Protecting Encryption Keys in Mobile Systems Against Memory Errors

Jianing Zhao
The College of William and Mary
jzhao@cs.wm.edu

Peter Kemper
The College of William and Mary
kemper@cs.wm.edu

ABSTRACT

Beyond its use to encrypt data, an encryption key can be used as a mean to control access to data on a mobile device by leveraging a cloud service. This implies that a key is present in a mobile device only when it is on demand and authorized by the cloud, the key is evicted when it is not in use. CleanOS is an example system that is based on this concept. For security reasons, keys are stored only in DRAM memory during execution, which makes them susceptible to memory errors. In this paper, we identify scenarios where a memory error that damages a key can escalate to an unrecoverable data loss in a mobile system.

Categories and Subject Descriptors

B.8.1 [**PERFORMANCE AND RELIABILITY**]: Reliability, Testing, and Fault-Tolerance

General Terms

Performance,Reliability,Measurement

Keywords

Mobile systems, encryption, fault injection, simulation, memory errors, software reliability

1. INTRODUCTION

Mobile devices such as smartphones and tablets are prevalent in modern societies. According to [11], there is a trend for apps on mobile devices to even overtake PC internet usage in the United States. People use mobile devices for personal as well as for business purposes such that phones process data that is sensitive. Since smartphone loss and theft is a common problem, much research has gone into the protection of sensitive data on mobile devices. Solutions typically rely on encryption and specific ways to manage the encryption key. A fundamental idea is that a remote key management provides the ability to remotely control and monitor access to otherwise encrypted data.

This leads to the classical concept of key-escrow, where "something (e.g., a document, an encryption key) is delivered to a third person to be given to the grantee only upon the fulfillment of a condition." [9]. In a key-escrow architecture, a trusted third party stores an encryption key and provides it on request (if legitimate). The concept lends itself to different applications including key backup and recovery, monitoring and logging access to sensitive data, remote access control and deletion of sensitive data.

The use of encryption with the key being stored at a remote server heavily relies on several assumptions: a) the remote server stores the key in a reliable manner, b) the communication channel is dependable and secure, and c) the use of keys and encryption on the local device is performed in a reliable manner. The latter for instance means that the correct key is used for encryption and decryption. In this paper, we will focus on the last assumption in a situation where memory errors may corrupt a key stored in memory.

Approaches presented in [15], [1], [12], use key escrow architectures in different ways but all assume that the key in memory is reliable, and it is correct during the encryption. However, if this assumption does not hold, it is possible that sensitive data is encrypted with a corrupt key so that the sensitive data can not be decrypted with the original key, therefore, the data is lost because of the inconsistency of the keys. Therefore, key management software by IBM [6], EMC [3] uses highly reliable hardware to store keys to avoid key corruption in traditional sever based systems.

Using a key escrow architecture in a mobile system faces additional challenges. Mobile system is a very competitive market that evolves quickly with very short hardware and software innovation cycles and a severe price pressure towards inexpensive solutions. So, in this paper, we investigate the research questions: **Can memory errors impact the correct operation of a key escrow system such as CleanOS for sensitive data on a mobile system?** We consider the following threat model: 1)We assume that memory content can be corrupted (single or multiple bits) at any moment in time due to some memory error. 2)One possible cause is seen in programming errors in kernel or application code which accidentally overwrite content. Programming languages such as C are infamous for the numerous possibilities to make mistakes with pointer arithmetic,

.

VALUETOOLS 2015, December 14-16, Berlin, Germany

DOI 10.4108/eai.14-12-2015.2262698

dangling pointers and so forth. 3)Another possible cause is seen in hardware errors on DRAM that is unprotected with ECC.

ECC protection for DRAM comes at a price in terms of increased production costs, reduced capacity and increased energy consumption. For low end smartphones, it is currently common that DRAM is not protected with ECC against memory errors. Although the probability of a memory error is small, it is still quite possible that it happens and a stored key may be corrupted by some hardware or programming error. We consider the following items as the main contributions of this paper:

- Recognizing that memory errors pose a severe risk to encryption-based mobile Android systems
- Identifying concrete scenarios for potential loss of sensitive data in the CleanOS system

2. CLEANOS: AN ENCRYPTION-BASED MOBILE OPERATING SYSTEMS

The CleanOS system is particularly designed to rigorously "manage sensitive data and maintain a clean environment at all times in anticipation of data theft" [14]. It is not focussing on malware, as its main focus is device theft or loss. It does so by defining sensitive data objects (SDOs), a logical collection of Java objects, files and database items that applications create and use to manage sensitive data. Such objects are identified and tracked with the help of TaintDroid [4]. The crucial step is to have a modified garbage collector for the Dalvik virtual machine that identifies SDOs that have not been accessed for some time, encrypts the data-bearing fields with a key that is escrowed in the cloud (idle eviction). Since the key is not stored on the mobile device, access to the sensitive and encrypted data results relies on a cloud service to provide the key. This provides ample opportunities for logging, monitoring and access control. So it is fundamental that a) keys are not stored locally and b) sensitive data in memory or on persistent storage is always encrypted (unless in memory and in use). Since mobile apps tend to use SQLite to store data, sensitive data and other data are mixed within tables and CleanOS replaces individual table entries that carry sensitive data with the corresponding cipher text and vice versa. The otherwise common granularity of a file for encryption is considered much too coarse for Android apps. The actual system includes further features to make sure that applications and operating system do not retain additional clear text copies in caches and that deletion operations securely erase content. It is an impressive demonstration of efforts necessary to secure data in a system design that is optimized for performance, e.g., by extensive use of caching and avoidance of unnecessary operations such as overwriting data for deletion. As evidenced by experiments, CleanOS manages to achieve its goals while retaining an acceptable performance. This is partly due to the fact that SDOs are a small minority of all objects, partly due to the concept of buckets. Its target application is one like an email client or online banking application that operates with sensitive data but is used in a bursty manner with long time intervals between periods of usage such that the idle eviction is effective and has only a marginal impact on performance and network traffic.

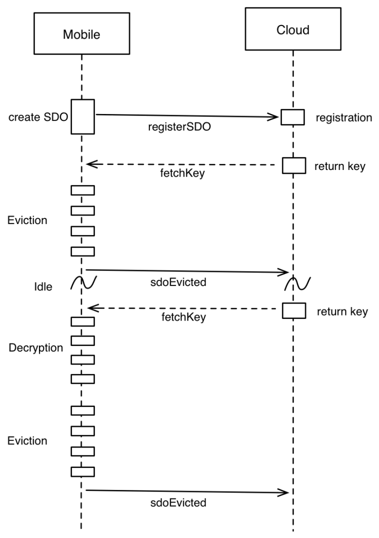

Figure 1: CleanOS communication between phone and cloud for the one time, initial registration of an SDO (top part) and subsequent, frequent, regular data access and key eviction (bottom)

Note that SDOs can vary substantially in size and content. While individual apps may induce very small scale and specific SDOs, the authors of [14] also establish coarse grain default SDOs at the kernel level such that apps can benefit from the CleanOS encryption with no changes to the application code. Examples for large default SDOs are a User Input SDO for all input a user types into the keypad or a SSL SDO for all objects read from incoming SSL connections. Obviously, loosing data of a large SDO can be very difficult to recover and go far beyond the inconvenience to retype a password or credit card number.

As an SDO is a collection of entities, which may have very different access patterns, CleanOS groups entities in so-called buckets and allows for different keys for individual buckets that are derived from a single key K_{SDO} that is SDO specific and stored in the cloud. According to [14], an K_{SDO} is obtained when an encrypted data field is accessed and then "cached onto the device and securely removed when the SDO as a whole is again evicted." Addition and removal of entities to and from an SDO is possible in a dynamic manner at runtime. This supports fine-grained eviction with coarse-grained SDOs. With bucketing, bucket keys are cached instead of SDO keys on the device.

Critical pieces of information: 1)The key: The SDO key is stored remotely for a particular SDO and fetched on demand. Bucket keys are derived from it locally on the phone. 2)The descriptor: The taint id contains an SDO id and a bucket id in a 32 bit field and serves as the descriptor for key and data. 3)The data: The sensitive data of an SDO object or in case of buckets all elements of a bucket.

To avoid a situation where encrypted data can not be decrypted again, there are some obvious necessary conditions that need to be satisfied:

1. *Descriptor consistency (DC) condition.* For a particular SDO, its descriptor in the cloud service and its descriptor on the phone must match.

2. *Key consistency (KC) condition.* For a particular SDO, its key stored in the cloud service and the key used for encryption of its data must match.

3. *Unique descriptor (UD) condition.* For a particular SDO, its descriptor is not allowed to match with the descriptor of another SDO.

Figure 1 illustrates the communication between phone and cloud service. There are three basic operations for the CleanOS communication protocol.

1. *registerSDO(sdoID,appName, description, key)*: If a new SDO is identified, a key is generated on the phone and the SDO is registered in the cloud with the descriptor (the app name and SDO id), the key, and a plaintext explanation (description).

2. *fetchKey(appName,sdoID,bucketID)*: If the key is not available at the phone and needed, it is fetched from the cloud with a method fetchkey that takes the descriptor information as its argument.

3. *sdoEvicted(appName,sdoID)* If an SDO is evicted, the cloud service is notified with a method sdoEvicted that takes the descriptor information as its argument. Eviction implies that on the phone, the key information is destroyed after encrypting the sensitive data.

So, if the DC or UD condition is violated, then the fetch key operation can not succeed. If the KC condition is violated, then the decryption operation on the encrypted data will fail when the key is fetched from the cloud service.

In the following, we investigate the impact of memory errors on basic CleanOS functionality by following the lifecycle of an SDO. We assume that a memory error could modify the content of a particular memory location in DRAM at any moment in time.

SDO creation Upon creation, a key and an SDO identifier are generated on the phone and the SDO is registered with the cloud service (method registerSDO). If a memory error modifies key or descriptor before registration, KC and DC conditions would not be violated as keys/descriptors match, however, there is a risk that key or descriptor are not functioning. With regard to AES encryption, we did not find any particular requirements for a key that would be violated by modifying its bit pattern in memory. The same holds for SDO and bucket ids that are a set of bits set in the Taint id. The only risk is to destroy the uniqueness of the descriptor such that two SDOs have the same id. It is reasonable to expect that the cloud service should fail the registration operation if the descriptor already exists.

Read access to SDO If the SDO has the requested content as clear text, then only memory errors in the data part may have an impact. If the requested content is encrypted, then the key need to be obtained. If the SDO or bucket key is cached, it can be obtained locally, otherwise it is fetched from the cloud service. The encrypted data is decrypted and accessed, the key is cached till eviction.

Vulnerability I.

The time between fetching the key and finishing decryption creates an opportunity for a memory error to corrupt the key. Caching the key substantially increases this time window. Using a corrupted key makes the decryption fail to reproduce the clear text data. If this goes undetected (as the decryption method transforms one bit pattern into another), the invalid clear text data will not allow the current application to perform. Even worse, if the invalid clear text data is used to overwrite the encrypted data, then this leads to a data loss. It is our understanding that CleanOS replaces encrypted data with clear text data on the spot (e.g. a specific SQLite data table entry) for a seamless integration of this concept into an existing Android architecture.

Vulnerability II.

A memory error that corrupts the descriptor of an SDO will make an operation to fetch the key from the cloud fail, which leads to data loss after idle eviction. The descriptor resides on the phone all the time and can not be recovered from the cloud. The time between the creation of the descriptor and its use in a fetch key operation is particularly long.

Write access to SDO Regardless of the data being in clear text or encrypted form, a write access will replace the current data with new data with no encryption or decryption involved. This will not allow memory errors on the key or descriptor to have an immediate effect on the newly written data.

Idle eviction of SDO If an SDO is not accessed for a sufficient amount of time, the Dalvik virtual machine in the CleanOS will perform an idle eviction. This means that the key is either obtained from a local cache or fetched from the cloud, the data is encrypted, and finally the key is evicted, i.e. its local copy is destroyed. As the local copy of the key is assumed to be consistent with the key stored on the remote server, the key is not communicated back to the server.

Vulnerability III.

The time distance between obtaining a key and using the key creates a time window for a memory error to corrupt the key before the encryption. If the key is cached, then that time window can be significant. If the key is fetched, that time window is expected to be very short. Encryption is not instantaneous, so the time during encryption also creates a time window for corrupting the key. If the encryption operation relies solely on a (good) copy of the key in a CPU cache, then a memory corruption in DRAM may have no effect. In general, if the key is corrupted between obtaining the key and finishing encryption, the encrypted data will be damaged and a subsequent decryption with the registered key will not succeed. As the encrypted data replaces the clear text data, this creates a vulnerability for data loss.

The destruction of an SDO completes its lifecycle. Besides, a notification to the cloud service may fail due to a corrupted descriptor, we do not expect any further activity than rigidly destroying the sensitive content. This does not create an opportunity for a memory error to destroy sensitive data.

In summary, we identify three scenarios where memory errors that corrupt a key or descriptor can lead to data loss in the CleanOS system.

3. SOURCE OF ERRORS

Hardware faults are one source of memory errors. For DRAM used in large scale data centers, studies by Hwang et al [5] and Schroeder et al [13] show that there is a rising rate in of memory error occurrences despite efforts in quality control of DRAM production and error-tolerance mechanisms. There are many causes such as environmental factors for memory error as shown in [8], [7], [10]. For mobile memory, we did not find corresponding numbers being published, however, as the high density and low cost design, we assume it is less reliable than PC DRAMs.

A second source of memory errors result from programming errors such as buffer overflow in software developed in languages such as C and C++.

4. RELIABILITY EVALUATION

In this section we present findings from a simulation study to demonstrate how vulnerable the handling of keys in a CleanOS system is to memory errors.

4.1 Evaluation using Mobius

We use Mobius [2] to establish and simulate the CleanOS system using SAN models. We use one submodel for CleanOS and one submodel for fault injection. We use an exponential distribution to model the time between memory faults, denoted as fault injection rate. The mean time between faults ranges between 0.87 and 2.4 hours. For the other activities, we assume a deterministic distribution for simplicity and use published average values from [14]. Idle time denotes the time between phases of active usage of an SDO, its parameter value was estimated from measurements of app usage for two volunteers in a period of two weeks. Measurements were recorded with the QualityTime app. We set the experiment time interval to 5000 hours as we assume the phone is used 14 hours per day. We do not describe details of the Mobius model for space limitations. We conducted simulations for different rates of fault injections, namely for a mean time between faults in [0.87,2.4] such that for an exponential distribution, $\lambda \in [0.4, 2.0]$. All simulations were performed with the Mobius simulator for a terminating simulation, confidence level setting of 95% and confidence interval setting of 0.1.

In Figure 2, we observe several measures as a function of the fault injection rate. Red lines show results for an average idle time of 6 minutes, blue lines for an idle time of 12 minutes between active phases for an SDO. A longer idle time reduces the number of times the system can go through a decryption, SDO active usage, SDO inactive timeout, encryption, key eviction cycle (a CleanOS cycle) in a fixed time horizon of 5000 hours. For sanity checks, we observed that the total number of cycles (correct or failed) being independent of the fault injection rate and that shorter idle times yield more CleanOS cycles.

Figure 2 shows that the basic CleanOS model experiences failures that increase with the fault injection rate and that the more CleanOS cycles are performed the higher the number of failures (difference between red line and blue line).

5. CONCLUSION

In this paper, we looked into CleanOS, an encryption-based Android OS that minimizes exposure of sensitive data in case of theft, to see how memory errors can affect its operation. We identified several scenarios where corruption of a key that is used for encryption can lead to loss of sensitive data. In the future work, we will detect the memory errors using hash tables and recover the keys by refetching the keys.

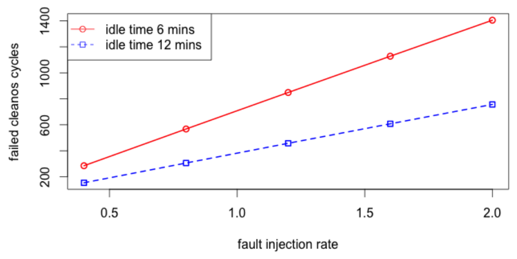

Figure 2: Failed cycles

6. REFERENCES

[1] D. Boneh and R. J. Lipton. A revocable backup system. In *USENIX Security Symposium 1996*.

[2] T. Courtney, D. Daly, S. Derisavi, V. Lam, W. H. Sanders, and W. H. S. The mobius modeling environment.

[3] EMC. Rsa key recovery manager.

[4] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI 2010*.

[5] A. A. Hwang, I. A. Stefanovici, and B. Schroeder. Cosmic rays don't strike twice: Understanding the nature of dram errors and the implications for system design. In *ASPLOS 2012*.

[6] IBM. Managing encryption.

[7] S. Khan, D. Lee, Y. Kim, A. R. Alameldeen, C. Wilkerson, and O. Mutlu. The efficacy of error mitigation techniques for dram retention failures: A comparative experimental study. In *SIGMETRICS 2014*.

[8] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *ISCA 2014*.

[9] R. D. Labati and F. Scotti. In *Encyclopedia of Cryptography and Security (2nd ed.)*.

[10] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu. An experimental study of data retention behavior in modern dram devices: Implications for retention time profiling mechanisms. In *ISCA 2013*.

[11] J. O'Toole. Mobile apps overtake pc internet usage in u.s. `http://money.cnn.com/2014/02/28/technology/mobile/mobile-apps-internet/`, 2014.

[12] R. Perlman. File system design with assured delete. In *SISW 2005*.

[13] B. Schroeder, E. Pinheiro, and W.-D. Weber. Dram errors in the wild: A large-scale field study. In *SIGMETRICS 2009*.

[14] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda. Cleanos: Limiting mobile data exposure with idle eviction. In *OSDI 2012*.

[15] Y. Tang, P. P. C. Lee, J. C. S. Lui, and R. Perlman. Secure overlay cloud storage with access control and assured deletion.