D. Nuñez, I. Agudo, and J. Lopez, "The fallout of key compromise in a proxy-mediated key agreement protocol", *31st Annual IFIP WG 11.3 Conference on Data and Applications Security and Privacy (DBSec'17)* vol. LNCS 10359, pp. 453-472, 2017.
http://doi.org/10.1007/978-3-319-61176-1_25
NICS Lab. Publications: https://www.nics.uma.es/publications

The fallout of key compromise in a proxy-mediated key agreement protocol

David Nuñez, Isaac Agudo, and Javier Lopez

Network, Information and Computer Security (NICS) Laboratory
Computer Science Department, University of Málaga, Spain
{dnunez,isaac,jlm}@lcc.uma.es

Abstract. In this paper, we analyze how key compromise affects the protocol by Nguyen et al. presented at ESORICS 2016, an authenticated key agreement protocol mediated by a proxy entity, restricted to only symmetric encryption primitives and intended for IoT environments. This protocol uses long-term encryption tokens as intermediate values during encryption and decryption procedures, which implies that these can be used to encrypt and decrypt messages without knowing the corresponding secret keys. In our work, we show how key compromise (or even compromise of encryption tokens) allows to break forward security and leads to key compromise impersonation attacks. Moreover, we demonstrate that these problems cannot be solved even if the affected user revokes his compromised secret key and updates it to a new one. The conclusion is that this protocol cannot be used in IoT environments, where key compromise is a realistic risk.

1 Introduction

Nguyen, Oualha, and Laurent presented at ESORICS 2016 an authenticated key agreement protocol based on an ad-hoc variant of symmetric proxy re-encryption [16], called AKAPR. This protocol is intended for highly-constrained IoT devices, and therefore, deliberately excludes the use of asymmetric primitives such as Diffie-Hellman key exchange due to its reliance on modular exponentiations. The proposed protocol is implemented on top of block ciphers, MACs, and simple modular arithmetic. The authors provide a formal verification of the mutual authentication property using ProVerif.

In this paper we identify some deficiencies in the proposed protocol that were not considered by the formal analysis and describe several attacks that exploit them. In particular, our attacks exploit the fact that encryption and decryption operations do not use secret keys directly, but intermediate secret values (which we call "encryption tokens"). These secrets are relatively exposed throughout the protocol and can be used to compromise the security of their corresponding owners (and in certain cases, other users). We show how compromise of these encryption tokens (or directly, of secret keys) makes it possible to break forward security and lead to key compromise impersonation attacks. An interesting (and devastating) consequence of the nature of these attacks is that these cannot be

prevented even if the affected user renews his secret key, given that the compromised secret can be linked to the encryption token of other users.

This paper constitutes an example of how providing a formal analysis of a protocol or system is often not enough, since this may be incomplete, incorrectly designed or directly flawed. In the case of the analyzed protocol, the authors did not consider the possibility of key compromise. However, in IoT environments (which is the target application of the protocol), key compromise is a realistic risk due to the broad attack surface of most devices, which includes physical attacks (e.g., use of JTAG and UART interfaces, power analysis, etc.), software vulnerabilities (e.g., buffer over-reads like Heartbleed in OpenSSL), privilege escalation through backdoor accounts, etc.

Related work The protocol we analyze in this paper is an example of authenticated key exchange (AKE), one of the most recurring topics in the literature when it comes to security protocols. It is also the basis for most Internet applications that relay on secure channels, as it is embedded in TLS and IPSEC. We can clearly distinguish two research trends in this area, one based on public-key cryptography, where Diffie-Hellman [7] is the current "standard", and the other on secret key cryptography. With respect to the latter, most proposals rely on a Key Distribution Center (KDC) that shares a secret key with all users in the system and supports them on agreeing on a session key. Since 1978, when the Needham-Schroeder symmetric protocol [14] was proposed, several authors have attempted to propose better symmetric AKE protocols, usually showing how previous ones can be attacked and fixing their weaknesses. For example, one of the weaknesses of the original Needham-Schroeder protocol was the inability to prove the freshness of the session key. Denning and Sacco [6] proposed the use of Timestamps as a way fix this problem. Later, the Kerberos protocol [15], also built on the idea of using timestamps, was proposed as the current "standard" AKE solution in the symmetric set-up. In the last years, new assumptions on the capabilities of the attackers have been defined as well as new application scenarios, security requirements and constrains that make this problem still an interesting research topic, particularly in those environment where public-key cryptography is not viable. Some recent works show that Elliptic Curve Cryptography (ECC) can be run in most wireless sensor platforms [12], although there are still some highly-constrained devices, such as passive RFID, that are not yet capable of using PKC and would then require the use of symmetric AKE protocols.

The analyzed protocol uses techniques from proxy re-encryption (PRE). This is a research topic with increasing popularity, with new use cases arising in different contexts (e.g, data sharing in the cloud, key management, etc.). Although the vast majority of PRE schemes are based on public-key cryptography [18], there have been some proposals based on symmetric cryptography. Syalim et al. [21] propose a symmetric PRE scheme based on the All-Or-Nothing Transform, although it assumes that both sender and receiver share a common secret. Cook and Keromytis [5] present a solution based on a double-encryption approach, but as in the previous case, a priori shared keys are needed. The key-homomorphic

PRF primitive by Boneh et al. [2] can be used to construct symmetric PRE schemes without the shared key requirement, although as noted by Garrison et al. [10], its computational cost is comparable or greater than traditional public-key cryptography. Sakazaki et al. [19] propose a symmetric PRE scheme that is essentially equivalent to the one of this paper; this is discussed further in Section 2.1.

Organization The rest of this paper is organized as follows: In Section 2, we describe the AKAPR protocol in detail, including actors, protocol flow and the underlying symmetric proxy re-encryption primitive. In Section 3, we present several attacks and weaknesses of the AKAPR protocol. In Section 4, we discuss some possible alternatives to the protocol. Finally, Section 5 concludes the paper.

2 Description of the Authenticated Key Agreement Protocol

In this section we briefly describe the protocol AKAPR (Authenticated Key Agreement mediated by a Proxy Re-Encryptor). The goal of this protocol is to provide an authenticated key exchange between two entities, using a proxy entity as a mediator. As in the general case of proxy re-encryption, although the proxy entity assists in the process, it should not learn any information. A critical restriction that guides the design of this protocol is that it assumes that the authenticating entities may not be capable of using common public-key cryptography primitives, such as Diffie-Hellman key exchange and digital signatures, due to the its reliance on modular exponentiations. Instead, AKAPR is designed solely on top of block ciphers, MACs, and simple modular arithmetic.

In the remaining of this section we first briefly explain the underlying symmetric proxy re-encryption primitive. Next, we describe the general setting of the protocol, which includes the description of actors, network architecture and trust assumptions. Finally, we detail the protocol flow.

2.1 Symmetric proxy re-encryption primitive

The authors propose a symmetric proxy re-encryption algorithm, which is composed of the following five functions:

- $\mathsf{KeyGen}(\lambda)$: On input the security parameter λ, the key generation algorithm KeyGen outputs the secret key sk and identity id. As an example, for user A, these are sk_A and id_A.
- $\mathsf{ReKeyGen}(sk_A, id_A, sk_B, id_B)$: On input the secret keys sk_A and sk_B, and identities id_A and id_B, the re-encryption key generation algorithm $\mathsf{ReKeyGen}$ computes the re-encryption key between users A and B as $rk_{A\to B} = h(sk_A \| id_B)^{-1} \cdot h(sk_B \| id_A)^{-1}$, where $h : \{0,1\}^* \to \mathbb{Z}_p$ is a hash function.
- $\mathsf{Enc}(sk_A, id_B, M)$: On input the secret key sk_A, the identifier of B, and a message M, the encryption algorithm Enc samples t from $\mathbb{Z}_p$, derives a fresh

key K from t using a key derivation function (KDF), and outputs ciphertext $C_A = (\mathsf{SymEnc}_K(M), t \cdot h(sk_A \| id_B))$, where SymEnc is a symmetric encryption algorithm.
- $\mathsf{ReEnc}(rk_{A\to B}, C_A)$: On input a re-encryption key $rk_{A\to B}$ and a ciphertext $C_A = (C_{A,1}, C_{A,2})$, the re-encryption algorithm ReEnc outputs ciphertext $C_B = (C_{A,1}, C_{A,2} \cdot rk_{A\to B})$.
- $\mathsf{Dec}(sk_B, id_A, C_B)$: On input the secret key sk_B, the identity of the original recipient A, and a ciphertext $C_B = (C_{B,1}, C_{B,2})$, the decryption algorithm Dec computes $t \leftarrow h(sk_B \| id_A) \cdot C_{B,2}$, derives key K from t using the KDF and decrypts the message M from $C_{B,1}$ using symmetric decryption algorithm SymDec.

We note that there is a previous proposal by Sakazaki et al. [19], further discussed in [23], which is essentially equivalent to this symmetric proxy re-encryption scheme. The only difference is that the original proposal by Sakazaki et al. uses the XOR operator instead of the modular multiplication. If instantiated correctly, both of them are equivalent, security-wise; however, Sakazaki et al.'s choice of the XOR operator can be implemented more efficiently than modular multiplication.

Additionally, we also remark that this symmetric proxy re-encryption scheme is not consistent with the prevalent idea of proxy re-encryption, where the sender does not need to know a priori the identity of the intended recipient. In this scheme, the sender fixes the recipient identity during encryption, and therefore, the resulting ciphertext can only be re-encrypted to this identity. In contrast, in the traditional idea of proxy re-encryption (regardless if it is public-key or symmetric), ciphertexts can be re-encrypted for any possible recipient, as long the corresponding re-encryption key exists. Although, for simplicity, we will continue to refer to this scheme as symmetric proxy re-encryption, we believe that it is actually some kind of multiparty encryption, where the sender and the proxy jointly create a ciphertext decryptable by an a priori fixed recipient.

2.2 Protocol setting

There are three main types of actors in the AKAPR protocol, which are the following:

- The Initiator (I) and the Responder (R), which are two entities located in separate subnetworks and that want to establish an authenticated session between them. It is assumed that these entities may lack the capacity to use asymmetric primitives such as Diffie-Hellman, which requires modular exponentiation, and therefore will only use symmetric techniques. Note, however, that a regular device can also participate in this protocol.
- The Delegatee[1] (D), which mediates in the key agreement protocol between the Initiator and the Responder, without being able to learn the negotiated session keys.

[1] Note that, in the proxy re-encryption literature, the term "Delegatee" is usually referred to the recipient of a re-encrypted ciphertext. Hence, a re-encryption from

– The Key Distribution Center (KDC), which initially generates all the secret keys and necessary re-encryption keys, and distributes them (secret keys to entities, and re-encryption keys to the delegatee).

Figure 1 depicts the network setting assumed by this protocol, as well as its main actors.

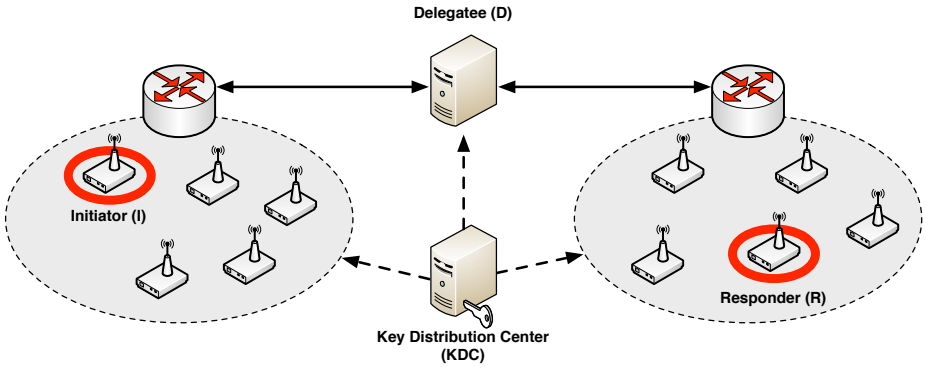


Fig. 1: Network architecture (adapted from [16])

2.3 Trust assumptions

The protocol has the following trust assumptions:

– The Delegatee is assumed to be honest-but-curious, which means it behaves correctly with respect to the protocol, but at the same time, it has an interest in reading the underlying information. We will latter show that if this assumption is relaxed (e.g., an attacker gains control of the delegatee, or simply, the delegatee cooperates with him), it makes several attacks possible.
– The Key Distribution Center is fully trusted. It is clear that, given that the KDC knows all the keys involved, it can gain full control of the network. The existence of an omniscient KDC can be seen as a single point of failure, precisely for this reason [13, Remark 13.3].

2.4 AKAPR Protocol Flow

We begin this subsection by defining some of the notation used in the protocol. Recall that one of the principal requirements is to use only symmetric encryption primitives. In particular, the protocol requires an authenticated encryption

user A to B can be seen as the delegation of decryption rights from a "delegator" A to a "delegatee" B. However, for consistency with the analyzed protocol, we will also refer to it as "delegatee".

scheme, denoted by AEnc, and a message authentication code MAC. The protocol assumes that each principal X has a shared key with the delegatee D, namely K_{XD}. This key is used to provide authenticity and integrity of communications between the principal and the delegatee. In addition, it also assumes that both initiator and responder maintain two counters for the communication between them, in order to protect against replay attacks. These counters are labeled CT_{IR} and CT_{RI}, respectively. Additionally, nonces N_I and N_R are used to ensure freshness of messages; a session identifier SID also contributes towards this goal. Table 1 summarizes the notation used in this paper.

Table 1: Notation

Symbol	**Description**
I	Initiator
R	Responder
D	Delegatee
KDC	Key Distribution Center
CT_{IR}, CT_{RI}	Counters
SID	Session identifier
N_I, N_R	Nonces
H	Hash function
KDF	Key derivation function
AEnc, ADec	Authenticated encryption/decryption primitives
MAC	Message authentication code
K_{ID}, K_{RD}	Pre-shared keys with the delegatee
$\mathbb{Z}_p$	Multiplicative group of integers modulo p (i.e., $\{1, ..., p-1\}$)

The AKAPR protocol is composed of 4 messages. In the the first two messages, the delegatee acts as intermediary between the initiator and responder, while the third and fourth messages are directly between them. Figure 2 shows the flow of messages of the protocol. Next, we describe these messages in detail.

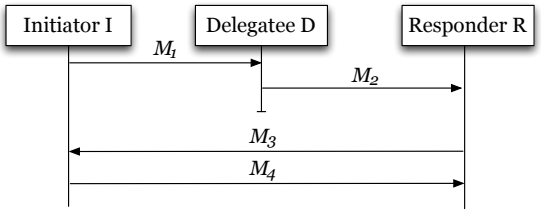


Fig. 2: Protocol flow

Message 1 ($I \rightarrow D$). The first message of the protocol, M_1, is created by an initiator I and sent to the responder R, via the delegatee D. The initiator performs the following steps:

1. $CT_{IR} \leftarrow CT_{IR} + 1$
2. $SID \leftarrow H(id_I \| id_R \| w)$, for a random w
3. $N_I \xleftarrow{R} \mathbb{Z}_p, \quad t \xleftarrow{R} \mathbb{Z}_p$
4. $AK \leftarrow \mathsf{KDF}(id_I, id_R, t)$
5. $AE_1 \leftarrow \mathsf{AEnc}_{AK}(SID \| id_I \| id_R \| N_I \| CT_{IR})$
6. $C_1 \leftarrow t \cdot h(sk_I \| id_R)$
7. $\bar{M}_1 \leftarrow SID \| id_I \| id_R \| AE_1 \| C_1$
8. $M_1 \leftarrow \bar{M}_1 \| \mathsf{MAC}_{K_{ID}}(\bar{M}_1)$

Message 2 ($D \rightarrow R$). Once the delegatee D receives a message from an initiatior I, he performs the following procedure in order to verify the validity of the message and to generate the message for responder R, namely M_2.

1. Verify that SID is not repeated and that MAC in M_1 with key K_{ID}; if failed, abort the protocol.
2. $C_2 \leftarrow rk_{I \rightarrow R} \cdot C_1$
3. $\bar{M}_2 \leftarrow SID \| id_I \| id_R \| AE_1 \| C_2$
4. $M_2 \leftarrow \bar{M}_2 \| \mathsf{MAC}_{K_{RD}}(\bar{M}_2)$

Note that after the re-encryption performed by the delegatee, the value of C_2 is $t \cdot h(sk_R \| id_I)^{-1}$.

Message 3 ($R \rightarrow I$). When the responder R receives the previous message from the delegatee D, he also verifies its validity, proceeds to decrypt it and extract the necessary information, as described below. Finally, he produces a new message M_3 that is sent directly to the initiator I, without intermediaries. Note that at the end of this process, the responder already knows the agreed session key K_S.

1. Verify MAC in M_2 with key K_{RD}; if failed, abort the protocol.
2. $t \leftarrow h(sk_R \| id_I) \cdot C_2$
3. $AK \leftarrow \mathsf{KDF}(id_I, id_R, t)$
4. $SID \| id_I \| id_R \| N_I \| CT_{IR} \leftarrow \mathsf{ADec}_{AK}(AE_1)$
5. Check that $CT_{IR} \geq CT_{RI}$; if failed, abort the protocol.
6. $CT_{RI} \leftarrow CT_{IR} + 1$
7. $N_R \xleftarrow{R} \mathbb{Z}_p$
8. $K_S \leftarrow \mathsf{KDF}(CT_{RI}, id_I, id_R, N_I, N_R)$
9. $M_3 \leftarrow \mathsf{AEnc}_{AK}(SID \| id_R \| id_I \| N_I \| t \| N_R \| CT_{RI})$

Message 4 ($I \rightarrow R$). When the initiator I receives the response from the responder R, he follows the procedure below to verify its validity and extract the necessary information. He produces a final message M_4 that is sent directly to the responder R, who verifies its validity using the previously generated session key K_S.

1. $SID \| id_R \| id_I \| N_I \| t \| N_R \| CT_{RI} \leftarrow \mathsf{ADec}_{AK}(M_3)$
2. Check that SID, N_I and t correspond to the original ones sent in M_1, and that $CT_{RI} = CT_{IR} + 1$; if failed, abort the protocol.
3. $K_S \leftarrow \mathsf{KDF}(CT_{RI}, id_I, id_R, N_I, N_R)$
4. $M_4 \leftarrow SID \| \mathsf{MAC}_{K_S}(SID \| id_I \| id_R \| N_I \| N_R)$

As a final remark, we note that the authors do not describe any procedure for the initial key distribution.

3 Attacks to the AKAPR Protocol

In this section we describe several attacks and weaknesses of the AKAPR protocol. Most of them stem from the use of encryption tokens as intermediate values for encryption and decryption, as well as the possibility of obtaining an encryption token associated to another user by means of the delegatee or analysis of past protocol traffic. This is particularly problematic in the event of compromise of long-term secret keys, to the point that even key revocation and update does not allow the affected user to recover the guarantees of secrecy and authentication with respect to other users. In the Appendix, we also show how the choice of an insecure keyed-hash construction can lead to length-extension attacks, which combined with our previous attacks, can be used to mount complex attack scenarios.

3.1 Breaking Forward Secrecy

A first and simple attack to this protocol is to recover previous session keys from past traffic, once the long-term secret of a user is leaked. The security goal we are breaking in this case is *forward secrecy*, formally defined as follows:

Definition 1 (Forward secrecy [3]). *A protocol provides forward secrecy if compromise of the long-term keys of a set of principals does not compromise the session keys established in previous protocol runs involving those principals.*

For this type of attack, we assume that an attacker has collected the messages of the protocol for one or several runs, and that, at a latter stage, he is able to get access to the secret key of the responder. The goal is to recover previous session keys. In the case of the AKAPR, the attacker proceeds as follows:

1. The attacker stores the protocol messages for one or several rounds. For simplicity, let us assume he stores the traffic for only the run he wants to extract its session key.
2. At some point, the attacker compromises the responder R and obtains his long-term secret key sk_R.
3. From message M_2 of the stored protocol run, he parses $t \cdot h(sk_R \| id_I)^{-1}$ and extracts t, since he can compute $h(sk_R \| id_I)$. The value t is used to compute the encryption key $AK \leftarrow \mathsf{KDF}(id_I, id_R, t)$

4. From message M_3, the attacker can decrypt the values N_R, N_I and CT_{IR} using the key AK computed in the previous step.
5. The attacker computes the session key K_S used during the stores run as described in the protocol:

$$K_S = \mathsf{KDF}(CT_{RI}, id_I, id_R, N_I, N_R)$$

Therefore, it can be seen this protocol does not fulfill forward secrecy. We note that if the attacker compromises the initiator I instead, then the attack is the same except for step 3, where he uses sk_I to extract t from $t \cdot h(sk_I \| id_R)$, contained in message M_1.

Countermeasure The messages M_1 and M_2 are transmitted over an authenticated channel using MACs, with pre-shared MAC keys K_{ID} and K_{RD}, respectively. A possible countermeasure to the previous attack is to assume additional encryption keys for securing the confidentiality of the channel, given that the assumption of pre-shared keys already exists. However, if attackers were able to compromise the secret key of one user, it is reasonable to assume that the corresponding MAC key is potentially leaked too.

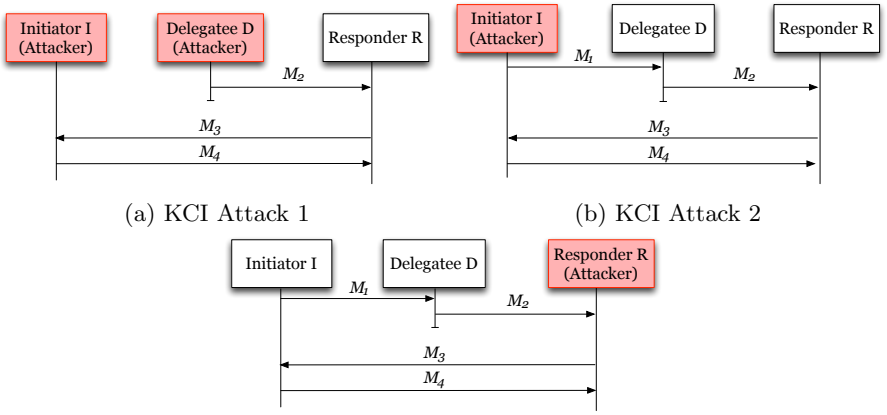
3.2 Key Compromise Impersonation Attacks

The previous subsection was devoted to attacking forward secrecy, a common concern of protocol designers. There are, however, other kinds of attacks that are not that well known, but can be potentially more hazardous. Key compromise impersonation (KCI) attacks occur when an adversary gains access to the secret key of a principal, and uses it to establish a session with him impersonating a different user. The attacker may use this session to actively gain new knowledge about the victim or causing him harm (e.g., by sending him malware), as the victim believes he is communicating with a legitimate user [4]. Therefore, KCI attacks can be more dangerous than breaking forward secrecy, which is limited to passive eavesdropping of past and future traffic. The following is a more formal definition of KCI, adapted from [20].

Definition 2 (Key compromise impersonation). *A key agreement protocol is vulnerable to key compromise impersonation (KCI) if compromise of the long-term key of a specific user allows the adversary to establish a session key with that user by masquerading as a different user.*

Here we describe three types of KCI attacks, although it may be possible that other variations exist. For the first two KCI attacks, we assume that the attacker compromises the secret key of one user and tries to impersonate an initiator I (which may be any user in the system chosen by the attacker); therefore, the victim acts as the responder R. Since the attacker needs to direct the attack via the delegatee, he needs at least one of the pre-shared MAC keys: either the key K_{RD} between the delegatee D and the responder R, or the key K_{ID} between the delegatee and the impersonated user I. The two first KCI attacks differ on

which MAC key is compromised. Finally, we also identify a third KCI attack that occurs in the opposite situation, when the attacker tries to impersonate a responder. Although in this case the attacker has to wait for a key agreement request from the victim (which can be forced by an out-of-band action), this attack is much easier to achieve, since it does not require any MAC key.



(a) KCI Attack 1

(b) KCI Attack 2

(c) KCI Attack 3

Fig. 3: Flow of KCI attacks

KCI Attack 1 If the attacker knows sk_R and K_{RD} then the strategy to impersonate an initiator I is to produce a message M_2 (i.e., the second message of the AKAPR protocol, which is between the delegatee and the responder), since this message does not depend on any secret from user I; the only requirement is to chose a counter value high enough. The responder R will accept this message as coming from user I via the delegatee, as depicted by Figure 3b, since its authenticity can be checked with K_{RD}. The attacker now only has to continue the protocol to establish a session key between him and R, although R thinks the session is between him and I. This attack works even in the attacker does not know sk_R, but only $h(sk_R \| id_I)$. The initial assumption of the attacker knowing K_{RD} is reasonable given he has access to sk_R.

KCI Attack 2 If the attacker knows sk_R and K_{ID} then the attacker needs to know also $h(sk_I \| id_R)$ in order to impersonate an initiator I. There are different ways to achieve this:

– A first option is to use past protocol traffic between R and I in order to extract this value, following a strategy similar to the forward secrecy attack: Knowing sk_R enables the attacker to compute the t value from the messages of a protocol run, which in turn, can be used to extract$h(sk_I \| id_R)$ from message M_1.

– A second option is to trick the delegatee into delivering this value by means of the re-encryption function, basically using it as a re-encryption oracle[2]. In order to do this, the attacker initiates a key agreement protocol between R and I, where C_1 is the component of message M_1 with value $t \cdot h(sk_R \| id_I)$. Next, he captures the response M_2 of the delegatee and parses the re-encrypted component C_2. Finally, he computes $h(sk_I \| id_R) = t \cdot (C_2)^{-1}$.
– A third option is to assume that the attacker colludes with the delegatee (or even that the attacker *is* the delegatee). In this case, $h(sk_I \| id_R)$ can be computed from the re-encryption keys since $rk_{I \to R} = h(sk_I \| id_R)^{-1} \cdot h(sk_R \| id_I)^{-1}$. In addition, the assumption of knowing K_{ID} is natural.

Once the attacker knows $h(sk_I \| id_R)$, he initiates a normal key agreement protocol with R via the delegatee, as shown in Figure 3b, using K_{ID} for computing the MAC for the first message. Note that the attacker can participate successfully in this key agreement since he does not need to know the secret sk_I, but the encryption token $h(sk_I \| id_R)$. As a final comment on this attack, note that it works even if the attacker initially does not know sk_R, but only the encryption token $h(sk_R \| id_I)$.

KCI Attack 3 Previous KCI attacks supposed that the leaked secret was of a user who later acted as the responder. Suppose now that the secret key of an initiator I is leaked. Figure 3c shows the protocol flow of this attack, where the attacker acts as the responder R. An attacker that knows the secret key sk_I of the initiator can impersonate any user R if he is able to obtain $h(sk_R \| id_I)$. As in the KCI attack 2, there are several ways to do this:

– Previous traffic between I and R. Knowing sk_I enables the attacker to compute the t value from the messages of a protocol run, which in turn, can be used to compute $h(sk_R \| id_I)$ from message M_2.
– Using the delegatee as a re-encryption oracle. This requires knowledge of K_{ID}.
– Assuming that the attacker controls the delegatee, colludes with it, or the corresponding re-encryption key is leaked somehow.

Once the attacker knows $h(sk_R \| id_I)$, he can respond to normal key agreement requests from I. As in previous KCI attacks, the attacker succeeds without actually knowing the secret sk_R, but the encryption token $h(sk_R \| id_I)$; similarly, this attack works even if the attacker only knows the encryption token $h(sk_I \| id_R)$ at the beginning.

As a final remark, KCI attacks 1 and 2 required the additional knowledge of one of the pre-shared MAC keys, since in both of them the attacker impersonated an initiator, whose messages are required to be validated, and therefore, is performing a proactive impersonation. On the contrary, in this KCI attack

[2] Note that in the traditional proxy re-encryption literature (i.e., in the public-key setting), the re-encryption oracle is considered as a delicate point. See for example [17], where several generic attacks using the re-encryption oracle are discussed.

the attacker impersonates a responder, which implies that the impersonation is reactive in this case (i.e., it requires that the initiator I starts the key agreement). This can be achieved either by waiting for a key agreement request to occur or by forcing it using some out-of-band mechanism (e.g., hard-resetting the initiator's device, social engineering, etc.).

3.3 Limited Scope of Key Revocation and Update

An interesting, yet not immediate, takeaway of the previous attacks is that they exploit the following undesired properties of the protocol: (1) encryption tokens can replace long-term secret keys, and (2) associated encryption tokens (e.g., $h(sk_I\|id_R)$ and $h(sk_R\|id_I)$) can be linked with each other through valid protocol messages. These issues are problematic when long-term secret keys are compromised, as illustrated by the attacks we identified.

A natural action that the affected principal performs once he becomes aware of the compromise of his secret key, is to initiate some kind of key revocation/update procedure. In principle, one can believe that key revocation is an effective countermeasure against a long-term key compromise event. However, we show next that this is not the case: once the long-term key of a user is compromised, and even after key revocation and update is realized, the protocol still remains vulnerable with respect to forward secrecy and key compromise impersonation, for all users whom the affected user communicated before. This is a consequence of the two undesired properties described above.

Breaking forward secrecy As an illustration, suppose that the responder R updates his secret key to sk'_R after his previous long-term secret sk_R is exposed, revokes previous re-encryption keys and asks for their update. In order for the protocol to be correct, the new re-encryption keys should be of the form $rk'_{I\to R} = h(sk_I\|id_R)^{-1} \cdot h(sk'_R\|id_I)^{-1}$, for all possible initiators I. Note that the term $h(sk_I\|id_R)$ does not change with respect to the previous re-encryption key, and therefore, the initiator I still uses this same encryption token when participating in a key agreement with the responder R. The consequence of this is that an attacker that was able to break forward secrecy before can extract $h(sk_I\|id_R)$ from previous rounds of the protocol, since $h(sk_I\|id_R)$ has not changed. Using this encryption token, the attacker can compute the corresponding t value from the first message of the protocol (i.e., M_1) of any future round, and break forward secrecy again.

Key Compromise Impersonation Suppose now that it is an initiator I who updates his secret key to sk'_I. Analogously to the previous case, it is still possible for an attacker to recover $h(sk_R\|id_I)$ from previous traffic between I and any other user R. However, in this case, the attacker does not limit himself to passively eavesdrop protocol messages as above (i.e., breaking forward secrecy), but can actively impersonate the responder R, given that he knows the value $h(sk_R\|id_I)$ necessary to decrypt message M_2. Notice how updating the secret key of I does not have any effect on the encryption token that protects the second message of the protocol.

Countermeasures The only possible countermeasures against this problem are either to issue a new identity for the affected user or to revoke also the users that established communications with him previously. Both options do not seem adequate nor practical.

4 Discussion

The previous attacks demonstrated that the main weaknesses of the AKAPR protocol is that the encryption tokens are, in fact, long-term secrets, just as the secret keys, and that the encryption tokens between two users can be linked to each other. These problems make it possible for an attacker to perpetually threaten the security goals of the protocol once he gains control of an encryption token. In this section we informally discuss some possible amendments to this protocol.

First, we note that if we assume that the initiator I is also capable of reading Message 2 (from the delegatee D to the responder R), then this means he can extract the encryption token $h(sk_R \| id_I)$. This is a plausible assumption in several settings compatible with this protocol (e.g., wireless environments), and it is also consistent with the philosophy of the Dolev-Yao model [8], by which one can consider the network to be open and all its messages public and subject to scrutiny by other entities. Therefore, it can be seen that this is functionally equivalent to the initiator I knowing this encryption token. Given that this encryption token is implicitly known by the responder (since he can generate it), then it acts as a sort of shared key between them. Therefore, a possible variation of the protocol is to simply distribute this encryption token to the initiator when he wants to commence a key agreement with responder R. This can be realized by transforming the delegatee D into a mere key server that distributes these encryption tokens to initiators. Therefore, instead of the delegatee re-encrypting Message 1 into Message 2, which requires it to know a re-encryption key linking two encryption tokens, he stores the "shared encryption tokens".

Note that this variation is still compatible with achieving the goal of secrecy with respect to the delegatee, simply by encrypting these encryption tokens with the secret key of the corresponding initiator. Therefore, re-encryption keys stored by the delegatee D are replaced by encryptions of encryption tokens. Specifically, each $rk_{I \to R}$ is substituted by an encrypted key $ek_{I \to R} = \mathsf{Enc}_{sk_I}(h(sk_R \| id_I))$. Note that this implies that $ek_{I \to R} \neq ek_{R \to I}$, and therefore, this doubles the number of keys managed by the delegatee D; nevertheless, this number is still of quadratic order (since it is necessary to store a key between each pair of users of the system), dominated by the number of users. This new type of keys eliminates the link between encryption tokens, reducing this way the applicability of some of the attacks we propose (although not completely). An interesting insight about this variation is that it is extremely similar to the traditional Needham-Schroeder symmetric key agreement protocol [14], with the main difference that in this case the key server (i.e., the delegatee) does not know the session keys because these are encrypted by a separate key server (i.e., the KDC).

However, it is important to note that this variant still suffers from the long-term nature of encryption tokens. An improvement to this respect could be to include a timestamp in the encryption token generation, in a similar way than Kerberos [15] improves over Needham-Schroeder protocol. Thus, encryption tokens would be of the form $h(sk_A \| id_B \| T_i)$, where T_i represents a time period. Although this could mitigate the attacks to a certain extent (since compromised encryption tokens would only be useful for a given time period), it also implies that the re-encryption keys should be recomputed by the KDC and distributed to the delegatee on each time period. This can represent a great inconvenience in some settings.

5 Conclusions

Proxy re-encryption (PRE) is a hot research topic nowadays, with new use cases arising in different contexts. Most PRE schemes are based on public-key cryptography, mostly because public key cryptography has become omnipresent in most environments. PRE schemes using symmetric cryptography would make an excellent contribution for IoT scenarios where devices are highly constrained, but it is true that properly capturing the essence of PRE and providing secure solutions in the same sense as in the public-key world is challenging.

In this paper we show how difficult is to properly define and implement PRE in the symmetric world by exploring the weaknesses of a particular scheme proposed at ESORICS 2016. This protocol tries to adapt previous proposals for symmetric key PRE to the IoT scenario, focusing not only on providing a lighter solution (not involving public-key cryptography) but also trying to avoid redundant messages that are needed in well-know and deeply-studied authentication and key exchange protocols (e.g., Needham-Schroeder).

Unfortunately the paper fails to accomplish its goals, mainly because of the use of two immutable encryption tokens for every pair of communicating parties that depend only on the secret keys of the initiator and the identity of the responder (and vice versa). The way the protocol is designed implies that compromising the secrets of one party allows to obtain the encryption token of the other party, which can be used to mount key compromise impersonation attacks and break forward security. Moreover, revocation of compromised keys turns out to be ineffective, which makes the protocol unusable in it present form.

Acknowledgments

This work was partly supported by the Junta de Andalucía through the project FISICCO (P11-TIC-07223) and by the Spanish Ministry of Economy and Competitiveness through the PERSIST project (TIN2013-41739-R). The first author is supported by a contract from the Regional Ministry of Economy and Knowledge of Andalucía.

References

1. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The Keccak SHA-3 submission. *Submission to NIST (Round 3)*, 6(7):16, 2011.
2. D. Boneh, K. Lewi, H. Montgomery, and A. Raghunathan. Key homomorphic prfs and their applications. In *Advances in Cryptology–CRYPTO 2013*, pages 410–428. Springer, 2013.
3. C. Boyd and A. Mathuria. *Protocols for authentication and key establishment*. Springer Science & Business Media, 2013.
4. K. Chalkias, F. Baldimtsi, D. Hristu-Varsakelis, and G. Stephanides. Two types of key-compromise impersonation attacks against one-pass key establishment protocols. In *International Conference on E-Business and Telecommunications*, pages 227–238. Springer, 2007.
5. D. L. Cook and A. D. Keromytis. Conversion functions for symmetric key ciphers. *Journal of Information Assurance and Security*, 1(2):119–128, 2006.
6. D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Commun. ACM*, 24(8):533–536, Aug. 1981.
7. W. Diffie and M. E. Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, 1976.
8. D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.
9. T. Duong and J. Rizzo. Flickr's api signature forgery vulnerability, 2009.
10. W. C. Garrison, A. Shull, S. Myers, and A. J. Lee. On the practicality of cryptographically enforcing dynamic access control policies in the cloud. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 819–838, May 2016.
11. D. H. Krawczyk and P. Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, Oct. 2015.
12. Z. Liu, X. Huang, Z. Hu, M. K. Khan, h. seo, and L. Zhou. On emerging family of elliptic curves to secure internet of things: Ecc comes of age. *IEEE Transactions on Dependable and Secure Computing*, PP(99):1–1, 2016.
13. A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*. CRC press, 1996.
14. R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.
15. B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications magazine*, 32(9):33–38, 1994.
16. K. T. Nguyen, N. Oualha, and M. Laurent. Authenticated key agreement mediated by a proxy re-encryptor for the internet of things. In *European Symposium on Research in Computer Security*, pages 339–358. Springer, 2016.
17. D. Nuñez, I. Agudo, and J. Lopez. A parametric family of attack models for proxy re-encryption. In *Proceedings of the 28th IEEE Computer Security Foundations Symposium*, CSF'15, pages 290–301. IEEE Computer Society, 2015.
18. D. Nuñez, I. Agudo, and J. Lopez. Proxy re-encryption: Analysis of constructions and its application to secure access delegation. *Journal of Network and Computer Applications*, 87:193–209, 2017.
19. H. Sakazaki, K. Anzai, and J. Hosoya. Study of re-encryption scheme based on symmetric-key cryptography. In *31st Symposium on Cryptography and Information Security (SCIS 2014)*, 2014.
20. M. A. Strangio. On the resilience of key agreement protocols to key compromise impersonation. In *European Public Key Infrastructure Workshop*, pages 233–247. Springer, 2006.

21. A. Syalim, T. Nishide, and K. Sakurai. Realizing proxy re-encryption in the symmetric world. In *International Conference on Informatics Engineering and Information Science*, pages 259–274. Springer, 2011.
22. G. Tsudik. Message authentication with one-way hash functions. *ACM SIGCOMM Computer Communication Review*, 22(5):29–38, 1992.
23. D. Watanabe, H. Sakazaki, and K. Miyazaki. Representative system and security message transmission using re-encryption scheme based on symmetric-key cryptography. *Journal of Information Processing*, 25:67–74, 2017.
24. Wikipedia. SpongeBob SquarePants — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/SpongeBob_SquarePants`, 2016. [Online; accessed 18-October-2016].

Appendix: Length-extension attacks

We now describe a completely different attack strategy, based on a potentially dangerous design choice in the AKAPR protocol. The symmetric proxy re-encryption primitive that underlies the key agreement protocol (see Section 2.1), uses a keyed hash function to derive the encryption tokens. Specifically, these values are of the form $h(sk_A \| id_B)$, where the identity of the recipient is used as the message of a keyed hash, with $h : \{0,1\}^* \to \mathbb{Z}_p$ as the hash function[3]. The authors state that the hash function is required to behave as a random oracle, which can be a rather strong assumption. However, it is known that real instantiations of hash functions do not necessarily behave as random oracles. In fact, widely used hash functions such as SHA-1 and SHA-256 (as well as others based on the Merkle-Damgård construction) suffer from length-extension vulnerabilities [22] that make possible to create new encryption tokens without knowing sk_A, under the assumption that identifiers can have variable length.

First, let us recall the idea of length-extension attacks. In this type of attacks, knowledge of the hash value $h(m_1)$ and the length of m_1 can be used to compute $h(m_1 \| pad \| m_2)$, for any message m_2 chosen by the attacker and some required internal padding pad. It is known that hash functions based on the Merkle-Damgård construction (e.g, MD5, SHA-1, SHA-256) are vulnerable to this type of attacks. This problem has been shown to permeate to real systems, such as the Flickr's API signature forgery vulnerability [9].

In the AKAPR protocol, if we assume that identifiers can have different lengths (i.e., $id \in \{0,1\}^*$), then it may be possible that, for some identity id_B, there exists an identity id_{BX} where id_B is a prefix. For example, let us assume that $id_B =$ "Bob" and that the attacker knows $h(sk_A \| id_B)$, but that sk_A remains secret. Then, by the length-extension vulnerability he can produce $h(sk_A \| id_{BE})$,

[3] Note that when we assume that h is implemented with a traditional hash function, it is also necessary an additional encoding from its output domain to $\mathbb{Z}_p$, as required in Section 2.1; usual encodings of this type (e.g, taking modulo p) are easily invertible, so we will omit this for simplicity.

where id_{BE} = "Bob Esponja"[4]. For the sake of illustration, let us ignore the internal padding *pad* for the moment.

As discussed in Section 3.2, an attacker that knows $h(sk_A \| id_{BE})$ can impersonate A in a key agreement with BE without knowing sk_A, regardless if he acts as an initiator or a responder. The length-extension attack is particularly worrisome when the attacker has access to previous traffic, or the delegatee is compromised or deceived by the attacker; in this case, the delegatee can be used in combination with the attack strategies presented in previous sections in order to compromise the security of non-involved users. For example, let us suppose that an attacker gains access to the private key of user B. Figure 4 shows a combination of some of the attacks described in this paper that eventually compromise the security of key agreements between users A and BE. An attacker that knows sk_B can trivially generate encryption tokens $h(sk_A \| id_B)$ associated to any other user A; knowing this, he can compute the opposite encryption tokens using strategies discussed in previous attacks (e.g., using the delegatee as an encryption oracle, compromising the delegatee, analyzing previous traffic between A and B). The resulting encryption tokens can be used as input to length-extension attacks, undermining this way the security of other users. The output of the length-extension attacks are also encryption tokens, which in turn, can be used again to obtain more encryption tokens. Therefore, it can be seen how the initial leakage of B's secret can potentially affect key agreements between other users, since once the attacker obtains $h(sk_A \| id_{BE})$ and $h(sk_{BE} \| id_A)$, he can potentially control all their past and future key agreements. Note that, although BE is some user whose identity has id_B as prefix (i.e., the choice of BE is not completely free), A is an arbitrary user, which is a serious threat to the application of this protocol in a real setting.

We next describe some possible attack scenarios that apply this strategy in combination with others of the previously identified attacks. In particular, we illustrate two scenarios: the first is reminiscent of sybil attacks in distributed networks, where a malicious entity gains control of multiple identities in order to increase its control over a system or network; the second is similar to a spear phishing attack, where the attacker targets specific victims aided by relevant, yet fake, contextual information that gains the trust of the victim.

Sybil-like attack scenario The goal of the attacker in this scenario is to gain as much encryption tokens as possible, each of them associated to a different identity, and to start multiple key agreements with a victim A, who will believe is communicating with different entities. In particular, the idea is to obtain encryption tokens between A and fresh "pseudonyms" (all of them with the same prefix in order to exploit the length-extension vulnerability, as described before).

[4] "Bob Esponja" is the Spanish name for "SpongeBob SquarePants" [24]. Not associated with the sponge-based hash function SHA3 [1], which, interestingly, is not susceptible to length-extension attacks.

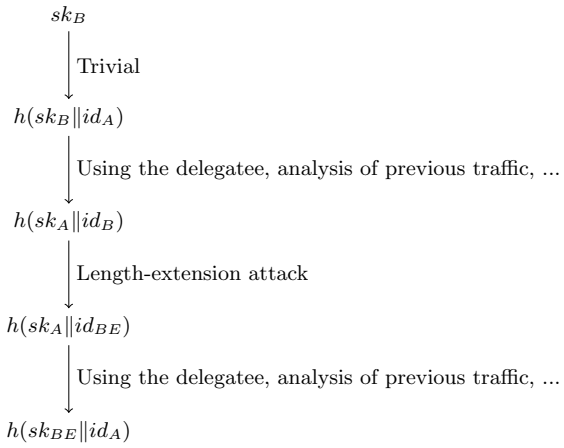


Fig. 4: Abstract flow of a length-extension attack

We initially assume either that the attacker is a legitimate user (e.g., user B), or that he gained access to the secret key sk_B of some user B; this does not matter from the point of view of the attack. Now, he can set up a sybil attack against any user A of his choosing as follows, also represented in Figure 5. First, knowledge of sk_B allows to compute $h(sk_B \| id_A)$ trivially; next, using the techniques described for previous attack strategies, the attack obtains the opposite encryption token $h(sk_A \| id_B)$. Now he proceeds to launch the length-extension attack against this hash value, by appending random i values to the end of id_B, obtaining as a result hashes of the form $h(sk_A \| id_B \| pad \| i)$. Therefore, fresh pseudonyms will be identities of the form $id_{Bi} = id_B \| pad \| i$. Note that in this case the internal padding pad that is necessary for the length-extension attack is not necessarily a problem (e.g., if identities have a numerical format); the only important requirement from the point of view of the sybil attack is that identities id_{Bi} are different from each other.

Finally, using once again the techniques for obtaining the opposite encryption token, the attacker eventually obtains a set of encryption tokens of the form $h(sk_{Bi} \| id_A)$, for different i values, which enable him to mount a sybil-like attack against entity A. The attacker can now start multiple key agreements with the victim A, making him believe that there are several independent entities B_i, which in fact are controlled uniquely by the attacker.

Spear phishing scenario Suppose that an attacker gains access to sk_B, where id_B is a common but dangerous prefix (e.g., "Bank"). Once again, following the previously discussed strategies, the attacker can obtain the encryption tokens between a victim A and user B, namely $h(sk_A \| id_B)$, and later use this hash value to derive new encryption tokens involving a tailored bait whose name starts with "Bank" (e.g., "Bank of America"). Recall that the attacker does not need

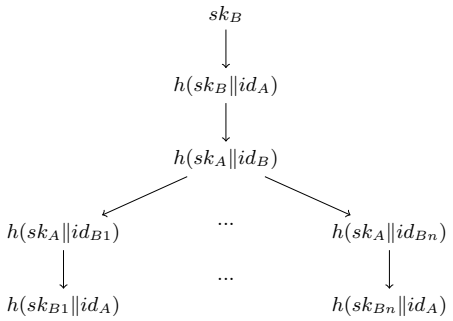


Fig. 5: Abstract flow of a sybil attack

to know the secret of "Bank of America", but uses instead our attack strategies to obtain an encryption token that compromises the security with respect to the victim A. This type of attack also works if, instead of compromising user B, we suppose B does not initially exist and the attacker is capable of registering it with the KDC as a new user.

There is, however, a small problem with the internal padding *pad* in this case. Following our example, the length-extension attack would allow the attacker to compute the encryption token $h(sk_A \| \text{"Bank"} \| pad \| \text{"of America"})$. Therefore, the forged identity is actually $\text{"Bank"} \| pad \| \text{"of America"}$, which can be problematic if the padding allows the victim to detect the attack. We will, however, illustrate with a real example how this problem can be overcome, and experimentally demonstrate the viability of the attack.

Suppose that the protocol uses SHA-256 as hash function and that secret keys are of 256 bits. In our experimental attack, the secret key is the hexadecimal value `3F` (which corresponds to the character '?') repeated 32 times; note that the specific value of this key is irrelevant for the attack and that it always remains unknown to the attacker. Table 2 shows the concatenation of the secret key and the identity "Bank", represented both as text and as hexadecimal strings, as well as the output of the hash function for this input. We suppose now that the attacker knows this output value (i.e., the encryption token $h(sk_A \| \text{"Bank"})$); knowing this, the attacker can use the length-extension vulnerability of SHA-256 to compute the encryption token $h(sk_A \| \text{"Bank"} \| pad \| \text{"of America"})$.

The concrete result for the length-extension attack applied to the previous values is also presented in Table 2. Note that the first four bytes of the extended identity correspond to the string "Bank" (`42 61 6E 6B`). Next, there is the internal padding *pad* (`80 00 ... 00 01 20`), followed by the characters that correspond to the extension string "of America" (`6F 66 ... 63 61`). It should be noted that the intermediate padding may be rendered differently depending on the implementation of the user agent (e.g., browsers, mobile apps, standalone GUI, etc.), and some options may be exploitable. For example, when the extended identity $\text{"Bank"} \| pad \| \text{"of America"}$ is rendered with the ISO 8859-

Table 2: Spear phishing based on a length-extension attack

Original encryption token $h(sk_A \| \text{“Bank”})$	
Hash input (string)	“????????????????????????????????Bank”
Hash input (hex)	3F 3F 3F 3F 3F 3F 3F 3F 3F 3F 3F 3F 3F 3F 3F 3F 3F 3F 3F 3F 3F 3F 3F 3F 3F 3F 3F 3F 42 61 6E 6B
Hash output (hex)	D2 7A D9 E5 FD FC 84 3E 8F 73 74 05 72 04 1D 80 72 48 F2 58 09 06 04 BF 5A 38 AA B7 B5 74 C8 CB
“Extended” encryption token $h(sk_A \| \text{“Bank”} \| pad \| \text{“of America”})$	
“Bank”$\|pad\|$“of America” (hex)	42 61 6E 6B 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 20 6F 66 20 41 6D 65 72 69 63 61
“Bank”$\|pad\|$“of America” (’latin1’)	“Bank of America”
Hash output (hex)	4F FA C6 1B A9 9F F2 AD 28 93 92 21 BD 7D 12 CF F4 01 BA C5 5F 41 C3 FB 64 41 F7 45 EE 17 5C A8

1 encoding (also called ‘latin1’), the intermediate padding may be ignored, depending on the implementation, since it is composed of NUL characters (`00`) and invalid codes (`80`). This is the case of Python 2.7.10 implementation, which we used for reproducing the length-extension attack, and that displays “Bank of America” as the extended identity when ‘latin1’ encoding is selected. This can be used to deceive users into believing they are interacting with the real Bank of America, since the only difference between the legitimate identity and the displayed extended identity are non-printable characters (i.e., the user cannot visually distinguish one string from another). This also can be reproduced with UTF-8 encoding if errors are ignored when printing UTF-8 encoded strings (in Python this is achieved by specifying the option `‘ignore’`).

We note that this example, although somewhat artificial, only makes relatively common assumptions, such as the use of SHA256, secret keys of 256 bits, ‘latin1’ encoding of strings, variable-size identities, etc.

Countermeasures As a simple and effective countermeasure to the length-extension attack, the protocol can require the use of a hash function resistant to length-extension attacks (e.g., SHA3), or even better, use an HMAC or a key derivation function (e.g., HKDF [11]). HMACs and KDFs are specifically designed to take a secret as input, and to be secure against length-extension attacks.