Accessing Secure Data on Android Through Application Analysis

Richard Buurke and Nhien-An Le-Khac(✉)

University College Dublin, Belfield, Dublin 4, Ireland
richard.buurke@ucdconnect.ie, an.lekhac@ucd.ie

Abstract. Acquisition of non-volatile or volatile memory is traditionally the first step in the forensic process. This approach has been widely used in mobile device investigations. However, with the advance of encryption techniques applied by default in mobile operating systems, data access is more restrictive. Investigators normally do not have administrative control over the device, which requires them to employ various techniques to acquire system data. On the other hand, application analysis is widely used in malware investigations to understand how malicious software operates without having access to the original source code. Hence, in this paper, we propose a new approach to access secure data on Android devices, based on techniques used in the field of malware analysis. Information gained through our proposed process can be used to identify implementation flaws and acquire/decode stored data. To evaluate the applicability of our approach, we analysed three applications that stored encrypted user notes. In two of the applications we identified implementation flaws that enabled acquisition of data without requiring elevated privileges.

Keywords: Android · Mobile device forensics · Application analysis · Secure data acquisition

1 Introduction

Mobile devices are becoming an increasingly important source of information in criminal investigations. These devices potentially store information about contacts, call logs, location history, images and other data which might be relevant to an investigation. Current forensic solutions aimed at mobile devices are primarily focused on post-mortem investigations. Often they are able to create a physical image of a device but are unable to process encrypted data from unsupported applications. This problem is quickly becoming more relevant since the use of encryption has become more widespread. In literature, most data acquisition techniques have their caveats in a practical environment. They are for example difficult to execute, require elevated privileges or are device specific [9,24,28,29].

Besides, application analysis is widely used in the domain of malware analysis to understand how malicious software operates without having access to the original source code. It uses a combination of behavioural and static analysis to

P. Gladyshev et al. (Eds.): ICDF2C 2021, LNICST 441, pp. 93–108, 2022.
https://doi.org/10.1007/978-3-031-06365-7_6

map the characteristics of an application. Therefore, in this paper, we propose a new approach based on this technique to assist the investigator acquire data where traditional solutions are unable to. Application analysis is relatively easily to perform, will work cross-platform and does not require elevated privileges. It also seems to be practically viable since improper API usage is seen as a common security threat on mobile operating systems [8,14,18,21]. In this study, we have limited the scope of our research to the Android operating system. The general approach should however be transferable to other mobile operating systems.

To evaluate the applicability of our approach, we analysed three applications that focus on protecting user's data. In two of the applications we identified implementation flaws that enabled acquisition of data without requiring elevated privileges. We also show that our methodology can be applied to a wide range of applications. By looking at our experimental results, combined with the available literature, we expect other applications to exhibit the same behaviour. We have proven that by analysing individual applications it is possible to acquire application data without the need for elevated privileges.

The main contribution of this research can be listed as follows:

- We performed an extensive literature survey to identify Android security mechanisms and various data acquisition techniques.
- We propose an application-analysis based approach for data acquisition. Our approach can be used by forensic investigators to get a global understanding of the application features and identify possible opportunities for data acquisition.
- We evaluate our approach to demonstrate its feasibility with three popular applications that focus on securely storing user notes: Safe Notes, Private Notepad and Secure Notes.

The rest of this paper is organised as follows: Sect. 2 provides background information on related work on Android data acquisition. Section 3 describes our approach for Android application analysis, then we present our experiment results in Sect. 4. We discuss our results in Sect. 5 and finally Sect. 6 concludes this paper.

2 Related Work

Smartphone evidence acquisition is a challenge for investigators [1]. In this section, we review various methods for data acquisition on Android devices. The results of our review and comparison of the various methods are summarised in Table 1, where the first column lists relevant characteristics used to compare these methods.

Binary exploitation and CPU specific vulnerabilities can lead to full control over the device or escalation of privileges [7,22]. However these attacks are difficult to perform and binary exploitation depends on specific versions of software to be present on the device. The same is true for exploiting the Trusted Execution Environment (TEE), although common exploit mitigations are often

missing in the secure world [3]. Another obstacle is the lack of publicly available basic tools such as a debugger for the secure world.

A cold boot attack requires flashing a custom recovery ROM onto the device and therefore needs an unlocked bootloader [25]. A copy of volatile memory can be created after quickly rebooting the device. Current versions of Android utilise the TEE for cryptographic operations, which makes it more complex to recover the disk encryption key [19]. Several other mitigations against cold boot attacks have been created but these are not currently implemented [13,20,30].

An evil maid attack is less difficult to execute and can also result in full control over the device. However this technique, just as the cold boot attack, requires an unlocked bootloader [12]. This makes it unlikely to be applicable in most real world scenarios. Using a duplicate device to trick the suspect into entering their code is still viable. Although it can also only be applied to devices for which the bootloader can be unlocked since a custom ROM is still required.

In the last few years we have seen various CPU specific attacks such as Spectre [17], DRAMMER [27] and TRRespass [11]. The aforementioned techniques can be leveraged to alter or leak information from volatile memory and affect the ARM architecture, which is used in the vast majority of smartphones [16]. This category of vulnerabilities enables the attacker to read or write arbitrary memory without the need for elevated privileges or existing software vulnerabilities. The lack of standardised tools and the required specialised skills make these techniques difficult to employ for the average forensic investigators.

Memory forensics has the most potential for gathering sensitive information, such as decrypted data and login credentials [4,6]. But it almost always requires elevated privileges on the device which must be obtained through exploitation [28]. Abusing the update protocol of a device can circumvent this restriction [29]. This technique requires a device of a specific brand. Also these vulnerabilities might be patched and public tools are unavailable. Migrating data using vendor specific tools can help to acquire a partial memory dump [9]. However it will not be possible to acquire data which only exists in volatile memory on the suspects device.

Network traffic analysis can be used to identify servers hosting information of interest. This information can then be acquired through legal procedures. In the vast majority of cases network traffic will be encrypted [15]. This will make it impossible to capture the plain text contents of packets without having elevated privileges on the device [10]. If the target application uses unencrypted connections, the investigator can acquire the data exchanged between the client and the server. Encrypted connections however still provide the investigator with relevant meta-data [6].

Authors in [2] proposed a wireless extraction of data for Android devices. However, it's only used for the logical extraction of the data from the Android file-system.

Table 1. Comparison of acquisition techniques (✓: Yes, X: No) - R: Required to apply this method; E: This method enables the investigator to do the following; P: A property of this method

		Improper API usage	Binary exploitation	TEE exploitation	Memory forensics					Cold boot attack	CPU Vulnerabilities		Evil maid attack		Network forensics
					Direct access	Loadable kernel modules	Kernel code injection	Update protocols	Data Migration		Fault induction	Branch prediction	Target device	Duplicate device	
R	Elevated privileges	X	X	X	✓	✓	✓	X	X	X	X	X	X	X	✓[a]
R	Unlocked bootloader	X	X	X	X	X	X	X	X	✓	X	X	✓	✓	X
R	Local code execution	X	✓[b]	✓	✓	✓	✓	X	X	X	✓	✓	X	X	X
R	Custom ROM	X	X	X	X	X	X	X	X	✓	X	X	✓	✓	X
E	Information leakage	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
E	Full control	X	✓[c]	✓[c]	N/A	N/A	N/A	X	X	X	✓[c]	✓[c]	✓[d]	✓[d]	X
E	Acq. non- volatile memory	Partial	✓[c]	✓[c]	N/A	N/A	N/A	X	✓	✓[d]	✓[c]	✓[c]	✓[d]	✓[d]	Partial
E	Acq. volatile memory	Partial	✓[c]	✓[c]	✓	✓	✓	✓	Partial	✓	✓[c]	✓[c]	X	X	Partial
P	Difficulty	Med	High	High	Low	Med	Med	High	Low	Med	High	High	Med	Med	Low
P	Cross-platform	✓	X	X	✓	X	X	X	X	X	✓	✓	X	X	✓
P	Tools available	✓	✓	X	✓	✓	✓	X	✓	✓	✓	✓	✓	✓	✓

[a] Only required for decrypting encrypted network traffic on the target device
[b] Some exploits can be triggered remotely
[c] Different types of exploits exist, ranging from information leaks to full control over the target device
[d] Since the bootloader is unlocked, the investigator has full access to the device when the password is obtained

Improper API usage is seen as the number one category of vulnerabilities on the Android platform by the OWASP community [21]. By analysing applications and identifying these vulnerabilities it is possible to access sensitive information without the need for elevated privileges. This technique also works cross-platform and is less difficult to execute than techniques that rely on the binary exploitation. Application analysis generally will not result in local code executing or full control over the device and is usually limited to a single application.

3 Proposed Approach

To detect improper API usage in Android applications we propose an application analysis approach consisting of four phases (Fig. 1). Our approach combines concepts from the forensic field (repeatability) with concepts from the field of malware analysis (behavioural and static analysis). Its four phases can be summarised as in Table 2.

Table 2. Four phases for detecting improper API usage

Phase 1: Preparation	Reset the test device to a pre-defined state and check time and date settings for any deviations
Phase 2: Installation	Install the target application and start a network capture using a proxy or by hooking networking functions.
Phase 3: Behavioural analysis	Interact with the target application and collect relevant artifacts for later analysis. Capture volatile memory of the target process during it's lifetime. Copy the artifacts and base APK file to the investigative machine.
Phase 4: Static analysis	Examine phase 3 preliminary results and identify Points of Interests (POIs). Use the checklist (Table 3) to identify additional POIs in the manifest file and code base.
Repeat	If there are uncovered scenarios then hook any functions of interest and restart the network capture and behavioural analysis

The investigator should have access to a test device, which supports the target application. The test device should be rooted that usually requires that the bootloader can be unlocked. The device would then be configured to include any required tools (e.g. Frida/gdbserver/LiME) and self-signed CA certificates. A bit-for-bit image of the device can be created using a custom recovery ROM. The device should be reverted to an initial state before analysis commences by re-flashing the backup image. This ensures repeatability of the experiment. The target application is installed in phase two and networking interception techniques are employed to capture any network (meta-)data.

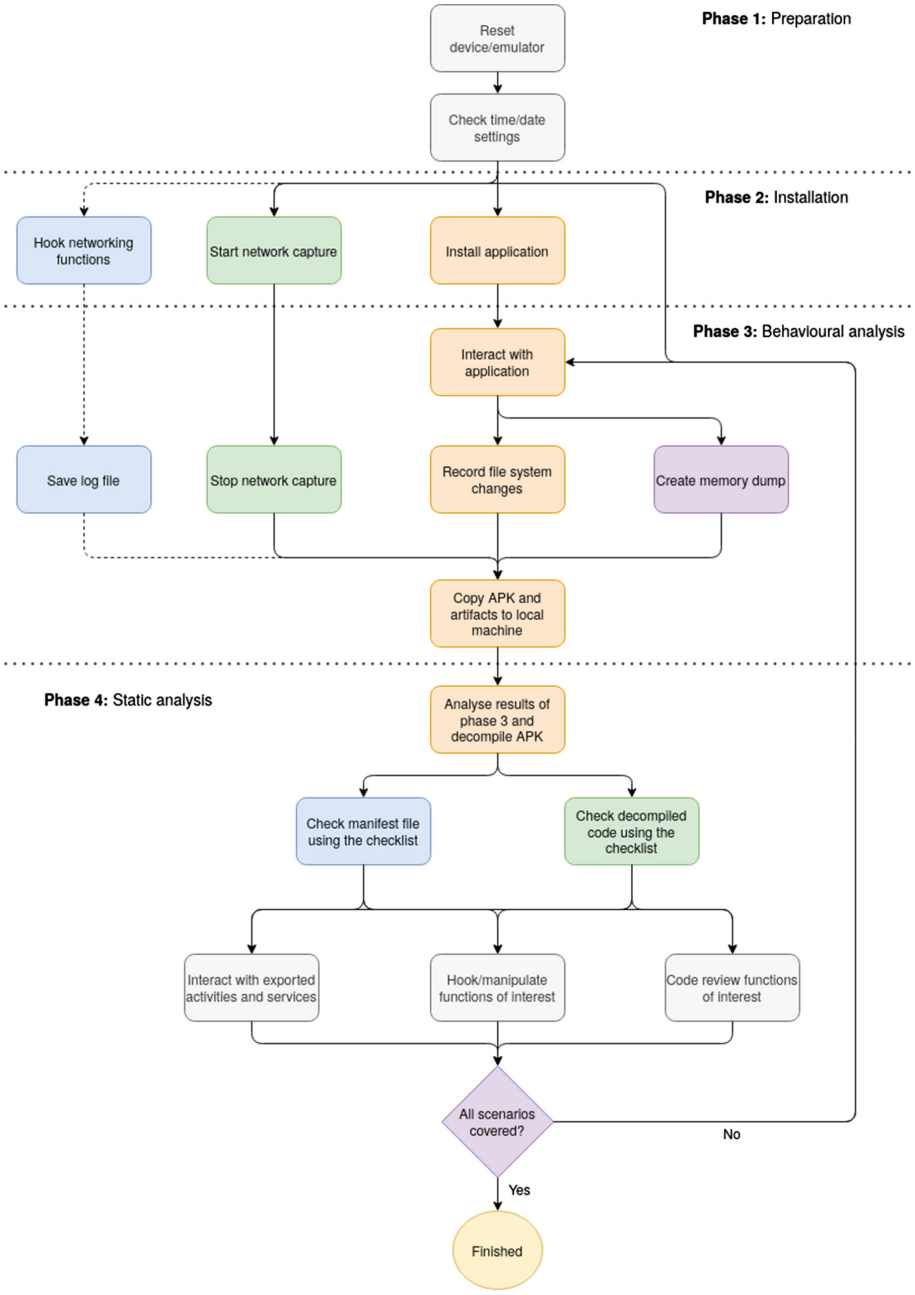

Fig. 1. Workflow for application analysis on Android

Dynamic analysis is a technique whereby we monitor the interaction between the application and other components such as the file system or remote servers. We first create a snapshot of the types of artifacts required for analysis, which will serve as our baseline. Then we run the application, perform some interactions and create another snapshot. Finally we compare the baseline with the application snapshot and determine which artifacts were created by our application. In our approach, we focus on the following locations:

- **File system** - We capture changes made to the file system by using standard Unix utilities.
- **API calls** - Systems calls made by the application are monitored using *strace* (if SELinux can be disabled) or they are hooked using *Frida*.
- **Volatile memory** - Application memory is captured by reading the */proc/<pid>/mem* block device. This process is automated by using the *PMDump* utility. We also use the GNU debugger to monitor live changes in process memory.
- **Network traffic** - Network traffic is captured through an HTTPS proxy using a self signed CA certificate. Analysis is performed using Wireshark and Moloch.

An example of behavioural analysis is determining file system changes using basic Unix tools such as "find". This enables identification of files that have been created or modified by the target application or operating system (Fig. 2):

```
FILE=$(mktemp); echo "Press any key to print file system changes ..."; read;
↪  find /data -newer $FILE
```

Fig. 2. Detect newly created or modified files in the /data folder

In Phase 4, we employed the well known tool "Frida" to hook or manipulate any functions of interest, which were detected using static analysis. This enables the investigator to follow the flow of data as the program runs or to trigger specific functionalities (such as a enabling a hidden debug menu). Static analysis is the process of analysing the compiled source code of an application. This can be achieved by using a decompiler such as "Jadx". Table 3 contains a non-exhaustive list of artifacts of interest. These artifacts are a great starting point for our static analysis. The manifest file of an Android application describes it's capabilities and is therefore also a valuable source of information.

The static analysis is aimed at identifying new code paths and to better understand previously observed behaviour. The behavioural analysis procedure should be repeated to trigger and analyse newly discovered functionality. The static analysis phase can be considered completed when no new code paths are identified. Figure 1 on page 6 contains a graphical representation of the procedure described in this section.

Table 3. Application analysis checklist - A: API Call; P: Permission; D: Directory; F: File; M: Manifest entry; V: Volatile memory; N: Network traffic

Artifacts of interest		Description
A	.getExternalFilesDir()	Data is possibly stored in publicly accessible locations
A	.getExternalCacheDir()	
P	READ_EXTERNAL_STORAGE	
P	WRITE_EXTERNAL_STORAGE	
D	/sdcard	
A	.bindService()	The application might expose a public interface which could allow extraction of data or exploitation
A	.onStartCommand()	
A	.onBind()	
A	LocalSocket()	
A	.bind()	
M	<activity name=“ exported”=“true” />	The application exports an activity that can be started externally
A	MessageDigest.getInstance()	The application uses cryptographic operations. Check the hashing or encryption algorithm and determine if the developer uses static keys
A	Cipher.getInstance()	
A	Cipher.init()	
A	.digest()	
A	.doFinal()	
A	KeyStore.getInstance()	Encryption keys are possibly stored in a trusted execution environment (TEE)
A	KeyChain()	
A	isInsideSecureHardware()	
A	setIsStrongBoxBacked()	
A	.load()	
A	.getEntry()	
V	Encryption keys	Volatile memory can hold sensitive information such as decryption keys, login credentials, etc.
V	Login credentials	
V	Traces of unencrypted data	
A	Socket()	The application uses internet functionality. This could be an indication of cloud storage
A	HttpURLConnection()	
A	.connect()	
P	INTERNET	
A	FirebaseFirestore.getInstance()	The application uses Google Firebase as a backend. Check if a Mutual Legal Assistance Treaty (MLAT) exists
A	FirebaseAuth.getInstance()	
A	FirebaseStorage.getInstance()	
A	FirebaseDatabase.getInstance()	
N	Hostnames/IP-addresses	Network meta-data can identify servers that store application data. Check for use of TLS encryption
N	Authentication tokens	
N	User credentials	
N	Decrypted data	
A	ContentProvider()	The application provides a custom content provider which could be used to extract data
A	.query(); .insert(); .delete(); .update()	

4 Experiment Results

To validate our approach we tested it against three applications that were specifically designed to securely store user notes. We selected our target applications based on the position in the list of recommended applications, rating, approximate number of downloads and the total number of ratings (Table 4).

Table 4. Top three "secure notes" applications in the Google Play Store

Application	Position in recommendations	Rating	Approximate no. downloads	No. ratings
Safe notes	1	4.5	>100,000	8,239
Private notepad	3	4.5	>1,000,000	35,706
Secure notes	4	4	>100,000	3,486

To get an initial idea of the possible attack vectors we compared the functionality of the selected applications based upon their description. We limited our research to functionality which was provided free of charge (Table 5).

Table 5. Application features overview

	Safes notes	Private notepad	Secure notes
Encryption algorithm	AES	AES	AES[a]
Cloud synchronization	✓	✓	✓[b]
Multiple device synchronisation	✓	✓	✓[b]
PIN lock	✓	✓	X
Pattern lock	X	✓	✓
Fingerprint lock	X	✓	X
Password recovery	X	X	✓
Image support	X	✓	X
Intruder image	X	✓	X
Self destruct	X	✓	X
Data hiding	X	✓	X

[a] The application advertises using AES symmetric encryption but actually implements DES

[b] Online features did not function

For our experiment we used a Samsung SM-935F device with Android version 8.0, build number R16NW.935FXXS6ESI4 and root access. The device also uses a TWRP recovery image which is used to make a byte-for-byte copy of the system and user data partitions. Before we start our analysis procedure we will first restore the device backup. This ensures that the initial state of the device is the same for every application analysed. We then install the application and populate it with data. For every experiment we will be using the same dataset to ensure repeatability and consistency. Since we are creating notes we chose to define three standard messages:

- Message 1: This is just a test message!
- Message 2: How is this data stored?
- Message 3: I'm encrypted. Nothing to see here.

4.1 Safe Notes

Using behavioural analyses we determined that notes are stored in the directory */data/data/com.protectedtext.android/shared_prefs/* and use the naming convention *com.protectedtext.n[note-number].xml*. They are encrypted by default using one of the following static keys which are hard coded into the application:

1. **7igb2h048io6fyv8h92q3ruag09g8h** + <note-number>
2. **7igb2h048io6fyv98hasdfil09g8h** + <note-number>

If a user password is specified then it is used to encrypt the note contents. The program uses AES/CBC/PKCS5 as the encryption algorithm. The decryption password is stored together with the encrypted text when the note is "unlocked". If the note is "locked" the stored decryption password is deleted. We also analysed the mandatory PIN protection mechanism and identified the hard coded debug PIN code **556711** using static analysis which displays a message in the following format:

```
debug-[prefHintColor]-[prefHintColor2]-[debugPinCodeInHex]-[SHA512OfDebugPin]
```

The fields *prefHintColor* and *prefHintColor2* are two random groups of six bytes extracted from the SHA512 hash of the user PIN code. This provides a high chance of hash collisions since we only need to calculate a SHA512 hash that contains these bytes. We created a script that can calculate a valid hash for the PIN code **123456** in less then a second. The application also enables the user to store their notes on the website *protectedtext.com* using a custom path. If the URL portion is known the notes can be downloaded using the following URL:

```
https://www.protectedtext.com/<custom-path>?action=getJSON
```

When the note is downloaded from the protectedtext website it is in a "locked" state. As a proof of concept (PoC) we wrote a multi-threaded application, which performs a dictionary attack on the encrypted note. Currently it can try approximately 87.500 passwords per second for a 20000 byte long note using all threads on an Intel i7 4790 processor running at 3.6 GHz. Performance can easily be improved, for example by utilizing the GPU.

4.2 Private Notepad

Using behavioural analysis we determined that the main database is located at */data/data/ ru.alexandermalikov.protectednotes/databases/notes.db*. It contains a SHA-1 hash of the user password, passcode and pattern lock. The key needed to decrypt the contents of a note is stored in the file */data/data/ ru.alexandermalikov.protectednotes/shared_prefs/protected_notes_preferences.xml* as the "encryption_key" variable.

The application uses AES/CBC/PKCS7 as the encryption algorithm with an empty initialization vector (IV). It first decrypts the key stored in *protected_notes_preferences.xml* using the Base64 encoded static value **4WJFtkwwUJqTHd+dJNtAaw==** as the decryption key. The resulting value is then used to decrypt the contents of the user notes.

Private notepad uses a Google Firebase backend with Google remote procedure calls (gRPC) over the HTTP/2 protocol. Since our software for analysing network traffic did not support this protocol we used Frida to hook API calls and monitor the flow of data between the client and server. Through this method we observed that the notes are decrypted before being sent to Google Firebase. This means that the developer has the possibility to decrypt this information. Since the information is stored on servers from Google it might be possible to obtain it through a Mutual Legal Assistance Treaty (MLAT) request.

Any documents retrieved from the remote database were also stored unencrypted in a local cache database. In our case they were stored in the file */data/data/ru.alexandermalikov.protectednotes/databases/firestore.%5BDEFAULT%5D.private-notepad-bd4a4.%28default%29.*

4.3 Secure Notes

Secure Notes stored its notes in the folder */sdcard/.innorriorsnotes/* which is publicly accessible by the user and other installed applications (that target API level 29 or lower [23]). The file */data/data/com.inno.securenote/databases/securenotepad* contains the password used for access control. If a pattern lock is enabled then it is stored in the file */data/data/ com.inno.securenote/files/pattern* as a byte array hashed with the SHA-1 algorithm.

The application uses the DES encryption algorithm although it advertises to use RSA. According to an article written by P. Zande in 2001 [26] for the SANS institute, the DES algorithm uses a 56-bits encryption key and is considered unsuitable for encrypting sensitive data. During the RSA Data Security Conference of 1999 it proved possible to break DES encryption within 28 h using distributed computing. This means that it should be feasible to decrypt any stored data without the need for knowing the password. However during the static analysis phase we discovered that the application uses the static password **kalandarkalandar** to decrypt any note stored on the file system.

When the user logs into the application any new or modified notes are sent to the URL "http://52.86.98.234/innorriors_securenotes/index.php" via a POST request, this URL is hard coded into the application. This even occurs without the user having enabled the synchronization feature of the application. The request contains the contents of the note and meta-data such as a timestamp, local filename and category. However it does not contain a user identifier, which means that any legitimate synchronisation features would be unable to function since the note is not associated with a user account. It is unclear if this is a bug or some form of malicious behaviour. At the time of writing it was not possible to create a working account for this application.

The login screen of the application includes a password recovery option. When activated the application sends the password and e-mail address to a script on the remote server. This scripts then sends an e-mail message containing the password to the specified e-mail address. If an investigator activates this feature he will be able to recover the password by acquiring network traffic since the connection is unencrypted. This also provides a method for a malicious user to send a legitimate looking phishing mail. The e-mail is sent from the mail address support@innorriors.com but the salutation and name of the app are supplied by the user. This enables an attacker to craft their own message originating from a legitimate address. The attacker can also specify the recipient of the message since it is part of the POST request to the server.

4.4 Memory Artifacts

Operations performed by an application, such as decryption or encryption of data, uses volatile memory to store results and intermediate values. If an investigator is able to capture this memory, he is able to acquire artifacts which are otherwise inaccessible. The following is a list of artifacts recovered from volatile memory during our experiments (Table 6):

Table 6. Recovered memory artifacts per application

	Safe notes	Private notepad	Secure notes
Decryption password	✓	✓	✓
Decrypted text	✓	✓	✓
Decrypted images	n/a	✓	n/a
Decrypted labels	n/a	✓	n/a
PIN code screen lock	✓	✓[a]	n/a
Pattern screen lock	n/a	✓	✓
Password screen lock	n/a	✓	✓
Cloud storage URLs	✓	✓	✓
Timestamps	✓	✓	✓
Filenames	✓	✓	✓
E-mail addresses	n/a	✓	✓
Account password	n/a	✓	n/a[a]
User ID	n/a	✓	n/a[b]

[a] The PIN code was was encountered multiple times since it was a 4 digit value. Therefore it is likely, but not conclusive, that one of these values was the actual code.
[b] Online features did not function.

We were able to recover all relevant artifacts from volatile memory. If a certain artifact is present in memory depends on user and application activity.

For example an image might only be loaded into memory if a certain note is viewed. Or a password is only present when a login activity is shown. Volatile memory eventually contains every bit of data used by the application, making it an invaluable source of information for the investigator, although it is currently very difficult to acquire.

5 Discussion

Application analysis proved to be a viable method for acquiring data. It should be used when other forensics solutions do not support acquiring data from a mobile device or when the software cannot decode/decrypt the application data. It is not an effective technique for acquiring a forensic image of a device or for trying to gain full control.

The main limitation of application analysis is that it is may be required to interact with the target application on the suspects device. This assumes that the investigator knows the access code of the suspect. However we have shown that our method can also be used to identify alternate storage locations such as cloud storage and determine how this information is most likely stored.

While we focused on the application as an attack vector, related research primarily looked at techniques that could be applied at the system level. For example by acquiring volatile memory or gaining executed privileges through exploitation of system components. These techniques are preferred over application analysis when system wide acquisition is the main goal of the investigation. They do however come with other prerequisites that cannot always be fulfilled.

Our methodology can be applied to a wide range of applications. By looking at our results, combined with the available literature, we expect other applications to exhibit the same behaviour. It also reinforces the idea that forensic investigators should not rely on commercial products alone. While these products are proven to be effective, their approach is still very traditional. Live data acquisition can yield interesting results and is worth the effort. We have proven that by analysing individual applications it is possible to acquire application data without the need for elevated privileges. While other methods can possibly acquire additional data, application analysis is relatively easy to employ and will work cross-platform.

Application analysis does not only provide a method for acquiring locally stored application data. By understanding how an application functions additional methods of acquisition can be identified. By applying our method to the Private Notepad application we identified Google Firebase as a secondary storage location.

6 Conclusion

Our research identified application analysis as a viable method for data acquisition. Although the Android operating system provides the developer with various methods for securely storing data, they are often not implemented. Developers

rely on custom security mechanisms and obfuscation for securing stored data and do not seem to have an adequate understanding of security standards.

In our research we also observed that volatile memory is an important source of information. Encrypted data stored on non-volatile memory can often be detected in an unencrypted state in volatile memory. Currently, acquisition of volatile memory is not a realistic option for the majority of devices because of the required privileges. More research is needed to identify viable methods for volatile memory acquisition.

During our network analysis we discovered that interception tools such as *Burp Suite* and *mitmproxy* are currently unable to capture HTTP/2 traffic. We were therefore unable to capture Google remote procedure calls (gRPC) used for Firebase communication. Because of this we had to rely on the more complicated method of function hooking. Additional research could enable easier analysis of HTTP/2 traffic.

Finally our current solution is aimed at the Android operating system. Although the general approach can be applied to other mobile operating systems such as iOS, some aspects need to be modified. For example, our checklist only applies to Android APK files and cannot be used for Mach-O binaries used on iOS. Also the checklist could be expanded to encompass more interesting artifacts. It would also be interesting to provide metrics for our analysis results, similar to the Common Vulnerability Scoring System (CVSS) [5]. This would allow us to compare applications and to estimate the value of discovered artifacts. Since the current procedure is largely a manual task, it would be desirable to automate the process by creating an automated vulnerability analysis framework.

References

1. Aouad, L., Kechadi, T., Trentesaux, J., Le-Khac, N.-A.: An open framework for smartphone evidence acquisition. In: Peterson, G., Shenoi, S. (eds.) DigitalForensics 2012. IAICT, vol. 383, pp. 159–166. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33962-2_11
2. Busstra, B., Kechadi, T., Le-Khac, N.-A.: Android and Wireless data-extraction using Wi-Fi. In: International Conference on the Innovative Computing Technology, pp. 170–175. IEEE (2014). https://doi.org/10.1109/INTECH.2014.6927769
3. Cerdeira, D., et al.: SoK: understanding the prevailing security vulnerabilities in TrustZone-assisted TEE systems. In: Proceedings of the IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA, pp. 18–20 (2020)
4. Chelihi, M.A., et al.: An android cloud storage apps forensic taxonomy. In: Contemporary Digital Forensic Investigations of Cloud and Mobile Applications, pp. 285–305. Elsevier (2017)
5. Common Vulnerability Scoring System SIG, February 2018. https://www.first.org/cvss. Accessed 24 Aug 2020
6. Daryabar, F., et al.: Forensic investigation of OneDrive, Box, GoogleDrive and Dropbox applications on Android and iOS devices. Aust. J. Forensic Sci. **48**(6), 615–642 (2016)

7. Davi, L., Dmitrienko, A., Sadeghi, A.-R., Winandy, M.: Privilege escalation attacks on android. In: Burmester, M., Tsudik, G., Magliveras, S., Ilić, I. (eds.) ISC 2010. LNCS, vol. 6531, pp. 346–360. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18178-8_30
8. Feng, H., Shin, K.G.: Understanding and defending the Binder attack surface in Android. In: Proceedings of the 32nd Annual Conference on Computer Security Applications, pp. 398–409 (2016)
9. Feng, P., et al.: Private data acquisition method based on system-level data migration and volatile memory forensics for android applications. IEEE Access **7**, 16695–16703 (2019)
10. Four Ways to Bypass Android SSL Verification and Certificate Pinning, January 2018. https://blog.netspi.com/four-ways-bypassandroid-ssl-verification-certificate-pinning. Accessed 10 Apr 2020
11. Frigo, P., et al.: TRRespass: exploiting the many sides of target row refresh. In: S&P, May 2020. https://download.vusec.net/papers/trrespass_sp20.pdf. https://www.vusec.net/projects/trrespassCode. https://github.com/vusec/trrespass
12. Götzfried, J., Müller, T.: Analysing android's full disk encryption feature. JoWUA **5**(1), 84–100 (2014)
13. Groß, T., Ahmadova, M., Müller, T.: Analyzing android's file-based encryption: information leakage through unencrypted metadata. In: Proceedings of the 14th International Conference on Availability, Reliability and Security, pp. 1–7 (2019)
14. Hayes, D., Cappa, F., Le-Khac, N.-A.: An effective approach to mobile device management: security and privacy issues associated with mobile applications. Digit. Bus. **1**(1), 100001 (2020)
15. HTTPS encryption on the web – Google Transparency Report, June 2020. https://transparencyreport.google.com/https/overview?hl=en_GB. Accessed 11 Jun 2020
16. Intel cuts Atom chips, basically giving up on the smartphone and tablet markets, April 2016. https://www.pcworld.com/article/3063508/intel-is-on-the-verge-of-exiting-the-smartphone-and-tablet-markets-aftercutting-atom-chips.html. Accessed 11 Jun 2020
17. Kocher, P., et al.: Spectre attacks: exploiting speculative execution. In: 40th IEEE Symposium on Security and Privacy (S&P 2019) (2019)
18. Liang, H., et al.: Witness: detecting vulnerabilities in android apps extensively and verifiably. In: 26th Asia-Pacific Software Engineering Conference (APSEC), pp. 434–441. IEEE (2019)
19. Loftus, R., et al.: Android 7 File Based Encryption and the Attacks Against It (2017)
20. Nilsson, A., Andersson, M., Axelsson, S.: Key-hiding on the ARM platform. Digit. Investig. **11**, S63–S67 (2014)
21. OWASP Mobile Top 10, June 2020. https://owasp.org/www-project-mobile-top-10. Accessed 13 Jun 2020
22. Security vulnerability search, April 2020. https://www.cvedetails.com/vulnerability-search.php?f=1&vendor=google&product=android&opgpriv=1. Accessed 15 Apr 2020
23. Storage updates in Android 11 j Android Developers, May 2021. https://developer.android.com/about/versions/11/privacy/storage. Accessed 8 Jun 2021
24. Thantilage, R., Le-Khac, N.-A.: Framework for the retrieval of social media and instant messaging evidence from volatile memory. In: 18th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, pp. 476–482. IEEE (2019). https://doi.org/10.1109/TrustCom/BigDataSE.2019.00070

25. Tilo, M., Michael, S., Freiling, F.C.: Frost: forensic recovery of scrambled telephones. In: International Conference on Applied Cryptography and Network Security (2014)
26. Van De Zande, P.: The day DES died. In: SANS Institute (2001)
27. Van Der Veen, V., et al.: Drammer: deterministic rowhammer attacks on mobile platforms. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 1675–1689 (2016)
28. Wächter, P., Gruhn, M.: Practicability study of android volatile memory forensic research. In: IEEE International Workshop on Information Forensics and Security (WIFS), pp. 1–6. IEEE (2015)
29. Yang, S.J., et al.: Live acquisition of main memory data from Android smartphones and smartwatches. Digit. Investig. **23**, 50–62 (2017)
30. Zhang, X., et al.: Cryptographic key protection against FROST for mobile devices. Clust. Comput. **20**(3), 2393–2402 (2017). https://doi.org/10.1007/s10586-016-0721-3