, ACCESS) executed 19 times                                |
| Wooribank | (/storage/emulated/0/NPKI/yessign/USER/<user-info>/signCert.der, OPEN)                      |

Table 4  Elapsed times during the transition to a victim screen

| Elapsed times (ms) | Amazon | Wish | eBay | Uber | Wooribank |
|-------------------|--------|------|------|------|-----------|
| (1)               | 1.771  | 1.145| 6.473| 673  | 1.763     |
| (2)               | 1.771  | 1.145| 6.473| 683  | 1.769     |
| (3)               | 1.830  | 1.216| 6.559| 733  | 1.844     |
| (4)               | 1.908  | 1.234| 6.762| 805  | 2.645     |
| (5)               | 1.901  | 1.233| 6.758| 801  | 2.623     |

As shown in Table 4, elapsed times are measured during the transition to a victim screen just after a user clicks a button on its preceding screen: (1) the elapsed time from the click to the occurrence of the first event among events to be monitored, (2) the elapsed time from the first event to the detection of the victim screen, (3) the elapsed time from the top-up of a phishing screen to the display of the victim screen. Such a completion prevents a user from perceiving our attack. Moreover, we measure the elapsed time from the display of the victim screen on which the attack is not made, as shown in the last row. This elapsed time to the display is almost equal to that of the attacked screen in all the apps. This result also reveals that a little CPU overhead can avoid user suspicion of our attack.

We suggest secure system and app designs to avoid the inishing attack. First, Android should prevent an app from using inotify APIs to monitor accesses to system files such as font and keyboard files in an internal storage. Such a restriction does not allow malware to detect the execution of a screen. Second, an app should not expose accesses to files (e.g., an authorization certificate file) for a screen handling a user’s credentials. To achieve such a hiding, the app can store the files in its own private directory (e.g., /data/data/app-package-name/files) that other apps cannot access.

Two approaches are introduced to avoid our attack on a credit card scanning app. The first requires an app to load a scanning library file at its launch, but not during the transition to a scanning screen on demand, even if the earlier loading can delay the app launching. Second, an app should be designed to download a scanning library file from a remote server to its own private directory.

Table 5 shows the access patterns used to detect the same victim screens shown in Table 2. The access patterns in Wish, eBay, and Uber are different from those of the Samsung Galaxy S3 smartphone, while Wooribank has the same access pattern. However, we cannot extract the access pattern in Amazon because any events are not generated on the transition to the victim screen. Moreover, the access patterns in Wish and eBay include events on a device-specific font file (i.e., LGSmartGothicKR-Bold.ttf). In contrast, the credit card scanning apps generate the same access patterns as those in Table 3 on the transition to a scanning screen because they use the scanning software package of the same company on the two smartphones. Therefore, these experiments show that our attack is applicable to various types of Android smartphones.

6. Discussion

We suggest secure system and app designs to avoid the inishing attack. First, Android should prevent an app from using inotify APIs to monitor accesses to system files such as font and keyboard files in an internal storage. Such a restriction does not allow malware to detect the execution of a screen. Second, an app should not expose accesses to files (e.g., an authorization certificate file) for a screen handling a user's credentials. To achieve such a hiding, the app can store the files in its own private directory (e.g., /data/data/app-package-name/files) that other apps cannot access.

Two approaches are introduced to avoid our attack on a credit card scanning app. The first requires an app to load a scanning library file at its launch, but not during the transition to a scanning screen on demand, even if the earlier loading can delay the app launching. Second, an app should be designed to download a scanning library file from a remote server to its own private directory.

7. Conclusions

Inishing is a powerful phishing attack that exploits the vulnerability of the inotify service in Android. Inotify allows an attacker to accurately detect the execution of a security-relevant UI screen by using its unique access pattern. In our experiment, such screens had unique access patterns in many popular apps, in which our attack peeked at user inputs from detected payment or login UI screens. Finally, we suggested some secure system and app designs that are expected to be implemented in the Android operating system and commercial apps in the future.
Acknowledgements

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (No.2011−0013781, NRF−2011−0023224). The present Research has been conducted by the Research Grant of Kwangwoon University in 2015. This work was supported by the Catholic University of Korea, Research Fund, 2015. This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (No.2015R1A5A7036384).

References

[1] R. Meier, Professional Android Application Development, Create-Space Independent Publishing Platform, 2014.

[2] Q.-A. Chen, Z. Qian, and Z.M. Mao, “Peeking into your app without actually seeing it: UI state inference and novel Android attacks,” Proc. USENIX Security Symp., 2014.

[3] C.-C. Lin, H. Li, X. Zhou, and X. Wang, “Screenmilker: How to milk your Android screen for secrets,” Proc. NDSS., 2014.

[4] S. Jana and V. Shmatikov, “Memento: Learning secrets from process footprints,” Proc. IEEE Symp. on Security and Privacy, pp.143–157, 2012.

[5] Filesystem notification series, http://lwn.net/Articles/605313.

[6] R. Prodduturi, Effective handling of low memory scenarios in Android, Master’s Thesis, Indian Institute of Technology, 2013.

[7] M. Niemietz and J. Schwenk, “UI redressing attacks on Android devices,” Proc. BlackHat Abu Dhabi, 2012.

[8] E. Chin, A.P. Felt, K. Greenwood, and D. Wagner, “Analyzing inter-application communication in Android,” Proc. MobiSys, pp.239–252, 2011.