Hindawi Publishing Corporation
Mobile Information Systems
Volume 2016, Article ID 8973828, 11 pages
http://dx.doi.org/10.1155/2016/8973828

Research Article

Security Analysis and Improvement of Fingerprint Authentication for Smartphones

Young-Hoo Jo,[1] Seong-Yun Jeon,[2] Jong-Hyuk Im,[2] and Mun-Kyu Lee[2]

[1]*Electronics and Telecommunications Research Institute, Daejeon 34129, Republic of Korea*
[2]*Department of Computer and Information Engineering, Inha University, Incheon 22212, Republic of Korea*

Correspondence should be addressed to Mun-Kyu Lee; mklee@inha.ac.kr

Received 6 November 2015; Accepted 24 January 2016

Academic Editor: Seung Yang

Currently, an increasing number of smartphones are adopting fingerprint verification as a method to authenticate their users. Fingerprint verification is not only used to unlock these smartphones, but also used in financial applications such as online payment. Therefore, it is very crucial to secure the fingerprint verification mechanism for reliable services. In this paper, however, we identify a few vulnerabilities in one of the currently deployed smartphones equipped with fingerprint verification service by analyzing the service application. We demonstrate actual attacks via two proof-of-concept codes that exploit these vulnerabilities. By the first attack, a malicious application can obtain the fingerprint image of the owner of the victimized smartphone through message-based interprocess communication with the service application. In the second attack, an attacker can extract fingerprint features by decoding a file containing them in encrypted form. We also suggest a few possible countermeasures to prevent these attacks.

1. Introduction

Recent advances in smartphone technologies enabled users to do various tasks using their smartphones. These tasks include not only simple ones such as playing mobile games and surfing the web, but also more critical ones, in particular, those dealing with private information and financial data. Therefore, a reliable mechanism is required to verify the identity of a person who tries to use the device. However, traditional secret knowledge-based solutions such as passwords, numeric PINs, and pattern locks have security issues such as password guessing attacks, brute-force attacks, and shoulder-surfing attacks. Moreover, they also have usability issues because a user must memorize some information and do a cumbersome task for log-on such as typing a password and drawing a pattern. In order to address these issues, fingerprint recognition is now being used for many smartphones, for example, iPhone 5s, Galaxy S5, and VEGA Secret Note. Fingerprint recognition is used both for unlocking a smartphone and for activating other security-critical functionalities in the smartphone, for example, for approving transactions in financial applications [1].

Therefore, it is very crucial to secure the fingerprint recognition service from possible threats such as intercepting a fingerprint image between an image sensor and a fingerprint recognition application and stealing the fingerprint data stored in a smartphone. Unfortunately, however, some of the currently deployed devices do not seem sufficiently safe against those threats. In this paper, we disclose the vulnerabilities in the fingerprint recognition service of VEGA Secret Note by analyzing the service application and demonstrate possible attacks against this service. (The VEGA series is one of the earliest smartphones with fingerprint recognition service, which is prior to recent popular ones such as iPhone 5s and Galaxy S5 [2]. The vulnerabilities were found on the device with Android 4.2.2 as of April, 2014. We reported these two vulnerabilities to the vendor. The second vulnerability was already addressed through a patch, and the vendor commented that the first vulnerability will also be addressed in the upcoming version.) VEGA Secret Note is an Android-based smartphone with a Qualcomm Snapdragon CPU (Krait 400), 3 GB RAM, and a 5.9-inch IPS touch display. It is equipped with an FPC fingerprint sensor on its back.

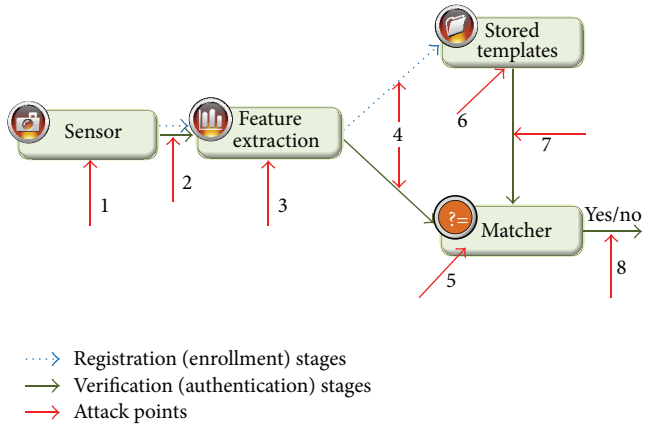


Figure 1: Generic structure of a biometric verification system and possible attack points (adopted from [4, 5]).

Our first attack is to enable a malicious application to acquire the fingerprint image of the owner of the victimized smartphone by accessing the memory space that the fingerprint recognition service application uses to temporarily store the image. In a nutshell, this attack exploits the design flaw of the service application which violates the principle of least privilege for access control [3]. To be precise, when a client application requests the service application to do fingerprint authentication, the service application activates a component which deals with the image of a scanned fingerprint. This component has been ill-designed so that it calls back an event handler in the client application with a reference to the memory location containing this image. As a result, the malicious client application can obtain the bitmap image by letting the component be activated and handling the event raised by that component.

Our second attack is to extract a stored template from the nonvolatile memory and restore fingerprint feature points by decoding the template. By identifying and analyzing a fingerprint service application on the target device, we identified the location of the stored template. In addition, we discovered that the template was encrypted, but the same key and initial vector (IV) are hard-coded and are the same for all devices. This design results in a vulnerability that a malicious user may be successfully authenticated if she/he overwrites a template by another template copied from his/her own device. In addition, by analyzing the structure of the decrypted template file, we were able to restore all feature points constituting the fingerprint template. This implies that a carefully forged template according to the file structure also may pass the authentication test.

Although we concentrated on a specific device in conducting our experiments, the technical flaws we have found in this device are a common trap that developers may fall into. Therefore, we suggest a few possible countermeasures to mitigate those vulnerabilities. We expect that the findings we obtained through our analysis may be used as a general guideline to design a secure biometric verification service on smartphones.

The remainder of this paper is organized as follows. Section 2 provides the preliminary information about the organization of a generic biometric system, a standard format for a fingerprint template, and the message-based communication mechanism between Android processes. In Section 3, two vulnerabilities and their possible consequences are explained in detail. A few possible countermeasures to these vulnerabilities are discussed in Section 4. Finally, we conclude the paper in Section 5.

2. Preliminaries

2.1. Threat Model against Biometric Verification. A generic biometric system can be cast in the framework of a pattern recognition system [4]. Figure 1, which was adopted from [4, 5], summarizes the typical stages in this generic system. A biometric system has two main procedures: registration (enrollment) of biometric data and verification (authentication) of biometric data, which are represented as blue dotted lines and green solid lines in Figure 1, respectively. The first stage of registration is to acquire the original biometric signal (typically, an image) using a sensor. The next stage is to extract invariant features from this original signal to construct a robust representation for biometric data that can uniquely determine an individual. The extracted features are stored as a form of a template. In the case of fingerprint recognition, a template contains fingerprint minutiae points. A minutia point is a peculiar point in a fingerprint image, for example, where a ridge either begins or divides into two ridges. A typical fingerprint may have tens of such points, and those points forming a template uniquely determine the characteristic of a specific fingerprint. Current fingerprint recognition systems are very accurate; in particular, they can provide a false rejection rate of 0.01% at a false acceptance rate of 0.1% [6].

The first and second stages of biometric verification are similar to those of registration. However, instead of storing the extracted features, the system runs a matching algorithm to compare the features derived from the current input biometric with those of the stored template. The matcher makes a decision, that is, whether to accept the user or not, based on the matching score.

Figure 1 also specifies eight places in the generic biometric system where attacks may occur. These points are represented as red lines in the figure and correspond to each item in the following list. This list is an extended version of the lists in [4, 5]. By a *passive attacker* we mean an attacker who steals or eavesdrops the secret information about the biometrics but who does not modify anything. On the contrary, an *active attacker* is an attacker who modifies the original biometric signal, template, or matching result to thwart a biometric verification service.

(1) A passive attacker may steal the original biometric signal by accessing the memory space the sensor uses to temporarily store this signal. In addition, fake biometrics such as a fake fingerprint, a copied signature, and a face mask can be presented for an active attacker to impersonate a legitimate user.

(2) A passive attacker may eavesdrop the original biometric signal sent from the sensor and store it in its own storage for later use. On the other hand, an active attacker may replay previously stored biometric signals bypassing the sensor. As a result, the attacker can impersonate the owner of that biometric. Note that a passive attacker may use the eavesdropped data to play a role of an active attacker.

(3) An active attacker may override the feature extraction module so that it produces only preselected features, ignoring the input from the sensor. A passive attacker may mount a backdoor which sends the extracted features back to him/her.

(4) The communication channel from the feature extraction module to either the template storage or the matcher may be tapped by a passive attacker. An active attacker may modify the packets and let the transmitted template be replaced with his/her own one. The purposes of these attacks are the same as those of the above type 3 attacks.

(5) An active attacker may corrupt the matcher so that it produces preselected matching scores without reference to the actual matching algorithm.

(6) A passive attacker may steal the stored templates, and an active attacker may modify the stored templates to force the system to authorize a fraudulent user or deny service to a legitimate user.

(7) The data sent from the template storage to the matcher may be intercepted by a passive attacker or modified by an active attacker. The results of these attacks are the same as those of the above type 6 attacks.

(8) The attacker may override the final decision with his/her intended result.

In this paper, we will present two passive attacks against the fingerprint recognition system of a VEGA Secret Note smartphone. Our first attack was to acquire the original biometric signal by injecting a malicious code independent of the original biometric application program and accessing the memory space where the biometric signal was stored. Our second attack was to directly access the stored template, not passing through the biometric application program. Therefore, the first attack can be viewed as a passive type 1 or type 2 attack, and the second attack can be viewed as a passive type 6 attack. We remark that even though we only demonstrate passive attacks, the output of our attacks may also be immediately used for active attacks. Although type 3, type 5, and type 8 attacks need an attacker's modification of the original biometric application, the effects of these attacks are the same as those of our attacks. Thus, we did not try to mount type 3, type 5, and type 8 attacks. In addition, type 4 and type 7 attacks were not required either, because stored templates were already manipulated by our type 6 attack.

2.2. Biometric Verification Using Fingerprint Minutiae. Many devices that deal with fingerprints, including our target device, use the fingerprint minutia formats based on ISO/IEC 19794-2 [7] and ANSI INCITS 378 [8]. According to these standards, four main characteristics of minutiae are considered. These four characteristics are the x and y coordinates of the minutia on the original fingerprint image, the angle (θ) of the ridge corresponding to this minutia point, and ridge types. Although there are many distinct ridge types, two major types among them, that is, a ridge ending (also known as a ridge termination) and a ridge bifurcation, are frequently used in most settings [7–10], where a ridge ending stands for a point where a ridge suddenly ends and a ridge bifurcation is a point where a ridge divides into two ridges. See Figure 1 in [11] for the concrete examples of these two ridge types.

For biometric verification, a matcher compares the features extracted from the current sensor image with the stored template which is composed of multiple minutia points. The comparison is done by comparing (x, y, θ) of each fingerprint minutia point in the stored template with those from the sensor. A matching score is increased whenever each point matches. If the score is larger than a predefined threshold, the user is permitted to access the target device (see Figure 2).

2.3. Message-Based Communication between Processes in Android. Android supports messages for interprocess communication (IPC) [12]. It enables an application to share an object with another application by sending a reference to the object to the target application. Figure 3 shows an example procedure where two applications communicate with each other through messages. As shown in this figure, a typical communication between two applications is done according to the following scenario. Throughout the paper, an item written in `typewriter` font represents a name of a class, an object, or their field.

(1) First, application A sends an intent to initiate a communication with application B, where an intent is a kind of signal to abstractly describe an operation to be performed [13]. An intent contains a parceled `Messenger` object which is a reference to the data which A wants to share. In addition, the intent specifies which component in B should use the parceled object and what this component should do with this object. That is, the application initiating

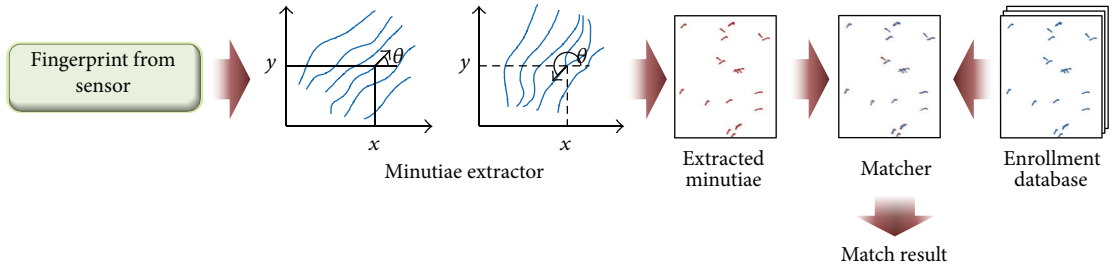


Figure 2: Matching of fingerprints (modified from Box 1 in [6]).

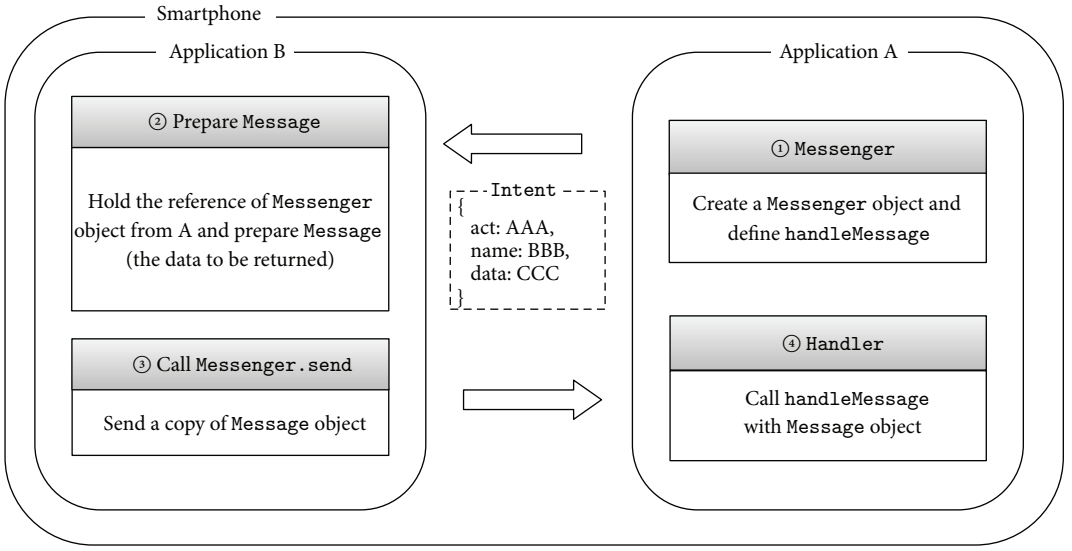


Figure 3: Message-based communication between Android processes.

the communication can designate a specific action the target application should do if only this action is defined in one of the components of the target application.

(2) While the specified component in B is being executed using the parceled object, this component may prepare the data to be returned to A, if required.

(3) The component then calls a public method of parceled object, `Messenger.send`, after setting the data to be returned, that is, a `Message` object, as its parameter. Among the various fields defined in a `Message` object are the `what` and `obj` fields. The `what` field specifies what kind of this message is and the `obj` field stands for data itself.

(4) Next, the `Handler` in the `Messenger` object, which has been used by A to initiate the communication, receives this object, and the `handleMessage` method of the `Handler` utilizes the data contained in the `Message` object.

3. Vulnerability Analysis

The fingerprint recognition service application on a VEGA Secret Note supports three main functionalities, registration, verification, and deletion.

(i) Registration. To register a fingerprint, a user is asked to swipe a fingerprint over the fingerprint sensor. For high reliability, the user should swipe his/her fingerprint multiple times. At the moment when the user's fingerprint is scanned, the scanned fingerprint image is displayed on the screen. See Figure 4.

(ii) Verification. The verification operation is usually used to unlock the smartphone. In this case, the user's task is just to scan his/her finger over the fingerprint sensor on the locked smartphone. The device recognizes the scanned fingerprint and decides whether to permit this user's access based on the matching result. In addition, other applications may request the fingerprint recognition application to activate the verification functionality to verify if the person who attempts to use the application is the legitimate owner of this smartphone.

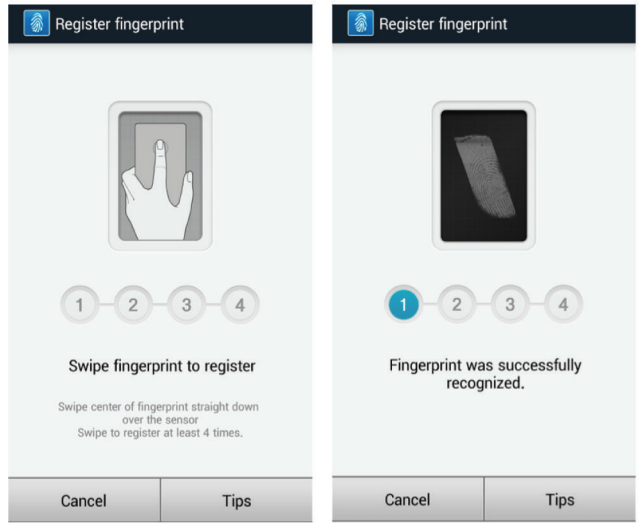


Figure 4: The example procedure for fingerprint registration on VEGA Secret Note.

(iii) Deletion. It is also possible to reset the registered fingerprint by conducting a deletion operation. After unlocking the smartphone by scanning the correct fingerprint, a user may delete the stored fingerprint by scanning his/her fingerprint once again. If this fingerprint matches the registered one, it is deleted from the database.

Because a scanned fingerprint image is displayed on the screen when a user's fingerprint is scanned over the fingerprint sensor, it should be the case that an Android `Bitmap` object in the `View` object related to the fingerprint registration interface is loaded on memory. Therefore, the original fingerprint image may be extracted if we can access the memory location that contains the corresponding `Bitmap` object. Our first attack is to find a way to access the fingerprint image on memory.

On the other hand, a registered fingerprint should be stored somewhere in nonvolatile memory storage for later use in fingerprint verification. Therefore, we may try to find the location of the stored template and restore the original minutia points. Our second attack is to achieve this objective.

3.1. Reverse Engineering of Fingerprint Service Mechanism. First of all, it is important to know where the binary code of the fingerprint recognition service application is located in flash memory. To find this location, we examined the list of running applications when the fingerprint service application is running as shown in Figure 5. To double-check, we also examined the result of the execution of a `ps` command through *Android Debug Bridge (adb)* [14]. As a result, we successfully identified application `com.pantech.app.fingerscan`. The next step was to extract the Android package file of this application for analysis. To this end, we ran an *adb* shell on a PC and tried to extract the package file using the backup functionality of *adb*, that is, by executing `adb backup -apk com.pantech.app.fingerscan`. After acquiring root user permission through rooting, we analyzed the package file. We remark that the root user permission is only required for the analysis of the application package file, but not all actual attacks such as the fingerprint disclosure attack explained in Section 3.2 require this permission.

Next, by analyzing the package file using a few tools such as *dex2jar 0.0.9.15* and *jd-gui 0.3.7*, we found out that this application uses JNI (Java Native Interface) to use the low-level functions implemented in a C++ library for fingerprint management, and we identified the path of this library loaded by the application. As a result, we successfully extracted an Android framework file, `framework.odex` (and its corresponding `framework.jar`), and a shared library file, `libfpc1080_jni.so`. We used `framework.odex` to understand the interaction between `class.dex` and `libfpc1080_jni.so`. For the analysis of `framework.odex`, we used a disassembler, *baksmali 2.0.3*.

The above implementation stack is summarized in Figure 6. According to our analysis, the library file, `libfpc1080_jni.so`, which is an ARM-based dynamic linking library, contains the core routines for fingerprint authentication, in particular, fingerprint image processing. Therefore, in order to find attack vectors against fingerprint authentication service, we traced a source code decompiled from `libfpc1080_jni.so` line by line. The detailed operation mechanism of this library will be explained in Sections 3.2 and 3.3.

3.2. Acquisition of Original Fingerprint Image through a Malicious Application. As briefly explained in the introductory part of this section, a scanned fingerprint image is displayed on the screen when a user's fingerprint is scanned. Therefore, an Android graphic data object such as a `Bitmap` object should be generated to show a fingerprint image while the fingerprint was being enrolled. We tried to find the code segments referring to this object in the decompiled source code, and, eventually, we successfully identified the following

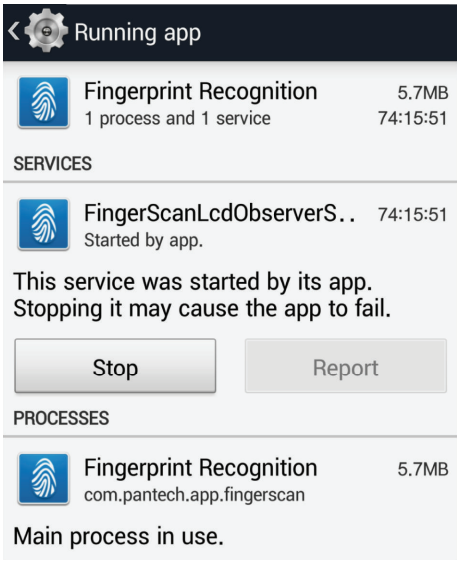


Figure 5: List of Android applications which are running currently.

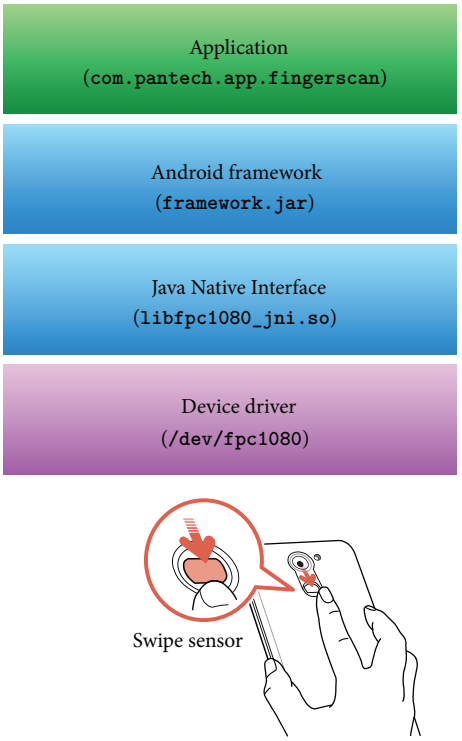


Figure 6: Implementation stack of fingerprint recognition service in VEGA Secret Note (the figure of a smartphone was adopted from [15]).

three locations, (1) setting up a `View` object for the fingerprint image during fingerprint registration; (2) getting a fingerprint image from the sensor during both fingerprint registration and verification; and (3) responding to an authentication request from an external application. Note that, in the last case, the external application may directly handle the `Bitmap` object corresponding to the fingerprint image if the response from the service application includes this object. This finding motivated us to analyze the communication procedure between the service application and the external client application requesting this service.

Figure 7 shows the analyzed result for the organization of the fingerprint authentication service in VEGA Secret Note. As shown in this figure, fingerprint authentication is conducted as follows.

(1) First, a client application A who wishes to use the fingerprint authentication service sends an

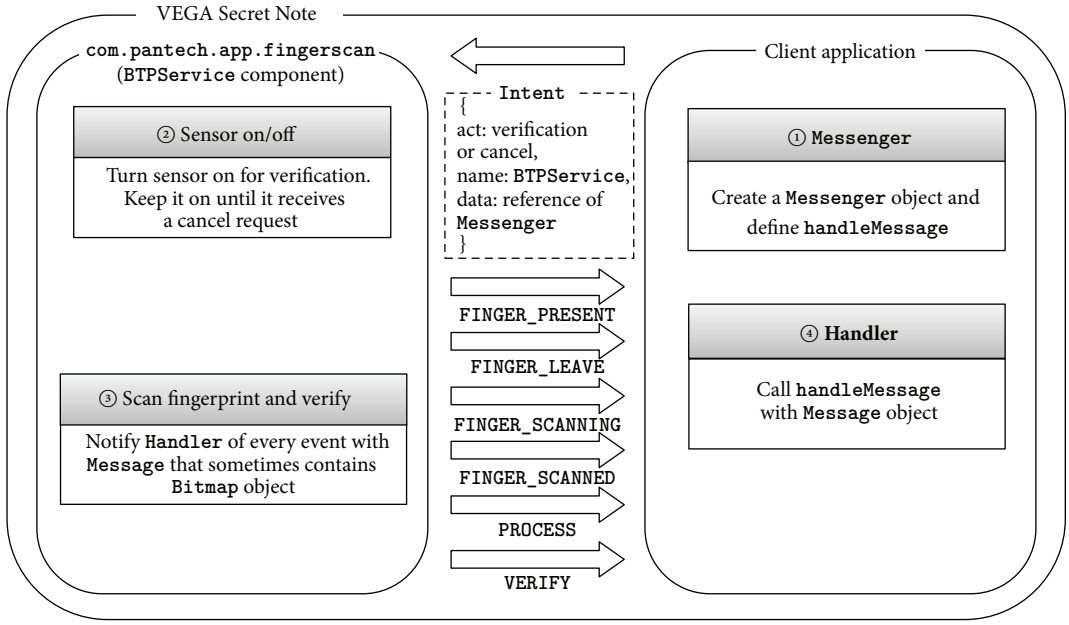


Figure 7: Fingerprint authentication service in VEGA Secret Note.

`Intent` object to initiate a communication with the fingerprint recognition service application, `com.pantech.app.fingerscan`. For this purpose, A does not have any data to share. The `Intent` contains the name of the target component, `BTPService`, and its requested action, `btp.intent.action.verification`. This request allows A to occupy the fingerprint sensor and prevent another application from using the fingerprint authentication service until `btp.intent.action.cancel` is sent by A.

(2) While `BTPService` in `com.pantech.app.fingerscan` is being executed, it turns on the fingerprint sensor and asks the user to scan his/her fingerprint. A `Bitmap` object is defined to contain the scanned fingerprint image.

(3) From the moment that a finger contacts the sensor, `BTPService` notifies A of every event that occurs, which can be one of the following seven events: `FINGER_PRESENT` (the finger touches the sensor), `FINGER_SCANNING` (the sensor is scanning the fingerprint), `FINGER_SCANNED` (the sensor completed a scan), `FINGER_LEAVE` (the finger leaves the sensor), `PROCESS` (the fingerprint verification operation is being done), `VERIFY` (the fingerprint verification operation has been completed), and `IGNORE_NOTIFY` (it seems that this event is not actually used). The procedure to send these notices is as follows. Whenever `BTPService` needs to send a notice, it first creates a new `Message` object and sets the `what` field in this `Message` to one of the constants corresponding to the current state of the sensor, which is defined in the Android framework file, `framework.odex`. When the event is `FINGER_SCANNED`, `BTPService` sets the `obj` field in `Message` to the `Bitmap` object containing the scanned fingerprint image. Finally, `BTPService` calls the function `Messenger.send` defined by A after setting the parameter to its own `Message`.

(4) The `Handler` in `Messenger` receives `Message` from `BTPService`, and then A can utilize the `Bitmap` object in `Message` in the way that its own `Handler.handleMessage` defines.

Our task is now to design a proof-of-concept (PoC) appliction that plays A's role. In addition, this PoC application should contain an event handler function, `Handler.handleMessage`, so that it may export the `Bitmap` object to a standard image file. To achieve this objective, we first analyzed the bitmap configuration of the object and found out that it was `Android.Bitmap.Config.ALPHA_8`. This constant is defined in the `Android.Bitmap.Config` class and it implies that pixels are stored as a single translucency channel. Next, we tried to use `Bitmap.compress` which is a typical method to export a bitmap image from memory to a file. Unfortunately, however, the `Bitmap.compress` method did not support the exportation of an image configured with `Android.Bitmap.Config.ALPHA_8`. To solve this problem, we converted the image into a 32-bit color image using the `Android.Bitmap.Config.ARGB_8888` configuration. However, because the original fingerprint is a grey-scale image, we set $R = 0, G = 0$, and $B = 0$, keeping the original alpha value unchanged. Finally, we succeeded in generating a png-format file containing the scanned fingerprint image.

The above attack vector was implemented in our PoC application. Figure 8 demonstrates its execution result. We call this attack a *fingerprint disclosure attack*. It should be

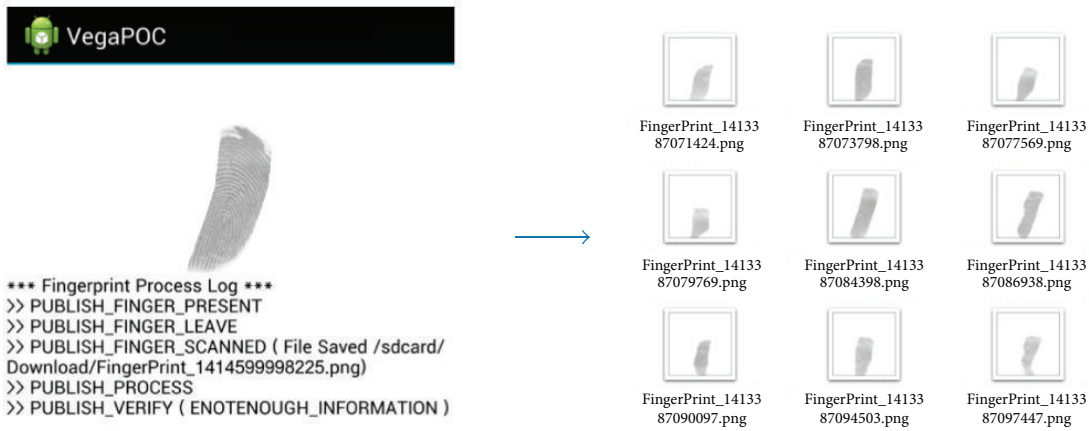


Figure 8: PoC that stores scanned images to files.

noted that this PoC application may be normally installed if only the smartphone owner accepts this application's request for a few application-level permissions. In other words, the above attack does not require any privilege such as root user permission.

The fingerprint disclosure attack can be viewed as a passive type 1 or type 2 attack explained in Section 2.1, though the actual extraction of the full image is done at the moment when the matcher makes the decision for a matching result. As the next step of this attack, an attacker may forge a fake fingerprint with the obtained fingerprint image to mount an attack fooling an image sensor, that is, an active type 1 attack, or to bypass the image sensor by injecting the disclosed image, that is, an active type 2 attack. Moreover, this fake fingerprint may be used for other environments where a fingerprint is used for authentication. For example, an attacker may unlock the victim's doorlock with a fake fingerprint obtained via a fingerprint disclosure attack against the victim's smartphone.

3.3. Extraction of Fingerprint Minutiae from an Encrypted Fingerprint Template. A registered fingerprint should be stored somewhere in nonvolatile memory storage for later use in fingerprint verification. If the fingerprint template is stored as a readable data file, we may try to analyze its structure and get minutiae points. Therefore, our second attack began with finding the location of the stored template file. As explained in Section 3.1, we analyzed the fingerprint application and identified a few essential functions in the library file `libfpc1080_jni.so` dedicated for fingerprint matching. In addition, by analyzing the code segment where the file storing the fingerprint data is accessed, we found out that the name and path of this file were hard-coded irrespective of a specific device. The file name was `csfp.tpl`.

According to our analysis, the file `csfp.tpl` starts with a 48-byte header and the remainder is a main body containing fingerprint data. To be precise, as shown in Figure 9, the header contains a 4-byte identifier (signature) which stands for `CSFP`, a 12-byte field which stands for file version, a 4-byte field which stands for file size, a 16-byte field which stands for MD5 checksum, and some additional data. This structure was found by tracing the header updating function in the library file.

The main body starting at the 49th byte is not in a readable form, but it is encrypted. By analyzing the point in `libfpc1080_jni.so` when a user's fingerprint template data is stored in `csfp.tpl`, we could decode the encryption logic. According to our analysis, the encryption of a template is performed using the CBC mode of AES [16, 17] with a 256-bit key and a 128-bit IV. We also found out that most routines used for this encryption procedure resemble those of OpenSSL [18], which was helpful for our analysis. The key and IV are generated using very simple `for` statements without involving any randomness. As a result, the key and IV are fixed as 0x00010203...1F and 0x00010203...0F, respectively, and all devices use the same key and IV.

We remark that the fact that all devices use the same key may be a critical issue. If the `csfp.tpl` in device A is copied and overwritten to another template file with the same name in another device B, then B will accept the fingerprint of the owner of A. This implies that virtually any attacker may bypass the fingerprint authentication. We will call this attack a *template replacement attack*, which may be regarded as an active type 6 attack according to the taxonomy in Section 2.1. Below is a practical scenario where this attack may be a potential threat. If an attacker has a temporary access to B when B is temporarily unattended (e.g., the owner of B may go to a restroom leaving his/her smartphone on the desk.), the attacker may inject his/her own `csfp.tpl` into B after rooting B. The attacker then may execute a financial transaction which is approved with fingerprint verification, for example, a payment on PayPal [1]. If time is sufficient, the attacker may recover the original template file and unroot B, which prevents the owner of B from recognizing what happened with his/her device.

Because the encryption mechanism and its key information have been analyzed, our next step was to decrypt the encrypted template file and obtain the information about the original template, which we call a *template restoration attack*. We wrote a C code to perform AES decryption using

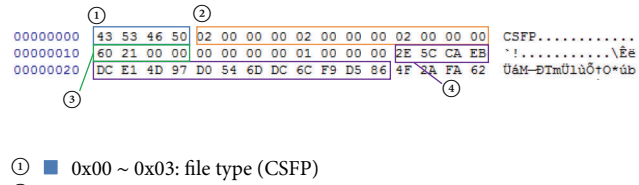


Figure 9: Header format of `csfp.tpl`.

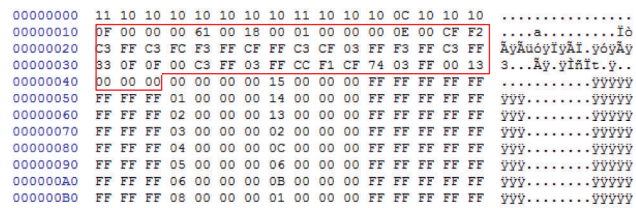


Figure 10: Decrypted content of `csfp.tpl` (extended from Figure 1 in [19]).

Table 1: Size and meaning of each field in a minutia in the decrypted `csfp.tpl` [19].

Index	Size (B)	Meaning
0x00	4	Node (minutia) id
0x04	2	x coordinate
0x06	2	y coordinate
0x08	4	Duplicate count (weight)†
0x0c	2	θ ($0 \leq \theta \leq 47$)
0x0e	32	Additional information of node
0x2e	1	Minutia type
0x2f	4	(Distance to next minutia in bytes)/16

†A weight is the number of occurrences of the same minutia point when a fingerprint is scanned multiple times to enhance accuracy.

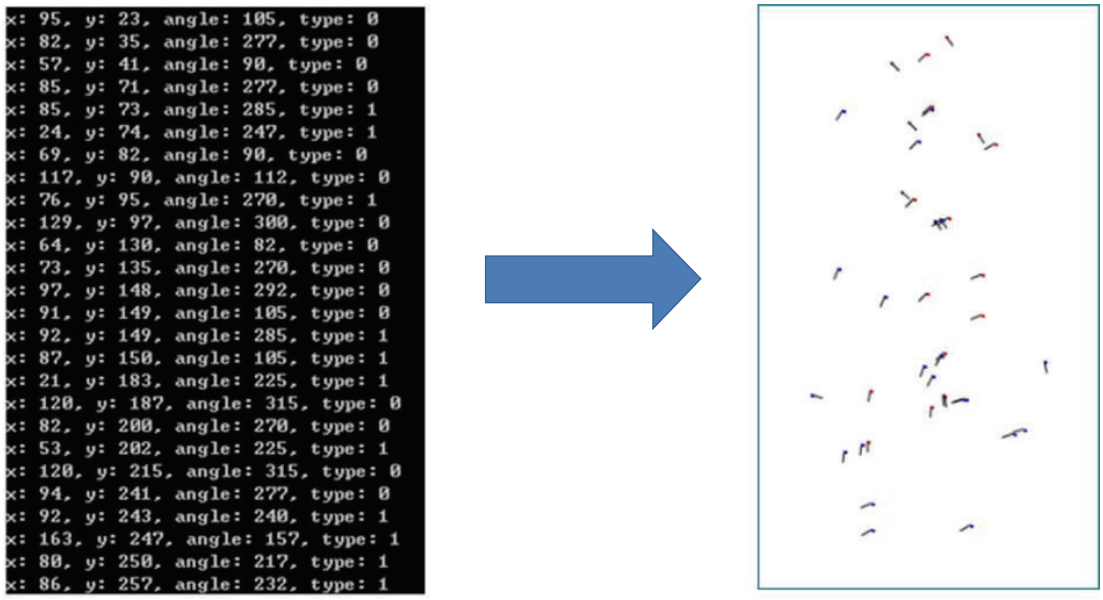
the procedures provided by OpenSSL [18]. For decryption, we used the fixed IV vector and key revealed in the above analysis. This C code reads the `csfp.tpl` as input and outputs the decrypted data. An example result is shown in Figure 10. At a glance, it does not seem to be feasible to identify the meaningful fields such as x, y, and θ and ridge types from the decoded binary data in Figure 10. However, we could identify x and y coordinates by analyzing the code segments in `libfpc1080_jni.so` which refer to the coordinates. In addition, we could identify θ by analyzing the code segment for rotation. According to our analysis, θ was represented in a metric system where a full rotation is defined as an integer, 48. In Figure 10, the highlighted region, which is 51 bytes long, stands for a single minutia point. The meaning of each field constituting this minutia is summarized in Table 1.

We were able to restore each and every minutia point in the original fingerprint template from the encrypted file, `csfp.tpl`, by using our PoC code. This result is shown in Figure 11. In the left-hand part of this figure, we enumerated restored points. These points can be graphically expressed using the convention in the literature. The result is presented in the right-hand part of Figure 11.

The restored template may be used to reconstruct the original fingerprint image [9, 20]. The reconstructed image can be used to forge a fake fingerprint to fool an image sensor. Moreover, the reconstructed image can be used for the same purposes as those of the fingerprint disclosure attack in Section 3.2, for example, to unlock a door. In some sense, however, the reconstruction through a template restoration attack may be a more severe threat because there is no need to execute a malicious application and to wait until the user scans his/her fingerprint.

4. Discussion on Countermeasures

The reason why our fingerprint disclosure attack was successful was that `com.pantech.app.fingerscan` has a component which provides the client application (the malicious application, in our case) with the bitmap image of a fingerprint. Because fingerprint recognition service should be available to client applications, it seems inevitable to allow these applications to call some component providing fingerprint recognition functionalities. However, this component should be designed so that it returns only the result of fingerprint authentication, not the original image itself. Therefore, we suggest that the `BTPService` component should be replaced with such a new component. We also remark that a customized permission `com.pantech.fingerprint.security` for fingerscan recognition has already been defined in the manifest in VEGA Secret Note. If an external application obtains this permission, it can access the fingerprint recognition service including the original fingerprint image, whose property was used by our PoC program. Therefore, we suggest that the permissions for the fingerprint recognition service should be subdivided



(a) (b)

Figure 11: Restored minutiae points and their graphical representation using x, y, and θ (reproduction of Figure 2 in [19]).

into several ones and the permission to access the original fingerprint image should be selectively given.

Regarding the template replacement attack, using the same key and IV vector on all devices is not recommended because an attacker can thwart the authentication test by overwriting the template file in the target device with that extracted from another device. Therefore, we suggest that a distinct key and a distinct IV should be used for each device. Then, even the same template will be transformed into a distinct ciphertext in each device, which implies that simply overwriting a template file does not work. For automatic generation of these device-dependent values, hardware characteristics such as a processor identifier could be used. For example, ARM Cortex-A series including Cortex-A7 processor embedded in Qualcomm Snapdragon CPU of VEGA Secret Note provide the device ID number in the `Primary part number` field in the `Main ID register (MIDR)` [21]. However, this patch does not completely solve the problem, given that the key generation and encryption procedures are easily recognizable by reverse engineering the library file. Thus, an attacker can still mount a template restoration attack that we demonstrated in Section 3.3, even if she/he cannot use the same template file. That is, a forged `csfp.tpl` file encrypted using the key of the target device and the minutiae points from a source device will let the owner of the source device be authenticated by the target device. Therefore, it would be desirable to design a fingerprint recognition procedure so that an extracted template should be useless for other devices even after being properly decrypted. To achieve this goal, cancelable fingerprints [10, 22] may be adopted. By using a noninvertible transform based on device-dependent parameters, an original template may be transformed into a new template before encryption. This transform can be viewed as a kind of error-tolerant one-way hash. Then, the comparison of templates for fingerprint verification is done over a transformed domain as in the case of traditional password authentication where passwords are compared over the hashed domain. Even after an attacker extracts the transformed template, the information about the original minutiae points is protected thanks to the one-way property of the noninvertible transform. This approach prevents a template restoration attack, though it cannot prevent a fake template synthesized according to the rules reverse-engineered from the target device. Therefore, the logic to generate a key and an IV vector should be obfuscated and made hard to be analyzed. For this purpose, we may use the well-known off-the-shelf obfuscation tools.

We may consider a more essential solution including hardware-based isolation technologies such as ARM TrustZone [23]. These techniques might be adopted for secure storage of fingerprint data and isolated execution of fingerprint recognition service.

We finally remark that there is an automated tool to analyze the cryptographic misuse in Android mobile applications [24]. This system only supports the analysis of Dalvik bytecodes, but applications such as our target application that invoke cryptographic primitives from native code cannot be analyzed.

5. Conclusion

By reverse engineering a fingerprint recognition service application, we have identified a few vulnerabilities in the fingerprint recognition service of VEGA Secret Note and demonstrated actual attacks against this service. The technical flaws we have found in this device are a common trap

that developers may fall into. To mitigate these vulnerabilities, we suggested possible countermeasures which may be implemented using well-known techniques in the literature. We expect that the findings we obtained through our analysis may be used as a general guideline to design a secure biometric verification service on smartphones. However, the proposed countermeasures cannot prevent all attacks, for example, a fake template synthesized using the reverse-engineered rules and keys of the target device. Therefore, it would be an important future research issue to develop a more robust countermeasure. In addition, it would be a good research issue to verify whether other smartphones such as Galaxy series and iPhones equipped with fingerprint recognition service are vulnerable or not to the attacks described in this paper.

Disclosure

Most of this research was done when Young-Hoo Jo was a student in Inha University. A preliminary version of this paper was presented at FutureTech 2015 [19].

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgment

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (Grant no. 2014R1A1A2058514).

References

[1] PayPal, "Pay faster with your fingerprint," 2014, https://www.paypal-pages.com/samsunggalaxys5/us/index.html.

[2] Pantech, "Pantech unveils VEGA LTE-A, world's first LTE-A with fingerprint recognition and rear touch," 2013, http://www.pantech.co.kr/en/board/reportBoardView.do?seq=5870&bbsID=report&ulcd=KO.

[3] M. Bishop, *Computer Security: Art and Science*, Addison-Wesley, Boston, Mass, USA, 2003.

[4] N. K. Ratha, J. H. Connell, and R. M. Bolle, "Enhancing security and privacy in biometrics-based authentication systems," *IBM Systems Journal*, vol. 40, no. 3, pp. 614–634, 2001.

[5] A. K. Jain, K. Nandakumar, and A. Nagar, "Biometric template security," *EURASIP Journal on Advances in Signal Processing*, vol. 2008, Article ID 579416, 17 pages, 2008.

[6] A. K. Jain, "Technology: biometric recognition," *Nature*, vol. 449, no. 7158, pp. 38–40, 2007.

[7] ISO/IEC, "Information technology—biometric data interchange formats—part 2: finger minutiae data," ISO/IEC International Standard 19794-2, 2011.

[8] ANSI and INCITS, "American National Statandard for information technology—finger minutiae format for data interchange," ANSI INCITS 378-2009, 2009.

[9] R. Cappelli, A. Lumini, D. Maio, and D. Maltoni, "Fingerprint image reconstruction from standard templates," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 29, no. 9, pp. 1489–1503, 2007.

[10] N. K. Ratha, S. Chikkerur, J. H. Connell, and R. M. Bolle, "Generating cancelable fingerprint templates," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 29, no. 4, pp. 561–572, 2007.

[11] T.-Y. Jea and V. Govindaraju, "A minutia-based partial fingerprint recognition system," *Pattern Recognition*, vol. 38, no. 10, pp. 1672–1684, 2005.

[12] Android Developers, android.os.Messenger, 2014, http://developer.android.com/reference/android/os/Messenger.html.

[13] Android Developers, android.content.Intent, 2014, http://developer.android.com/reference/android/content/Intent.html.

[14] Android Debug Bridge, 2014, http://developer.android.com/tools/help/adb.html.

[15] Pantech VEGA Service, 2014, http://www.pantechservice.co.kr.

[16] National Institute of Standards and Technology, *Recommendation for Block Cipher Modes of Operation*, NIST Special Publication 800-38A, National Institute of Standards and Technology, Gaithersburg, Md, USA, 2001.

[17] NIST Federal Information Processing Standards Publication 197, *Advanced Encryption Standard (AES)*, 2001.

[18] OpenSSL, "The Open Source Toolkit for SSL/TLS," 2014, http://www.openssl.org/.

[19] Y.-H. Jo, S.-Y. Jeon, J.-H. Im, and M.-K. Lee, "Vulnerability analysis on smartphone fingerprint templates," in *Advanced Multimedia and Ubiquitous Engineering*, vol. 354 of *Lecture Notes in Electrical Engineering*, pp. 71–77, Springer, Berlin, Germany, 2016.

[20] J. Feng and A. K. Jain, "Fingerprint reconstruction: from minutiae to phase," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 2, pp. 209–223, 2011.

[21] ARM, *Cortex-A7 MPCore Technical Reference Manual*, Revision r0p3, 2012.

[22] D. Moon, J.-H. Yoo, and M.-K. Lee, "Improved cancelable fingerprint templates using minutiae-based functional transform," *Security and Communication Networks*, vol. 7, no. 10, pp. 1543–1551, 2014.

[23] ARM, "TrustZone," 2014, http://www.arm.com/products/processors/technologies/trustzone/index.php.

[24] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in Android applications," in *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*, pp. 73–83, ACM, Berlin, Germany, November 2013.