Device attacker models: fact and fiction

Jiangshan Yu and Mark D. Ryan

School of Computer Science
University of Birmingham
{jxy223, m.d.ryan}@cs.bham.ac.uk

Abstract. According to standard fiction, a user is able to securely keep long term keys on his device. However, in fact his device may become infected with malware, and an adversary may obtain a copy of his key. We propose an attacker model in which devices are "periodically trustworthy" — they may become infected by malware, and then later become trustworthy again after software patches and malware scans have been applied, in an ongoing cycle. This paper proposes a solution to make the usage of private keys by attackers detectable by using public transparently-maintained logs to monitor the usage of long-term secret keys.

Keywords: attacker model · key compromise detection · key usage monitoring

1 Introduction

Encryption is an important way to ensure the confidentiality of digital messages exchanged between a sender and recipient. However, there is a gap between the assumption that a secret decryption key is only known by the owner, and the fact that some party (e.g. an attacker) may have a way to gain access to the secret key. The security of existing systems cannot be guaranteed if the secret key is abused after being exposed to other parties.

Currently there are no effective methods to detect improper usage of a decryption key. In the scenario where symmetric key encryption is used, the communication parties share the same secret key for encryption and decryption, and the key owners cannot detect the condition that the key is compromised. Similarly, in the scenario where public key encryption is used, the key owner who has a pair of public key and private key for message encryption and decryption, respectively, cannot detect when his private key has been compromised. So, in both cases, the secret key owner is not prompted to revoke the compromised secret key or take other actions. Hence, the security that the system guarantees is broken until the secret key has expired. However even when a key is expired or revoked, the new key may also become compromised.

For example, if a user Alice wants to send a private message to a domain server Bob through the transport layer security (TLS) protocol, then Alice needs

to obtain Bob's public key certificate, and encrypt and send a session key establishment message to Bob to establish a session key. Here, even if Alice has obtained an authentic copy of Bob's TLS certificate, if attacker Eve can somehow compromise Bob's private key corresponding to the certificate Alice obtained, then Eve is able to block and decrypt the cipher text from Alice, and play man-in-the-middle attacks by providing her Diffie-Hellman (DH) key establishment contributions to both Alice and Bob, without being detected.

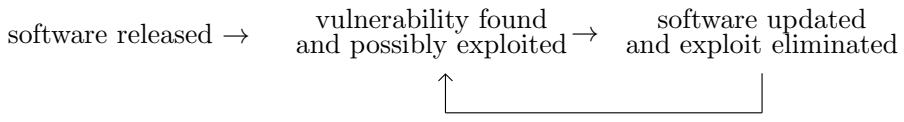
Detection of private key usage when the private key is a signing key rather than a decryption key is easier, because security can be leveraged from the party that needs to verify and accept the signature. Such a party is a witness who saw the usage of the compromised signing key. Certificate transparency (CT) [1], accountable key infrastructure (AKI) [2], enhanced certificate transparency (ECT) [3], and distributed transparent key infrastructure (DTKI) [4] are existing key infrastructures that can audit such key usage by recording signatures (i.e. certificates) into the log.

Inspired by the above mentioned systems, we propose a mechanism to convert systems that use long-term symmetric or asymmetric keys for encryption to systems that can detect unauthorized usage of long-term secret keys, thus providing a better security guarantee.

2 Overview

The problem. Suppose that Alice wants to send a confidential message m to Bob. Suppose Bob has a public/private key pair for encryption; that is, he has a decryption key dk_B which is kept private, and an encryption key ek_B which is published. According to the common practice, Alice encrypts the message m using Bob's encryption key ek_B, creating a ciphertext. If only Bob has possession of dk_B, then only Bob can decrypt the ciphertext (Fig. 1). However, if dk_B has become exposed to an attacker, then the attacker can decrypt the ciphertext. Bob has no reliable way to be aware that this has happened. Similarly, when Alice and Bob want to use symmetric key encryption for exchanging sensitive messages, the same problem would occur.

Adversary model Almost all systems (e.g. Windows or Mac OS) and protocols (e.g. TLS) suffer from an undesired life cycle of security:



Based on the above life cycle of security, we consider an adversary who can get all (long-term and short-term) secrets of victims. The attacker can repeatedly fully control the victim's device, but each time only for a limited period.

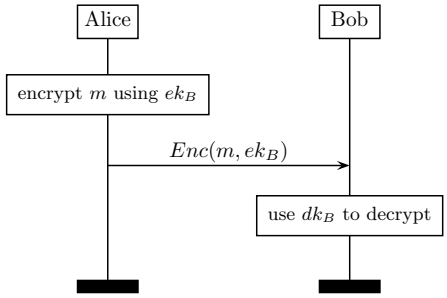


Fig. 1. An abstract representation of the original protocol. In this protocol, Alice encrypts message m by using Bob's public key ek_B, and sends the cipher-text $Enc(m, ek_B)$ to Bob who can decrypt it by using his private key dk_B.

For example, after the implementation of a system (e.g. Windows) or a protocol (e.g. SSL) has been released, there are always some undiscovered security flaws. During the time that the system has not been patched, an attacker is able to get some credential (e.g. a secret key) out from a victim's device. However, some time later, the security flaw is discovered and fixed, and the attacker's ability to control the device is removed.

The security properties we want to guarantee are *forward secrecy* and *auditable-backward secrecy*. *Forward secrecy* is a property ensuring that the plain-text of transmitted messages will not be exposed if the associated long-term secret is compromised in the future; and *auditable-backward secrecy* is a property ensuring that if a compromised long-term secret will have exposed future messages, then the genuine owner of the long-term secret can detect the fact of the compromised secret.

The solution. The basic idea is that the commitments about the usage of long-term private keys are recorded in a log. This enables a key owner to monitor the usage of a key. For a long-term asymmetric secret key, the commitment could be a signature, or a proof of knowledge. For a long-term symmetric key, the commitment can be done by using symmetric key encryption, or by using hash-based message authentication code (HMAC). In this paper, we focus on the asymmetric key case. (The protocol for symmetric key case is similar to the protocol for asymmetric key case.)

In more detail (as shown in Fig. 2), Bob uses a long-term public/private key pair (sk_B, vk_B) for signature generation and verification rather than for

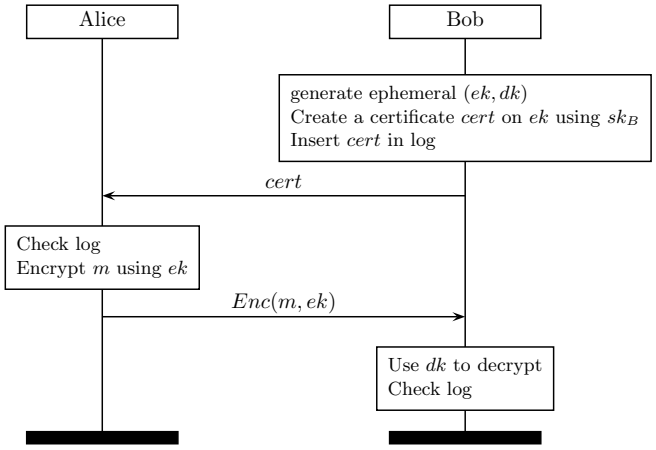


Fig. 2. The basic idea of our proposed system. In this abstract protocol, Bob has a pair (sk_B, vk_B) of long term keys for signature generation and verification, and Alice is assumed to have the verification key vk_B. Bob generates an ephemeral key pair (ek, dk) for encryption and decryption, issues certificate *cert* on the ephemeral encryption key ek, then inserts the certificate into his associated log. After receiving *cert*, Alice checks that it is present in the log, then sends a message m to Bob encrypted with ek. Bob checks in the log whether there are certificates for his keys not generated by him.

message decryption and encryption, respectively. Bob periodically[1] generates an ephemeral key pair (ek, dk) for encryption and decryption, issues certificate *cert* on the ephemeral encryption key ek, then publishes *cert* in the log. After receiving *cert*, Alice verifies the certificate. If *cert* is valid, she checks that (a) it is present in the log; (b) it is not revoked; and (c) the log is maintained in an append-only manner. If all checks succeed, she encrypts sensitive message[2] m using ek, and sends the ciphertext to Bob encrypted with ek. Bob should check in the log whether there are certificates for his keys not generated by him. Bob also keeps his own record of generated certificates, and periodically, he verifies that the certificates published in the log agree with his own record of published information.

Remark. Alice and Bob can be simultaneously online (synchronous) or the receiver may be offline at the time the sender sends a message (asynchronous). In the case they are synchronous, Alice can get the ephemeral certificate from

[1] The period is the lifetime of the generated S. It will be denoted as δ in the security discussion section. The smaller the δ, the more secure the system is.

[2] Note that m could be any sensitive data, such as a session key which will be used to encrypt messages in this communication.

Bob directly. In the other case, Bob can upload a pre-generated ephemeral certificate into a third party where Alice will get the ephemeral certificate from. Since Alice can verify the log, the third party is not required to be trusted. In addition, it is not necessary for Bob to be the log maintainer. He can ask some third party to maintain his log. The log maintainer is also not required to be trusted. Moreover, instead of generating a single key pair and putting it in the log each time, Bob can generate a sequence of keys, and publish the digest (with constant size) of the sequence into the log. The digest can be a hash chain of the sequence, or the root value of a hash tree that records the sequence. In this case, Alice needs to ask an additional proof that the obtained certificate is an element of the sequence that is presented by the digest. The proof is with size $O(\log n)$ if a hash tree is used, where n is the size of the tree (and the size of the sequence).

Properties of the log The security of the method requires that an attacker cannot remove information from the log. To achieve this, the log is typically stipulated to be append-only. It is a requirement that users of the log (including Bob) can verify that no information has been deleted from the log. For this purpose, the log can be organised as a Merkle tree [5] in which data is inserted by extending the tree to the right. Such a log was designed and introduced in *certificate transparency* [1]. The log maintainer can provide efficient proofs that (A) some particular data is present in the log, and (B) the log is being maintained in an append-only manner. Proof A is referred to as *proof of presence* and proof B is referred to as *proof of extension*. Certificate transparency has been extended to provide proofs of absence and proofs of currency [3]. Proof of absence demonstrates that any data having a given attribute is absent from the log. Proof of currency demonstrates that some data is the latest valid data associated to a given attribute (e.g. a key associated to a user identity). The extension allows revocation to be supported. In addition, certificate transparency has been extended to support multiple logs and to avoid the requirement of trusted parties [4]. Another interesting design for certificate logs is through the checks-and-balances proposed in AKI [2] and its enhancement (ARPKI [6]).

It is also a requirement that the log maintainer cannot maintain different versions of the log which it shows to different sets of users. Gossip protocols are a known technique to ensure that.

On the other side, user privacy is a concern. This can be solved by letting the log maintainer sign each user identity, then put the hash value of the signature into the log (rather than putting the user identity directly into the log). The signature scheme used should be deterministic and unforgeable, as suggested in a previous work [7]. Hence, users that have the recipient's identity can request the signed user identity from the log maintainer, and verify it; but an attacker who has downloaded the entire log cannot recover the identity of users, based on the unforgeability of the chosen signature scheme.

In addition, it would be difficult to maintain and sync all logs in real-time when they become huge, as any delay of receiving new update queries would make logs inconsistent. One countermeasure is to update the log periodically

rather than update it in real-time. In this arrangement, log maintainers collect all new log update queries, synchronise with each other, then update the log when all log maintainers have agreed the same set of data for update.

Security discussion We assume that all sets of pre-generated ephemeral keys of the same user have the same lifetime, denoted δ. In addition, according to our adversary model, we know that Bob's system can be compromised repeatedly, but not continuously.

Suppose Bob's system is compromised at time t_i and fixed at time t'_i, for $i \in \{1, 2, 3, \ldots\}$ (see Fig. 3). If the attacker recovers the plain-text of messages exchanged at a time outside of the period $(t_i - \delta, t'_i + \delta)$, then the attacker has to leave some evidence in the log, which can be readily detected by Bob.

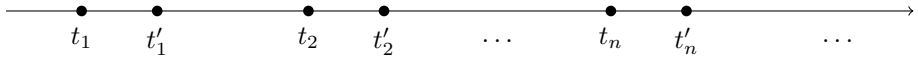


Fig. 3. An example time-line. In this time-line, for all $i \in \{1, 2, 3, \ldots\}$, the security of Bob's system is broken between time t_i and t'_i.

3 Application to SSH

Key usage detection (KUD) can be applied to many scenarios where the secret keys might be compromised or escrowed by other parties.

For example, in existing identity based signature [8] systems, the identity provider can derive any of its client's signing key, and therefore can sign on behalf of any of its client. Similarly, in existing identity based encryption [9] systems, the identity provider can derive any of its client's decryption key, and therefore can decrypt a ciphertext sent to the client without informing the client [10]. One way to solve the above key escrow problem is adopting multiple private key generators to perform the secret generation process. So, no single party can derive the client secret. However, this solution is hard to achieve in practice.

In the case of identity based signature, key usage detection mitigates the problem in another way by recording all signatures in the public log, and allows a member to detect that an unauthorised signature (e.g. one made by the identity provider) has been issued. Key usage detection cannot be directly used to solve the key escrow problem in identity based encryption; however, it can solve the problem indirectly — instead of using identity-based encryption, the client obtains an identity based signing key from the identity provider, generates an

ephemeral key pair for decryption and encryption, signs the ephemeral encryption key by using the obtained signing key, and publishes the signature into the log.

Each time when a sender wants to send a message, the sender downloads the signed encryption key of the recipient, verifies the signature according to the recipient's identity, and encrypts and sends the message if the verification succeeds. Note that the log does not need to be universal. Depending on the scenario, logs can be maintained by identity providers, by clients, or by third parties.

In this way, clients can use the user identity to verify the signed encryption key, can detect the un-authorised usages of the long-term signing key, and are still be able to do encryption.

Another example is to detect un-authorised key usage in the Secure Shell (SSH) protocol. Secure Shell (SSH) is an industry-standard cryptographic network protocol for securing data communication, and is one of the most popular cryptographic protocols on the Internet. It aims to establish a secure channel over an insecure network in a client-server architecture, connecting an SSH client application with an SSH server. SSH users can easily setup (or connect to) an SSH server on (resp. by using) their devices, e.g. a desktop or laptop, for secure communication and data sharing. The server authentication is done by using public-key cryptography, and the client authentication can be done by either using public-key cryptography-based authentication, or using password-based authentication.

We explain the problem and illustrate how to apply key usage detection to SSH as follows.

3.1 The problem

The encryption used by SSH is intended to provide confidentiality and integrity of data over an unsecured network, such as the Internet. However, if the private key of a server (or a client) is exposed to an attacker (e.g. because the presence of security bugs [11] or malware), then the security is broken. In practice, the server operator or the client may periodically perform malware scans, operating system upgrades, and software updates. These actions will bring their devices back into a trustworthy state again. Unfortunately, since all secret keys are already exposed to the attacker, the regained trustworthy state will not help victims to regain the security.

To regain the security, victims need to revoke their keys and generate new ones; however, since they do not know when compromises take place, they are not motivated to revoke their keys. In practice, it is impractical to ask users to revoke their keys and distribute new ones after every security update. So, the security is broken from the time that a server or client device is compromised onwards.

3.2 The solution

Key usage detection (KUD) can be applied to SSH to reduce the damage of a compromised long-term secret key. It could be applied to the client side, or the server side, or both. We present an abstract protocol to show how to apply KUD to the server authentication of SSH. In the modified protocol, the server has a pair of long-term signing key and verification key. The modified protocol has the same assumption as SSH, namely the clients have an authenticate copy of the server's long-term public key.

The server creates a public log, then periodically (e.g. every δ-long period)

Step 1. generates a pair of ephemeral decryption key and encryption key;
Step 2. issues a certificate on the newly generated decryption key;
Step 3. inserts the certificate into the log; the new certificate will revoke previous ones;
Step 4. verifies that all certificates in the log are generated by itself.

For server authentication, the user sends to the server a request for establishing a new communication session, together with a digest of the log that s/he has observed in the last session. The server sends to the user the currently valid certificate together with the following proofs:

- (**Proof of currency**) the received certificate is present in the log and is the latest valid certificate of the server; and
- (**Proof of extension**) the current log is an extension of the log that he has previously observed.

The user verifies the above proofs, then uses the obtained certificate to authenticate the server. All participants of the protocol should also run gossip protocols to detect inconsistent logs.

3.3 The public log structure and proofs

Similar to CT [1], the public log in this example is simply organised as an append-only Merkle tree. However, we do not have the problem that CT faces — namely, to efficiently prove that a given certificate in the log is not revoked. This is due to the fact that our log only contains one valid certificate, which is stored in the rightmost leaf of the Merkle tree. This design allows a log maintainer to efficiently generate the proof that a certificate is the latest one in the log (by proving the certificate is located on the rightmost leaf of the Merkle tree), and this proof can be efficiently verified by users. (We will detail it later.)

Log structure The public log is organised as an append-only Merkle tree. A Merkle tree (a.k.a. hash tree) [5], is a binary tree where each non-leaf node is labelled with the hash of the labels of its children; and where each leaf is labelled with some data. Suppose a node has two children labelled with hash values h_1 and h_2. Then the label of this node is $\mathsf{h}(h_1, h_2)$. Relying on the properties of the

hash function, a Merkle tree allows one to provide cryptographic proofs of the existence of some data in a Merkle tree; the size of those proofs is logarithmic in the size of the tree.

An append-only Merkle tree is a Merkle tree where the only allowed operation is appending a new data by extending the tree to the right side. Append-only Merkle trees allow one to provide cryptographic proofs that one version of the tree is an extension of a previous version; the size of those proofs is also logarithmic in the size of the tree. Note that an append-only Merkle tree is a not necessarily a balanced tree. The two trees in Figure 4 and 5 are examples of append-only Merkle trees. Figure 4 shows a Merkle tree containing data items $d_1, \ldots, d_5$ stored at the leaf nodes. Figure 5 shows a larger Merkle tree containing data items $d_1, \ldots, d_7$.

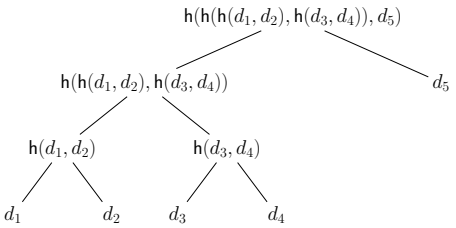


Fig. 4. Append-only Merkle tree T containing 5 leaves.

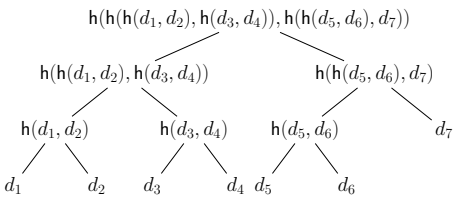


Fig. 5. Append-only Merkle tree T' containing 7 leaves. It is obtained after appending d_6 and d_7 into T by extending T to the right.

Proof of presence To demonstrate that a hashed certificate d_3 is present in the tree T, it is sufficient to provide the additional data $d_4, \mathsf{h}(d_1, d_2), d_5$, i.e. one data item per layer of the tree. The verifier can then verify the correctness of the root hash $\mathsf{h}(\mathsf{h}(\mathsf{h}(d_1, d_2), \mathsf{h}(d_3, d_4)), d_5)$.

Proof of currency To prove that a hashed certificate d_7 is the latest one in the tree T', the prover issues the proof of presence of d_7 in T'. The verifier verifies the proof, and additionally verifies that the path of d_7 is of the form $r \cdot r \ldots r$, where r is the path from a parent node to the right child node.

Proof of extension Proving that one Merkle tree extends another can also be done in logarithmic space and time, by providing at most one hash value per layer. For example, to demonstrate that T' in Figure 5 is an extension of the one in Figure 4, it is sufficient to provide the data $d_5, d_6, \mathsf{h}(\mathsf{h}(d_1, d_2), \mathsf{h}(d_3, d_4)), d_7$.

For user authentication, we can use a protocol similar to the protocol for server authentication. One difference is that rather than only recording the

server's key usage (i.e. the ephemeral certificate) in the log, the usages of all clients' secret key should also be recorded. Since the log will contain one current valid ephemeral certificate for each user, we need to use a log structure that supports efficient proof of currency when the log contains multiple valid certificates of different clients. One possible log structure is proposed in ECT [3].

3.4 Security discussion

Let's assume that an attacker has compromised the server at time t such that $t_i \leq t \leq t'_i$ for some i, has obtained secrets stored on the server, is remaining in possession of the obtained secrets, and wants to attack at time t' such that $[t'_i + \delta \leq t' \leq t_{i+1} - \delta]$ (see Figure 3).

Let δ be the lifetime of ephemeral key pairs. We have that in a time period $[t'_i + \delta, t_{i+1} - \delta]$, all server authentication messages are protected by using an ephemeral encryption key that is generated when the device is secure. So, the attacker does not have the possession of the corresponding ephemeral decryption key, and the previously compromised decryption key does not help with the decryption.

Since clients will only accept an ephemeral certificate if it is accompanied by proofs of presence in the log, the attacker has to either sign a key and put it in the log, or generate a fake log and fool the user. The former will be readily detected by the server, and the later will eventually be detected by gossip protocols, as the inconsistent logs will be detected by users.

4 Conclusion

Unauthorized usage of secret keys may occur if computer systems are attacked and keys are compromised, or if backup keys or escrowed keys are abused. This paper proposes a direction in which we aim to make the usage of private keys detectable. This is achieved by using public transparent logs to monitor the usage of long-term secret keys.

We consider an attacker able to periodically compromise secret keys, for example by introducing malware onto a device. We assume that the legitimate user will eventually gain control of the device again, by applying software patches and malware scans and clean-ups.

Acknowledgements

The authors thank Ross Anderson, Daniel Thomas, and all other attendees of International Workshop on Security Protocols for their comments and discussions. Jiangshan Yu is supported by the EPSRC project EP/H005501/1.

References

1. Laurie, B., Langley, A., Kasper, E.: Certificate Transparency. RFC 6962 (Experimental) (2013)
2. Kim, T.H.J., Huang, L.S., Perrig, A., Jackson, C., Gligor, V.: Accountable key infrastructure (AKI): A proposal for a public-key validation infrastructure. In: the 22nd International World Wide Web Conference (WWW 2013). (2013)
3. Ryan, M.D.: Enhanced certificate transparency and end-to-end encrypted mail. in network and distributed system security. In: NDSS. (2014)
4. Yu, J., Cheval, V., Ryan, M.: DTKI: a new formalized PKI with no trusted parties. CoRR **abs/1408.1023** (2014)
5. Merkle, R.C.: A digital signature based on a conventional encryption function. In: CRYPTO. (1987) 369–378
6. Kim, T.H., Gupta, P., Han, J., Owusu, E., Hong, J.I., Perrig, A., Gao, D.: ARPKI: attack resilient public-key infrastructure. In: ACM CCS. (2014)
7. Melara, M.S., Blankstein, A., Bonneau, J., Freedman, M.J., Felten, E.W.: CONIKS: A privacy-preserving consistent key service for secure end-to-end communication. IACR Cryptology ePrint Archive (2014)
8. Shamir, A.: Identity-based cryptosystems and signature schemes. In: CRYPTO. (1984) 47–53
9. Boneh, D., Franklin, M.K.: Identity-based encryption from the weil pairing. In: CRYPTO. (2001) 213–229
10. Al-Riyami, S.S., Paterson, K.G.: Certificateless public key cryptography. In: ASIACRYPT. (2003) 452–473
11. CVE: Common vulnerabilities and exposures list. Retrieved Feb. 2015. https://cve.mitre.org/cve/index.html.