International Journal of Research in Engineering, Science and Management
Volume-2, Issue-12, December-2019
www.ijresm.com | ISSN (Online): 2581-5792

A Novel Technique to Protect the Stealing of Authorization Code Using Hybrid Approach

M. Christina Ranjitham[1], C. Karpagavalli[2], D. Asir[3]
[1,2,3]*Assistant Professor, Department of Computer Science and Engineering, St. Mother Theresa Engineering College, Vagaikulam, India*

***Abstract*: In personal identification and online banking authorization codes play an important role require users to provide a code for authorization purposes. However, authorization codes in application can be stolen and forwarded by attackers, which introduces serious security concerns. In this paper, we propose Code Tracker, a hybrid approach to track and protect authorization codes. When the authorization code is sent out via either SMS messages or network connections (taint sinks), we extract the taint tag of the data and enforce standard security policies to prevent the code from being hacked and attacked from unauthorized access. SMS-stealing Android malware samples to evaluate the system. In this paper we used cryptographic algorithm to protect from SMS stealing from unauthorized access. The results show that Code Tracker a lightweight and hybrid approach can effectively track and protect SMS authorization codes with a high performance.**

***Keywords*: Tags, short message service (SMS) authorization**

1. Introduction

Smart phones are widely used in our daily life. Increasingly more users leverage smart phones for online transaction, bank transfers and other operations. Simultaneously increasingly more websites and applications (apps for short). leverage codes delivered via short message service (SMS) messages to authorize users. We call this type of code an authorization code in this paper. For instance, an SMS authorization code can be required when users log into a banking application or reset their passwords. Leveraging SMS codes for authorization is convenient; however, it may present security concerns. If the code is stolen by attackers, it can cause financial losses to users.

On the other hand, SMS-stealing malware is emerging [1], [2]. A research report from the Qihoo 360 companies [3] revealed that 6:1% of mobile malware is stealing information. Before Android version 4.4 (KitKat) [5], malicious apps could intercept SMS messages to retrieve authorization codes and then block the SMS broadcasting stealthily without informing users. However, starting with Android version 4.4, the SMS mechanism has been changed. Malicious apps are unable to block SMS broadcasting, and the system SMS app will get the SMS messages. However, malicious apps can still steal SMS messages by registering a broadcast receiver that listens to certain system events or requesting the READ_SMS permission to retrieve SMS messages from the database. Noted that a number of systems have been proposed to protect SMS authorization codes. For instance, TISSA [6] can provide null or bogus values instead of real data, which avoids data leakage (including SMS authorization codes). However, TISSA is currently implemented on legacy Android's Dalvik runtime and not the newly designed ART runtime. Secure SMS [7] is another system used to protect SMS messages by changing the Android framework. In particular, when an SMS message arrives, Secure SMS searches the message text. If a predefined keyword is found in the message, it adjusts the apps' receiving sequence of text message sin the system so that the default SMS app can get the text message first. Then, it stops the SMS broadcasting to prevent malicious apps from getting the message. This system works but may cause compatibility issues in some benign apps that rely on received text messages. In addition, starting with Android version 4.4, the SMS broadcasting mechanism has been changed, and the new unordered broadcasting cannot be blocked. From another perspective, because SMS authorization codes are a type of sensitive data in smart phones, they can be protected with the well-known taint tracking technique. Taint Droid [8] is such a system for real-time privacy monitoring that can be used to protect authorization codes. However, Taint Droid is implemented on the Dalvik virtual machine under Android version 4.4 and has not been applicable for the newly introduced ART runtime since Android 4.4. Taint ART [9] implements a practical multi-level information-_own tracking system on Android's ART virtual machine and can be used to track and protect private data. However, its extensibility is an issue because the bit length of a register (32 bits) for taint indication is limited. ARTist [10] is a system in Android that tracks private data by incrementing apps using a customized dex2oat tool. ART is an excellent system; however, it only works for intra-application tracking and lacks the inter-application tracking that is necessary for SMS authorization code protection. In this paper, we propose Code Tracker, a lightweight approach to track and protect SMS authorization codes in Android SMS messages. Specially, Code Tracker adds taint tags to mark the authorization code at the very beginning of the incoming SMS messages, and it modifies the related array structure, array operations, string operations, IPC (Inter-Process Communication) mechanism, and _le operations for the

secondary storage of SMS authorization codes to ensure that the tags cannot be removed. Finally, when the authorization code is sent out (via either SMS or the network), Code Tracker extracts the tag of the data and checks with predefined security policies. By doing so, it prevents authorization codes from being stolen by attackers. We have developed a prototype of Code Tracker on an Android system with the ART runtime, and we collected1; 218 state-of-the-art SMS-stealing Android malware samples to evaluate the system. The evaluation results show that Code Tracker can track and protect SMS authorization codes from being stolen. We further analyzed the remote server addresses where the stolen authorization codes are forwarded to, and we found that 87:66% of them are located in China.1This may be due to the popularity of third-party app stores in China with less regulation. The evaluation of the performance overhead shows that Code Tracker incurs a large overhead. In summary, this paper makes the following contributions:

- We propose a lightweight approach with data-_own tracking to protect SMS authorization codes in Android smart phones, called Code Tracker.
- We have implemented a prototype of Code Tracker in the Android ART runtime. Code Tracker adds taint tags to the SMS authorization code at the very beginning of the incoming SMS messages and ensures that the tags cannot be removed when propagating through the system.
- When the authorization code is sent out, Code Tracker protects the code by enforcing predefined security policies.
- We have evaluated our system with a collection of malware samples. The evaluation results demonstrate the effectiveness and low performance overhead of our system. The remainder of the paper is structured as follows.
- We proposed the cryptographic algorithm to protect the SMS stealing from third parties it's also take less time to perform

The remainder of the paper is structured as follows.

We first introduce the necessary background information in Section II. We illustrate the design, implementation, and evaluation of our system in Section III, Section IV, and Section V, respectively. We discuss the limitations and potential improvements to our system in Section VI, and the related work is presented in Section VII. Finally, we conclude the paper in Section VIII.

2. Background

In this section, we will briefly introduce the key concepts of the Android SMS system, as well as the Android runtime environment, to provide necessary background information for our proposed approach.

A. Android SMS system

In Android, when receiving a text message, the system sends the message from the RLI (Radio Layer Interface) layer to the framework layer. The framework layer then packs the text message into an SMS PDU and sends a broadcast indicating the receiving of an SMS message. All apps with the RECEIVE_SMS permission will receive the broadcast along with the SMS message if they have registered the SMS_RECEIVED_ACTION action. Before Android version 4.4, SMS broadcasting was ordered, and apps with higher priority could access SMS messages first and then discard the messages, which make apps with low priority unreachable to the SMS messages. This mechanism has been abused by malware to intercept SMS messages [4]. In addition, if a malicious app has the permissions (READ_SMS or WRITE_SMS) to directly operate on the SMS data base, it could monitor the database continuously. Once an SMS authorization

Code is received, it could steal the code and then delete it. Starting with Android version 4.4, the SMS system has been changed. When the system receives a text message, the framework layer encapsulates the text message into an SMS PDU and sends it with two types of broadcasting. One type is ordered broadcasting, i.e., SMS_DELIVER_ACTION, in which only the default SMS app can receive it. In other words, only the default SMS app has the permission to delete and insert the text messages to the SMS database. The other type is unordered broadcasting, i.e., SMS_RECEIVED_ACTION, in which the broadcasting cannot be interrupted, and all apps can receive SMS messages by registering the broadcasting. Due to this difference, malicious apps cannot intercept and delete the received SMS messages, but they still can steal and forward the SMS messages to remote servers.

B. Android runtime environment

On an Android system, each app is running inside a separated runtime environment and has its own unique running environment. This runtime environment was called the Dalvik runtime in old Android versions and is called the ART runtime in Android versions 5.0 and above. Dalvik is a register based virtual machine that will translate a dex _le into an odex _le with the dexopt command and then execute it. To further improve the performance of Android, Google introduced a new Android runtime, i.e., ART (Android Runtime) [11], which adopts the AOT (ahead of time) mechanism. When an Android app is being installed, the ART virtual machine leverages the dex2oat tool to transform the app's dex _le into an oat _le, which actually compiles the byte code into native machine code. When the app is running, the machine code will be directly executed, which greatly improves the performance. The transition from the Dalvik to ART runtime leads to several challenges to the taint tracking system. For instance, Taint Droid [8] is implemented in Dalvik, which stores the taint tags by applying extra space adjacent to the variable sin the stack of

the Dalvik virtual machine. In the ART runtime, some of the parameters are stored directly in registers. To support taint tracking in the ART runtime, the method of n storing taint tags should be changed accordingly. This is only one challenge, and we will illustrate how to implement taint tracking on the ART runtime in Section IV.

C. Initial study of the SMS-stealing malware samples

1) Permission analysis

To systematically understand the malicious behaviors of SMS-stealing malware, we have collected 1; 218 malicious samples from three websites, i.e., Virus Total [14], Virus Share [15], and Contagion Mobile [16]. We found that most samples request the permissions related to SMS and the network. Among these 1; 218 collected malware samples,1; 015 of them (83:33%) request RECEIVE_SMS permission,799 (65:60%) of them request READ_SMS permission,1; 020 samples (83:74%) request SEND_SMS permission, and1; 005 samples (82:51%) request INTERNET permission.

2) Two methods to obtain SMS authorization codes

We then decompiled these samples to further understand the methods used to steal the SMS authorization code. We found that these samples usually leverage two different methods to steal SMS messages. One is through the SMS broadcast receiver, and the other is through the SMS database monitoring mechanism. Among the 1; 218 samples that we collected, 429 of them steal messages through the SMS broadcast receiver, 120 of them steal messages by monitoring the SMS database, and 336 of them steal messages using both mechanisms.

- *SMS broadcast receiver:* As mentioned in Section II-A, before Android version 4.4, a malicious app could obtain text messages ahead of other apps by setting its priority to a higher value and then it blocks the SMS broadcast. This could prevent other apps (the system SMS app for example) from receiving the SMS messages. In Android versions 4.4 and above, a malicious app can still receive text messages via registering a broadcast receiver, but it cannot block the message broadcast.
- *Monitoring SMS database:* Malicious apps with READ_SMS permission are able to monitor the SMS database. When a new SMS message is received and inserted into the SMS database, the monitor will be informed.

3) Two methods to send out SMS authorization codes

We further installed the samples on an LG Nexus 5 phone and captured. The results show that there are two main methods that the samples use to send out the stolen SMS messages: through SMS messages and through the network interface.

- *SMS forwarding:* If a malicious app has the SEND_SMS permission, it is easy for the app to forward the stolen authorization code through another SMS message by invoking the well-defined system APIs, e.g., send Text Message().
- *Network forwarding:* If a malicious app has the INTERNET permission, it can forward the authorization code through the network interface. The possible channels include emails, HTTP requests, and direct TCP/UDP sockets.

3. System design

A. Threat model

In this work, the SMS authorization code is the user's private data that need to be protected. Third-party apps installed on the system are not trusted either because they are malicious or vulnerable. These apps can steal SMS authorization codes in smart phones and forward the codes to a remote server. Similar to other works, trust the underlying Android framework and the operating system. While the physical security of the devices (including the smart phones and the SIM cards) is out of the scope of this work.

B. Overall design

The goal of our work is to track and protect authorization codes in SMS messages. To achieve this, there are several challenges that need to be addressed. First, we must determine whether a text message contains an authorization code and then mark it with the taint tag. Second, the SMS authorization code could be processed in many locations, e.g., it might be copied or passed to a new variable or be saved to the SMS database. The taint tags need to be reversed and propagated in these scenarios. Third, we need to determine the correct place to enforce pre- security policies to ensure that the SMS authorization code cannot be stolen. To overcome these challenges, we propose a lightweight approach to track and protect SMS authorization codes, called Code Tracker. The overall design of Code Tracker is shown in Figure 1. As marked in digital numbers in Figure 1, there are 15 possible steps in the processing of SMS authorization codes. These steps are described as follows: 1, Android receives an SMS message; 2-3, according to the predefined rules ,the SMS message is marked as a potential SMS authorization code by adding a t_p tag; 4, an SMS message with the t_ptag is sent to the default system SMS app and the third part apps that register as an SMS broadcast receiver; 5,the system SMS app inserts the SMS content into the SMS database and adds the t_p tag as an extra extended attribute to the SMS database _le; 6, a third-party app fetches an SMS message from the SMS database, and the content of the message is marked with the t_p tag, which is obtained from the extra extended attribute of the SMS database _le; 7, a database taint tag, i.e., t_d, which represents that the data are read from the SMS database, is added to the SMS message;8-9, it determines whether the message contains an authorization code; if so, it adds a t_a tag to the SMS message;10, a third-party app obtains the SMS authorization coded at from the SMS database, which contains three taint tags ,i.e., t_a, t_d, and t_p. These tags are denoted as t_a|d|p;11-12, the SMS messages are sent through the SMS interface or the

network interface; 13-15, it extracts the tags of the data to be sent out and then processes them according to the predefined security policies.

C. *Identify the SMS authorization code*

To identify an SMS authorization code and then apply the taint tag, our system has to determine whether an SMS message contains an authorization code. First, we need to decide when to identify the authorization code. Note that the Android SMS system mainly obtains SMS messages via SMS broadcasting or by reading from the SMS database. Therefore, we only need to determine whether an SMS message contains an authorization code before the SMS broadcasting and after the message is fetched from the SMS database

However, because the framework layer of Android will not have decoded the message content before the SMS broadcasting, it is difficult for us to recognize the authorization code by searching the content of the message. Therefore, we leverage the sender address of the SMS message to determine whether the message possibly contains an authorization code; if so, we mark it as a potential SMS authorization code. We maintain a list of sender addresses of SMS authorization codes, and we treat all the SMS messages that originate from these addresses as messages potentially containing SMS authorization codes. After the SMS message can be read from the SMS database, we search the content of the message to obtain the string pattern of the authorization code to determine whether the message contains an authorization code. After identifying an SMS message that contains an authorization code (or potentially contains such a code), we mark and track the message by adding a tag (or taint tag) to it (the marked message is called a taint source). It is important to note that if we add tags to all the variables in the system, it can better track the data, but the memory overhead will become a concern. We observe that an SMS message is generally stored in a character or byte array; therefore, we only need to add tags in character and byte arrays. In addition, we add one tag for each array to reduce the memory overhead. However, because the framework layer of Android will not have decoded the message content before the SMS broadcasting, it is difficult for us to recognize the authorization code by searching the content of the message. Therefore, we leverage the sender address of the SMS message to determine whether the message possibly contains an authorization code; if so, we mark it as a potential SMS authorization code. We maintain a list of sender addresses of SMS authorization codes, and we treat all the SMS messages that originate from these addresses as messages potentially containing SMS authorization codes. After the SMS message can be read from the SMS database, we search the content of the message to obtain the string pattern of the authorization code to determine whether the message contains an authorization code. After identifying an SMS message that contains an authorization code (or potentially contains such a code), we mark and track the message by adding a tag (or taint tag) to it (the marked message is called a taint source). It is important to note that if we add tags to all the variables in the system, it can better track the data, but the memory overhead will become a concern. We observe that an SMS message is generally stored in a character or byte array; therefore, we only need to add tags in character and byte arrays. In addition, we add one tag for each array to reduce the memory overhead.

Fig. 1.

The taint tags are defined and applied in the Android native layer; this is transparent to the application and Android framework.

D. *Propagate taint tags*

Ensuring that the taint tags cannot be removed during the internal processing of the system is a challenge. Because the SMS message is saved in an array that is created in the heap, the taint tag will not be removed during general operations. e.g., function calls. However, in the processing of multiple cases, the Android system can lose a tag carried by an array. These cases include (1) IPC, (2) string operations, (3) single element processing in an array, and (4) the secondary storage of the data.

E. *Enforce security policies*

To prevent the SMS authorization code from being stolen, we enforce the corresponding policies at the endpoints where the code could be sent out (these endpoints are called taint sinks). Our initial study of the malware samples shows that a stolen authorization code can be sent out through SMS messages or the network interface (Section III-B). For the first case, we modify the SMS manager to detect whether an authorization code is being sent out. Specifically, when an app sends an SMS message, we extract the tag of the data and enforce the security rules. For instance, we could allow or prevent the sending, warn the user, record the target address, etc. For the second case, there are different ways in the Android application layer to send out the data through the network interface. However, all of these ways will eventually call the native layer method through Posix. Therefore, we can extract the taint tag of the data and enforce similar security policies at this endpoint.

4. Implementation

We have implemented a prototype of Code Tracker. Our system consists of three components: authorization code identification module, taint tag propagation module, and

security policy enforcement module.

A. Authorization code identification

In the following, we will present how the taint tags are defined and how the authorization code is identified. We also present the way that we apply the taint tags to the authorization code.

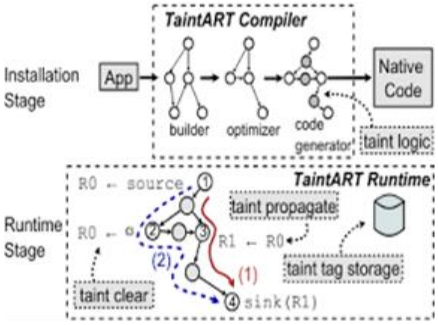


Fig. 2. Taint tag

1) The definition of a taint tag

In our system, each byte or character array contains a taint tag, as defined in Figure 2. A taint tag is a 32-bit integer data, and each bit has a specific meaning. We only define the meaning of the lower three bits, and we leave the remaining bits for future extension. If the data contain a taint tag, the corresponding bit of the tag of the data will be set to 1; otherwise, it will be set to 0. To apply taint tags to the SMS authorization code, we define several different tags. We define the t_n tag as 0x00000000, which represents no tag, and data containing the t_n tag denotes that the data contain no taint tags and thus are general data. We define the t_p tag as 0x00000001, which represents the potential tag, and data containing the t_p tag are data possibly containing an authorization code. We define the t_d tag as 0x00000002, which represents the database tag, and data containing the t_d tag are data that are fetched from the SMS database. We define the t_a tag as 0x00000004, which represents the authorization code tag, and data containing the t_a tag are data that contain an authorization code. Note that it is possible for data to contain more than one tag. For instance, some data might contain the t_d and t_a tags (t_a|d for short), which denotes that the data are fetched from the SMS database and contain an authorization code.

2) The storage of the taint tag

The purpose of our system is to protect SMS authorization codes. Notice that an SMS authorization code is stored in a string or a character/byte array; therefore, we only need to add a taint tag to the character and byte arrays. To save memory, there is no need to store tags for other types of data. To this end, we modify the method of the Array class in Android. Because the taint tag is 32-bit (i.e., 4 bytes) integer data, the memory space occupied by an Array object will increase by 4 bytes. Thus, we modify the Compute ArraySize() method of the Array class.

We store the taint tag at the end of an array. Note that the content of an Array object is stored in a variable array, called elements_. To get the taint tag, the tag's offset in the array must be calculated. We define a static method to calculate the actual memory address of the taint tag; this is easy to do with the array's starting address, the type of the array content and the length of the array as parameters. We could not access the taint tag using the array's subscript, and the taint tag is transparent to the app.

3) The operations of the taint tag

In the native layer, we define operations to add or get tags for different types of data objects. These types include byte array, character array, and string. Then, we register these methods as internal methods in the virtual machine. We only define the operations to add or get the taint tags, not operations to delete or update the tags. This is to prevent the app from abusing these operations to remove the tag. We have internal ways that cannot be observed by apps to update or remove the tags.

4) Identifying authorization codes before SMS broadcasting

We leverage the source address of the SMS message to determine whether the message possibly contains an authorization code. If it might, we mark the message with a t_p tag, which represents a potential tag. To this end, we need to collect the phone numbers that send out SMS authorization codes. We observe that such phone numbers are very different in different countries and regions. For example, in China, the phone numbers that send SMS authorization codes usually start with 95 or 106 or are special numbers. Thus, we maintain a list of source addresses of SMS authorization codes, which currently includes, for example, 106∗, 95???, 12306, 10086, 10000, and 10010. Among the list, '106∗' stands for phone numbers starting with 106, and the length of such numbers is not fixed but is no longer than 20 bits.

5) Identifying authorization codes after getting from the sms database

When getting an SMS message from the SMS database, Android will return a Cursor object using the query() method, and then, it gets the message content with the getString() method of the Cursor object. After getting the message content, we will add a t_d tag (i.e., 0x00000002) to the string of the message, which represents that the string is fetched from the SMS database. Then, we determine whether the string contains an authorization code by search- in the content. Specifically, we search for ''authorization'' or ''password'' key words. If either key word is found, we con- duct a subsequent search; if the string contains four or more digits or a sub-string of four or more digits or English characters, then we think that the string contains an authorization code and add a t_a tag (i.e., 0x00000004) to the message data. As a result, when an SMS message is fetched from the SMS database, the general message will carry a tag as t_d or t_d|p. It is possible for a message from the SMS database to contain a t_p tag; as for the SMS database, all the records in the database share the same tag set that is stored in the extra extended attribute of the database file. If the SMS database contains one record with the potential tag, then

all the records fetched from the database will carry such a tag.

B. Taint tag propagation

In the following, we will illustrate how the taint tags are propagated in the system. In Android, array objects are stored in the heap. In addition, the taint tags of the array object might be lost when the object is copied, moved or saved into a file. In our system, we made changes to the string methods, compiler, Parcel class for inter-process communication, and methods related to file operations to propagate the taint tag.

1) Modification to the string methods

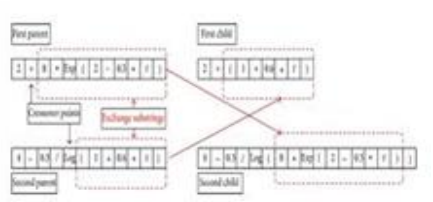
In Java, the String class contains a private array that contains the actual data. Assigning elements from an old array to a new target array is usually done by the System_array copy TUnchecked() method in the native layer.

2) Instrumentation to the compiler

In Android, many apps encode the data in text messages before they send it through the Internet, similar to the URL coding process. The encoding operations might modify a single array element of the source array and copy it to a new array, which can lead to a loss of the taint tag. There- fore, we need to instrument the compiler of the ART virtual machine. In particular, we first append a new field (i.e., tag_container) to the Thread class to store the taint tag carried with the thread; then, we modify the process of array assignment. When this process copies an element of the source array to the target array, we extract the taint tag from the source array and save it into the current thread's tag_container field. Later, we get the taint tag from the tag_container field of the current thread and add it to the target array. Array access operations will be converted into AGET and APUT operations in the dex byte code and eventually translated into the corresponding machine code by the compiler. To maintain the taint tag, we instrument the compiler for byte-and character-related AGET and APUT operations. We take the byte-related operations.

When the system takes out an element from a byte array with an AGET-BYTE operation, it gets the taint tag of the source array and the taint tag from the tag container field of the current thread. Then, we get the result of the bitwise OR computation for these two tags. Finally, we store the result as a new tag to the tag_container field of the current thread. On the other hand, when the system assigns an element to a byte array with an APUT-BYTE operation, we first get the taint tag from the tag_container field of the current thread and the taint tag of the target array. The original compiler of the sys- tem produces six lines of assembly code for AGET-BYTE, i.e., lines 01-03 and 12-14. First, it loads the array length to the r3 register (line 01); then, it stores the array subscript into memory(lines02-03).Before loading the array element to the r5 register (line 14), it judges if the array is out of boundary (line 12); if so, it jumps to an exception handling (line 13).

To maintain the taint tag, we instrument the compiler to add 8 instructions (lines 04-11 marked by *). Specifically, it first calculates the memory location of the taint tag of the array (lines 04-07) and loads the tag to the r1 register (line 08); then, it loads the taint tag saved in the current thread to the r12 register (line 09) and calculates the new tag by a bitwise OR operation for these two tags (line 10). Finally, it saves the new tag to the current thread for later use in APUT-BYTE (line 11).



```
01* ldr.w r2, [r9, #44]      #Load tag of current thread to r2
02* mov r7, r0               #Store array address to r0
03* ldr r3, [r7, #8]         #Load array length to r3
04* movs r0, #1              #Store size of type BYTE to r0
05* mul r12, r3, r0          #Calculate size of the array
06* adds r12, r12, #12       #Calculate offset of the tag
07* adds.w,r12, r12, r7      #Calculate memory location of tag
08* ldr.w r1, [r12, #0]      #Load the tag to r1 register
09* orr r2, r1               #Get new tag with a bitwise OR
10* str.w r2, [r12, #0]      #Store new tag in target array
11  ldr r2, [r7, #8]         #Load array length to R2 register
12  cmp r2, #1               #Is the array out of boundary?
13  bls +22                  #If so, jump to exception
14  strb r5, [r7, #13]       #Store r5 register into memory
```

The original compiler of the sys- tem produces four lines of assembly code for APUT-BYTE, i.e., lines 11-14. First, it loads the array length to the r2 register (line 11), and then, it judges if the array is out of boundary (line 12). If so, it jumps to an exception handling (line 13); otherwise, it stores the value in r5 into the target array (line 14). To maintain the taint tag, we instrument the compiler to add 10 instructions (lines 01-10 marked by *). Specifically, it first loads the taint tag of the current thread to the r2 register (line 01). Then, it calculates the memory location of the taint tag of the array (lines 02-07) and loads thetagtother1 register(line08). Finally, it calculates the new tag with a bitwise OR operation for these two tags (line 09) and saves the new tag to the target array (line 10).

3) File operations

When an SMS message is stored into the SMS database file, the taint tag carried by the text message will be lost. In Android, because the file operations are performed by calling the native layer methods through the Posix class, which is in the framework layer, we modify the operations associated with the Posix class for this purpose. Specifically, when a byte array is stored in a file, we first extract the taint tag of the array, and then, we add it to the extra extended attribute of the target file. Later, when we read the data from a file, we first get the taint tag from the file's extra extended attribute and then add it to the corresponding byte array.

C. Security policy enforcement

The malicious apps could forward the stolen SMS authorization code through SMS Manager or network interface (taint sinks). Therefore, to catch such behaviors, we need to

modify the corresponding interfaces in Android's framework layer and apply corresponding security policies. When it forwards the message through the SMS Manager, we extract the tag of the data to be sent. When it forwards the data through the network interface, it could be in several ways, e.g., by email, with HTTP request, and with TCP/UDP sockets. However, in any way, the network data will eventually be submitted to the system call of the kernel, which is performed through the Posix class. Therefore, we could detect and protect the SMS authorization data by monitoring the network-related operations in the Posix class.

1) Security policies

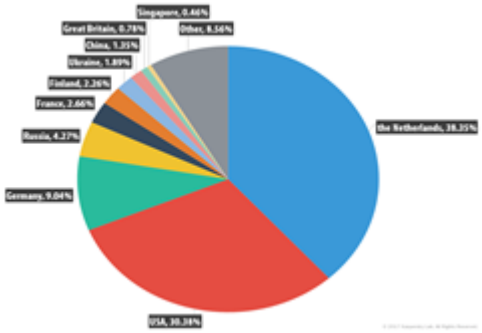
If we get a taint tag from a byte or character array, we may possiblygetseveralvalues.Amongthesevalues,0x00000000 (i.e., t_n) represents that the data do not contain any taint tags; 0x00000001 (i.e., t_p) represents that the data potentially contain an authorization code and that the data are directly obtained through SMS broadcasting; 0x00000002 (i.e., t_d) and 0x00000003 (i.e., t_d|p) represent that the data are fetched from the SMS database; and 0x000000007 (i.e., t_a|d|p) represents that the data are fetched from the SMS database and contain an authorization code. When the value is 0x00000001 or 0x00000007, we manipulate the data according to our pre-defined rules (e.g., prohibit sending, warn the user, or send a bogus value). It is important to note that if an app sends out data with a tag of 0x00000001 (i.e., t_p), we think that it is a dangerous operation. This is because the data are directly obtained through SMS broad- casting, and then, the app is attempting to send it out. This is a malicious action, as a benign app always fetches an SMS message from the SMS database and then sends it out.

5. Evaluation

In this section, we evaluate the effectiveness as well as the performance overhead of our prototype system.

A. Effectiveness

We ran our prototype to track and protect the SMS authorization code on an LG Nexus 5 smart phone. To evaluate the effectiveness of our system, we ran each sample every time and sent real authorization code messages through some popular shopping and banking websites to the smart phone. The prototype printed log information about the authorization codes. With this log information, we then retrieved the evaluation results. The evaluation results show that among the 1,218 malicious samples, 885 of them obtained the messages, and the remaining 333 samples failed to obtain them. Of the 885 samples that obtained messages, 429 of them obtained messages through SMS broadcasting, 120 of them obtained messages by reading the SMS database, and 336 of them obtained messages both through the SMS broadcasting and by reading the SMS database. Concerning subsequent actions, in the 885 samples that obtained messages, 312 of them did not forward the messages, 152 of them forwarded the messages, although we did not get any taint tags for the data, and 421 of them forwarded the messages, but we did obtain the authorization code tag. Among the 421 samples that forwarded messages with taint tags, 212 of them transmitted the data through SMS Manager, 108 of them transmitted the data through the network interface, and 101 of them transmitted the data through both the SmsManager and the network interface. The distribution map of the 1,218 SMS-stealing malware samples.



Among the 333 samples that failed to retrieve SMS messages, we found that the main reason for their failure was the expiration of their software license or software errors. For the 312 samples that did not forward the messages after obtaining them, we found that the system did not print any log information about SMS forwarding. We believe that this is reasonable because these samples may need certain conditions to be satisfied to trigger the behavior. How to automatically trigger this malicious behavior remains an ongoing research problem. Among the 152 samples that forwarded messages but where we did not get any taint tags, we checked the log information and found that these samples did not forward the messages successfully. The main reasons for this are that these samples failed to connect to a remote server before sending the authorization code data, the mailbox's password had been changed when sending through email, or the license expired. In summary, our system effectively captured the taint tags for the 421 samples that actually forwarded the SMS authorization code and blocked such attempts. In the log information produced by the 1,218 samples, we also collected 1,311 target IP addresses and 294 target phone numbers for the stolen authorization codes. We found that the top three target addresses are located in mainland China, USA, and Hong Kong, and 87.66% (1,407/1,605) of the target addresses are located in mainland China. The distribution map of target addresses for SMS-stealing samples is shown in Figure 7. We will release the addresses that we collected to the community to improve the detection of such malware.

B. Performance overhead

To measure the performance overhead introduced by Code Tracker, we have performed several micro benchmarks, i.e., a compiler micro benchmark, a Java micro benchmark, and an

IPC micro benchmark. The evaluation is conducted on an LG Nexus 5 phone with our system.

1) Compiler microbenchmark

We measure the size of the oat files and the total compilation time. For the compiler micro benchmark, we select the ten most popular apps in Google play as our evaluation datasets. The ten apps are Free VPN, Google Translate, Youtube, Instagram, Lantern, Taobao, Twitter, Alipay, Google Game, and Tumblr. We com- pile each app every time with the original compiler and our instrumented compiler and record the size of the produced at file and the time of compilation. We conduct the benchmarks ten times for each app and calculate the average. Table 1 shows the evaluation results of the compiler microbench- mark. On average, Code Tracker introduces an approximate 0.07% overhead with respect to the size of oat files and an approximate 1.79% overhead with respect to the compilation time. Compared to TaintART [9], Code Tracker has a better performance as TaintART incurs about 19.9% overhead with respect to the compilation time.

- *Java microbenchmark:* Because the Java micro benchmark can accurately reflect the runtime overhead introduced by Code Tracker. The maximum overhead introduced by Code Tracker is 6.92% (String score), and the minimum loss is 0.01% (Sieve score). The average performance overhead incurred by Code Tracker with Caffeine Mark is 1.33%, which is much better than TaintART [9] that introduces about 14%. Our approach is thus a lightweight solution.

2) IPC microbenchmark

To perform the evaluation of the IPC micro benchmark on Code Tracker, we have developed a pair of client/server apps that communicate through Binder in Android. Specifically, the client records the time (t_0) before sending a message to the server. The server will also send a message back to the client after receiving the client's message. Then, the client records the time (t_1) at which it receives a message from the server.

Therefore, `t1-t0' represents the elapsed time. We repeat the test ten thousand times and record the execution times. We also calculate the memory usage or the client and server apps during their communication.

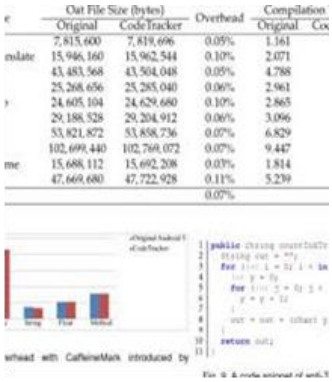


Table 2 shows the test results. The overhead of the IPC execution time is 0.90%, and the memory usage overheads for the client and server are 0.66% and 1.17%, respectively. In contrast, the overheads introduced by Taint ART [9] are about 4.35% for the IPC execution time and 4% for the memory usage, which are both higher than Code Tracker.

6. Discussion

In this section, we discuss some possible limitations of Code Tracker and potential future work.

A. Future work

First, Code Tracker is designed for the protection of SMS authorization codes, not for the protection of general text messages. However, in our decompilation process, we found that many malware apps steal general messages. Therefore, in the future, we can easily extend Code Tracker into a pro- to type system to protect all text messages by applying the taint tags to SMS messages and changing the security policies accordingly. Second, Code Tracker requires changes to the underlying framework; it cannot be transparently supported as a user-level solution. We using Greedy algorithm to protect the information from the third parties.

7. Related work

A. Protection of SMS messages

A variety of systems have been designed to prevent SMS messages from being leaked in smart phones. For example, Secure SMS [20] and other similar systems[21]–[23]leverage cryptographic algorithms to encrypt the SMS messages for confidentiality, integrity and authentication services, which is a different goal compared to Code Tracker. Secure SMS [7] attempts to protect SMS messages by adjusting the app's receiving sequence of text messages in the system so that the default SMS app can get the text message first. Then, it blocks the SMS broadcasting to prevent malicious apps from getting the message. However, Secure SMS only works in Android versions prior to 4.4. Other systems [24]–[27] have also been proposed to prevent phishing messages. Specifically, these systems search the content of SMS messages to find URLs that might link to malicious apps for installation and then block users' dangerous operations. In contrast to these apps, Code Tracker aims to provide protection for authorization codes in SMS messages.

B. Confinement of smartphone apps

A number of systems have been implemented to limit apps' access to sensitive data. For example, Kirin [39] confines apps by preventing third-party apps from accessing private data. FlaskDroid [40] achieves this goal by hooking Android system services. AppCage [41] leverages two complimentary user-level sandboxes to interpose and regulate an app's access to sensitive APIs. To prevent potential privacy leakage, Aurasium [42], AppGuard [43], TISSA [6], and RetroSkeleton [44] have been proposed to enforce finegrained access control on sensitive data. All these systems may be able to be leveraged to

provide protection for sensitive data (including SMS authorization codes) on legacy runtimes (i.e., Dalvik) in Android, but not on the ART runtime. In contrast, Code Tracker works well on Android's ART runtime and can provide protection as well as tracking for authorization codes in SMS messages.

8. Conclusion

In this paper, we design a dynamic lightweight approach for tracking and protecting authorization codes in Android, called Code Tracker. Specifically, we leverage the taint tracking technique and mark authorization codes with taint tags at the origin of the incoming SMS messages and propagate the tags through the system. Then, we apply security policies at the endpoints where the tainted authorization code is being sent out. The evaluation results on real malware samples demonstrate the effectiveness of our system, and the introduced performance overhead is low (< 2% on average).

References

[1] Code Tracker: A light Weight Approach to protect the message SMS 2018.
[2] We Steal SMS: An Insight into Android. KorBanker Operations. Accessed: Dec. 26, 2017. [Online]. Available: https://www.fireeye.com/blog/threat-research/2014/09/we-steal-sms-an-insight-into-android-korbanker-operations.html
[3] SophosLabs Report Explores Mobile Security Threat Trends, Reveals Explosive Growth in Android Malware. Accessed: Dec. 26, 2017. https://news.sophos.com/en-us/2014/02/24/sophoslabs-report- explores-mobile-security-threat-trends-reveals-explosive-growth-in- android-malware/
[4] Special Report on Android Malware in 2016. Availablehttp://zt.360.cn/1101061855.php?dtid=1101061451&did=490301065
[5] Y. Zhou and X. Jiang, ''Dissecting Android malware: Characterization and evolution,'' in Proc. IEEE Symp. Secure. rivacy, May 2012, pp. 95–109.
[6] Android 4.4. https://developer.android.google.cn/about/versions/kitkat.html
[7] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh, ''Taming information-stealing smart phone applications (on Android),'' in Proc. 4th Int. Conf. Trust Trustworthy Compute., 2011, pp. 93–107.
[8] D. Kim and J. Ryou, "Secure SMS: prevention of SMS interception on Android platform,'' in Proc. 8th Int. Conf. Ubiquitous Inf. Manage. Commun., 2014, p. 32.
[9] W. Enck et al., ''TaintDroid: An information-flow tracking system for real time privacy monitoring on smart phones,'' ACM Trans. Compute. Syst., vol. 32, no. 2, p. 5, Jun. 2014.
[10] M. Sun, T. Wei, and J. C. S. Lui, ''TaintART: A practical multi-level information-flow tracking system for Android runtime,'' in Proc. ACM SIGSAC Conf. Compute. Commun. Secure., 2016, pp. 331–342.
[11] M. Backes, S. Bugiel, O. Schranz, P. von Styp-Rekowsky, and S. Weisger- ber, ''ARTist: The Android runtime instrumentation and security toolkit,'' in Proc. IEEE Eur. Symp. Secur. Privacy, Apr. 2017, pp. 481–495.
[12] GoogleIO 2014. Accessed: Dec. 26, 2017. [Online]. Available: https://www.google.com/events/io/io14videos/b750c8da-aebe-e311-b297-00155d5066d7
[13] MazarBOT. https://www.tripwire.com/state-of-security/featured/mazarbot-android-malware-distributed-via-sms-spoofing-campaign/
[14] Bankbot. Available: https:// github.com/bemre/bankbot-mazain
[15] VirusTotal. Available: https:// www.virustotal.com/
[16] VirusShare. Available: https:// virusshare.com/
[17] Contagio Mobile. https://contagiominidump.blogspot.com/
[18] Pendragon Software Corporation. CaffeineMark 3.0. http://www.benchmarkhq.ru/cm30/
[19] G. S. Babil, O. Mehani, R. Boreli, and M.-A. Kaafar, ''On the effectiveness of dynamic taint analysis for protecting against private information leaks on Android-based devices,'' in Proc. Int. Conf. Secure. Cryptogr., 2015, pp. 1–8.
[20] AntiTaintDroid. https://github.com/gsbabil/AntiTaintDroid
[21] N. Saxena and N. S. Chaudhari, ''SecureSMS: A secure SMS protocol for vas another applications" J. Syst. Softw., vol.90, pp.138–150, Apr.2014.
[22] A. De Santis, A. Castiglione, G. Cattaneo, M. Cembalo, F. Petagna, and U. F. Petrillo, ''An extensible framework for efficient secure SMS,'' in Proc. Int. Conf. Complex, Intell. Softw. Intensive Syst., 2010, pp. 843–850.
[23] H. Harb, H. Farahat, and M. Ezz, ''SecureSMSPay: Secure SMS mobile payment model,'' in Proc. Int. Conf. Anti-Counterfeiting, Secur. Identificat., 2008, pp. 11–17.
[24] G. C. C. F. Pereira et al., ''SMScrypto: A lightweight cryptographic framework for secure SMS transmission,'' J. Syst. Softw., vol. 86, no. 3, pp. 698–706, 2013.
[25] LinkScanning. Accessed: Dec. 26, 2017. [Online]. Available: https://play.google.com/store/apps/details?id=com.directionsoft.linkscan&feature
[26] S-GUARD. https://play.google.com/store/apps/details?id=kr.co.seworks.sguard
[27] AntiSmishing. https://play.google.com/store/apps/details?id=com.nprotect.antismishing
[28] T-GUARD. http:// www.tstore.co.kr/userpoc/game/viewProduct.omp?t_top=DP000504&dpCatNo=DP04003&insDpCatNo=DP04003&insProdId=0000329718&prodGrdCd=PD004401&stPrePageNm=DP04003&stActionPositionNm= 06&stDisplayOrder=1
[29] S. Arzt et al., ''Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps,'' in Proc. 35th ACM SIGPLAN Conf. Program. Lang. Design Implement., 2014, pp. 259–269.
[30] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, ''Analyzing inter-application communication in Android,'' in Proc. Int. Conf. Mobile Syst., Appl., Services, 2011, pp. 239–252.
[31] F. Wei, S. Roy, X. Ou, and Robby, ''Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps,'' in Proc. ACM SIGSAC Conf. Comput. Commun. Secur., 2014, pp. 1329–1341.
[32] S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden, ''DroidForce: Enforcing complex, data-centric, system-wide policies in Android,'' in Proc. 9th Int. Conf. Availability, Rel. Secur., 2014, pp. 40–49.
[33] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, ''Chex: Statically vetting Android apps for component hijacking vulnerabilities,'' in Proc. ACM Conf. Comput. Commun. Secur., 2012, pp. 229–240.
[34] Y. Zhou, K. Singh,andX.Jiang,''Owner-centricprotectionofunstructured data on smartphones,'' in Proc. 7th Int. Conf. Trust Trustworthy Comput., 2014, pp. 55–73.
[35] C. Qian, X. Luo, Y. Shao, and A. T. Chan, ''On tracking information flows through JNI in Android applications,'' in Proc. 44th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN), Jun. 2014, pp. 180–191.
[36] DroidBox. https://github.com/pjlantz/droidbox
[37] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, ''Riskranker: Scalable and accurate zero-day Android malware detection,'' in Proc. Int. Conf. Mobile Syst., Appl., Services, 2012, pp. 281–294.
[38] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, ''Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets,'' in Proc. Annu. Netw. Distrib. Syst. Secur. Symp., 2014, pp. 50–52.
[39] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu, ''Malton: Towards on-device non-invasive mobile malware analysis for art,'' in Proc. USENIX Secur. Symp., 2017, pp. 1–19.
[40] W. Enck, M. Ongtang, and P. McDaniel, ''On lightweight mobile phone application certification,'' in Proc. 16th ACM Conf. Comput. Commun. Secur., 2009, pp. 235–245.

[41] S. Bugiel, S. Heuser, and A. R. Sadeghi, ‘‘Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies,’’ in Proc. USENIX Secur. Symp., 2013, pp. 131–146.
[42] Y. Zhou, K. Patel, L. Wu, Z. Wang, and X. Jiang, ‘‘Hybrid user-level sandboxing of third-party Android apps,’’ in Proc. ACM Symp. Inf., Comput. Commun. Secur., 2015, pp. 19–30.
[43] R. Xu, H. Saïdi, and R. Anderson, “Aurasium: Practical policy enforcement for Android applications,” in Proc. USENIX Secur. Symp., 2012, pp. 1–14.
[44] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. Styp-Rekowsky, “Appguard–enforcing user requirements on Android apps,” in Proc. Int. Conf. Tools Algorithms Construction Anal. Syst., 2013, pp. 543–548.
[45] B. Davis and H. Chen, ‘‘Retro skeleton: Retrofitting Android apps,’’ in Proc. Int. Conf. Mobile Syst., Appl., Services, 2013, pp. 181–192.