Computer Standards & Interfaces 49 (2017) 71–78

Contents lists available at ScienceDirect

Computer Standards & Interfaces

journal homepage: www.elsevier.com/locate/csi

Mobile cloud security: An adversary model for lightweight browser security

Shasi Pokharel [a], Kim-Kwang Raymond Choo [b,a,*], Jixue Liu [a]

[a] School of Information Technology & Mathematical Sciences, University of South Australia, GPO Box 2471, Adelaide, SA 5001, Australia
[b] Department of Information Systems and Cyber Security, University of Texas at San Antonio, One UTSA Circle, San Antonio, TX 78249-0631, USA

ARTICLE INFO

Article history:
Received 7 March 2016
Received in revised form
18 August 2016
Accepted 8 September 2016
Available online 9 September 2016

Keywords:
Mobile cloud security
Lightweight browser security
UC Browser
Dolphin
CM Browser
Samsung Stock Browser

ABSTRACT

Lightweight browsers on mobile devices are increasingly been used to access cloud services and upload / view data stored on the cloud, due to their faster resource loading capabilities. These browsers use client side efficiency measures such as larger cache storage and fewer plugins. However, the impact on data security of such measures is an understudied area. In this paper, we propose an adversary model to examine the security of lightweight browsers. Using the adversary model, we reveal previously unpublished vulnerabilities in four popular light browsers, namely: UC Browser, Dolphin, CM Browser, and Samsung Stock Browser, which allows an attacker to obtain unauthorized access to the user's private data. The latter include browser history, email content, and bank account details. For example, we also demonstrate that it is possible to replace the images of the cache in one of the browsers, which can be used to facilitate phishing and other fraudulent activities. By identifying the design flaw in these browsers (i.e. improper file storage), we hope that future browser designers can avoid similar errors.

1. Introduction

In recent years, we have seen a rapid shift in Internet browsing behaviors from the use of personal computers (PCs) to mobile devices, particularly accessing cloud services and storing data in the cloud [20,45,23]. In other words, Internet browsing is increasingly being conducted on mobile devices [51]. This has also resulted in an increasing use of lightweight browsers on mobile devices.

Lightweight browsers are popular for their speedy resource loading capabilities, particularly for viewing large media files or for gaming. However, the trade-off is reduced user functionalities and weakened security mechanisms [57,58]. For example, basic browser security requirements defined by W3C [46] implemented in typical browsers, such as Google Chrome and Mozilla Firefox, may not be installed on the lightweight browsers [3].

Browsers are security sensitive applications, as they are able to access personally identifiable information (PII) and sensitive data such as bank account details. Browser communications can be targeted at various stages of the communication, such as on client devices, during network transmission, and at the server. Security issues and mitigation strategies relating to the network and the server have gained significant interest (see [14,15,52]). The security of browsers in mobile devices, however, appears to be an understudied area. For example, the question whether cache and other files are securely stored by browsers so that they cannot be accessed by unintended person or apps has not been well studied (e.g. are cache and other files encrypted or stored with the appropriate file permission?).

In this paper, we attempt to evaluate the security of user information stored by the lightweight browsers on mobile devices. Using an adversary model adapted from the security literature, we examine four popular lightweight browsers for Android device and reveal previously unpublished vulnerabilities. We regard the contributions to be two-fold:

1) An adversary model designed to study the security of lightweight mobile browser; and
2) Identification of previously unpublished vulnerabilities in four lightweight browsers.

The rest of the paper is organized as follows. Background materials and related literature are described in Sections 2 and 3, respectively. In Section 4, we present the proposed adversary model and the prototype app. The experiment setup and findings are respectively outlined in Sections 5 and 6. The last section discusses potential mitigation strategies and concludes the paper.

* Corresponding author at: Department of Information Systems and Cyber Security, University of Texas at San Antonio, One UTSA Circle, San Antonio, TX 78249-0631, USA. Tel.: +1.210.458.7876.
E-mail address: raymond.choo@fulbrightmail.org (K.-K. Choo).

http://dx.doi.org/10.1016/j.csi.2016.09.002
0920-5489/

2. Background: Mobile browsers

Browsing a webpage requires the loading of multiple sets of resources, such as HTML, CSS, JavaScript and media files. For example, according to Wang et al. [54], loading of such resources can be slower on mobile devices than on PCs due to the architectural differences and computational constraints.

Speed during website browsing is a key user concern. For example, a one second delay in webpage loading could reportedly result in 11% reduction in webpage views and 16% reduction in customer satisfaction [39]. Similar observations were echoed in the studies by Amazon [33] and Google [10].

Lightweight browsers apply client side efficiency solutions to improve the browsing speed, and consequently user's quality of experience. This includes creating a larger cache storage and avoiding any plugins that can delay the loading of web resources. Cache is the temporary storage to save downloaded web resources. If a user attempts to access a previously accessed same page or URL, the browser checks whether the content exists in the cache. If the contents exist, then the browser loads the resources from the cache; thus, saving time and network resources.

Popular browsers, such as Google Chrome, Mozilla Firefox, and Opera, use standard Web Storage to store cache data. Web Storage was introduced as a part of HTML5 and is being standardized by World Wide Web Consortium (W3C). Web Storage contains two major parts, namely: Local Storage and Session Storage, whose behavior is similar to that of persistent cookies and session cookies, respectively. Session storage stores web resources until the webpage is open. In the case of Local Storage, the generated cache remains on the device even when the browser is closed [55].

In both PC and mobile device environments, Web Storage is considered more secure than the native browser cache. According to W3C, Web Storage can be used to store sensitive user information, if implemented properly [55]. On Android devices, Web Storage typically uses the device's internal storage (e.g. /data/data/PackageName/ directory). Therefore, the items stored in these cache storage cannot be accessed by other users or apps, with the exception of the owner's app.

However, Web Storage is limited by cache size. For general use, W3C recommends the use of 5MB storage size per website, but this can be reduced when implemented on mobile devices. Lightweight browsers mostly rely on large cache storage to improve the browser's loading speed. Therefore, these browsers store large amount of cache data outside of Web Storage, often in external storage (e.g. SD card).

For Android devices, internal storage is generally considered a more secure storage location for application data, because, by default, stored data can be accessed or modified only by the creator app. In comparison, any resources, stored in external storage can be accessed, modified or deleted by any applications that have READ_ EXTERNAL_STORAGE Permission [16].

3. Related work

Web (application) security has been a research focus for a number of years [40]. Browsers for PCs, laptops and mobile devices share the underlying rules for loading webpages and communicating with servers. Therefore, existing literature on browser security tend to be focused on 'traditional' browsers (for PCs and laptops), as well as focusing on either network security or on detecting malicious websites (see [18,19,21,24,50]).

In 2014, Wadkar, Mishra and Dixit proposed the 'system call' monitoring approach to prevent information leakage from the browser [53]. System call is an interface between the browser application and Operating System (Linux) kernel, which is invoked during the execution of browser process. The researchers proposed an intermediate layer between the Kernel and the application layer that controls the system calls and filters the personal information being leaked during the browsing.

Virvilis et al. [52] evaluated the effectiveness of the Blacklist filtering approach on browsers, designed to prevent users from visiting rouge or malicious webpages. In another related study, Amrutkar et al. [2] presented a threat model, which allows the discovery of architectural weakness on mobile devices and browsers. The researchers demonstrated that attack vectors, such as display ballooning, Cross Site Request Forgery (CSRF) and clickjacking, can be used for phishing or directly stealing information from the users' device (Table 2).

More recently in 2015, Amrutkar, Traynor and van Oorschot [3] evaluated the security indicators (based on the security guidelines of W3C – [46]) used in popular mobile browsers. For example, they check to determine whether the browser displays identity of the site owner and certificate issuer and whether the browser uses the

Table 1
Lightweight browsers.

S. no.	Browser Name	Version No.	in Google Play Store downloads (in millions; as of Sep 2015)	Remarks
1.	UC Browser	10.6.2	100–500	
2.	Dolphin	11.4.19	50–100	
3.	CM Browser	5.20.06	10–50	
4.	Samsung Stock Browser	N/A	N/A	Pre-installed with Samsung mobiles, so total user number and application version cannot be identified.

Table 2
Targeted cache and file storage locations of the browsers in the study.

Browser	Targeted Cache Location	Important Contents
Dolphin	/sdcard/TunnyBrowser/cache/speeddial_covers	URLS saved as speed dial
	/sdcard/TunnyBrowser/cache/tablist_cache	screenshot image files
	/sdcard/TunnyBrowser/cache/webViewCache	All cache files (HTML, CSS, JavaScript, media)
UC Browser	/sdcard/UCDownloads/cache/	All cache files
	/sdcard/UCDownloads/config/	TrafficStatus.db; contains client server communication timing and response
	/sdcard/UCDownloads/offline/	ApplicationCache.db; contains data for cache loading management
CM Browser	/sdcard/CheetahBrowser/.data/	Browsers URL history
Samsung Stock Browser	/data/data/com.sec.android.app.sbrowser/files/	Screenshot image files

anti-phishing URL filter. A summary of the study for Google Chrome and Mozilla Firefox is presented in Table 4.

Previous studies identified cache as a potential vector to compromise a user's privacy over browser communication [47,6,8]. For example, Bernstein [7] demonstrated that an attacker can identify a user's web browsing behaviors by calculating the content loading time. Since then, several solutions have been proposed to mitigate such an attack [28,30,37].

Existing studies generally focus on protecting the users' information when their data is being transmitted over the network. Due to the popularity of mobile devices, browsers are an increasingly target for cybercriminals seeking to exploit the inherent security issues, such as exploitation of user permission and file permission. For example, Hay [22] showed how the security flaws of Opera browser can be easily exploited by an attacker to steal the user's personal information by exploiting the file permission of Opera browser's cache.

Jia et al. [26] evaluated the security strength of five desktop and 15 mobile browsers cache to determine whether they are vulnerable to Browser Cache Poisoning (BCP) in a Man in The Middle (MITM) attack. The researchers concluded that all the five desktop browsers and most of the mobile browsers examined in their study are vulnerable to such an attack. In a related work, Jia et al. [27] presented an approach to identify the mobile device user's geo location (e.g., country, city and neighborhood) by sniffing on the browser cache and by measuring the timing of browser cache queries. A similar approach was presented by Liang et al. [32], who demonstrated that user's browsing history can be hijacked using timing attack over browser cache.

Storage security in Android has also attracted the attention of researchers in recent times. In 2015, for example, Liu et al. [34] studied the data storage behaviors of Android apps. They examined the data stored by popular communication apps (i.e. Weibo, Facebook, Instagram, LINE, Skype, and Viber) to determine how much private data an attacker is able to steal from the device without the users' knowledge. The study suggests that security of users' private data is dependent on whether data is defined as sensitive or insensitive by the app designer. Data considered insensitive by the app designer will be stored in shared memory of the external storage, which is accessible to other users. However, there is no uniform definition for sensitive and insensitive data. It was demonstrated that in some cases, user's phone number, contact list and other private information could be obtained from the publicly shared files.

The study by Zhang et al. [59] also reported that data remnants left by Android apps after their uninstallation can reveal sensitive information to an attacker. In their experiment, they examined the data remnants recovered from system services and system-app service. They were able to recover sensitive information, such as users' login credential, public/private keys, URI, and Pending Intent content. Similar findings were reported in the forensic analysis of mobile apps [5,13,31,42,43,44,35,36,41,48].

Liu et al. [34] demonstrated how an attacker with the capability to install an app with WRITE_EXTERNAL_STORAGE, READ_-EXTERNAL_STORAGE and INTERNET permissions could obtain unauthorized access to files stored in public storage on Android devices. Similar to the approach of Liu et al. [34], Zhou et al. [60] used an adversary model to model an attacker's capability to steal and send data via browser using the URI ACTION_VIEW intent. However, this method (zero permission adversary model) can only transmit data when the device's screen is on or running. The use of adversary model in mobile app security is also found in the study of Do, Martini and Choo [16].

Table 3
Summary of findings.

	Browsers	Adversary Goals			
		Browser history	Searched Terms	Extract (knowledge) of web content	Change content of Cache
1	Dolphin Browser	Yes	Yes	Yes	Yes
2	CM Browser	Yes			
3	UC Browser	Yes	Yes	Yes	
4	Samsung Stock Browser	Yes		Yes	

Table 4
A comparative summary of security indicator implementations on mobile browsers.

Browser Suites [3]			
Browsers	Anti-Phishing Filter	Display of providers identity	Display of Certificate
Google Chrome	Y	Y	Y
Mozilla Firefox	Y	Y	Y
Lightweight Browsers (Result from our evaluation)			
Dolphin	**N**	Y	Y
UC Browser	**N**	**N**	**N**
Samsung Stock Brower	**N**	Y	Y
CM Browser	**Y**	Y	Y

4. Our proposed adversary model and a prototype App

4.1. Adversary model

In a recent work, Do, Martini and Choo [17] proposed the first adversary model for Android covert data exfiltration. In this model, the adversary has the capabilities to intercept, inject, modify, delete, encrypt, decrypt, transmit, and listen to user communication. In a related work, D'Orazio & Choo [12] presented an adversary model which captures the real-world capability of a digital rights management (DRM) attacker for mobile devices. In this paper, we present a weaker (and probably, more realistic) model of Do, Martini and Choo [17] and D'Orazio and Choo [12], and demonstrate that an adversary in such a model can also be used to discover and exploit vulnerabilities in lightweight browsers for Android devices.

We consider a mobile browser to be insecure if any of these goals are met.

- Goal 1: The adversary learns the URL history of the browser.
- Goal 2: The adversary learns the user's search terms.
- Goal 3: The adversary learns the content of the webpage (e.g., user's email content, bank account information).
- Goal 4: The adversary can modify the content (e.g., image file) in the cache.

4.2. Prototype App

To demonstrate the utility of our adversary model, we prototype an Android app which is able to read files and write files in the device's external shared memory (via the READ_ EXTERNAL_STORAGE and WRITE_EXTERNAL_STORAGE permissions). We remark that the READ_ EXTERNAL_STORAGE permission is granted automatically if the app is granted access to the WRITE_EXTERNAL_STORAGE permission. In order to transmit data

from a user device, the app will require the INTERNET permission. However, Android apps do not need to notify or request for user's approval for this permission because this permission is granted by default to all apps, if it is declared in manifest file. Another permission that our app requires is ACCESS_NETWORK_STATE, which allows the adversary to know the type of network the device is connected to (e.g., Wi-Fi or 3G/4G). In other words, our app will require user's approval for WRITE_EXTERNAL_STORAGE and ACCESS_NETWORK_STATE permissions.

A key challenge for any malicious app (including our app) is to avoid detection by users and malware filters. Malware filter algorithms are generally based on the evaluation of user permissions [1,4,49] and activity behavior. These algorithms consider an app as suspicious if it acquires too many permissions or permissions associated in cost incurring activities, such as sending of SMS or MMS, and making of calls (e.g., to premium numbers) [25,9,61]. Therefore, our app is designed to use few, common and non-cost incurring related permissions to avoid detection.

Excessive drainage of battery or high consumption of network bandwidth will also result in detection or raise a user's suspicion. Therefore, to avoid excessive battery power usage, our app is designed to use Android's in-built listener functions only. Similarly, to avoid excessive bandwidth usage, our app will upload user information to a third-party server controlled by the adversary, only when the device is connected to a Wi-Fi network. If the user is browsing webpages using 3 G/4 G, our app will copy the browser's cache data to the app's private folder and wait for the device to be connected to the Wi-Fi. In other words, the app runs in Wi-Fi and 4 G network.

4.2.1. Activity 1: Determining when browser starts running

Android provides multiple methods to return current running apps, and one effective method is to create a broadcast receiver for targeted browser app using ActivityManager.getRunningAppProcess(). However, this feature has been deprecated in Android API level 21. Therefore, we will use Android's native fileObserver feature, which notifies the app when changes in the target file or directory occurs. FileObserver uses inotify subsystem of Linux kernel, which extends the fileSystems to fire notifications when a given directory is accessed by the browser.

4.2.2. Activity 2: Copying of cache files

Since the Android does not have "cp" command, we will use the native InputStream and OutputStream to copy the files from the browser's cache directory (see Algorithm 2) after we have identified the files to be copied using Algorithm 1. Copied files will be stored in the prototype app's cache directory (in internal storage) to avoid raising the user's suspicion. To minimize the use of resources, we will limit the prototype app's cache directory to 5 MB and the system will automatically delete older files when this limit exceeds.

Algorithm 1. Listing of cache files.

```
File applicationCache=this.getCacheDir();
File browserCache = new File(Environment.getExternalStorage
  ().getAbsolutePath() +"/cacheDirectoryName");
String[] fileName = browserCache.list();
for (int i = 0; i < browserCache.listFiles().length; i++) {
        copyFiles(new File(browserCache, fileName[i]),
              new File(applicationCache, fileName[i]));
}
```

Algorithm 2. Copying of cache files.

```
copyFiles(File source, File dest){
InputStream toCopyFile = new FileInputStream(source);
        OutputStream copiedFile = new FileOutputStream
  (dest);
        byte[] buff = new byte[1024];
        int leng;
        while ((leng = toCopyFile.read(buff)) > 0) {
            copiedFile.write(buff, 0, leng);
}
        toCopyFile.close();
        copiedFile.close();
}
```

To improve the efficiency of our prototype app, we can avoid copying the same file more than once by implementing a 'file already exists' check.

4.2.3. Activity 3: Checking network connection and file transfer

Frequent uploading of cache files from client devices could raise suspicion due to bandwidth consumption (e.g., when the user has a limited data plan). Therefore, our prototype app is designed to check for Wi-Fi connection (see Algorithm 3).

Algorithm 3. Checking of Wi-Fi connection.

```
ConnectivityManager manager = (ConnectivityManager) get-
  SystemService(CONNECTIVITY_SERVICE);
    NetworkInfo wifiConn = connManager.getNetworkInfo
  (ConnectivityManager.TYPE_WIFI);
return wifiConn.isAvailable();
```

If a Wi-Fi connection is detected, then data stored in the private storage of our app will be uploaded to the designated server.

Uploading of files to the server can be a lengthy and resource consuming process, depending on the Wi-Fi connection. As our app runs as a service in the background, we will use 'asyncTask' to upload the files. The app will loop multiple times until the upload is completed – see Fig. 1. The files will then be deleted from the app's private storage.

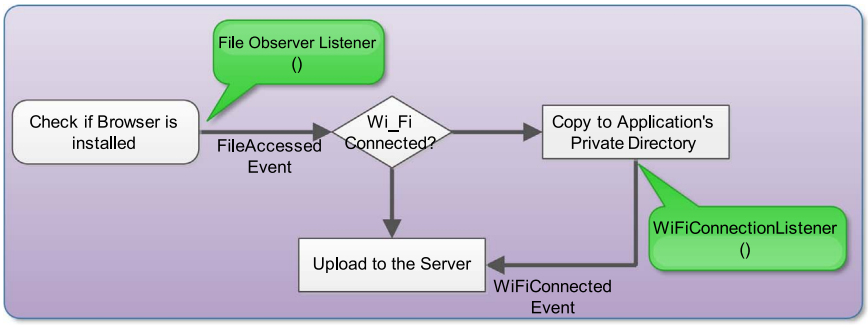

Fig. 1. Uploading of files to designated server.

4.2.4. Activity 4: Insertion of image in cache (cache poisoning)

Using the WRITE_EXTERNAL permission, our app is able to insert image to the cache directory of the browser – see Algorithms 4 and 5. Whether the browser will load the inserted image depends on a number of conditions, such as expiry time, type of cache file, and frequency of user's access to the associated URL.

Algorithm 4. Determining whether existing cache file is an image.

```
BitmapFactory.Options imageOptions = new BitmapFactory.
  Options();
    imageOptions.inJustDecodeBounds = true;
    BitmapFactory.decodeFile(file.getPath(), imageOptions);
    if(options.outWidth!= −1 && options.outHeight!= −1)
return true;
```

Algorithm 5. Replacing of image file.

```
byte[] buff = new byte[1024];
      int leng;
      while ((leng = new FileInputStream(toBeCopied).read
  (buff)) > 0) {
      toBeReplaced.write(buff, 0, leng);
      }
      toBeCopied.close();
      toBeReplaced.close();
```

5. Experiment setup

According to statistics from Google Play Store, Google Chrome, Mozilla Firefox and UC browsers are the most popular browser Android apps. Chrome and Firefox are browser suites, which have cross-platform browsing compatibility and implementation of various security measures. However, lightweight browsers are also popular with users as they have a significantly shorter resources loading speed [11], especially loading of large media files and in the gaming environment.

Therefore, in this study, we selected three most popular lightweight Android browsers, namely: Dolphin, CM Browser and UC Browser (as of September 2015, based on Google Play Store downloads). As Samsung is a popular Android model, we also included Samsung's stock browser (package name: com.sec.android.app.sbrowser) which is pre-installed on Samsung Galaxy S4 with Android version 4.4.2.

Table 1 shows the information about selected browser applications. We now briefly explain the cache storage behavior of selected browser applications.

UC browser is one of the most popular browsers for Android with more than 100 million downloads. Upon an initial inspection, we discover that the UC browser stores a large portion of cache files in external shared memory, including HTML files of visited web pages, JavaScript, CSS, image files and range of databases filled with users browsing activity. These databases not only reveals the information about which webpages were browsed by the user, but also shows the number of times the webpages were accessed, the times when they were accessed and what was the response from the server every time. It also reveals how much data (bandwidth) was used while accessing those pages individually.

Dolphin browser is one of the oldest lightweight browser introduced for Android. Initial inspection of its cache storage shows that Dolphin browser stores most of the cache resources in external shared memory in /sdcard/TunnyBrowser/cache/ directory. It has three sub-directories, namely: speeddial_covers, tablist_cache and webviewCache. As the name suggests, speed dial allows a user to save their favorite or frequently used URLs on the browser's front screen thumbnails. Speeddial_covers directory stores the URL and associated files of those websites.

We determined that Dolphin stores the screenshots of the displayed web content. For example, when a user browses more than one tab at a time, web content of the tab being viewed will be captured as a screenshot, which is around 50 KB and saved in the tablist_cache directory. All other cache files, including HTML, JavaScript, CSS and media files, are stored in the webViewCache directory.

Initial inspection indicated that CM browser stores most of the cache files in the internal memory, and we determined that Local Storage is implemented in the app_webview directory, similar to the standard cache storage proposed in HTML5. However, visited URLs are listed in a file stored in the shared memory.

In the case of Samsung stock browser, cache files, including HTML, JavaScript and media files, are stored in internal storage. Similar to the Dolphin browser, it also saves the screenshots as a bitmap file each time new content is loaded in the browser. This bitmap might be used to display the preview of running applications when user accesses the fast app switcher key in device or when user attempts to change the tabs in browser.

By default, files stored in internal memory are considered private and are only accessible by the owner app. However, this permission can be changed by the creator app, by using MODE_APPEND, MODE_WORLD_READABLE or MODE_WORLD_WRITABLE flags when creating such a file. This is considered a 'serious security flaw' by Android and removed from Android API level 17. However, such a flaw is found in Samsung's stock browser as revealed by our investigation. More specifically, we determined that the screenshot image stored by the Samsung stock browser in its cache is assigned with chmod 644 (rw-r–r–) user permission, which means it can be accessed by any other apps.

In our case study, we used Samsung Galaxy S4 (unrooted) running Android 4.4.2 version as the main experiment device. We then installed the prototype app (see next section) on the device, which runs as a background process. We also ran each browser for at least five minutes, opened multiple URLs in multiple tabs. To verify the effectiveness of our adversary model, we repeated the same experiment on a Samsung Galaxy S5 (unrooted) with Android version 5.0.

6. Findings

In this section, we describe the findings of our study.

6.1. Dolphin

We were successful in copying all files in the root directory of the cache storage (/sdcard/TunnyBrowser/Cache) and its subdirectories. The files were also successfully uploaded to our server.

An inspection of the uploaded files, we were able to retrieve the browser's URL history, view the content (including media files) of the visited webpage, etc. We were also able to recover the private content of the webpage, such as image and wall post of user's Facebook page, by accessing the link in HTML file. From the uploaded screenshot images, we were also able to obtain emails, bank account details and other sensitive user information.

We also determined that Dolphin saves the cache images without modifying the header. Generally, when a user attempts to access the same URL, the browser checks with the server cache to determine whether the stored cache has been modified. However, Dolphin was unable to identify the modified cache in our study. More specifically, we browsed Amazon homepage (https://www.amazon.com/) and determined that all product images were

stored in the cache. We then replaced several of these images with other images, which have the same filename. We immediately browsed the same page again, and we were presented with the replaced images (instead of the original images). We also note that the time between the first and second browsing was less than two minutes. Findings were similar for both devices – Samsung Galaxy S4 and Samsung Galaxy S5.

6.2. UC browser

Similar to the findings described in Section 6.1, we were successful in copying and uploading data stored in the cache root directory (/sdcard/UCDownloads). We were not able to identify the screenshots, although we were successful in recovering the web browsing history and identifying the user's browsing behavior, based on information gleaned from other cache files and databases. For example, we were able to determine what webpage was accessed at what time, and the number of visits a site was accessed. We were also able to recover the content of the accessed webpage. We found similar results for both devices.

6.3. Samsung stock browser

As previously explained in Section 5, Samsung Stock browser stores all cache resources in the device's internal storage. However, on the Samsung Galaxy S4 device, we were able to copy and upload the screenshot images from the device, once new content has been loaded in the browser. From these images, we were able to identify user's browsing activities as well as the contents of the visited sites. We were also able to recover other information such as email messages, and bank account details. As the content is stored using chmod 644 permission, we were not able to make any modification.

We found that this browser is installed on Samsung Galaxy S5 devices with Android Version 5.0 by default. However, our inspections of other Samsung mobile phones and tablets show that this browser is installed on devices with Android version 4.4.2.

6.4. CM Browser

Surprisingly, the CM browser stores all cache files, including screenshot for the multi-tabs navigation, database, HTML files and media files, on the device's internal storage in private mode. The only information we could recover was the list of URLs visited by the user, saved as a single file in /sdcard/CheeahBrowser/.data/. Similar findings were reported in both devices.

6.5. Summary

Table 3 summarizes the findings from the study of the four browsers.

Table 4 summarizes the security implementations (used in the study of [3]) of the lightweight browsers studied in this paper, as well as two other popular browser suites. It is clear that Dolphin, UC Browser and Samsung's stock browsers lack the basic security mechanisms.

7. Discussion

There are a number of commercial apps designed to track and monitor users' activities on Android devices, such as FlexySpy and MobileSpy. These apps allow the user to read browser history and bookmarks on unrooted devices, but they require the mobile device owner to grant the app permissions such as *"com.android.browser permission. READ_HISTORY_BOOKMARKS"* to access browser history and *"android.permission.ACCESS_FINE_LOCATION"* or "*android.permission.ACCESS_COARSE_LOCATION*" to track user location. These apps are not capable of taking screenshots of browser activity or access the content (e.g., email, personal account detail) unless the device is rooted and are granted additional specific permissions.

We demonstrated that our prototype app can read the targeted browsers history and content (through the cache), as well as capturing the screenshots of the Samsung's stock browser. In Dolphin browser, our app can also replace the image files in the cache and display the replaced images (instead of the legitimate web files).

Of the vulnerabilities identified in this paper, three vulnerabilities are regarded by the Open Web Application Security Project (OWASP) as the top 10 major security vulnerabilities for mobile apps [38] – see Table 5.

Insecure data storage includes the practice of storing sensitive data without encryption or protection in unsafe location. For example, in our study, we determined that the Samsung Stock browser stores screenshots in internal memory with a weak file permission. Our prototype app was able to inject malicious content in the cache storage of Dolphin browser. Therefore, this browser is vulnerable to a Client Side Injection attack.

Unintended data leakage could result from the storing of data in unsecure locations (e.g. data stored can be accessed by unauthorized person or apps). For example, the four browser apps studied in this paper store data in external shared memory, which could be accessed by other apps.

8. Conclusion and future work

In this paper, we presented an adversary model that can be used to study the security of lightweight browsers, which are a popular way of accessing cloud services. To demonstrate the practicality of the adversary model, we constructed a prototype app. Using four popular lightweight browsers as case studies, we determined that Dolphin, CM browser and UC browser store

Table 5
Top ten mobile app security vulnerabilities identified by OWASP.

Top ten OWASP mobile app security vulnerabilities	Vulnerabilities identified in the browsers			
	Dolphin	UC	CM	Samsung Stock
Weak Server Side Controls				
Insecure Data Storage	Yes	Yes	Yes	
Insufficient Transport Layer Protection				
Unintended Data Leakage	Yes	Yes		Yes
Poor Authorization and Authentication				
Broken Cryptography				
Client Side Injection	Yes			
Security Decisions Via Untrusted Inputs				
Improper Session Handling				
Lack of Binary Protections				

sensitive user information on external shared storage. Although Samsung stock browser stores sensitive user information on the device's internal memory, we demonstrated that an attacker could exploit the weak file permissions to access the stored information.

The vulnerability is due to a design flaw; that is, improper file storage by browsers. This also reinforces the need to ensure for Android's file storage and file permission system to be improved, in order to avoid unauthorized users from accessing files created by other apps. From a performance perspective, loading content from the internal memory is more efficient [29] without compromising on security. However, it appears that browsers are still designed to use external memory rather than internal memory, perhaps due to the storage size. Therefore, we do not believe that a naïve recommendation for browsers to store files in internal memory will be a viable mitigation strategy, unless the risk associated with file permission is also addressed (see [34,56]). Therefore, a short-term solution could be for browsers to store non-sensitive or large files (e.g. video clips) in external storage. However, users should also be alerted whenever these files are accessed by other apps.

Acknowledgments

The views and opinions expressed in this article are those of the authors alone and not the organizations with whom the authors are or have been associated. The authors would also like to thank the editor and the anonymous reviewers for providing constructive and generous feedback. Despite their invaluable assistance, any errors remaining in this paper are solely attributed to the authors.

References

[1] O.S. Adebayo, N.A. Aziz, Static code analysis of permission-based features for Android Malware Classification using Apriori Algorithm with Particle Swarm Optimization, J. Inf. Assur. Secur. 10 (4) (2015).
[2] C. Amrutkar, K. Singh, A. Verma, P. Traynor, VulnerableMe: Measuring Systemic Weaknesses in Mobile Browser Security, Information Systems Security, Springer 2012, pp. 16–34.
[3] C. Amrutkar, P. Traynor, P.C. van Oorschot, An empirical evaluation of security indicators in mobile Web browsers, Mobile Comput. IEEE Trans. 14 (5) (2015) 889–903.
[4] Z. Aung, W. Zaw, Permission-based Android malware detection, Int. J. Sci. Technol. Res. vol. 2 (3) (2013) 228–234.
[5] A. Azfar, K.-K.R. Choo, L. Liu, An Android communication App forensic taxonomy, J, Forensic Sciences (2016), http://dx.doi.org/10.1111/1556-4029.13164].
[6] C. Bansal, S. Preibusch, N. Milic-Frayling, Cache Timing Attacks Revisited: Efficient and Repeatable Browser History, OS and Network Sniffing', ICT Systems Security and Privacy Protection, Springer 2015, pp. 97–111.
[7] D.J., Bernstein 2005, Cache-timing attacks on AES.
[8] B.B. Brumley, 'Cache Storage Attacks', Topics in Cryptology-–CT-RSA, Springer 2015, pp. 22–34.
[9] I. Burguera, U. Zurutuza, S. Nadjm-Tehrani, Crowdroid: behavior-based malware detection system for android, in: Proceedings of the 1st ACM workshop on Security and Privacy in Smartphones and Mobile Devices, ACM, 2011, pp. 15–26.
[10] L. Bustos, Speed Kills Conversion Rates, 2012 <⟨http://www.getelastic.com/site-speed-infographic/⟩>.
[11] P. Chris, Best Android browsers, 2015 edition: speed, features, and design, updated 8 April 2015, Phonearena.com, viewed 12 September 2015, <⟨http://www.phonearena.com/news/Best-Android-browsers-2015-edition-design-features-and-performance_id67848⟩>.
[12] C. D'Orazio, K.-K.R. Choo, An adversary model to evaluate DRM protection of video contents on iOS devices, Comput. Secur. 56 (2015) 94–110.
[13] D. Daryabar, A. Dehghantanha, B. Eterovic-Soric, K.-K.R. Choo, Forensic Investigation of OneDrive, Box, GoogleDrive and Dropbox Applications on Android and iOS Devices, Aust. J. Forensic Sci. (2016), http://dx.doi.org/10.1080/00450618.2015.1110620].
[14] M.L. Das, N. Samdaria, On the security of SSL/TLS-enabled applications, Appl. Comput. Inf. 10 (1) (2014) 68–81.
[15] L. Desmet, M. Johns, Real-time communications security on the web, Internet Computing, IEEE 18 (6) (2014) 8–10.
[16] Q. Do, B. Martini and K.-K.R. Choo, Enforcing File System Permissions on Android External Storage. In Proceedings of 13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 2014), IEEE Computer Society Press, 2014, pp. 949–954.
[17] Q. Do, B. Martini, K.-K.R. Choo, Exfiltrating data from Android devices, Comput. Secur. 48 (2015) 74–91.
[18] W. Du, L. Yang, J. Kizza & X. Yuan, New hands-on labs on browser security, in: Proceedings of the 45th (ACM) Technical Symposium on Computer Sience Education, (ACM), 2014, pp. 717–717.
[19] A.P. Felt, R.W. Reeder, H. Almuhimedi & S. Consolvo, 2014, Experimenting at scale with google chrome's SSL warning, in: Proceedings of the 32nd Annual ACM Conference on Human factors in Computing Systems, ACM, pp. 2667–2670.
[20] X. Fu, X. Sun, Q. Liu, L. Zhou, J. Sh, Achieving efficient Cloud search services: multi-keyword Ranked Search over Encrypted Cloud data supporting parallel computing, IEICE Trans. Commun. E98-B (1) (2015) 190–200, 2015.
[21] M. Hanif, M.S. Vighio, Z. Hussain, N.A. Memon, Comparative Study of Top-Ranked Web Browsers, Bahria Univ. J. Inf. Commun. Technol. 8 (1) (2015) 93.
[22] R., Hay, Opera Mobile Cache Poisoning XAS, September 2011.
[23] D. He, D. Zeadally, L. Wu, 'Certificateless public auditing scheme for cloud-assisted wireless body area networks, IEEE Syst. J. (2016), http://dx.doi.org/10.1109/JSYST.2015.2428620.
[24] C. Hothersall-Thomas, S. Maffeis & C. Novakovic, BrowserAudit: automated testing of browser security features, in: Proceedings of the 2015 International Symposium on Software Testing and Analysis, ACM, 2015, pp. 37–47.
[25] T. Isohara, K. Takemori & A. Kubota, Kernel-based behavior analysis for android malware detection, Computational Intelligence and Security (CIS), 2011 Seventh International Conference on, IEEE, 2011, pp. 1011–1015.
[26] Y. Jia, Y. Chen, X. Dong, P. Saxena, J. Mao, Z. Liang, 'Man-in-the-browser-cache: Persisting HTTPS attacks via browser cache poisoning', Comput. Secur. 55 (2015) 62–80.
[27] Y. Jia, X. Dong, Z. Liang, P. Saxena, 'I know where you've been: Geo-inference attacks via the browser cache', Internet Computing, IEEE 19 (1) (2015) 44–53.
[28] E. Käsper, P. Schwabe, 'Faster and timing-attack resistant AES-GCM', Cryptographic Hardware and Embedded Systems-CHES 2009, Springer 2009, pp. 1–17.
[29] Kim, H., Agrawal, N. & Ungureanu, C. 2011, 'Examining storage performance on mobile devices', in: Proceedings of the 3rd ACM SOSP Workshop on Networking, Systems, and Applications on Mobile Handhelds, ACM, p. 6.
[30] R. Könighofer, 'A fast and cache-timing resistant implementation of the AES', Topics in Cryptology–CT-RSA 2008, Springer 2008, pp. 187–202.
[31] M.D. Leom, K.-K. R. Choo and R. Hunt, Remote wiping and secure deletion on mobile devices: A review. Journal of Forensic Sciences, 2016 (In press).
[32] Liang, B., You, W., Liu, L., Shi, W. & Heiderich, M. 2014, 'Scriptless timing attacks on web browser privacy', Dependable Systems and Networks (DSN), 2014 44th Annual (IEEE)/IFIP International Conference on, (IEEE), pp. 112–123.
[33] G., Linden 2006, Make data useful.
[34] X. Liu, Z. Zhou, W. Diao, Z. Li, K. Zhang, 'An Empirical Study on Android for Saving Non-shared Data on Public Storage', ICT Systems Security and Privacy Protection, Springer 2015, pp. 542–556.
[35] B. Martini, K.-K.R. Choo, Cloud Storage Forensics: ownCloud as a Case Study, Digit. Investig 10 (4) (2013) 287–299.
[36] F. Norouzi, A. Dehghantanha, B. Eterovic-Soric, K.-K.R. Choo, Investigating Social Networking Applications on Smartphones: Detecting Facebook, Twitter, LinkedIn, and Google+ Artifacts on Android and iOS Platforms, Aust. J.4 Forensic Sci. (2016), http://dx.doi.org/10.1080/00450618.2015.1066854].
[37] D.A. Osvik, A. Shamir, E. Tromer, 'Cache attacks and countermeasures: the case of AES', Topics in Cryptology–CT-RSA 2006, Springer 2006, pp. 1–20.
[38] OWASP 2014, OWASP Mobile Security Project: Top 10 Mobile Risk, <⟨https://www.owasp.org/index.php/OWASP_Mobile_Security_Project#tab=Top_10_Mobile_Risks⟩>.
[39] The performance of Web Application 2008, Aberdeen Group. <⟨http://v1.aberdeen.com/launch/report/research_report/5136-RR-performance-web-application.asp⟩>.
[40] V. Prokhorenko, K.K.R. Choo, H. Ashman, Web application protection techniques: a taxonomy', J. Netw. Comput. Appl. 60 (2016) 95–112.
[41] D. Quick, K.-K.R. Choo, Google drive: forensic analysis of Cloud Storage Data Remnants, J. Netw. Comput. Appl. 40 (2014) 179–193.
[42] D. Quick, K.-K.R. Choo, Digital Droplets: Microsoft SkyDrive forensic data remnants, Future Gener. Comput. Syst. 29 (6) (2013) 1378–1394.
[43] D. Quick, K.-K.R. Choo, Dropbox Analysis: Data Remnants on User Machines, Digit. Investig. 10 (1) (2013) 3–18.
[44] D. Quick, K.-K.R. Choo, Forensic collection of cloud storage data: does the act of collection result in changes to the data or its metadata? Digit. Investig. 10 (3) (2013) 266–277.
[45] Y. Ren, J. Shen, J. Wang, J. Han & S. Lee, Mutual Verifiable Provable Data Auditing in Public Cloud Storage,' Journal of Internet Technology, vol. 16, 2, pp. 317-323.
[46] T., Roessler & A., Saldhana, Web Security Context: User Interface Guidelines, 2010 <⟨http://www.w3.org/TR/wsc-ui/⟩>.
[47] V. Saraswat, D. Feldman, D.F. Kune, S. Das. Remote cache-timing attacks against AES' Proceedings of the First Workshop on Cryptography and Security in Computing Systems, ACM, 2014, pp. 45–48.
[48] M. Shariati, A. Dehghantanha, K.-K.R. Choo, SugarSync forensic analysis, Aust. J. Forens. Sci. 48 (1) (2016) 95–117.
[49] K.A. Talha, D.I. Alper, C. Aydin, APK Auditor: permission-based Android

malware detection system, Digit. Investig. 13 (2015) 1–14.
[50] N. Tsalis, N. Virvilis, A. Mylonas, T. Apostolopoulos, D. Gritzalis, Browser Blacklists: A Utopia of Phishing Protection, Springer, 2015.
[51] The U.S. Mobile App Report 2014, ComScore.
[52] N. Virvilis, A. Mylonas, N. Tsalis, D. Gritzalis, Security busters: web browser security vs. rogue sites, Comput. Secur. 52 (2015) 90–105.
[53] H. Wadkar, A. Mishra, & A. Dixit, 2014, 'Prevention of information leakages in a web browser by monitoring system calls', Advance Computing Conference (IACC), 2014 IEEE International, IEEE, pp. 199–204.
[54] Z. Wang F.X. Lin L. Zhong M. Chishtie. 'Why are web browsers slow on smartphones? in: Proceedings of the 12th Workshop on Mobile Computing Systems and Applications, ACM 2011 91 96.
[55] Web Storage, W3C (2015).
[56] D., Wu & R.K., Chang 2011, Indirect File Leaks in Mobile Applications.
[57] Y. Yang, H. Cai, Z. Wei, H. Lu and K.-K.R. Choo, 2016. Towards lightweight anonymous entity authentication for IoT applications, in: Proceedings of 21st Australasian Conference on Information Security and Privacy - ACISP 2016, Melbourne, Australia, Volume 9722/2016 of Lecture Notes in Computer Science (pp. 265–280), Springer-Verlag, 4–6 July.
[58] Y. Yang, J. Lu, K.K.R. Choo and J. Liu, 2015. On lightweight security Enforcement in Cyber-physical Systems, in: Proceedings of International Workshop on Lightweight Cryptography for Security & Privacy (LightSec 2015), Bochum, Germany, Volume 9542/2016 of Lecture Notes in Computer Science (pp. 97–112), Springer-Verlag.
[59] X., Zhang, K., Ying, Y., Aafer, Z., Qiu & W., Du, Life after App Uninstallation: Are the Data Still Alive? Data Residue Attacks on Android', NDSS, 2016.
[60] X. Zhou S. Demetriou D. He M. Naveed X. Pan X. Wang C.A. Gunter K. Nahrstedt. 'Identity, location, disease and more: Inferring your secrets from android public resources', in: Proceedings of the 2013 ACM SIGSAC conference on Computer communications security ACM, 2013, pp. 1017–01028.
[61] Y. Zhou, X. Jiang, X. 2012, 'Dissecting android malware: Characterization and evolution', Security and Privacy (SP), 2012 IEEE Symposium on, IEEE, pp. 95–109.