# Automated Cash Mining Attacks on Mobile Advertising Networks

Woojoong Ji[1(✉)], Taeyun Kim[1], Kuyju Kim[2], and Hyoungshick Kim[1]

[1] Sungkyunkwan University, Suwon, Republic of Korea
{woojoong,taeyun1010,hyoung}@skku.edu
[2] AhnLab, Seongnam, Republic of Korea
kuyju.kim@ahnlab.com

**Abstract.** Rewarded advertisements are popularly used in the mobile advertising industry. In this paper, we analyze several rewarded advertisement applications to discover security weaknesses, which allow malicious users to automatically generate in-app activities for earning cash rewards on advertisement networks; we call this attack *automated cash mining*. To show the risk of this attack, we implemented automated cashing attacks on four popularly used Android applications (`Cash Slide`, `Fronto`, `Honey Screen` and `Screen Stash`) with rewarded advertisements through reverse engineering and demonstrated that all the tested reward apps are vulnerable to our attack implementation.

## 1 Introduction

In rewarded advertisement services, the most important security issue is the detection of (artificially created) fraudulent user engagement activities that have no intention of generating value for the advertiser [6]. Recently, there have been few studies [2,3] that analyze the potential security risks in this domain. Cho et al. [2] demonstrated that six Android advertising networks were vulnerable to automated click fraud attacks through the Android Debug Bridge (ADB).

In this paper, we extend Cho et al's attack model of relying on automated input sequences at the user interface level into a more sophisticated attack called *automated cash mining*, which allows an attacker to automatically generate in-app activities at the network packet level. This is a significant advancement from previous studies [2,3] that merely showed potential weaknesses in rewarded advertisement applications.

To show the feasibility of our attack, we analyzed four popularly used reward apps (`Cash Slide`, `Fronto`, `Honey Screen` and `Screen Stash`) by reverse engineering and packet analysis, and we found that all tested reward apps are vulnerable to automated cash mining attacks.

## 2 Mobile Advertising Network

To provide a better understanding of automated cash mining attacks, we first present the typical model of a mobile advertising network for reward applications.


J. Jang-Jaccard and F. Guo (Eds.): ACISP 2019, LNCS 11547, pp. 679–686, 2019.
https://doi.org/10.1007/978-3-030-21548-4_40

In the advertising network model, there are four main entities: (1) publisher, (2) advertising network, (3) advertiser and (4) reward app.

A publisher (i.e., app developer) develops a reward app with the SDK library for an advertising network and releases it to users. Advertisers can add new advertisements to the advertising network when they want. The reward app can periodically fetch a list of advertisements from the advertising network via its SDK library incorporated in the app itself. In general, the advertising network manages publishers and advertisers as a moderator in this model.

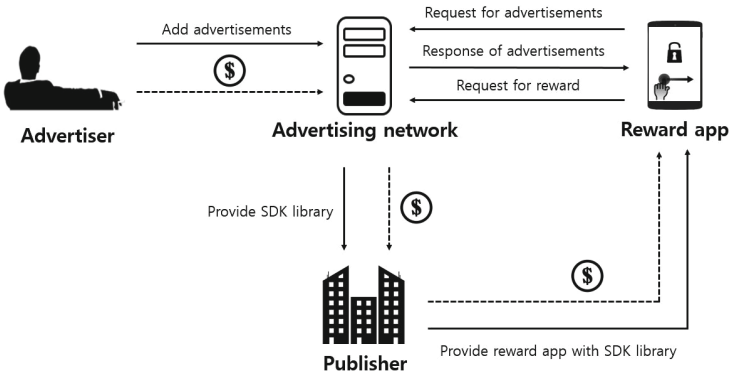


**Fig. 1.** Business model of advertising networks with reward apps.

Whenever a user watches advertisements on the reward app, the app then reports the user engagement activities referred to as '*impressions*' to the advertising network. This reporting process is triggered by sending a request message for a reward from the reward app to the advertising network. Consequently, the user would be rewarded for performing such activities within the reward app. For this rewarding process, advertisers pay money to the advertising network depending on the number of impressions they receive; the advertising network pays money to the publishers; and a publisher finally pays money to its reward app users. This process is depicted in Fig. 1.

## 3 Automated Cash Mining Attack

Our goal is to generate network traffic that emulates a real reward app. To artificially generate reward request messages for automated cash mining attacks, we must know how a reward app interacts with its advertising network. Therefore, we carefully analyzed the network messages exchanged between a target reward app and its advertising network server by a web debugging proxy.

From our traffic analysis of the four reward apps tested (`Cash Slide`, `Fronto`, `Honey Screen` and `Screen Stash`), we found that reward apps' request messages

generally include an *authentication code* to verify the authenticity of reward messages. The authentication code is *newly* calculated every time in a *secret* function $f$ (e.g., encryption or cryptographic hash function) with a reward app's internal parameters such as advertisement identifier, user identifier, timestamp, reward amount, etc. to prevent replay attacks. Therefore, to implement automated cash mining attacks, the most challenging task is to analyze how such an authentication code is computed in a function $f$, which is internally implemented in each reward app. To achieve this, we examined the procedure of authentication code generation in each reward app through reverse engineering.

If we are able to compute authentication codes with the internal parameters of a reward app, we can systemically generate reward request messages containing valid authentication codes.

## 4 Implementation

To perform automated cash mining attacks, we need to intentionally craft reward request messages to deceive a victim advertising network. Therefore, we first analyze the structure of network messages used in genuine reward apps and then generate reward request messages based on our analysis results.

### 4.1 Analysis of Network Messages

To generate reward request messages, it is necessary to analyze the messages exchanged between the advertising network and the reward app. To analyze the HTTPS traffic, we used *Fiddler* (https://www.telerik.com/fiddler) to mount a man-in-the-middle attack. In practice, there are several methods (e.g., certificate pinning [5]) to prevent man-in-the-middle attacks on HTTPS but all the tested reward apps failed to prevent our traffic analysis.

There are two types of advertisements in reward apps: (1) advertisements that have a reward and (2) advertisements that do not have a reward. After receiving a request to deliver the current advertisement list to the reward app, the advertising network server responds with the requested advertisement list in JavaScript Object Notation (JSON) format.

We found that the reward amounts can be changed dynamically depending on the type of advertisement (e.g., some advertisements do not have a reward at all). Therefore, to maximize their gain, attackers must first obtain the information about the reward amount for each advertisement so that they can selectively generate request messages only for advertisements with a (high) reward. For reward apps, it is essential to maintain up-to-date reward amount information for advertisements. In the reward apps on Android, such information is typically stored on a system cache and/or a database file. We experimentally observed that a request message for new advertisements would be generated in most reward apps when they are restarted after erasing the system cache and database file for advertisements. Therefore, in a reward app, we can try to analyze the structure of the request message for new advertisements by intentionally

erasing the system cache and database file and restarting the app itself. If we completely analyze the structure of the request message for new advertisements, we can also generate the request message based on our analysis results and send the message to obtain the information about advertisements. Thus, we can selectively generate request messages only for advertisements with rewards to boost the efficiency of automated cash mining attacks.

### 4.2 Analysis of Authentication Code Computation

As explained in Sect. 3, in reward apps, the authentication code is typically used to verify the integrity and authenticity of reward messages. For example, in `Cash Slide`, *key* and *ts* are used to compute the authentication code. `Cash Slide` calculates a *key* value by combining the user's name (*c_nickname*) and timestamp for the purpose of preventing replay attacks. The *ts* field provides the timestamp that is used when generating the *key* value. As a result of the analysis, we discovered that only the *key* and *ts* fields are periodically refreshed, and the rest of the fields are always fixed. Hence, the attacker only needs to dynamically calculate the *key* and *ts* fields to make valid request messages.

Therefore, we need to analyze the reward apps' codes for computing authentication codes and reimplement them to automatically generate valid reward request messages. To achieve this, we analyzed the APK files of each reward app. We note that Android apps are written in Java, and they are compiled to Java byte code and then translated into the Dalvik executable (DEX) format [4]. Using an APK extractor, we first extracted a target reward app's APK file from the app and then converted the extracted APK file into JAR files. Next, we used a decomplier (e.g., *JD-decompiler*) to decompile the JAR files and analyze the decompiled source codes. In the decompiled source codes, we can find a few candidate functions using text keyword matching (e.g., crypto, key and AES).

```
public static String a = "1a2b3c4d5e6f7g8h9i1j2k3l4m5n6o7p";
public static String b = "1a2b3c4d5e6f7g8h";
 ...
try
{
  Object abc = new javax/crypto/spec/SecretKeySpec;
  ((SecretKeySpec)abc).<init>(a.getBytes(), "AES");
  Object def = b;
  def = Cipher.getInstance("AES/CBC/PKCS5PADDING");
                          ...
  abc = ((Cipher)def).doFinal(paramString.getBytes());
}
 ...
```

**Fig. 2.** Function for authentication code computation in `Cash Slide`.

To identify the candidate functions, we used dynamic analysis tools (*Frida*, https://www.frida.re/ and *AppMon*, https://dpnishant.github.io/appmon/) to analyze how an authentication code is computed by using a cryptographic algorithm (e.g., AES or MD5) with its parameters. To use *Frida* and *AppMon* on an Android smartphone, the smartphone must be rooted. However, none of the apps tested had any anti-rooting mechanisms.

Among the apps tested, two apps (`Cash Slide` and `Fronto`) only used an encryption algorithm to compute authentication codes. The remaining apps (`Honey Screen` and `Screen Stash`) only used a cryptographic hash function (e.g., MD5) instead. Figure 2 shows the decompiled code for computing authentication codes in `Cash Slide`. From this code, we can see that AES in CBC mode with PKCS5 padding algorithm with the hard-coded encryption key and initial vector is used to compute authentication codes.

### 4.3 Generation of Messages to Mimic Reward Apps

The overall process of our automated cash mining attack is as follows:

1. Send a request message to obtain information about the advertisement (e.g., advertisement's identifier, user identifier, timestamp and reward amount).
2. Send reward request messages periodically for advertisements with a (high) reward.

The reward request messages that will be sent to the advertising network can be easily generated using Request to Code [1], which is a `Fiddler` extension.

After making such a request, we use the advertisement identifier and other information of advertisements with rewards from the list to automatically calculate authentication codes.

## 5 Experiments

We analyzed four popularly used reward apps (`Cash Slide`, `Fronto`, `Honey Screen` and `Screen Stash`) and implemented automated cash mining attacks to generate *valid* reward request messages to mimic the messages generated by human users using those reward apps. To evaluate the performance of our attack implementations, we created log files to record the response messages from the advertising network. We demonstrate that our implementation of automated cash mining attacks can be used to obtain rewards from reward apps in an automated manner, and we confirmed that there were only a few defense mechanisms in the four reward applications that we investigated. We also discovered that it is possible to implement automated cash mining attacks to financially damage real-world mobile advertising networks with all four reward apps. The detailed experiment results are explained in the following sections.

### 5.1 Data Protection

We analyzed the features that the developers used to protect their data, such as hash and cryptographic functions. As shown in Table 1, two of the four apps used AES encryption but the encryption keys and IV vector values were stored in plaintext form in the APK files. As a result, malicious users can easily access these values. In addition, we found that both the encryption keys and the IV vector values were fixed, and they were not changed following encryption. Attackers can decrypt the encrypted messages by obtaining the encryption keys and IV vector values that the app uses to encrypt messages.

**Table 1.** Security mechanisms used in reward apps.

| Application | Data protection | | Reward policy | Defense mechanisms | | | |
|---|---|---|---|---|---|---|---|
| | Encryption | Hash | Fixed reward | Rooting check | Hooking check | TLS certificate | Code obfuscation |
| **Cash Slide** | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ |
| **Fronto** | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✓ |
| **Honey Screen** | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ |
| **Screen Stash** | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ |

✓=used; ✗=not used

As shown in Table 1, the two remaining apps used the MD5 hash function instead of AES encryption, and the hash function was applied to one of the fields in the message. Since the hash function is not an encryption method, the attackers can generate the same output by analyzing the input value through source code analysis.

### 5.2 Reward Policy

Many reward applications have mechanisms to protect themselves. The reward apps have reward policies, such as a time limit to request a reward or verification of a reward. We analyzed the reward polices of the four reward apps. As shown in Table 1, in two of the four apps, the users receive a fixed amount of rewards. For the other two apps, the attackers can change the reward value. In this case, the attackers can obtain more rewards than the intended amount of rewards since they can manipulate the reward value.

### 5.3 Defense Mechanisms

In all the apps we tested, we found that there is no proper defense mechanism (e.g., anti-debugging) except simple code obfuscation to hide package/class/variable names. Therefore, we can effectively analyze the procedures to compute the authentication code by tracing crypto APIs with *Frida*.

Another straightforward defense solution is to limit the number of reward request messages in a specific time interval. However, `Cash Slide` did not limit

the number of request attempts. Another defense mechanism is to limit the number of request attempts within a fixed time interval, but automated cash mining attacks can still be financially damaging over long periods of time.

### 5.4 Attack Results

To show the feasibility of an automated cash mining attack, we performed the attack for a total of three days using the vulnerabilities discovered against the four reward apps (see Table 2).

**Table 2.** Results of performing *automated cash mining* attacks on reward apps

| Application | Normal user(Avg.) | | Attacker | | Time interval | Reward manipulation |
|---|---|---|---|---|---|---|
| | Amount of reward | Number of reward | Amount of reward | Number of reward | | |
| **Cach Slide** | $0.04 | 27 | $127.84 | 28,533 | Unlimited time | ✗ $0.005 |
| **Fronto** | $0.06 | 21 | $0.56 | 70 | 25m | ✗ $0.009 |
| **Honey Screen** | $0.04 | 23 | $0.6 | 21 | 40m | ✓ ($0.005 → $0.03) |
| **Screen Stash** | $0.02 | 16 | $0.52 | 19 | 40m | ✓ ($0.005 → $0.03) |

✓=possible ✗=not possible

When normal users use the four reward apps without launching an automated cash mining attack for three days, they can receive an average of $0.04 (`Cash Slide`), $0.06 (`Fronto`), $0.04 (`Honey Screen`) and $0.02 (`Screen Stash`) for each app. Hence, the reward app in which the users can receive the highest reward is Fronto with $0.06, and the reward app in which the users can receive the lowest reward is `Screen Stash` with $0.02. Additionally, the number of advertisements with rewards displayed on the lock screen within three days was 27 (`Cash Slide`), 21 (`Fronto`), 23 (`Honey Screen`) and 16 (`Screen Stash`) on average for each reward app. `Cash Slide` displayed the most number of such advertisements (27 times), and `Screen Stash` displayed the least (16 times).

When launching automated cash mining attacks to show the security weakness of reward apps, we were able to earn $127.84 (`Cash Slide`), $0.56 (`Fronto`), $0.6 (`Honey Screen`) and $0.52 (`Screen Stash`), respectively, for each of the four reward apps within one day. For `Honey Screen` and `Screen Stash`, the amount of rewards in reward request messages can be modified even though the maximum allowable value is $0.03. This is six times higher than the default reward value ($0.005) while the amount of rewards for `Cash Slide` and `Fronto` were fixed to $0.005 and $0.009, respectively. In the case of `Cash Slide`, however, the most serious financial damage occurred because we can generate reward request messages without any limitation.

To test how effectively the automated cash mining attack obtains rewards, we compared the case when a normal user uses the app for three days with the case when the automated cash mining attack is launched. Compared to the

case of a normal user, the amount of rewards received for three days increased by 3,196 times, 9.3 times, 15 times, 26 times when the automated cash mining attack was launched. Further, compared to the case of a normal user, the number of advertisements with rewards increased by 1,057 times, 3.3 times, 0.9 times, 1.2 times during the same time period. As a result, we confirmed that we could obtain rewards with 0.9 times to 1,057 times more efficiency using automated cash mining attack.

## 6 Conclusion

In this paper, we analyzed the security flaws present in mobile advertising networks. We introduced automated cash mining attacks to automatically generate in-app activities at the network packet level. While previous studies [2,3] have only demonstrated the feasibility of automated attacks by emulating human click behaviors at the UI level, we implemented the first *fully automated* and working tool that is capable of generating reward request messages in a massive manner.

In our attack experiments, we found that all tested adverting networks failed to detect our automated cash mining attacks. This could be explained from the economic incentives of a security failure in current mobile adverting network models. We expect that in automated cash mining attacks, the advertisers incur the financial losses rather than the advertising networks. To make matters worse, the success of automated cash mining attacks is not a loss in an advertising network but rather a profit. Because of this disincentive, we surmise that adverting networks might not be sufficiently motivated to detect automated cash mining attacks. To fix this problem, we suggest that the interaction between adverting networks and reward applications should be audited and monitored regularly by an external third party in order to properly regulate adverting networks.

**Acknowledgments.** This work was supported in part by NRF of Korea (NRF-2017K1A3A1A17092614) and the ICT Consilience Creative support program (IITP-2019-2015-0-00742).

## References

1. Fiddler Extension (Requset to Code). http://www.chadsowald.com/software/fiddler-extension-request-to-code. Accessed 28 Feb 2019
2. Cho, G., Cho, J., Song, Y., Choi, D., Kim, H.: Combating online fraud attacks in mobile-based advertising. EURASIP J. Inf. Secur. **2016**(1), 2 (2016)
3. Crussell, J., Stevens, R., Chen, H.: Madfraud: investigating ad fraud in android applications. In: Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services (2014)
4. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A study of android application security. In: Proceedings of the 20th USENIX Security Symposium (2011)
5. Evans, C., Palmer, C.: Certificate pinning extension for HSTS (2011). https://tools.ietf.org/html/draft-evans-palmer-hsts-pinning-00
6. Immorlica, N., Jain, K., Mahdian, M., Talwar, K.: Click fraud resistant methods for learning click-through rates. In: Deng, X., Ye, Y. (eds.) WINE 2005. LNCS, vol. 3828, pp. 34–45. Springer, Heidelberg (2005). https://doi.org/10.1007/11600930_5