Modelling Analysis and Auto Detection of Cryptographic Misuse in Android Applications

Shao Shuai[*1], *Dong Guowei*[*1], *Guo Tao*[*1]

[*1]*China Information Technology Security Evaluation Center*
Beijing, China
shaoshuaib@163.com, dgw2008@163.com,
guotao@itsec.gov.cn

Yang Tianchang[*2], *Shi Chenjie*[*2]

[*2]*Beijing University of posts and telecommunications*
Beijing, China
gunzai-00@163.com, happy_chengjie@bupt.edu.cn

Abstract*—Cryptographic misuse affects a sizeable portion of Android applications. However, there is only an empirical study that has been made about this problem. In this paper, we perform a systematic analysis on the cryptographic misuse, build the cryptographic misuse vulnerability model and implement a prototype tool Crypto Misuse Analyser (CMA). The CMA can perform static analysis on Android apps and select the branches that invoke the cryptographic API. Then it runs the app following the target branch and records the cryptographic API calls. At last, the CMA identifies the cryptographic API misuse vulnerabilities from the records based on the pre-defined model. We also analyze dozens of Android apps with the help of CMA and find that more than a half of apps are affected by such vulnerabilities.*

Keywords—Modelling Analysis; Cryptographic Misuse; Vulnerability; Android.

I. Introduction

Mobile applications store a large number of users' sensitive data, such as username, password, location, credit card number, chat log and so on. Many mobile application developers use cryptography to protect the confidentiality, integrity and authentication in their apps. In theory, cryptography can provide a good protection for the sensitive data storage, transmission and user authentication, authorization. Unfortunately, many developers didn't use the cryptographic algorithms in a correct way. Veracode[1] detected the cryptographic defects in the source codes of mobile applications and concluded that cryptographic issues affected a sizeable portion of Android (64%) and iOS (58%) applications. With Android phones being ubiquitous, Android apps have become a worthwhile target for security and privacy violations. Attacker may take advantage of the cryptographic misuse vulnerabilities to acquire sensitive information of users', or even for targeted attacks.

Recently, a number of efforts have been made to investigate the cryptographic misuse problem. In 2008, Bhargvan et al.[2] concerned on the security protocol of TLS, they relied on a combination of model-extraction and verification tools, and implemented automated symbolic cryptographic verification and computational cryptographic verification. Fahl et al.[3] attempted to better understand the potential security threats posed by Android apps that use the SSL/TLS protocols to protect data they transmit. They created MalloDroid, an Androguard extension that performs static code analysis to analyze the apps' vulnerabilities against Man-in-the-Middle (MITM) attacks due to the inadequate or incorrect use of SSL. Georgiev et al.[4] presented an in-depth study of SSL connection authentication in non-browser software, focusing on how diverse applications and libraries validate SSL server certificates. They performed both white-box and black-box techniques to discover vulnerabilities in validation logic and uncovered a wide variety of SSL certificate validation bugs. Egele et al.[5] made an empirical study of the cryptographic misuse. They proposed a light-weight static analysis approach that checked for common flaws made by developers who had used the cryptographic APIs in an incorrect way so that IND-CPA security couldn't be provided. Sounthiraraj et al.[6] presented SMV-HUNTER, a system for the automatic, large-scale identification of SSL/TLS Man-in-the-Middle vulnerabilities in Android apps.

These papers analyzed APK files to discover the cryptographic misuse vulnerabilities, which achieve more convenience then source codes. However, instead of perform a systematic analysis, all the existing researches focus on a special aspect of that. In this paper, we perform a systematic analysis on the cryptographic misuse vulnerability, and present a conduct experiments on a number of Android applications. First, we build the models and types of cryptographic misuse vulnerabilities. Then, we show the lines of code where the vulnerabilities exist in Android apps. To identify the vulnerabilities effectively and efficiently, we build a prototypical identification tool that combines both static and dynamic analysis, called Crypto Misuse Analyser (CMA). With the help of CMA, we can monitor the cryptographic APIs invoked in the applications, and determine if the applications have the cryptographic misuse vulnerabilities by analysing the runtime information.

Contributions. We investigate the problem of cryptographic misuse in Android apps and our contributions are the followings:

- **Cryptographic Misuse Model.** A collection of misuse models is built in this paper, which will be helpful in identifying the cryptographic misuse.
- **Crypto Misuse Analyzer (CMA).** An automatic tool is implemented, which can identify the cryptographic misuse effectively and efficient.

The rest of our paper is organized as follows: In section 2, we present the definition and the scope of cryptographic misuse vulnerability. The codes that have cryptographic

978-1-4799-5079-9/14 $31.00 © 2014 IEEE
DOI 10.1109/DASC.2014.22

misuse vulnerabilities in Android applications are studied in section 3, followed by the methodology used in our system in section 4. Experiment results are presented in section 5. Finally, future work is discussed in section 6.

II. Modelling Analysis

We define the cryptographic misuse vulnerability as the improper use of cryptographic algorithms. Any code that does not use cryptography will not have such vulnerabilities. For an example, the information leakage caused by unencrypted sensitive data is not included, for there is no use of cryptographic algorithm. In most cases, cryptographic algorithms are predefined as APIs or libraries and what the application developers need to do is to call them in a proper way. However, not all the programmers know how to use the APIs or libraries correctly. Such vulnerabilities can be divided into two categories determined by its causes.

- **Misusing the cryptographic APIs.** Developers may invoke the wrong API functions, set incorrect parameters, and check the return values improperly and so on.
- **Lacking necessary steps.** Many cryptographic APIs need to be invoked in a predefined manner. For an example, in order to encrypt a message using a symmetric algorithm, the generation of a random initialization vector should be performed first. The vulnerability will exist in the code if the developer didn't generate the random initialization vector.

According to the algorithm types that the misuses occur, the cryptographic misuse vulnerabilities can be divided into 3 classes: the symmetric encryption algorithm misuse, the asymmetric encryption algorithm misuse and the hash algorithm misuse. Then we consult the Common Weakness Enumeration[7] and give a more detailed model to describe how the vulnerabilities are caused.

A. Symmetric Encryption Algorithms

S1: Use a non-random IV for CBC encryption. Using a non-random Initialization Vector (IV) with Cipher Block Chaining (CBC) Mode causes algorithms to be vulnerable to dictionary attacks. If an attacker knows the IV before he gets the next plaintext, he can check his guess about plaintext of some block that was encrypted with the same key before[9].

S2: Use ECB-mode for encryption. Symmetric encryption scheme in Electronic Code Book(ECB) Mode do not immune to the chosen plaintext attack, so symmetric encryption algorithms with ECB mode cannot provide sufficient protection for users.

S3: Insufficient key length. To reduce the possibility of brute force attack, the key length of symmetric algorithm should be no less than 128 bits. If the key length is less than 128 bits, we believe it is vulnerable.

S4: Use a risk or broken algorithm for Symmetric encryption. Maybe some algorithm was believed to be secure. But now days computing power is much cheaper than before, so the commercial hardware can creak it without much difficult. DES is one of them. Since DES is of only 56 bit key length, various proposals for a DES-cracking machine have been advanced. In 2008, commercial hardware costing less than USD 15,000 could break DES keys in less than a day on average[10]. DES is long past its sell-by date.

S5: Use the same cryptographic key multiple times or hard-coded cryptographic keys for encryption. For various reasons, such as, to reduce the development cycle or to reduce the technical difficulty, some developers use the same cryptographic key multiple times or hard-coded cryptographic keys for encryption. If such keys with problem are used, malicious attackers may recover the encrypted data, even gain the keys.

B. Hash Algorithms

H1: Reversible one-way hash. Maybe some algorithm was believed to be secure, but researchers found effective attacking algorithms. Two of the reversible one-way hash algorithms are MD4 and MD5. Gaëtan Leurent[11] broke the MD4 in 2008, and RFC 6150 stated that RFC 1320 (MD4) is obsolete in 2011. In 2010, Cert believed that MD5 "should be considered cryptographically broken and unsuitable for further use"[12]. The malware Flame exploited the weaknesses in MD5 and fake a Microsoft digital signature to make itself installed as a legal code[13].

C. Asymmetric Encryption Algorithms

A1: Inadequate key length. The key of asymmetric encryption algorithms is not long enough to prevent the brute force attack. For an example, we believe that the key length of RSA should be no less than 1024 bits, or else it is considered to be vulnerable. For exploits using 512-bit code-signing certificates that may have been factored were reported in 2011[14], It is currently recommended that the key be at least 2048 bits[15].

A2: RSA algorithm without OAEP. The programs use the RSA algorithm but do not incorporate Optimal Asymmetric Encryption Padding (OAEP) [16], which might weaken the encryption.

A3: Improper Certificate Validation. The programs didn't validate the certificate properly. So an attacker may use an invalid certificate to impersonate a trusted entity and make a man-in-the-middle (MITM) attack[17]. In an MITM attack, an attacker is able to intercept and modify network traffic between the client and the server. The improper certificate validations include the following situations:

A3-1: Missing Validation of Certificate

A3-2: Improper Check for Certificate Revocation

A3-3: Improper Validation of Certificate Expiration

A3-4: Improper Validation of Certificate with Host Mismatch

A3-5: Improper Following of a Certificate's Chain of Trust

D. Key Management Errors

K1: Use of Hard-coded Cryptographic Key or Password. The use of a hard-coded cryptographic key or

password significantly increases the possibility that encrypted data may be recovered. The attacker can simple disassemble the binary codes and acquire the hard-coded cryptographic key.

K2: Key Exchange without Entity Authentication. Performing a key exchange will preserve the privacy of the information sent between two entities, but this will not guarantee the entities are who they claim they are. This may enable a set of "man-in-the-middle" attacks. Typically, this involves a victim client that contacts a malicious server that is impersonating a trusted server. If the client skips authentication or ignores an authentication failure, the malicious server may request authentication information from the user. Then the malicious server can use this authentication information to log in to the trusted server using the victim's credentials, sniff traffic between the victim and trusted server, etc.

K3: Reusing a Nonce, Key Pair in Encryption. The nonce should be used for the present occasion during the program running and only once. Nonce is often bundled with a key in a communication exchange to produce a new session key for each exchange. An attacker may be able to replay previous legitimate commands or execute new arbitrary commands.

K4: Use of a Key past its Expiration Date. Software uses a cryptographic key or password past its expiration date, which diminishes its safety significantly by increasing the timing window for cracking attacks against that key. While the expiration of keys does not necessarily ensure that they are compromised, it is a significant concern that keys which remain in use for prolonged periods of time have a decreasing probability of integrity. For this reason, it is important to replace keys within a period of time proportional to their strength.

III. Cryptographic misuse vulnerabilities in Android Applications

Android apps complete cryptographic functions by taking advantage of the Java Cryptography Architecture (JCA) which provides cryptographic services and specifies the developers how to invoke Cryptographic APIs on Android platform. The JCA uses a provider-based architecture and contains a set of APIs for various purposes, such as encryption, key generation and management, certificate validation, etc.

The JCA implementations are included in the package java/security, javax/crypto and javax/security. Through the interface provided by Engine class such as Cipher, Signature, KeyStore, SecureRandom, MessageDigest and MAC, a specific cryptographic service is accessible to an application. We study the Android API calls that have cryptographic misuse vulnerabilities, and the following three examples show the vulnerabilities in Android apps.

S1: Use a non-random IV for CBC encryption. Before obtain the service of symmetric encryption scheme in Android, the developer should select the Cipher Engine and invoke the Cipher.getinstance() factory method, and initialize the object new established, specify the ENCRYPT_MODE, key and IV if the mode is CBC. As shown in the Fig. 1, the third parameter of Cipher.init() is set as "iv" in line 8, which is an IvParameterSpec instance. However, the instance is initialized with a constant vector in line 2. So, it matches the model of **S1**.

```
1   public static String encode(String key, byte[] data) throws
    Exception {
2    byte[] ivbyte = { 1, 2, 3, 4, 5, 6, 7, 8 };
3    DESKeySpec dks = new  DESKeySpec(key.getBytes());
4   SecretKeyFactory keyFactory =
      SecretKeyFactory.getInstance("DES");
5   Key secretKey = keyFactory.generateSecret(dks);
6   Cipher cipher = Cipher.getInstance("DES/CBC/PKCS5Padding");
7   IvParameterSpec iv = new IvParameterSpec(ivbyte);
8   cipher.init(Cipher.ENCRYPT_MODE,secretKey,iv);
9   byte[] bytes = cipher.doFinal(data);}
```

Fig. 1. *Non-random IV for CBC encryption sample*

```
1   private static byte[] encrypt(byte[] raw, byte[] clear)
    throws Exception{
2    SecretKeySpec skeySpec = new
     SecretKeySpec(raw, "AES");
3    Cipher cipher = Cipher.getInstance("AES");
4    cipher.init(Cipher.ENCRYPT_MODE, skeySpec);
5    byte[] encrypted = cipher.doFinal(clear);
6    return encrypted; }
```

Fig. 2. *ECB-mode for Encryption sample*

S2: Use ECB-mode for Encryption. As shown in the Fig. 2, the parameter of Cipher.getinstance() is set as "AES" in line 3 and the cipher mode and padding is omitted, so the ECB mode and PKCS7Padding are used as the default value, which matches the model of **S2**.

```
1  public boolean VerifyAdmin(String password) {
2  if (password.equals
3    ("68af404b513073584c4b6f22b6c63e6b")) {
4    System.out.println("Entering Diagnostic Mode...");
5    return true;
6  }
7  System.out.println("Incorrect Password!");
8  return false;
```

Fig. 3. *Use of hard-coded cryptographic key*

```
1   public class MD5 {
2    public static String getMD5(String val)
       throws NoSuchAlgorithmException{
3     MessageDigest md5 = MessageDigest
     .getInstance("MD5");
4     md5.update(val.getBytes());
5     byte[] m = md5.digest();
6     return getString(m); } }
```

Fig. 4. *Reversible one-way hash sample*

K1: Use of Hard-coded Cryptographic Key or Password. The code in Fig. 3 attempts to verify a password using a hard-coded cryptographic key. The key value is a hard-coded string value, the value is compared to the password for verification that if the password is equivalent to the hard-coded cryptographic key.

H1: Reversible one-way hash. Before obtain the service of message digest scheme, the developer should select the MessageDigest Engine and invoke the MessageDigest.getinstance() factory method, call the update()

and digest() to generate the Message-Digest. As shown in Fig. 4, the parameter of getInstance() is selected as "MD5" in line 3. Unfortunately, the message digest algorithm MD5 has been demonstrated to be reversible.

IV. Methodology

To identify the cryptographic misuse vulnerabilities in Android applications, we proposed CMA. In this section, we describe our system architecture and introduce the analysis implementation in detail.

We only analyze the symmetric encryption, hash, and asymmetric encryption misuse vulnerabilities in CMA for simplicity. CMA first performs static analysis on each app, and determines if the encryption APIs are used. When detecting the cryptographic APIs, CMA builds the Control Flow Graph (CFG) and Call Graph(CG)of the target app, and analyses which branch the cryptographic APIs are on. Then it runs the app following the branch detected and generates runtime logs. By comparing the logs with the models defined in section 2, CMA determines whether the apps have the cryptographic misuse vulnerabilities.

The CMA is designed to meet the following major requirements:

- **Efficiency**: The CMA performs branch analysis to select the branches that invoke the cryptographic APIs in the runtime, which reduces the computation cost of modelling analysis.
- **Accuracy**: The CMA analyzes vulnerabilities based on the runtime records, which achieve more accuracy than static analysis[4].

A. Branch Analysis

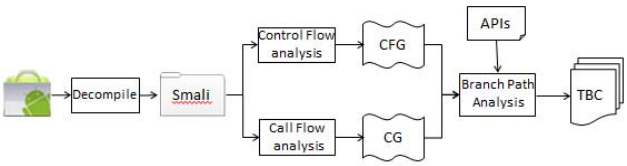

Fig. 5. *Branch Analysis*

In branch analysis, APK files are disassembled firstly and the dalvik bytecodes are analyzed to reveal the functionalities of the app. Then the Control Flow Graph (CFG) and Call Graph (CG) of each app are constructed and the Target Branch Collection(TBC) is built by the branch path analysis. The branch analysis process is shown in Fig. 5.

In CMA, we use apktool[18] to unpack the app and acquire the classes.dex file, which contains dalvik byte codes. And then reveals the sequence of the dalvik instructions of every method implemented in the app. The CMA analyses the dalvik instructions and selects the apps who invoke the cryptographic APIs.

Then we build the CFG and CG of each app. An android program can be divided into different small blocks, each block contains several dalvik instructions which can be invoke by dalvik virtual machine in a certain order. CFG is a graph represents the special control flows in a block and between blocks for an application. We construct the CFG by analyzing the sequence of dalvik instructions. The sequence often contains a lot of jump instructions. Android jump instructions use direct jump and the destination addresses can be statically obtained. In the process of CFG construction, we first get the block boundaries by analysing the jump instructions, then we link the blocks boundaries based on the destination address of jump instructions.

CG is a collection of nodes and edges and it has a single start and end point. The nodes stand for the function entry points, which consist of function name, parameter and return value. The edges are between two nodes, each edge records an entry point and an exit point and describes the relationship between the two nodes. We consider the Android app as a graph G consisting of a set N(G) of vertices and a set E(G) of direct edges. An edge $e \in E(G)$ is an ordered pair $e = (i; j)$, where $i; j \in V: i \neq j$.

To identify the execution branch which contains the cryptographic API, the CMA performs the Branch Path analysis on the app by performing reverse traversal on CFG and CG. Through the path analysis, the Target Branch Collection(TBC) was generated at last.

B. Record Generation and Modelling Analysis

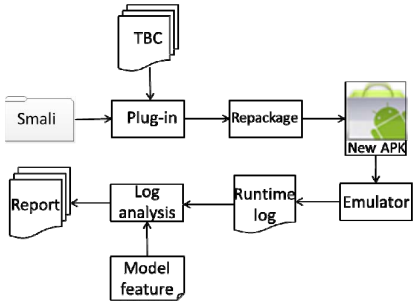

Fig. 6. *Record Generation and Modelling Analysis*

We mainly use API trace[19] and dynamic analysis to generate the cryptographic API invoked records, and compare the records with the model defined in section 2. The CMA adds logging code and repackages the APK. Then it runs each repacked app and records runtime logs. The record generation and modelling analysis process is shown in Fig 6.

The record generation is tightly coupled with the dynamic analysis tool called API Monitor[20]. API Monitor is a dynamic analysis tool that can add logging codes into Smali code of the app. With the information collected from the branch analysis, the CMA can accurately locate the cryptographic APIs and add instrumentation code. CMA monitors the API according to a configuration file which specifies the APIs that CMA will monitor. Runtime logs related to them will be generated when the repackaged APK is running. A demo of API list file is showed in Figure 6.

A runtime log records the executive information of the instrumentation in detail, the content of the record was separated into multiple items by semicolons, from left to right, the record items are: the first item is the package name and the invoker class, the second item consists of the method name

before the parentheses, arguments and assignment in the parentheses, the last item is return value and its type.

We define the model feature set for all cryptographic algorithms misuse mentioned in section 2. The log analyser checks each item of the log records, extracts the features of the record, and examines if the record matches the misuse model.

```
1  # DEFAULT API LIST
2  # Digest
3  Ljava/security/MessageDigest;->getInstance
4  Ljava/security/MessageDigest;->update
5  Ljava/security/MessageDigest;->digest
6  # Cipher
7  Ljavax/crypto/Cipher;->getInstance
8  Ljavax/crypto/spec/SecretKeySpec;-><init>
9  Ljavax/crypto/Cipher;->init
9  Ljavax/crypto/Cipher;->doFinal
```

Fig. 7. *Configuration List*

V. Experiment

To estimate the impact of the cryptographic misuse problem in Android applications, we analyzed dozens of mobile apps and found more than half of them were affected by such vulnerabilities. The tested apps were downloaded from http://as.baidu.com/, which is one of the most popular mobile application download site in China.

A. Test Subjects

We selected the following 5 type of apps: shopping, instant communication, mobile bank client, social network, payment & finance. All of the selected apps involve the sensitive data store or transport, thus it is critical for them to use the cryptographic algorithms correctly. Most of the selected apps' have been downloaded more than 1,000,000 times. The numbers of apps are shown in Table 1.

TABLE I. THE NUMBER OF APPS

App types	*shopping*	*instant communication*	*mobile bank client*	*social network*	*payment & finance*	*total*
Numbers	9	9	11	8	8	45

B. Experiment Results

We only found 5 apps have no cryptographic misuse vulnerabilities. The numbers of apps that are affected by the vulnerabilities are shown in Table 2.

TABLE II. THE NUMBER OF APPS AFFECTED

vulnerabilities	*S1*	*S2*	*S3*	*S4*	*H1*	*A1*	*A2*	*A3*
total numbers	8	7	0	8	38	1	3	0
shopping	4	2	0	2	9	0	0	0
instant communication	3	1	0	2	8	0	1	0
mobile bank client	0	2	0	0	6	1	0	0
social network	0	0	0	3	8	0	2	0
payment & finance	1	2	0	1	7	0	0	0

From table 2 we can see that H1 affect the most apps, for most apps choose md5 in computing digest. Since the most U.S. government applications now require the SHA-2 family of hash functions[21], this will be improved years later. The reason why asymmetric encryption misuse is much fewer than symmetric encryption misuse is the use of asymmetric encryption is much fewer than symmetric encryption.

C. Analysis

To check the reported cryptographic misuse vulnerabilities, we examined the apps manually. The rest of section 5 shows examples of how the vulnerabilities were identified.

```
1  V/CMA(16248): Ljavax/crypto/spec/SecretKeySpec;-
     ><init>([B={122, 121, 33, 67, 97, 68, 51, 123, 51, 54, 42, 119,
     53, 54, 35, 99} | Ljava/lang/String;=AES)V
2  V/CMA(16248): Ljavax/crypto/Cipher;-
     >getInstance(Ljava/lang/String;=AES/ECB/PKCS7Padding)Ljava
     x/crypto/Cipher;=javax.crypto.Cipher@40637c38
3  V/CMA(16248): Ljavax/crypto/Cipher;->init(I=1 |
     Ljava/security/Key;=javax.crypto.spec.SecretKeySpec@4a8)V
   107, 83, 106}
```

Fig. 8. *Example of S2*

S2: Use ECB-mode for Encryption. Using Symmetric cryptographic algorithms in ECB mode may weaken the security of the application, and CBC mode is recommended. A popular shopping client use AES in their mobile client, part of its runtime log is shown in Fig. 9. Line 2 shows the developer set the argument as "AES/ECB/PKCS7Padding", which means the algorithm use the encryption mode "ECB".

```
1   V/CMA( 9230): Ljavax/crypto/Cipher;-
      >getInstance(Ljava/lang/String;=DES)
    Ljavax/crypto/Cipher;=javax.crypto.Cipher@4242ead8
2   V/ CMA ( 9230): Ljavax/crypto/Cipher;->init(I=1
      | Ljava/security/Key; =javax.crypto. spec.SecretKeySpec@bc)V
3   V/CMA( 9230): Ljavax/crypto/Cipher;->doFinal
      ([B={-18, 52, -44, -116, 44, 40, -53, 6, 9, -120, 79, -117, -53, 113,
      28, 40})[B={83, 121, 115, 67, 108, 105, 101, 110, 116, 74, 115}
```

Fig. 9. *Example of S4*

S4: Using a risk or broken algorithm. DES has been exposed to be vulnerable to brute force attack. But there are still many apps use DES to encrypt the sensitive data. Runtime log in Fig. 10 is recorded when P Bank client is running. Line 1 shows that the argument of Cipher.getInstance() is set as "DES" .

```
1  V/CMA(18739): java/security/ MessageDigest; >getInstance
   (Ljava/lang/String;=MD5)
    Ljava/security/MessageDigest;=MESSAGE DIGEST MD5
2  V/CMA(18739):Ljava/security/MessageDigest;-
   >update([B={100, 101, ……, 0, 0, -70, 47, 0, }
3  V/CMA(18739): Ljava/security/MessageDigest;->digest()
   [B={-67, -32, 93, -45, -19, -13, 4, 81, -36, 25, -
    107, -81, -66, -43, -98, -125}
```

Fig. 10. *Example of H1*

H1: Reversible one-way hash. To ensure the effectiveness and integrity of the transmission content, one-way hash algorithm is used to generate message digest. As describe in section 2, MD5 has been known as vulnerable. According to the report provided by CMA, many applications still use the improper one-way hash algorithm. As the runtime log in Fig. 11, N Bank client uses MD5 to get message digest

value. Line 1 shows the client calls the MessageDigest.getInstance() to construct the MessageDigest instance, whose argument is specified as “MD5”.

```
1 V/CMA(6225): Ljavax/crypto/Cipher;->getInstance
(Ljava/lang/String;=RSA/ECB/PKCS1Padding)
Ljavax/crypto/Cipher;=javax.crypto.Cipher@422c3590
2 V/CMA(6225): Ljavax/crypto/Cipher;->init(I=1|
Ljava/security/Key;
=OpenSSLRSAPublicKey{modulus=b4…9,publicExponent=10001
})V
3 V/CMA (6225): Ljavax/crypto/Cipher;->doFinal([
B={65, 65, 65, 56, 51, 56, 51, 53, 49, 50, 51, 51, 53, 53})
4 [B={54, 101, 103, -18, 88, -105, -72, -106, 39, 34, 46, -34, 7, -58, -
11, 66, -116, -15, -128, 75, 22, 9, -81, -12, -123, -39, -121, 100, -88,
-75, 106, 100, -25, -58, -49, -30, -31, 82, -2, -94, -123, -70, 106, 75,
89, -42, 57, -53, 71, 77, 107, -68, 27, -40, -117, 120, -9, 27, -56, -
44, 121, -99, 121, 43}
```

Fig. 11. *Example of A1*

A1: Inadequate key length. The key length of RSA is an important factor to its reliability. Fig. 11 shows the CMA log of M Bank who use RSA to protect the communication data. The key is in line 4, whose length is 64Bytes, equals to 512bit.

```
1 V/CMA(19121): Ljavax/crypto/Cipher;-
>getInstance(Ljava/lang/String;=RSA)Ljavax/crypto/Cipher;=javax.
crypto.Cipher@406a0cc0
2 V/CMA(19121): Ljava/security/KeyFactory;-
>getInstance(Ljava/lang/String;=RSA)Ljava/security/KeyFactory;=j
ava.security.KeyFactory@4069fbe0
3 V/CMA(19121): Ljava/security/KeyFactory;-
>generatePublic(Ljava/security/spec/KeySpec;=java.security.spec.R
SAPublicKeySpec@4063b548)Ljava/security/PublicKey;=RSA
Public Key\n modulus: 85..b4c862b\n public exponent:
10001\n
4 V/CMA(19121): Ljavax/crypto/Cipher;->init(I=1 |
Ljava/security/Key;=RSA Public Key\n modulus:
85…b4c862b\n public exponent: 10001\n)V
```

Fig. 12. *Example of A2*

A2: RSA algorithm without OAEP. Part of runtime log of a famous social network client is showen in Fig.13. The client initializes the RSA Cipher object with the parameter “RSA”, while the padding is none. This implementation matches the model of A2.

VI. Conclusion

In this paper, we systematic studied the cryptographic misuse vulnerabilities and perform an investigation of the current cryptographic misuse vulnerabilities in Android apps. We concluded the cryptographic misuse model and build an prototype system named Crypto Misuse Analyzer(CMA), which can effectively identify the crypto misuse vulnerabilities based on the pre-defined model. Our investigation shows that more than half of apps have the cryptographic misuse vulnerabilities. But our work is limited in the cryptographic APIs misuse. In future work, we plan to continue our current work on Android native codes and other kind of misuse vulnerabilities.

ACKNOWLEDGMENTS

This work was supported by the National High Technology Research and Development Program of China (863 Program) (2012AA012903), the National Natural Science Foundation of China (61100047).

REFERENCES

[1] Veracode “State of Software Security Report （ Volume5)”, http://www.veracode.com/ resources/ state-of-software-security
[2] Bhargavan, Karthikeyan, et al. "Cryptographically verified implementations for TLS." Proceedings of the 15th ACM conference on Computer and communications security. ACM, 2008.
[3] Fahl, Sascha, et al. "Why Eve and Mallory love Android: An analysis of Android SSL (in) security." Proceedings of the 2012 ACM conference on Computer and communications security. ACM, 2012.
[4] Georgiev, Martin, et al. "The most dangerous code in the world: validating SSL certificates in non-browser software." Proceedings of the 2012 ACM conference on Computer and communications security. ACM, 2012.
[5] Egele, Manuel, et al. "An empirical study of cryptographic misuse in Android applications." Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. ACM, 2013.
[6] David Sounthiraraj, Justin Sahs, Garret Greenwood, et al. “SMV-HUNTER: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps.” NDSS 2014, February, 2014, San Diego, CA, USA.
[7] Common Weakness Enumeration: CWE, https://cwe.mitre.org/
[8] Bellare, Mihir, and Phillip Rogaway. "Introduction to modern cryptography." UCSD CSE 207 (2005): 207.
[9] Al Fardan, Nadhem J., and Kenneth G. Paterson. "Lucky thirteen: Breaking the TLS and DTLS record protocols." Security and Privacy (SP), 2013 IEEE Symposium on. IEEE, 2013.
[10] Kumar, Sandeep, et al. "How to Break DES for BC 8,980." SHARCS ‘06–Special-purpose Hardware for Attacking Cryptographic Systems (2006): 17-35.
[11] Leurent, Gaëtan. "MD4 is not one-way." Fast Software Encryption. Springer Berlin Heidelberg, 2008.
[12] Vulnerability Note VU#836068 MD5 vulnerable to collision attacks. http://www.kb.cert.org/ vuls/id/836068
[13] Sotirov, Alex. "Analyzing the MD5 collision in Flame." Presentation at SummerCon, slides available at http://www. trailofbits. com/resources/flame-md5. pdf (2012).
[14] RSA-512 certificates abused in-the-wild. http://blog.fox-it.com/2011/11/21/rsa-512- certificates-abused -in-the-wild/.
[15] Has the RSA algorithm been compromised as a result of Bernstein's Paper? What key size should I be using? http://www.emc.com/emc-plus/rsa-labs/historical/has-the-rsa-algorithm -been-compromised.htm
[16] Brown, Daniel RL. "What Hashes Make RSA-OAEP Secure?." IACR Cryptology ePrint Archive 2006 (2006): 223.
[17] Clark, Jeremy, and Paul C. van Oorschot. "SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements." Security and Privacy (SP), 2013 IEEE Symposium on. IEEE, 2013.
[18] http://code.google.com/p/android-apktool/.
[19] Y. Jing, G.-J. Ahn, Z. Zhao, and H. Hu. Riskmon: Continuous and automated risk assessment for mobile applications. In Proceedings of 4th ACM Conference on Data and Applications Security (CODASPY), pages 99–110. ACM, 2014.
[20] http://code.google.com/p/droidbox/.
[21] Nist's Policy On Hash Functions. “http://csrc.nist.gov/groups/ST/hash/policy.html”