IJMTST is UGC Approved Journal with Journal Id: 43137

International Journal for Modern Trends in Science and Technology, 8(S08): 185-190, 2022
Copyright © 2022 International Journal for Modern Trends in Science and Technology
ISSN: 2455-3778 online
DOI: https://doi.org/10.46501/IJMTST08S0832
Available online at: https://www.ijmtst.com/vol8si08.html

Detecting Hidden Attacks through the Mobile App-Web Interfaces

Kalpana Gottumukkala | G. Ramachandrarao

Computer science and engineering, Chalapathi Institute of Engineering and Technology, Guntur, AP, India

To Cite this Article
Kalpana Gottumukkala and G. Ramachandrarao. Detecting Hidden Attacks through the Mobile App-Web Interfaces. International Journal for Modern Trends in Science and Technology 2022, 8(S08), pp. 185-190. https://doi.org/10.46501/IJMTST08S0832

Article Info
Received: 26 May 2022; Accepted: 24 June 2022; Published: 28 June 2022.

ABSTRACT

Mobile users are increasingly becoming targets of malware infections and scams. Some platforms, such as Android, are more open than others and are therefore easier to exploit than other platforms. In order to curb such attacks it is important to know how these attacks originate. We take a previously unexplored step in this direction and look for the answer at the interface between mobile apps and the Web. Numerous in-app advertisements work at this interface: when the user taps on an advertisement, she is led to a web page which may further redirect until the user reaches the final destination. Similarly, applications also embed web links that again lead to the outside Web. Even though the original application may not be malicious, the Web destinations that the user visits could play an important role in propagating attacks. In order to study such attacks we develop a systematic methodology consisting of three components related to triggering web links and advertisements, detecting malware and scam campaigns, and determining the provenance of such campaigns reaching the user. We have realized this methodology through various techniques and contributions and have developed a robust, integrated system capable of running continuously without human intervention. We deployed this system for a two-month period and analyzed over 600,000 applications in the United States and in China while triggering a total of about 1.5 million links in applications to the Web. We gain a general understanding of attacks through the app-web interface as well as make several interesting findings, including a rogue antivirus scam, free iPad and iPhone scams, and advertisements propagating SMS Trojans disguised as fake movie players. In broader terms, our system enables locating attacks and identifying the parties that intentionally or unintentionally let them reach the end users and, thus, increasing accountability from these parties..

KEYWORDS: - Virus, Application Permissions, Redirect and Malicious URL

1. INTRODUCTION

Android is the predominant mobile operating system with about 80% worldwide market share [1]. At the same time, Android also tops among mobile operating system in terms of malware infections [2]. Part of the reason for this is the open nature of the Android ecosystem, which permits users to install applications for unverified sources. This means that users can install applications from third party app stores that go through no manual review or integrity violation. This leads to easy propagation of malware. In addition, industry researchers are reporting [3] that some scams which traditionally target desktop users, such as random ware and phishing, are also gaining ground on mobile devices. In order to curb Android malware and scams, it is important to understand how attackers reach users. While a significant amount of research effort has been spent analyzing the malicious applications themselves, an important, yet unexplored vector of malware propagation is benign, legitimate applications that lead users to websites hosting malicious applications. We call this the app-web interface. In some cases this occurs through web links embedded directly in applications, but in other cases the malicious links are visited via the landing pages of advertisements coming from ad networks.

2. LITERATURE SURVEY

Mobile users are increasingly becoming targets of malware infections and scams. Some platforms, such as Android, are more open than others and are therefore easier to exploit than other platforms. In order to curb such attacks it is important to know how these attacks originate. We take a previously unexplored step in this direction and look for the answer at the interface between mobile apps and the Web. Numerous in app advertisements work at this interface: when the user taps on an advertisement, she is led to a web page which may further redirect until the user reaches the final destination. Similarly, applications also embed web links that again lead to the outside Web. Even though the original application may not be malicious, the Web destinations that the user visits could play an important role in propagating attacks. In order to study such attacks we develop a systematic methodology consisting of three components related to triggering web links and advertisements, detecting malware and scam campaigns, and determining the provenance of such campaigns reaching the user. We have realized this methodology through various techniques and contributions and have developed a robust, integrated system capable of running continuously without human intervention. We deployed this system for a two-month period and analyzed over 600,000 applications in the United States and in China while triggering a total of about 1.5 million links in applications to the Web. We gain a general understanding of attacks through the app web interface as well as make several interesting findings, including a rogue antivirus scam, free iPad and iPhone scams, and advertisements propagating SMS trojans disguised as fake movie players. In broader terms, our system enables locating attacks and identifying the parties (such as specific ad networks, websites, and applications) that intentionally or unintentionally let them reach the end users and, thus, increasing accountability from these parties.

3. METHODOLOGY

Our methodology for analyzing app-web interfaces will involve the following three conceptual components:

- Triggering. This involves interacting with the application to launch web links, which may be statically embedded in the application code or may be dynamically generated (such as those in the case of advertisements).
- Detection. This includes the various processes to discriminate between malicious and benign activities that may occur as a result of triggering.
- Provenance. This is about understanding the cause or origin of a detected malicious activity, and attributing events to specific entities or parties. Once a malicious activity is detected, this component provides the information required in order to hold the responsible parties accountable. Different processes and techniques may be plugged-in to these different components almost independently of what goes into the other components. The rest of this section elaborates on these three components, describing the various processes we incorporate into each of them.

3.1. Triggering App-Web interfaces

Recall from previous discussion that web links in applications are often dynamically generated (such as from advertisements). Thus a static approach of extracting web links is not sufficient. Therefore, in order to trigger web links from within the application, we run the applications in a custom dynamic analysis environment. To enable scalability and continuous operation, running applications on real devices is not a feasible option. Therefore, each application is run in a virtual machine based on the Android emulator. The applications we are interested in are primarily GUI oriented and therefore we need to navigate through the GUI automatically to trigger app-web interfaces. The rest of this subsection describes techniques that we leverage from past research in order to accomplish this, as well as some new techniques designed to overcome issues specific to the app-web interface.

- Application UI Exploration:

Application user interface (UI) exploration is necessary to trigger app web interfaces. Researchers have come up with a number of systems for effective UI exploration catering to varied applications and incorporating different techniques (Section VIII). An effective UI explorer will offer high coverage (of the UI, which may also translates to code coverage) while avoiding redundant exploration. For our work, we used the heuristics and algorithms that we had developed earlier in Apps Playground [6].

• We briefly describe these next UI exploration generally involves extracting features (the widget hierarchy) from the displayed UI and iteratively constructing a model or a state machine of the application's UI organization, i.e., how different windows and widgets are connected together. A black-box (or grey-box) technique, such as Apps Playground, may apply heuristics to identify which windows and widgets are identical to prevent redundant exploration of these elements. Window equivalence is determined by the activity class name (an activity is a code-level artifact in Android that describes one screen or window). Widget equivalence is determined by various features such as any associated text, the position of the widget on the screen, and the position in the UI hierarchy. In order to prevent long, redundant exploration, thresholds are used to prune the model search.

• Handling Web views: While studying advertisements. We faced a significant challenge: most of the in-app advertisements are implemented as customizations of Web views (these are special widgets that render Web content, i.e., HTML, JavaScript, and CSS). Web views and some custom widgets are opaque in the UI hierarchy obtained from the system, i.e., the UI rendered inside them cannot be observed in the native UI hierarchy and thus interaction with them will be limited. To the best of our knowledge, previous research has not proposed a satisfactory solution to this problem. Certain open source projects, such as Selendroid [7], may be used to obtain some information about the internals of the Web view. We developed code around Selendroid to interact with Web views. However, our experience was that it is difficult to use the information provided from Web views to trigger advertisements. Advertisements often include specific buttons (actually decorated links) that should be clicked to trigger the ads. They may also present other features such as those relating to users' preferences, but which are irrelevant for our purposes. The relevant links cannot easily be distinguished from the irrelevant ones. Often times the click-able link is represented by images instead of text. If we click the irrelevant links, ads may not get triggered, resulting in low click-through rates. In order to overcome this issue of essentially flat (i.e., with no hierarchical structure in the UI debug interfaces provided by Android) Web views, we apply computer graphics techniques in order to detect buttons and widgets as a human would see them. Algorithm 1 presents the detection algorithm.

3.2. Algorithm 1Button detection algorithm

- Perform edge detection on the view's image
- Find contours in the image
- Ignore the non-convex contours or those with very
- Small area Compute the bounding boxes of all remaining contours.

The first step, edge detection, is the technique of identifying sharp changes in an image. Fundamentally, it works by detecting discontinuities in image brightness. We specifically use the Canny edge detection algorithm, a classical, yet generally well performing edge detection algorithm. In the second step we compute contours of images, using the computed edges, to obtain object boundaries. Since buttons typically have a convex shape and a large enough area so that a user can easily tap on them, we ignore non-convex contours and those with a small area within a threshold parameter. Numerous contours such as those arising out of text or the non convex or open contours in embedded images are eliminated in this step. For the remaining contours, we compute the bounding boxes, or the smallest rectangles that would contain those contours. This step is simply to identify a central point where a tap can be made to simulate a button click. The resulting bounding boxes signify the buttons that would be visible to a human being. We have not performed a thorough evaluation of the accuracy of our technique but the results are good in the cases we have examined.

3.3. Detection

As the links are triggered, they may be saved for further analysis and detection of malicious activity such as spreading malware or scam. We would like to capture the links, their redirection chains, and their landing pages. The links, redirection chains, and the content of the landing pages may then be further analyzed using various methods.

4. PROPOSED SOLUTION

We implemented most of our system in Python. For UI exploration, we make use of the source code of the Apps

Playground tool [10]. However, the existing version of the tool is unable to run on current versions of Android, and we therefore re-implemented the system to work on current Android versions with the same heuristics as are described in the Apps Playground paper. Furthermore, instead of using Hierarchy Viewer for getting the current UI hierarchy of the application, we used UI Automator, which is based on the accessibility service of Android. This had a significant and positive effect on the speed of execution. The graphics algorithms used for button detection were provided by the Open CV library and appropriate thresholds were chosen after repeated testing. To improve speed of dynamic analysis, we take advantage of KVM accelerated virtualization. To use this, we use Android images that can run on the x86 architecture. About 70% Android applications have no native code and so can run without problem on such targets. Other applications contain ARM native code and cannot run on x86 architecture without proprietary library support. We therefore excluded applications containing native code. Despite this we believe the study results are generally representative. Furthermore, not being able torun ARM native code is not a fundamental limitation of our approach: third party Android emulators, e.g., Geny motion, or the use of a dynamic ARM-to-x86code translation library (libhoudini) can allow running ARM code on hardware-accelerated x86 architectures [11], [12].For post trigger analysis, our entire framework is managed through Celery [13], which provides job management with the ability to deploy in a distributed setting. Once this stage is completed, any recorded redirection chains are queued through a REST API into the Celery-managed queue together with information about the application and part of the code that was responsible for the triggering of the intent that led to the redirection chain. Tasks are pulled from the queue to perform further analysis on the landing pages and scan the files and URLs with Virus Total as described above. The whole system has proper retry and timeout mechanisms in place and could run for multiple months without significant need of human attention. All the resulting analysis data is stored in MySQL and MongoDB databases. Since the framework works in a distributed, concurrent manner, server-based SQL engines such as MySQL were more appropriate than server less implementations like SQLite. SQL commands are additionally wrapped with SQL Alchemy, a library that provides object-relational mapping (ORM), generally easing the programming. We implemented the analysis of the landing pages or the final URLs in the redirection chains on top of Chromium web browser using Watir and the Selenium Web driver framework. We use Watir and Web driver to script browser actions for automatically loading web pages, clicking on links, automatically download content that is available on clicking links, as well as going back to the original page if a new page loads after clicking on links. All the processing is done headlessly (i.e., without any GUI) using the Xvfb display server, which is an X server implementation that does not present a screen output. Applications are run in the virtualized environment for a maximum of five minutes, with the average running time less than two minutes. The post-trigger analysis, especially the analysis of landing pages, is allowed to run for a maximum of fifteen minutes. We allow for such a long time as our crawler may traverse many links and each link may have complex redirection mechanisms that may trigger only after a short wait. A systematic static analysis methodology to find ad libraries embedded in applications and dynamic analysis methodology consisting of three components related to triggering web links, detecting malware and scan campaigns, and determining the provenance of such campaigns reaching the user.

5. EVALUATION AND RESULTS

5.1. Application Collection Our application dataset consists of 492,534 applications from Google Play and 422,505 applications from four Chinese Android application stores: 91, Anzhi, App China, and Mumayi. Google Play has a proprietary API for searching and downloading applications from the store and it further requires Google account credentials to do these tasks. We used Play Drone, which is an open source project to crawl Google Play [14]. Google implements rate limiting based on Google accounts and IP addresses and bans accounts and IP addresses if there are two many requests in a given period of time. Play Drone mitigates this problem by seamlessly allowing the use of multiple Google accounts and deploying the crawler over multiple machines in a distributed manner. We used the multiple Google accounts feature but simplified the system by using a single machine and setting multiple IP addresses for that machine. In our deployment, every new connection to Google's servers randomly chooses from among twenty source IP addresses. To crawl applications from Chinese application stores, we used our own in house tool. These third-party stores have a much simpler API than Google Play and typically have a public http/https URL associated with each application. While there can be sophisticated ways to search for each application, the technique we employed was based on the observation that applications in all these stores have identifiers in a small integer range. Requesting URLs constructed for each possible identifier sufficed to completely scrap these applications stores. After removing applications that were redundant among these stores, the total number amounts to 422,505. About 30% applications have native code and due to implementation reasons mentioned in Section IV cannot be tested on our system. Our entire usable application dataset therefore consists of a little over 600,000 applications.

5.2. Deployment We deployed our system to gather results over a period of about two months from mid-April 2015 to mid-June 2015 in two locations, one at Northwestern University campus in the US and the other at Zhejian University campus in China. The deployment ran continuously with little manual intervention, and restarts were necessary only when we needed to update the system for fixing bugs or adding features. To have a realistic setting, the Northwestern University location ran applications from Google Play (only the applications available from the US) while the Chinese university location ran applications from Chinese application stores. The location where the apps are run is important because much of advertising, which forms bulk of the app-web interaction we are studying, is targeted based on location. The advertisements that are seen in one location may not be shown in another location.

5.3. Overall Findings

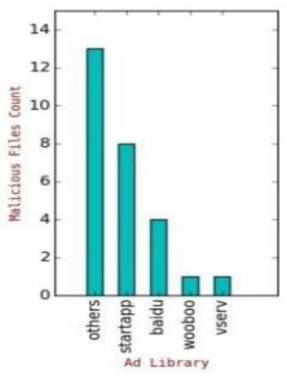

Fig. 4. Malicious files downloaded through ad libraries and through other links not affiliated with any ad libraries in US deployment. Libraries not resulting in malware downloads are not shown. Tapcontext malware numbers are not shown here as they are too high.

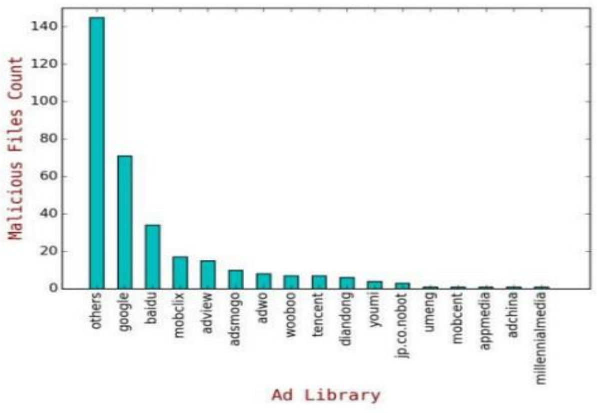

Fig. 5. Malicious files downloaded through ad libraries and through other links not affiliated with any ad libraries in Chinese deployment. Tapcontext and libraries not resulting in malware downloads are not shown.

Overall, we recorded a total of slightly over 1 million launches of app-to-web links in the US deployment. In the Chinese deployment, this number was 415,000. Note that this is not a direct correspondence with the applications:

Some applications may result in more than one launch while others may not result in any. In the US, we detected a total of 948 malicious URLs coming from 64 unique domains. For the Chinese deployment we detected 1,475 malicious URLs that came from 139 unique domains. We also downloaded several thousands of files of which many were simple text files or docx files. As for the number of Android applications, the US deployment collected 468 unique applications (from the Web, outside Google Play) of which 271 were found to be malicious. A large chunk (244)of these malicious applications comes from the antivirus scam reported in Section VI-A. Excluding this anomalous number of 244, we find that one in six applications downloaded from the Web (outside Google Play) are malicious. The file numbers above do not include the applications hosted on Google

Play. We accounted for such separately: there were 433,000 landing Google Play landing URLs, i.e., http URLs with play.google.com domain or URLs with market scheme (beginning with "market://"). These Google Play landing URLs led to a little over 19,000 applications on Google Play. About 5% of these labels are labeled as malicious (based on our criterion of being flagged by at least 3 anti viruses) on Virus Total. Based on our manual check of the antivirus labels, however, all of these appear to bead ware. On the Chinese deployment side, we collected 1,097 unique files of which 435 are malicious. 102 of these files are from the antivirus scam of Section VI-A. Figures 4 and 5 present the distribution of malware downloads through various ad libraries in the US deployment and in the Chinese deployment respectively. The "others" bar presents the downloads through web links not embedded in advertisements. Both the higher diversity and higher number of malicious downloads in the Chinese deployment are noteworthy. This is likely because the North American Android ecosystem is centered around Google Play and application downloads outside it are rare. However, the Chinese ecosystem depends much more on the Web and third-party Android application stores.

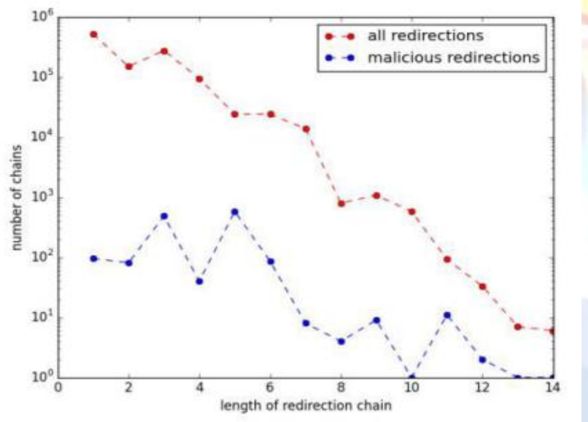

Fig. 6. Redirection Chain lengths in US Deployment

We also plot the length of redirection chains in both North American and Chinese Deployments. Note as the length of the chains increases, the two curves come closer, i.e., we have a greater fraction of malicious chains when they are longer. This was also observed by [5] and can possibly used to enhance our detection in future work.

6. CONCLUSION

In order to curb malware and scam attacks on mobile platforms it is important to understand how they reach the user. In this paper, we explored the app-web interface, where in a user may go from an application to a Web destination via advertisements or web links embedded in the application. We used our implemented system for a period of two months to study over 600,000 applications in two continents and identified several malware and scam campaigns propagating through both advertisements and web links in applications. With the provenance gathered, it was possible to identify the responsible parties (such as ad networks and application developers). Our study shows that should such as system be deployed, the users can be offered better protection on the Android ecosystem by screening out offending applications that embed links leading to malicious content as well as by making ad networks more accountable for their ad content.

Conflict of interest statement

Authors declare that they do not have any conflict of interest.

REFERENCES

[1] Smartphone OS market share, q1 2015,‖ http://www.idc.com/prodserv/ smartphone-os-market-share.jsp.

[2] Malware infected as many android devices as windows laptops in 2014,‖ http://bgr.com/2015/02/17/android-vs-windows-malware-infection /.

[3] Android phones hit by ‗ransomware',‖ http://bits.blogs.nytimes.com/ 2014/08/22/android-phones-hit-by-ransomware/? r=0.

[4] A. Zarras, A. Kapravelos, G. Stringhini, T. Holz, C. Kruegel, and G. Vigna, ―The dark alleys of madison avenue: Understanding malicious advertisements,‖ in Proceedings of the 2014 Conference on Internet Measurement Conference. ACM, 2014, pp. 373–380.

[5] Z. Li, K. Zhang, Y. Xie, F. Yu, and X. Wang, ―Knowing your enemy: understanding and detecting malicious web advertising,‖ in Proceedings of the 2012 ACM conference on Computer and Communications Security. ACM, 2012, pp. 674–686.

[6] V. Rastogi, Y. Chen, and W. Enck, ―Apps Playground: Automatic Security Analysis of Smartphone Applications,‖ in Proceedings of ACM CODASPY, 2013.

[7] Selendroid: Selenium for android,‖ http://selendroid.io/.

[8] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, ―Fast, scalable detection of piggybacked mobile applications,‖ in Proceedings of the third ACM conference on Data and application security and privacy. ACM, 2013, pp. 185–196.

[9] Symantec, ―Air push begins obfuscating ad modules, ‖November 2012, http://www.symantec.com/connect/blogs/ air push-begins obfuscating-ad-modules.

[10] V. Rastogi, Y. Chen, and W. Enck, ―Apps playground: automatic security analysis of smart phone applications,‖ in Proceedings of the third ACM conference on Data and application security and privacy. ACM, 2013, pp.\