TechCom ComTech Company
2147 G.G. Brown Laboratory
MI 48109

To: Dr. Vigiletti (CTO) & Members of "Imagician" Development Team
From: Tianyao Gu, Research Engineer
Subject: Security Defects in "Imagician" App and Best Practices for Digital Watermarking
Date: April 16, 2022

I. Foreword

In the company's value proposition, the "Imagician" App (referred to as "App" below) aims at protecting the copyright of users' digital assets via digital watermarking technology. However, I recently found serious security flaws in the App that threaten users' property and privacy. Therefore, I researched the attack and defense mechanisms in digital watermarking and determined the best practices for implementing the App. This memo presents my research results and proposes urgent changes to the current App design before the team proceeds with the implementation.

II. Summary

The memo reviews the current design of the App and identifies its vulnerability to forgery, modification, overlay, phishing, and denial of service attacks. The memo also discussed why the current design could leak users' privacy data, including the watermark messages, the original images, and the creation timestamps. To mitigate these risks, I recommend that the team takes the following steps:

1. Implement a sign-in system to prevent forgery, modification, and denial of service attacks.
2. Let the App check existing watermarks before embedding a new one. Also, let the backend server verify the App's authenticity through the Google Play Integrity API.
3. Let the App encrypt all the watermark messages with a random key generated locally, embed the key in the image, and use the hash of the key as the watermark record ID.

III. Introduction to Current Security Design

As "security" is one of the key characteristics in the company's value proposition, the development team has made several attempts to guard the App against malicious operations. I will introduce these attempts in this section and discuss why they are inadequate in section IV and V. The recommended solutions can be found in section VI.

Currently, the team has implemented a sophisticated machine learning model known as "StegaStamp" [1] to secretly embed information in images. The model is robust to modification, compression, and format transformation. Experiments show that the watermark is retrievable even when the picture is printed out and re-scanned with a camera. The biggest drawback of the model, however, is its low data capacity. The model can only store 56 bits of information in a 400×400 colored image, which is insufficient for embedding the complete copyright messages

directly. As a compromise, the team chose to store all the copyright messages in the server's database and embed only a tag in each image for retrieving the message. While the design choice enhances the App's usability, it also brings about challenges in data safety.

To ensure the security of data transmission, the front-end and the back-end communicate through the HTTPS protocol [2]. The App allows users to enter a password to further encrypt the watermark messages. The App does not require users to upload their images to the server, but applies a checksum to ensure that adversaries cannot reuse the identifier on another image or modify the image in an undetectable way. Moreover, the App can determine which part of the image is modified - the current approach is to split the image into multiple blocks and verify the checksum of each of them.

Figure 1 visualizes the current design of the system's architecture, as is described above.

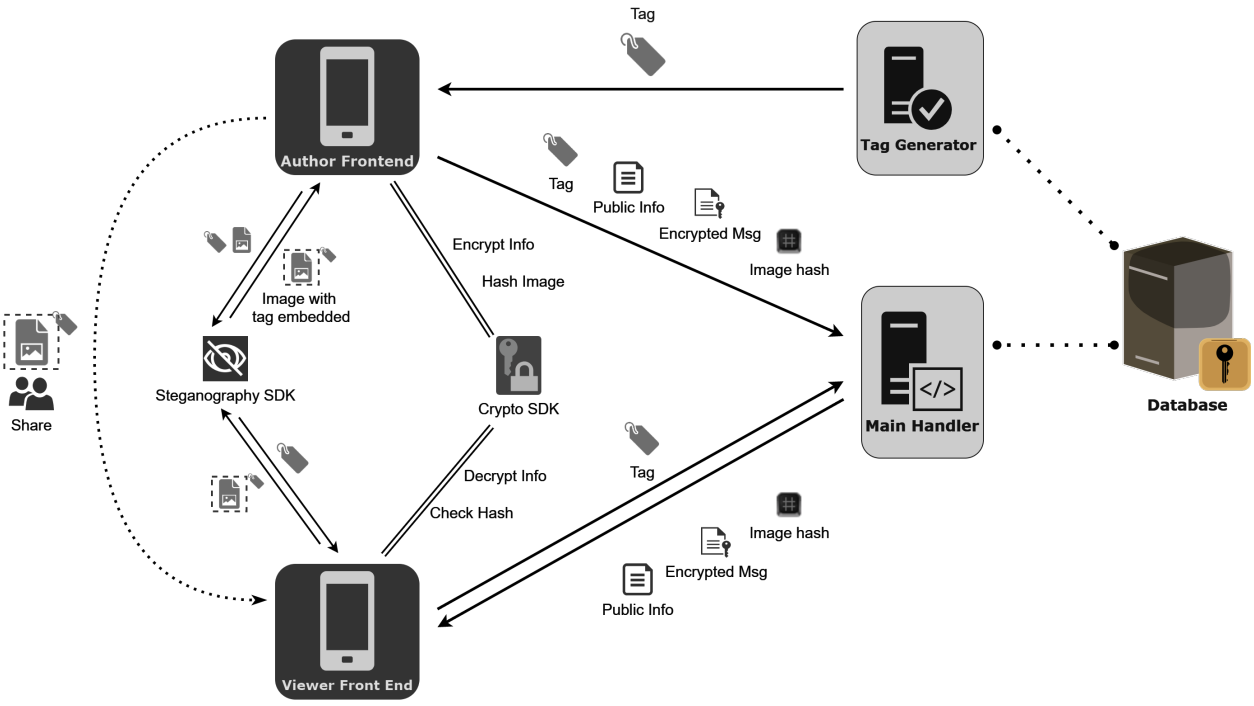

Figure 1: Initial System Architecture Design

IV. Security Threats, Assumptions, and Criteria

It's important to first identify the security threats before scrutinizing the design. Since the App is designed mainly for commercial purposes, there exist sufficient economic incentives for malicious attacks. By the client's requirement, the project should be open-source on GitHub. Namely, everyone can access the source code of the App front-end and back-end. The attacker could intentionally tamper with the watermark and distribute the image after modification. The attacker could pretend to be a normal user and send malicious requests to the back-end server. The attackers could also act as a Person-In-The-Middle or eavesdropper on the Internet. The only assumption we will make is that the attackers cannot intrude upon the user's Android system and

steal the local credentials. The assumption is reasonable as our App is designed to run only on Android 10+, which offers a much better system-level protection than the older versions.

The user interview shows that a viable product for digital watermarking should satisfy the following security criteria.

- **Anti-forgery:** Others cannot forge the digital watermark of the author.
- **Anti-modification:** Modification to the image can be detected using the watermark.
- **Anti-removal:** Attackers should not be able to remove the watermark without damaging the image.
- **High availability:** The service should almost always be available to the users.
- **Privacy:** Sensitive data should not be sent to / stored on the server unencrypted.

In the next section, I will show why these criteria are not met in the current design.

V. Security Vulnerabilities in Current Design

Forgery Attack

In the current design, a user is allowed to embed any message in an image, which means an attacker can forge another user's watermark for malicious purposes, such as fraud and slander.

Modification Attack

In a modification attack, the adversaries modify an image while still letting others believe it is sent by the claimed author. The attacker can exploit the same vulnerability as in the forgery attack. After modifying the image, the attacker can forge the author's watermark and upload the new checksum of the image. The modification would be undetectable unless the original image is given.

Overlay Attack

Even though the digital watermarking in [1] is robust to removal, an attacker can cover the existing watermark with another layer of the watermark using the App to change the ownership of the image.

Phishing Attack

An attacker could trick the user to connect to a phishing server by publishing a maliciously modified copy of the App or manipulating the Internet routing (through DNS hijacking, for example). The attacker would then be able to access the user's watermark data and alter them arbitrarily.

Denial of Service Attack

Since the disk space of our server is limited, attackers can keep uploading large files as messages to overwhelm the server, so that it can no longer respond to new requests from other honest users. In addition to malicious attackers, some end users might misuse our App as a cloud storage platform, consuming large amounts of disk space.

Privacy Exposure

- **Malicious Read**

In the current design, the embedded identifier is just the database record id, which is consecutive and increasing. Since both the App and the steganographic algorithm are open-source, there's no way to prevent users from creating a fake identifier. An attacker can hence request for any record in the database by exploiting the app's back-end API and enumerating all the valid identifiers, even if they do not have access to the corresponding images.

- **Exposure of Messages**

The embedded messages are stored as plain text on the server by default, which could leak due to the malicious read vulnerability. Although the author may choose to encrypt the message with a password, it is not an ideal solution.

On the one hand, an author may want to embed their contact information in plain text so that the image viewers can ask for permission before reproducing it. However, it doesn't mean that the author is willing to expose the information to everyone on the Internet.

On the other hand, people tend to choose weak passwords and reuse them fairly often. Study of password habits [3][4] show that 12.45% of the passwords can be cracked within a thousand attempts by brute force, and each password is used at about 6 different sites on average. Therefore, it's better not to fully count on the authors to keep their private data well protected.

- **Exposure of Images**

The base image for a digital watermark may contain sensitive information. Even if the image itself is not uploaded, its checksum could expose some information. In the current design, an image is divided into blocks and a checksum is calculated for each block, which is known as the Electronic Codebook (ECB) mode [5]. Unfortunately, the scheme can expose certain information about the image. As is shown in Figure 2, the overall pattern of an image is still identifiable after hashing, especially when the image contains large areas of uniform color. The checksum also exposes the size of the image, since a larger image has a longer checksum. In addition, an attacker can check whether the image contains a certain pattern if the block size is too small.

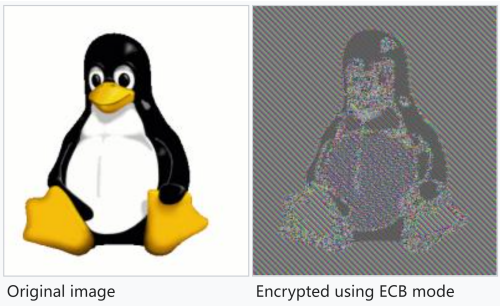

Figure 2: Large-scale features of the image remain recognizable despite the ECB encryption[6]

- **Exposure of Creation Date**

The vulnerability exists because the embedded identifier is generated in increasing order. The attacker can estimate the creation date of the watermark by extracting and comparing the

identifiers. For instance, the attacker may request a dummy watermark every day to keep track of how large the identifier has grown. Then the attacker can determine the exact date when a watermark is created.

VI. Solutions

Towards Forgery Attack

To prevent forgery attacks, we need a mechanism to verify the author's authentication when retrieving the watermark.

- **Cryptographic Signature based on Public-Private Key Mechanism**

One solution, as is suggested in [7], is to append the author's cryptographic signature of the image hash in the watermark. As long as the adversaries do not know the author's private key, they cannot forge the signature in the watermark.

The solution is secure from the cryptography perspective. However, it requires the author to publish their public key on a trusted platform so that others can verify the signature. [1] In practice, an author may not want to post a 256-bit random code on their social media homepage, and most image viewers would never bother verifying the code. To overcome this problem, we would need to provide a name service to bind each public key with a human-readable identifier, such as a unique token set by the author. Another problem is that the private key might get lost when the author uninstalls the App or changes the device. We would need to offer a backup service to overcome this problem, which brings extra risks of leaking.

- **Sign-in System**

For mobile apps, a better defense approach would be to require users to register and log into an account before creating any digital watermark, so that the server can auto-complete the author's information in the watermark and give a guarantee of its genuineness. To forge the author's identity, attackers have to crack the user's sign-in password. Common prevention practices include rejecting simple passwords, hashing the password with random salts before storing them in the database, and limiting the number of failed login attempts.

Towards Modification Attack

The sign-in system in the previous section also mitigates the modification attack problem. As long as the adversaries cannot forge the author's watermark, they cannot modify the image in an undetectable manner, since any modification would change the checksum of the image, leading to a mismatch with the value in the database. Although adversaries can still modify the image and add another watermark, they cannot pretend to be the original author.

Towards Overlay Attack

- **Revise the Steganographic Algorithm**

To prevent the overlay attack, one possible approach would be to revise the steganographic algorithm so that any trace of old watermarks in the image would invalidate the new watermark.

[1] Note that it's insufficient to just put the public key into the watermark message. The author needs to show that the key belongs to herself/himself.

However, there is no off-the-shelf solution to achieve this goal without hurting the robustness of the algorithm. It is challenging for the development team to come up with a working solution from scratch.

- **Embed Watermark in the Cloud**

A more feasible way is to let the App check if there is an existing watermark in the image before embedding a new one. However, the adversaries can circumvent this defense. Since the source code is public, attackers can compile a modified version of the App that skips the step of checking existing watermarks. They can even reproduce the steganographic algorithm without using our app. Therefore, we need additional security mechanisms so that the back-end API can reject these unauthorized watermark.

One solution would be to disable the current API for uploading watermark messages and move the whole process of steganography into a Cloud server. The server process will abort if it finds an existing watermark in the image provided by the user. Otherwise, the server will write the watermark data directly into the database and return the image with the new watermark. Even if the adversaries can embed an identifier in the image offline, they cannot inject the corresponding watermark messages into our database.

The solution is secure and easy to implement. However, it obviously breaks the Privacy rule in section IV. Users may not be willing to upload their private images, even if the images will be deleted from the server after the computation.

Besides, putting the steganographic computation in the cloud will significantly increase the running cost of our service. The embedding and extracting processes would consume lots of CPU and GPU resources. To guarantee high availability, the company needs to rent more machines. Also, uploading and downloading images require high Internet bandwidth, thus increasing the Internet traffic cost.

- **Check App Integrity by Close-Sourcing Part of the Source Code**

Another way to prevent unauthorized embedding is to let the server verify the integrity of the user's front-end before handling any sensitive request from it.

One way to implement this idea is by close-sourcing part of the source code, in which the App should check the integrity of itself, as described in [8]. To prevent the adversary from skipping the check, the close-sourcing code should also provide a proof of execution. For example, the program may run a secure key-hash function [9] to calculate an authentication code for a dynamic message offered by the server. Hashing prevents attackers from stealing the key by capturing network packages and the dynamic message prevents attackers from performing replay attacks.

The drawback is that the close-sourcing code may still be reverse-engineered regardless of how well we pack the App. If any secret key in the code leaks, the defense will become an empty shell.

- **Check App Integrity with Google Play Integrity API**

Considering that the App is only for Android, I suggest the team utilize Google Play Integrity API [10], which helps Android App developers to determine whether they are interacting with an

unmodified binary that's recognized by Google Play.

Figure 3 demonstrates how the API works. For our use case, every time the App server received a request for adding a watermark, it first returns a unique nonce. The App then calls the Play Integrity API, transferring the nonce as a parameter. The API will return an integrity verdict token, which the App should pass back to the server. Finally, the App server verifies the token by sending it to Google Play's server and decides whether to proceed with the request.

One disadvantage of this approach is that the user's Android device needs to access the Google Play services, which could be a problem in countries like China and Cuba. For example, Google Play services are currently not available on some new Huawei device models due to the US government restriction [11].

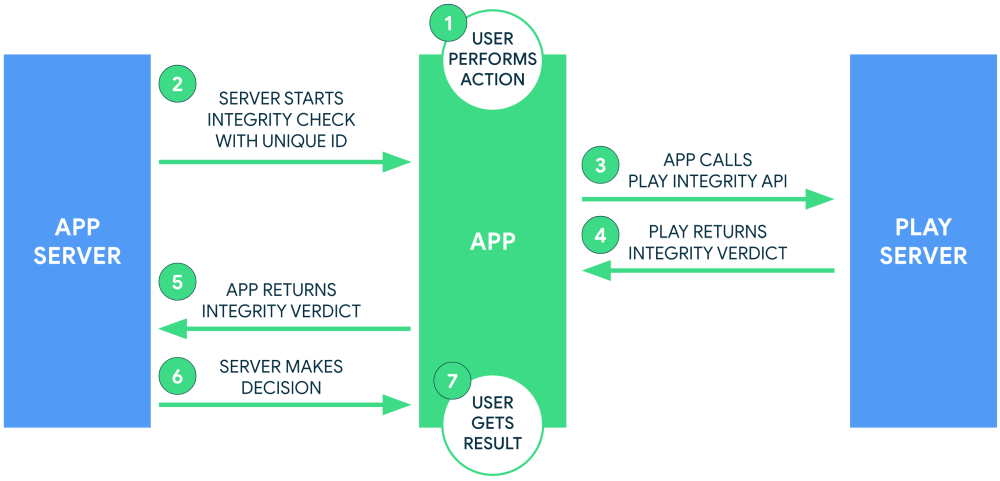

Figure 3: Flowchart for the Google Play Integrity API [10]

Towards Phishing Attack

To avoid the phishing attack, the company should encourage users to download the App from trusted platforms, such as the Google Play store. The App can then ensure that the routing is correct by verifying the server's certificate.

Towards Denial of Service Attacks

The App can apply the reCAPTCHA service [12] to avoid spamming. The App should also limit the number of requests per day for each user and the maximum file size in each request.

Towards Privacy Exposure Problems

- **Against Malicious Read and Exposure of Creation Date**

The solution to these problems is to randomize the embedded identifier. The App front-end could generate a pseudo-random 56-bit value as the identifier in each request, and the server just needs to check that it has never been used before. By calculation, even if the App handled a million requests, it would consume only 1/70000000000 of the valid identifiers. Therefore, it's statistically impossible for an attacker to read any watermark records by guessing the identifier.

Meanwhile, since the identifiers are not generated in a fixed order, the problem of exposing the creation date doesn't exist anymore.

- **Against the Exposure of Messages and Images**

Even if the attacker cannot read the watermark records directly, user data might still leak due to the phishing attack. Also, the team cannot guarantee that the database is 100% secure. Therefore, it's ideal to encrypt all the messages (including the image's checksum), whether the author set a password or not. The App should embed the encryption key together with the identifier in the image so that the viewers can decrypt the messages, and upload only the identifier to the server. Unfortunately, the capacity of the steganography algorithm is very limited. In the original paper [1], each image can only store 56 bits of information, which is by no means sufficient for storing both the identifier and the key.

One improvement approach is to set all the 56 bits as the key and let both the author's and viewer's front-end apply a secure hash function to obtain the identifier. Namely, whoever has access to the image can calculate the identifier based on the key, but it's difficult for the attacker to calculate the key based on the identifier. In fact, the 56-bit random key is much more secure than most user-specified keys, which have an entropy of about 42 bits on average [13]. If the hash function takes 0.01 seconds on the user's device, and the attacker has 10,000 times more computation power than the user, it will still cost the attacker about $\frac{1}{2} \cdot 0.01 \cdot \frac{1}{10000} \cdot 2^{56} sec \approx 1142$ years to crack the key of a target victim user by the enumeration. Also, since the output of a good hash function is still pseudo-random, the attackers still cannot perform malicious reads.

Note that the defense does not prevent the attack on all the records as a whole, which might be a problem when the service reaches a certain scale. For instance, if there are one million records in the database, it will take only a few hours to crack a random one of them based on the previous settings. Nonetheless, it's generally not interesting for an attacker to hack into the database and spend a large amount of computation power just to decrypt the watermark of some random users.

VII. Conclusion - The High-level Mindsets

The memo reviewed many technical approaches to improve the security level of the application. However, you shouldn't expect the users to understand them. The App shouldn't require users to perform unconventional operations, such as exporting a private key. Even better, you may hide all those technical details from the users. For example, users need not know that their watermark data are stored on back-end servers rather than directly in the images. These kinds of information can be listed separately in a "Terms of Service and data safety" document.

By the client's requirement, the project needs to be open-source on GitHub, which could bring about more attacks. On the other hand, however, this requirement may help you throw away the view of "Security through obscurity" and design the system according to the "Kerckhoffs's principle"; that is, a cryptosystem should be secure, even if everything about the system, except the key, is public knowledge [14].

Finally, cybersecurity today is like a game of cat and mouse - attackers and defenders are always in a back-and-forth battle, so please be prepared to react to any new emerging threats.

References

[1] M. Tancik, B. Mildenhall, and R. Ng, "Stegastamp: Invisible hyperlinks in physical photographs," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[2] E. Rescorla, *HTTP Over TLS*, RFC 2818, May 2000. DOI: 10.17487/RFC2818. [Online]. Available: https://www.rfc-editor.org/info/rfc2818.

[3] D. Wang and P. Wang, "On the implications of zipf's law in passwords," in *Computer Security – ESORICS 2016*, I. Askoxylakis, S. Ioannidis, S. Katsikas, and C. Meadows, Eds., Cham: Springer International Publishing, 2016, pp. 111–131, ISBN: 978-3-319-45744-4.

[4] D. Florencio and C. Herley, "A large-scale study of web password habits," in *Proceedings of the 16th International Conference on World Wide Web*, ser. WWW '07, Banff, Alberta, Canada: Association for Computing Machinery, 2007, pp. 657–666, ISBN: 9781595936547. DOI: 10.1145/1242572.1242661. [Online]. Available: https://doi.org/10.1145/1242572.1242661.

[5] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*, ser. Discrete Mathematics and Its Applications. CRC Press, 2018, ISBN: 9780429881329. [Online]. Available: https://books.google.com/books?id=YyCyDwAAQBAJ.

[6] L. Ewing, *File:tux ecb.jpg - wikimedia commons*, May 2006. [Online]. Available: https://commons.wikimedia.org/wiki/File:Tux_ecb.jpg.

[7] I. J. Cox, M. L. Miller, J. A. Bloom, J. Fridrich, and T. Kalker, "Chapter 10 - watermark security," in *Digital Watermarking and Steganography (Second Edition)*, ser. The Morgan Kaufmann Series in Multimedia Information and Systems, I. J. Cox, M. L. Miller, J. A. Bloom, J. Fridrich, and T. Kalker, Eds., Second Edition, Burlington: Morgan Kaufmann, 2008, pp. 335–374.

[8] K. Król, *Enhanced android security: Build integrity verification*, Jun. 2020. [Online]. Available: https://www.nomtek.com/blog/enhanced-android-security.

[9] M. Bellare, R. Canetti, and H. Krawczyk, "Keying hash functions for message authentication," Springer-Verlag, 1996, pp. 1–15.

[10] Google, *Play integrity api*, Jan. 2022. [Online]. Available: https://developer.android.com/google/play/integrity.

[11] T. Ostrowski, *Answering your questions on huawei devices and google services. - android community*, Feb. 2020. [Online]. Available: https://support.google.com/android/thread/29434011?hl=en.

[12] Google, *Recaptcha v3*, Oct. 2021. [Online]. Available: https://developers.google.com/recaptcha/docs/v3.

[13] J. Massey, "Guessing and entropy," in *Proceedings of 1994 IEEE International Symposium on Information Theory*, 1994, pp. 204–. DOI: 10.1109/ISIT.1994.394764.

[14] A. Kerckhoffs, "La cryptographie militaire," *Journal des Sciences Militaires*, pp. 161–191, 1883.