Storing data that is updated regularly on a client machine securely

Artem Los (artem@artemlos.net)

July 9, 2021

Abstract

Licensing of software that will run in offline environments introduces constraints on the licensing model that can be supported. The difficulty arises in cases where information needs to be recorded about the usage of the software, for example in a consumption-based licensing model. We describe a method that makes it harder for an adversary to tamper with the recorded data as well as ways for software vendors to detect if tampering with this information would still occur.

1 Introduction

A challenge when licensing software that does not have direct access to the internet is that it is more difficult to support licensing models that need to record data and send updates to a server. Examples of such licensing models are usage-based (aka. consumption based) or floating licensing. In the first case, there is a need to record information about usage that a vendor can use to bill a client correctly. In the second case, the difficulty is in keeping track of the number of concurrent machines securely.

We present a method that consists two parts. In the first part, we attempt to make it harder for an adversary to tamper with the data that is recorded on disk. In the second part, we explore several ways that a vendor can detect if tampering of the data would still occur.

2 Method

The method that is proposed consists of two stages. The first stage is to securely record information that is updated regularly in such a way that it is harder for an adversary to make changes once the information has been committed. The second stage is to detect if it is likely that an adversary has succeeded in tampering with the data when this data is analyzed by the vendor. In the end, we will describe alternative solutions and why the method that we have presented is better.

2.1 Storing information securely

The idea behind our method is to store information in a chain-like structure where each update is encrypted using vendor's public key. Each time new data is added, the previously encrypted data is included into the new block that is saved on the client machine.

Let us assume that there is a function E that, given a public key e and message m, returns the encrypted message c using a public-key cryptosystem. Furthermore, let us define a function C that combines two messages into one. The first time data is recorded, the result c_1 from $E(e, m_1)$ is written to disk. Next time data is recorded, the new data m_2 is combined with c_1 and encrypted with E. The result c_2 is written to disk. In other words, $c_2 = E(e, C(c_1, m_2))$. Decryption is performed by using the private key to decrypt the messages in reverse order. It is assumed that only the vendor has access to the private key.

When implementing this method, three design choices can be made. The first is the choice of the asymmetric cipher. We chose RSA 2048 (with Pkcs1 padding) since the implementation is widely available. A cryptosystem based on elliptic curves could be used as an alternative, with the advantage that the encrypted message will be smaller. The next design choice is the symmetric cipher to use for encryption of the data. We chose AES 128 with zero padding and CBC for the speed and availability. As a result, the data that is actually stored on disk is the initialization vector, the encrypted symmetric key (using RSA) and the data (encrypted using AES). The third is the choice of the method used for serialization of data. We chose MessagePack [1] since the serialized result was smaller than when using JSON format. Any other binary serializer should work also.

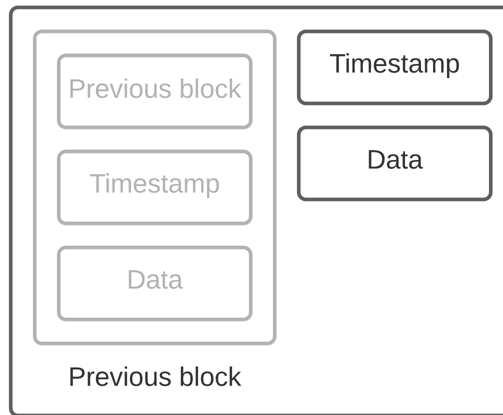

Figure 1: In this example, three updates have been made. The first is stored in the grayed out box. The second is the grayed box itself and the third is the entire block in the image.

2.2 Alternative approaches

We thought of two alternative approaches that could be used to store data that is updated regularly: either by storing information in clear text or encrypt each data block individually using public-key cryptosystem. They will be described below with reasons given as to why they where not chosen.

Storing usage in plain text In this method, the idea is to either record the history of the updates in clear text or by encrypting it using a symmetric cipher. The downside of this method is that it allows an adversary to easily tamper with the content of the updates. Moreover, an adversary always has access to the information that is being stored. By encrypting the contents of the updates with a symmetric cipher, it can make it harder for an adversary to scrutinize the contents and modify it, but we assume that the symmetric encryption key can easily be obtained. Tampering might still be detected by the vendor though.

Encrypting each data block individually In this method, the idea is to encrypt each new update using a public-key cryptosystem and store the results separately in a list. As a result, an adversary has no information about the content stored in each update. However, since each block is encrypted separately, it is still possible to remove certain updates from the list. To prevent removal of updates stored in the list, a cryptographic hash of the previous update can be stored in the new update, making it more difficult to remove updates.

If each update that is encrypted stores the hash of the previous update, this method can be a sound alternative to the main method that is described in this report (where the last update is encrypted together with the new update). The reason we chose the method that encrypts the entire history together with the new information is to make it harder for an adversary to find the number of updates that are stored on disk as well as to make tampering with the data harder. For example, assuming that an adversary has not stored the last update that was written to disk, it is impossible to make any modifications if all updates were re-encrypted. On the other hand, it would be easier to make changes if each update would be encrypted separately.

2.3 Possible threats and tampering detection

There are at least two methods that an adversary can tamper with the data: either by backing up the old file or by simulating usage themselves. Both methods are explained below:

Backing up the old file Each time an update is written to disk, it includes the history that was loaded from disk at an earlier point. This means that an adversary could take a backup of the old file and replace the new file that was saved at a later point. This way, the next time the application loads the file, newly committed updates would be lost. To mitigate this, the application can have a set schedule of how many times it will write updates to disk. This way, if an adversary would replace the new file with an old one, this could be detected by the vendor by examining the frequency of the updates and the timestamps.

Simulating usage with fake data In the previous attack, we assumed an adversary did not have access to the method that is used to commit the updates nor any information on how frequently the vendor expects these updates

to be saved. Let us assume that both the method to save updates and the frequency is known. In such case, the only parameter left is the data itself that a vendor can use to analyze for any anomalies. For example, if historical updates have already been collected, it might be possible to train an autoencoder to detect possible manipulations of the data, as described in [2]. If each update contains information about consumption (e.g. of a certain feature), then a vendor could analyze the usage value, the time when it was committed and optionally which feature the usage is related to.

2.4 Applications in different licensing models

There are at least three licensing models where our method can be used to make it harder for an adversary to tamper with the data.

Storing information about consumption Let us suppose that a client is offline and we need to record usage information so that this can be used in billing. Using our method, new updates can be written to disk as they arrives, or a summary can written on a periodic basis. The vendor can then use this file to upload the information into the central database and compute the actual usage that was consumed.

Detecting time-tampering Let us assume that you have a product sold as a subscription to a larger company and you need a way to make sure any time tampering can be detected. The safest way to solve this is to have an RTC module used as an independent time source. However, if this is not possible, our method can be used to reduce the risk of time tampering, by asking clients to send the vendor the file with all timestamps recorded on a regular basis. This can then be used by the vendor to detect any time changes.

Monitoring floating license usage Let us suppose that floating licenses are managed by a locally installed license server that does not have access to the internet. Our method can then be used to store the log of machines that have been recorded or a summary of the number of active machines at any given time. Our method does not offer any security advantage to ensure that the limit of maximum number of concurrent users has not been achieved. Instead, the idea in the case of floating licenses would be to have a secure way to get logs on software usage, with smaller risk of them being tampered with.

3 Discussion and Conclusion

In this report we have presented a method to securely store data that is updated regularly on a machine that does not have direct access to the internet. Our focus was on exploring how to store updates in a way that makes it harder for an adversary to tamper with. We presented ideas on how tampering can be detected by the vendor, which we think can be explored further in a separate project.

4 References

1. `https://msgpack.org/index.html`, Last accessed 2021-06-04.
2. A. Los, '*Finding anomalies in software licensing logs using unsupervised methods*', Dissertation, 2020.
3. Ferguson, N., Schneier, B., & Kohno, T. (2010). Cryptography engineering : Design principles and practical applications.