Symmetric key authentication using verification in public*

Jaap-Henk Hoepman

Department of Computer Science, University of Twente
P.O.Box 217, 7500 AE Enschede, the Netherlands
hoepman@cs.utwente.nl

Abstract Several protocols for authentication using memory-constrained smart cards or tokens based on symmetric key cryptography are described. Key property of these protocols is that verification on the server can be performed in public based on (more or less) publicly known information. The protocols use a symmetric cipher in an asymmetric fashion: the verifier uses a verification key that cannot be used to generate a valid authentication response.
Verification in public means that the verification server does not have to be equipped with some tamper resistant device storing the verification keys. Nor is it necessary to secure the server to similar strength in a different fashion. The protocols are especially suitable for environments in which smart cards or active tokens are used as authentication devices. In these environments symmetric key protocols are extensively used, and will be used for some time to come, to reduce both system cost and transaction time.

1 Introduction

We present three protocols for authentication based on symmetric key (instead of public key) cryptography, where the verifier does not need to store any secret information. Therefore the verifier does not have to be implemented using tamper resistant hardware or be run on a secure server, but instead can be built as a software program running on off-the-shelf hardware. The protocols use a symmetric cipher in an asymmetric fashion: the verifier uses a verification key that cannot be used to generate a valid authentication response.

Originally, we studied the problem of software authentication in the context of an already fielded electronic purse smart card system. This has influenced the choice of threat model, system requirements and in turn the design of our protocols. In particular, using public key cryptography was (and still is) not an option under these circumstances. We elaborate on this issue in Sect. 2.

* Id: chip-ident.tex,v 3.3 2000/08/29 10:24:46 hoepman Exp
[1] A preliminary version of this paper was presented at the *Financial Cryptography 2000* conference.

However application of our protocols is certainly not limited to smart card based systems. In fact, they are equally suitable for similarly resource challenged systems like active tokens or Personal Digital Assistants (PDAs). We discuss several applications of our protocols at the end of the paper, in Sect. 7.

In a smart card environment, a very large number of both smart cards and verification terminals are in use. The cost of a smart card is determined by its cryptographic capabilities and the amount of memory available on-chip. Similarly, the cost of a terminal can be reduced significantly if it does not contain special tamper resistant (or tamper evident) hardware. Cheaper systems should therefore satisfy the following two constraints.

- The smart cards or tokens must be simple devices that only use simple cryptographic primitives and have only limited memory available to store information about previous authentications.
- The verification servers (e.g. smart card terminals) cannot rely on dedicated tamper resistant hardware to store verification keys. The choice of authentication protocol depends on the threat model, that captures the amount of public access to the verification data one assumes exists.

All three protocols described in this paper have been designed to meet these two constraints.

The core of the first protocol (presented in Sect. 4) is the use of diversified keys (cf. Sect. 2.1) to compute verification keys. The verifier obtains a verification key for a user from a central back office. To prove identity, prover and verifier engage in a classic challenge response protocol using this verification key. In this protocol the adversary is allowed to view data stored by verifiers, except the verifier under attack. This models the case where verifiers may collude to defraud other verifiers, except themselves.

The core of the second protocol (presented in Sect. 5) is a novel application of the one-time password scheme by Lamport [Lam81]. The main difference is that the hashed secret stored by the verifier is not obtained from the user directly, but instead from a central back office. This protocol is developed for a slightly stronger adversary that is also allowed to use data stored by the verifier under attack. Moreover, provisions are added to this protocol to allow billing based on the number of authentications performed.

The core of the third protocol (presented in Sect. 6) is a novel application of the one-time signatures of Merkle [Mer89] inspired by a protocol of Wu and Sung [WS96]. This protocol prevents an active querying attack, which is only prevented in our second protocol by relying on the vigilance of the user. The protocol of Wu and Sung [WS96] does not provide this protection (i.e. their protocol is vulnerable to active querying attacks on the prover), as explained in Sect. 6.4. Moreover, their protocol requires that the prover stores some secret data for each verifier to which it connects. This violates the first constraint above.

We see that the choice of protocol depends on the choice of adversary against which it must protect. These threat models are discussed in Sect. 3, after a general description of a typical application area of these protocols — namely smart card based systems — in the next section.

2 Case: the Smart Card

A smart card is a plastic card with a single embedded chip, containing a small 8–16 bit CPU, 256–512 bytes of RAM, 8–16kB EEPROM to store keys and data while the chip is disconnected from power, and 16–32kB ROM containing the operating system and permanent keys. The chip communicates with the environment through a 6 through 8 pin connector fitted directly on top of it. It also receives its' power through these contacts. Typically, the chip is made tamper resistant using several protection mechanisms [KK99]. For more information we refer to Dreifus and Monk [DM98] for the technical perspective and Allen and Barr [AB97] for the business perspective on smart cards.

For reasons of cost and transaction speed, most operational smart card systems use symmetric key cryptography (e.g. DES [FIP77] or triple DES) to protect the integrity and confidentiality of commands and data. This means that both the smart card and the smart card reader (a.k.a. the terminal) must store secret key material. On the terminal, a special smart card called the Secure Application Module (SAM) is used as secure storage for this purpose.

Generally, in a smart card system, the smart card itself is merely used as a secure storage device for keys and other sensitive and/or valuable application data. The active part of a smart card application (like for instance an electronic purse) is complete implemented on the terminal, or, for security reasons, on the SAM it contains.

The situation will not improve in the near future. Although more advanced smart cards, capable of handling public key cryptography (RSA) or implementing interpreters on the smart card (JavaCard), are being developed by smart card manufacturers, their cost and concern over increasing transaction times prohibit full scale commercial application at the moment.

This clear separation of the data and the active part of a smart card application has two consequences.

- Usually, the smart card is capable of handling a new application without modification to its operating system, although sometimes extra data fields will have to be created. This is covered by most smart card standards.
- The SAM is highly geared towards a specific application. The terminal, and specifically the SAM, *do* have to be changed to accommodate the new application.

This makes introducing new smart card applications on an already fielded smart card system hard, because of the additional cost of replacing SAMs in all terminals in the field. Moreover, for high throughput applications where a lot of authentications have to be performed in a small period of time, SAMs prove to be a large performance bottleneck.

This paper shows that for a particularly important application like user authentication, the aforementioned problems with using SAMs can be avoided. We present three protocols where the verification can be implemented in the terminal in software only, without the need for a SAM at all, while not leaking valuable secret information. This means that our protocols yield a solution using

both cheap terminals and cheap smart cards, while retaining high transaction speeds. A purely public key solution would require more expensive smart cards and would increase the transaction time considerably.

Our protocols make very pessimistic assumptions about the amount of storage available on a smart card. In fact, our protocols are more or less stateless when considering only the smart card (except for the storage of the keys).

2.1 Basic Authentication Using Smart Cards

Usually, authentication protocols are based on a simple challenge-response mechanism which proves to the verifier that the prover has the secret key (cf. [Sch96]). For smart card systems this simple scheme is impractical, because it would imply that either all cards hold the same key, or that each SAM contains all keys of all fielded cards (which may run in the millions).

The way forward is to use so-called *diversified keys* [AB96]. These keys are given to the provers, and are all derived from a single master key which is stored by the verifier. Each prover has a different key, and the verifier can derive this key using the master key and the identity of the prover. Other ways of deriving this key are impossible, so that given the key of one prover it is not possible to compute the key of any other prover. One particular method of diversifying keys is to encrypt the identity of the prover with the master key using DES.

$$k_i = \{i\}_{k_M} \tag{1}$$

where $\{m\}_k$ denotes encryption of message m using key k (where the encryption method is implicit), k_i is the key of prover i (stored on the smart card) and k_M is the master key (stored in the SAM).

Authentication is performed by challenge response as before, except that the verifier first diversifies the key for the prover given his identity and the master key, before verifying the response returned by the prover.

3 Protocol Requirements and the Threat Model

We wish to implement an authentication protocol where the verifier can be implemented without the need for tamper resistant hardware. To be precise, we wish to implement an authentication protocol between a prover P and a verifier V that allows V to verify the claimed identity of P. Each protocol run is initiated by the verifier. At the end of run of the protocol V must decide either to accept P (viz. it is convinced of P's identity), or to reject the run. P also marks any runs in which it detects an error as rejected; it marks all other runs as accepted. P and V may also be engaged in unfinished runs (in which P and V have not decided yet).

Given an adversary to be described in the next subsection, we require for each run of the protocol that if V accepts P, P also accepts the run, and that all

messages received by V were sent (possibly relayed by other parties)[1] by P after V initiated this run.

3.1 Threat Model

Within our threat model we distinguish the following entities (with in parentheses the corresponding entities in a smart card system):

- a *prover* (a smart card),
- a *network* (between prover and verifier, and between verifier and back office),
- a *verifier* (a terminal),
- a *back office*, (which issues the cards, manages the terminals and provides the terminals with data). We assume that the terminal has some way to authenticate the back office (e.g., using a public key protocol), but *not* the other way around (so the terminal only stores the public key of the back-office), and
- an *adversary* that tries to break the protocol.

The adversary does not have unlimited power. In fact we restrict the adversary by allowing it only to do the following (cf. [BGH+92]):

1. to intercept, store-and-replay, modify or create new messages on the network, both between prover and verifier and verifier and back office,
2. to know the authentication protocol and the format of all messages exchanged,
3. to start simultaneous sessions with the back office or any number of provers and verifiers, including parallel sessions involving the same verifier or prover, and
4. to see the data stored inside a verifier,
   (a) except for the verifier under attack (for the first protocol).
   (b) including the verifier under attack (for the second protocol).

We assume that the adversary cannot do any of the following:

- see or modify any data within the prover, except by sending valid commands to the prover,
- see or modify any data stored in the back office, except by sending valid commands to the back office,
- modify any data within the verifier, except by sending valid commands to the verifier,
- invert a one-way hash function, compute a valid MAC or decrypt a message without knowing the secret key, or
- derive the secret key in either the prover, the verifier under attack, or the back office, given any set of plaintext-ciphertext pairs that can be collected during the lifetime of the system.

[1] We cannot rule out the possibility that an adversary simply forwards all messages to the prover and sends the responses back to the verifier (cf. [BGH+92]).

3.2 Discussion of the Threat Model

Originally, we studied the problem of software authentication in the context of an already fielded smart card system. The choice of threat model clearly reflects this. For instance, the assumption that the data stored by the prover cannot be modified or seen is natural in this case. For smart card systems we really have two possible assumptions on the power of the adversary w.r.t. the terminal.

Terminals contain SAMs mainly because they contain information the verifier (viz. the merchant) is not allowed to see, as he can abuse that information for 'personal' gain. We assume that in order to read the data inside the terminal, cooperation of the verifier is necessary. We further assume that the verifier has nothing to gain from giving away information that will create fake authentications on his own terminal; for him there are much easier ways to bypass the terminal altogether. In this model, the protocol based on diversified keys, as described in Sect. 4, is appropriate.

If the terminal is located in a less secure environment, which makes it likely that data in the terminal can be read without the verifier's cooperation, then we must assume that the adversary knows all data stored by the terminal under attack. Under these circumstances, the protocol based on one-time-passwords described in Sect. 5 or the protocol based on one-time signatures described in Sect. 6 must be applied.

4 Using Diversified Keys for Authentication

The simple challenge response approach outlined in Sect. 2.1 requires a secret key to be used by the verifier, which could be abused to generate fake authentication protocol exchanges with other verifiers. Hence this key must be stored securely by the verifier.

In this section we present an authentication protocol which does not suffer from this drawback: the verifier can be implemented completely in software, assuming that the adversary can see all data stored by any verifier except the verifier under attack (cf. Sect. 3, item 4 (a)).

The protocol does require that the verifier occasionally contacts a central back office system to obtain new authentication data[2].

4.1 The Protocol

The protocol appears, schematically, in Fig. 1. We see that each entity stores the following data.

[2] This may not be a major drawback, because an important case under consideration is upgrading existing smart card systems used for payments to include authentication functionality. These terminals (that implement the verifier) are already required to contact the back office regularly to deposit collected payments or to get approval to perform a transaction.

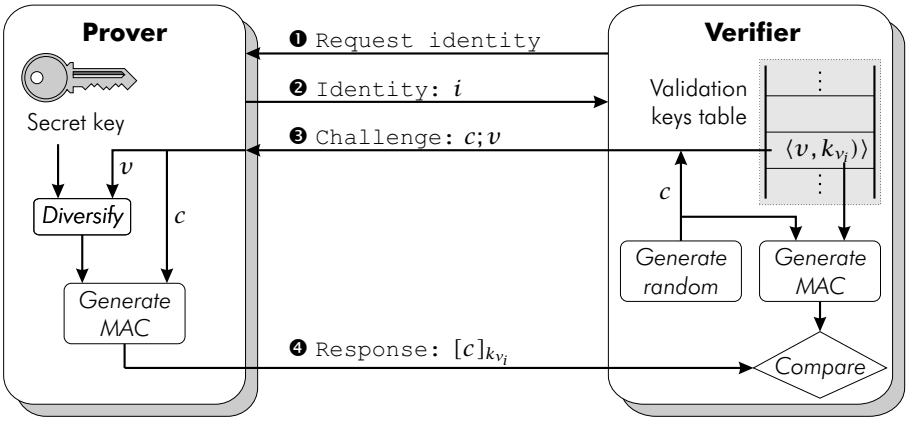

Figure 1. Authentication protocol using diversified keys

- the prover (with identity i): stores a diversified symmetric key k_i, where $k_i = \{i\}_{k_M}$ for some masterkey k_M stored by the backoffice.
- the verifier: storing for each user/prover i a tuple $\langle v, k_{v_i}\rangle$ — where k_{v_i} is a diversified symmetric *verification key* generated by the back office using i and v as input. In fact $k_{v_i} = \{v\}_{k_i}$.
- the central and secure back office system with access to the master key k_M, from which k_i and k_{v_i} (given i and v) can be derived.

Here, $[m]_{k_i}$ denotes the message authentication code (MAC) for message m generated using key k_i. Again the exact cryptographic protocol is implicit; ANSI X9.19 [ANS86] (DES based) is assumed.

The authentication protocol now runs as follows.

1. The verifier requests the identity form the prover.
2. The prover sends his identity i.
3. The verifier checks to see if it owns a verification key k_{v_i} for i. If so, continue with step 4.
   (a) Otherwise, authenticate the back office, set up an encrypted and integrity preserving communication channel with it using a random session key, and request a new verification key for user i.
   (b) The central back office computes $k_i = \{i\}_{k_M}$, generates a random number v, and computes $k_{v_i} = \{v\}_{k_i}$.
   (c) The central back office sends v and $k_{v_i} = \{v\}_{k_i}$ to the verifier, who stores the tuple $\langle v, k_{v_i}\rangle$ for user i. Optionally, the back office may send additional identifying information to be stored in the tuple, for instance a provers name, address or the like.
4. The verifier generates a random challenge c, selects the tuple for user i and sends c and v to the prover.

5. The user/prover agrees to authenticate[3]. Otherwise the prover rejects the run and aborts.
6. The prover computes $k_{v_i} = \{v\}_{k_i}$, and then computes the response $r = [c]_{k_{v_i}}$, which it sends to the verifier.
7. The verifier receives r as response and checks whether r equals $[c]_{k_{v_i}}$ (which the verifier computes locally). If so, then the verifier accepts i, otherwise authentication fails and the verifier rejects.

A potential problem with this protocol is the large database of tuples a verifier may have to store locally. However, verifiers can save space if they can afford to go on-line more often, using a caching like strategy to only keep tuples for recent visitors and throwing away tuples that haven't been used for some time. Actually, the security of the system would improve if the verification keys were expired regularly by throwing away tuples more than a few days old.

4.2 Security

We split the proof of security in two parts. We first consider the authentication protocol between prover and verifier. Next we consider the reloading protocol to obtain a fresh tuple with a verification key for a user i from the central back office.

Between Prover and Verifier. To prevent replay attacks, fresh challenges must be sent in each run of the protocol. This is indeed the case due to the randomness of c generated by the verifier.

For a cipher like DES we may assume that knowledge of k_i and k_{v_i} does not reveal any information on k_j or k_{v_j} for different j, if we assume that v generated by the back office is fresh. Again this is the case due to the randomness of v. Also, given k_{v_i}, k_i cannot be deduced.

Hence passive eavesdropping (including looking inside any other verifier) reveals no information on k_i and k_{v_i}.

Given a fresh challenge $c; v$, to compute the valid response $[c]_{k_{v_i}}$ either k_i or k_{v_i} must be known. From the previous discussion we conclude that neither of these are known to the adversary, and hence the valid response must be computed by the prover.

Between Verifier and Back Office. To verify the responses from prover i, the verifier needs to obtain a tuple $\langle v, k_{v_i} \rangle$ from the back office. To ensure that only i can compute a response that is considered valid by the verifier, we must guarantee that

- v is fresh (see previous section),
- k_{v_i} received by the verifier indeed equals $\{v\}_{k_i}$ where $k_i = \{i\}_{k_M}$, and
- k_{v_i} remains secret.

[3] E.g., by entering a PIN to unblock the smart card

k_{v_i} is correctly computed using a fresh v by the back office. To transfer these safely to the verifier requires authentication of the back office and an integrity preserving, encrypted communication channel between the verifier and the authenticated back office.

5 Using a One-Time Password Scheme for Authentication

The protocol using diversified keys has a major drawback, in that it assumes that the adversary does not have access to the data stored by the verifier under attack. This may not be a reasonable assumption in all circumstances. In this section we present another authentication protocol to get rid of this assumption. However, the protocol does require that the verifier contacts, more frequently, a central back office system to obtain new authentication data. Moreover, the protocol requires the prover to repeatedly apply a one-way hash function to compute a response. The maximum number of such hashes is a protocol parameter.

The protocol is based on the one-time password scheme [Lam81, HMNS98], which we briefly outline in Sect. 5.1. Next, in Sect. 5.2, we present our authentication protocol.

5.1 OTP: the One-Time Password Scheme

The one-time password scheme OTP works as follows. Initially, the prover generates a secret and sends a multiple hash of it to the verifier (presumably through some secure channel to prevent modification of this hash). That is, the prover generates s and sends the tuple $\langle n, h^{n+1}(s)\rangle$ to the verifier, which stores it. Here h is a one-way hash function like SHA-1 [FIP95] or MD5 [Riv92]. To establish the identity of the prover, the verifier sends the index n as the challenge to the prover, which must respond with the n-th hash $h^n(s)$ of its secret. The verifier checks whether hashing the response (i.e. $h(h^n(s))$) equals the hash $h^{n+1}(s)$ stored in the tuple. If so, authentication succeeds and the verifier stores the new tuple $\langle n-1, h^n(s)\rangle$. When n reaches 0 a new secret and tuple must be generated. Security of the protocol relies on the hash to be a one-way function which is difficult to invert[4].

5.2 The Protocol

The authentication protocol we propose combines the standard challenge response authentication protocol with the one-time password approach. Instead of replying directly to the challenge, the response is hashed as many times as indicated by the index (which is also part of the challenge). This hash is sent to the verifier which compares the hash of the response with a locally stored tuple for this particular user. The secret s in the OTP scheme is derived from

[4] For a formal definition of one-way-ness we refer to [GB96].

the secret key k_i and the challenge c, by computing a MAC over c using k_i. Note that this way, unlike the simple challenge response authentication protocol, the challenge c is used repeatedly. By appending n to the challenge it is made fresh again.

The protocol appears, schematically, in Fig. 2. Each entity stores the following data.

- the prover (with identity i): stores diversified symmetric keys k_i and $k_{i'}$. We set $k_{i'} = \{i + a\}_{k_M}$, for a constant a such that for no i, j in the range of valid identities, $j = i + a$.
- the verifier: storing for each i a tuple $\langle \{c; V; n\}_{k_{i'}}, h^{n+1}([c; V]_{k_i}) \rangle$ — where c is the challenge, V is the identity of the verifier, and n is the current index (which is decremented with every authentication), and h is a publicly known one-way hash function, and
- the central and secure back office system with access to the master key k_M, from which k_i and $k_{i'}$ can be derived.

The tuple stored by the verifier can be thought to represent some sort of *authentication credit*, and is requested from and computed by the back office. This request includes the number of times N the verifier wishes to be able to authenticate the prover. Every time the credit has been used up (i.e. if n becomes 0), the verifier must refresh the authentication credit at the back office. The back office picks a new challenge c at random and computes $h^{N+1}([c]_{k_i})$ as well as $\{c; V; N\}_{k_{i'}}$ and sends both computed values back to the verifier. Note that c is generated afresh for every tuple request. Therefore c is different for each user i.

Instead of storing the challenge c and the index n in the clear, they are stored and sent encrypted with another secret key $k_{i'}$. Together with a response, the prover returns the next encrypted challenge $\{c; V; n - 1\}_{k_{i'}}$ This makes it impossible (both for the verifier and the adversary) to generate a valid challenge message containing different values for n or c. This serves two purposes:

- The verifier cannot increase the index and the credit to allow for more authentications. This is important if the verifier is charged for the number of authentications performed.
- The adversary cannot send a challenge like $c; 0$ to the prover to obtain the response $[c]_{k_i}$ from which a whole range of valid authentication responses can be derived.

Moreover, this encrypted challenge stores the identity of the verifier which is displayed to the user/prover before she is requested to agree to authenticate. Also, the name of the verifier is part of the hashed mac response. Therefore, responses are only valid for the intended verifiers.

The authentication protocol now runs as follows.

1. The verifier requests the identity from the prover.
2. The prover sends his identity i.
3. The verifier checks to see if it owns a tuple for i.

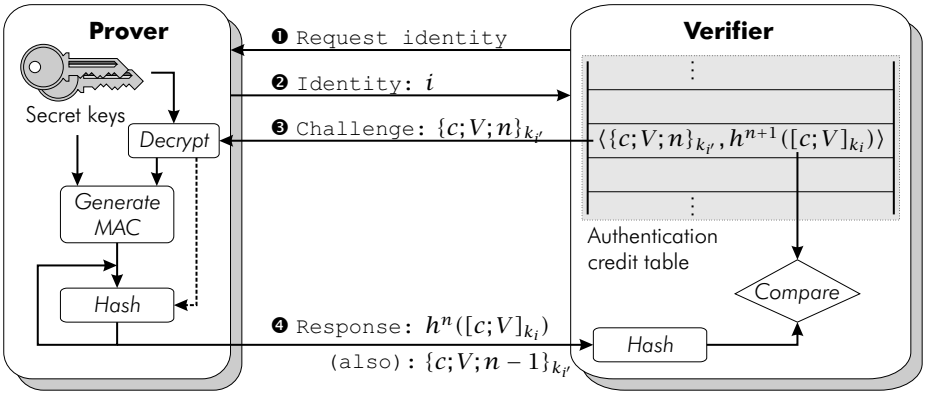

Figure 2. Authentication protocol using OTP

(a) If so, continue with step 4, unless the first item in the tuple equals `null` (indicating $n = 0$).
(b) Otherwise, authenticate the back office and set up an integrity preserving communication channel with it, using a random session key.
(c) Request a new tuple from the central back office sending the identity i, his own identity V and some constant N. N equals the number of times the verifier wishes to verify the identity of i without the need to go online. This is also equals the maximal number of times a prover will have to apply a hash function to compute a response.
(d) The central back office computes $k_i = \{i\}_{k_M}$, and $k_{i'} = \{i + a\}_{k_M}$, generates a random challenge c, and computes $h^{N+1}([c;V]_{k_i})$ as well as $\{c;V;N\}_{k_{i'}}$.
(e) The central back office sends $\{c;V;N\}_{k_{i'}}$ and $h^{N+1}([c;V]_{k_i})$ to the verifier which stores the tuple $\langle\{c;V;N\}_{k_{i'}}, h^{N+1}([c;V]_{k_i})\rangle$. Optionally, the back office may send additional identifying information to be stored in the tuple, for instance a provers name, address or the like.

4. The verifier selects the tuple and sends the encrypted challenge $\{c;V;n\}_{k_{i'}}$ the prover.
5. The prover decrypts the challenge — using his secret key $k_{i'}$ — to obtain c, V and n, and displays V to the user. The user/prover agrees to authenticate[5]. Otherwise the prover rejects the run and aborts.
6. Then the prover computes $[c;V]_{k_i}$ and hashes this n times to compute the response $r = h^n([c;V]_{k_i})$. The prover also computes $\{c;V;n-1\}_{k_{i'}}$, which is set to `null` if $n - 1 = 0$. Then r and $\{c;V;n-1\}_{k_{i'}}$ are sent to the verifier. All computations should take place inside the prover[6].

[5] E.g. unblocking the smart card with a PIN.
[6] In the case of a smart card based system, this approach requires the implementation of a smart card command `MAC-HASHED`(c,n), which accepts parameters in encryp-

7. The verifier receives r and checks whether $h(r)$ equals $h^{n+1}([c;V]_{k_i})$ stored in the tuple. If so, $r = h^n([c;V]_{k_i})$, which means authentication succeeded; the verifier accepts i and replaces the tuple for user i with $\langle \{c;V;n-1\}_{k_{i'}}, r\rangle$. If not, authentication failed, and the verifier rejects, and the tuple for user i is discarded.

As with the previous protocol, a potential problem with this protocol is the large database of tuples a verifier may have to store locally. Again, verifiers can save space if they can afford to go on-line more often, using a caching like strategy to only keep tuples for recent visitors and throwing away tuples that haven't been used for some time. Actually, the security of the system would improve if the random challenges were expired regularly by throwing away tuples more than a few days old.

5.3 Security

We split the proof of security in two parts. We first consider the authentication protocol between prover and verifier. Next we consider the reloading protocol to obtain a fresh tuple with authentication credit for a user i from the central back office.

Between Prover and Verifier. To prevent replay attacks, fresh challenges must be sent in each run of the protocol. Freshness of messages is guaranteed by the randomness of c generated by the back office and the fact that the index n is decremented with every authentication request. Hence an earlier response of the prover is never valid for the current run of the authentication protocol and will be rejected by the verifier.

Passive eavesdropping (including looking inside the verifier) may reveal tuples $\langle \{c;V;n\}_{k_{i'}}, h^{n+1}([c;V]_{k_i})\rangle$ for several values of c, V, n, k_i and $k_{i'}$. For tuples with the correct values for k_i and c, n will be larger than the current challenge. Hence if the adversary is able to calculate a valid response $h^n([c;V]_{k_i})$ on a fresh challenge c, n, this implies that the adversary is able to invert the one-way hash function h. This is impossible by assumption. Tuples with values for c and k_i incompatible with the current run do not contain information because $[c;V]_{k_i}$ and $[d;V]_{k_j}$ are completely unrelated by assumption.

Actively querying the prover is a more interesting case. To obtain a response $h^n([c;V]_{k_i})$ the adversary needs to know $\{c;V;n\}_{k_{i'}}$. This encrypted challenge is only available for the current (or next) challenge or any previous challenges

ted mode. Computing a MAC and hashing it n times is usually not implemented as a single function inside a smart card. As an alternative, and depending on the application and the appropriate trust model, the hashing can also take place inside the terminal into which the smart card is inserted. This terminal should be appropriately tested and certified not to leak information about the value to be hashed, and should not be identical with the verifier, so this situation only applies to 'remote' authentication over a computer network (e.g. Internet).

(that compute already known responses). However, intercepting $\{c;V;n\}_{k_{i'}}$, the adversary can obtain $h^n([c;V]_{k_i})$ *and* $\{c;V;n-1\}_{k_{i'}}$ which in turn can be used to obtain $h^{n-1}([c;V]_{k_i})$ *and* $\{c;V;n-2\}_{k_{i'}}$, etc. This allows the adversary to, in principle, sequentially obtain $h^0([c;V]_{k_i})$ without involvement or knowledge of the verifier. However, each time the prover is asked to supply a response, the user/prover must agree to authenticate to the indicated verifier V. This prevents the attack above, assuming the user is well aware of the fact that to authenticate agreement to do so has to be given only once, and only if the user wants to authenticate to the indicated verifier V. Depending on the trust model, this measure may be adequate.

Between Verifier and Back Office. To verify the responses from prover i, and to send new challenges, the verifier needs to obtain $\langle\{c;V;N\}_{k_{i'}}, h^{N+1}([c;V]_{k_i})\rangle$. from the back office. To ensure that only i can compute a response that is considered valid by the verifier V, we must guarantee that

- c is fresh (see previous section),
- the verifier receives the correct values $h^{N+1}([c;V]_{k_i})$ and $\{c;V;N\}_{k_{i'}}$ on request $i;N$.

Given the right values for V and $i;N$, $h^{N+1}([c;V]_{k_i})$ and $\{c;V;N\}_{k_{i'}}$ are correctly computed using a fresh c by the back office. To transfer these safely to the verifier requires authentication of the back office and an integrity preserving channel between the verifier and the authenticated back office. The back office does not have to verify the true identity V of the verifier; the user will do this when asked to agree to authenticate. Note that encrypting the channel is not necessary, because the channel only transports public information that the adversary is allowed to see already.

It remains to show that information obtained from the back office (either by eavesdropping or by purporting to be a verifier) cannot be used to break the protocol. Eavesdropping reveals no more information than looking inside the verifier; this case was already dealt with in the previous section. Asking authentication credit from the back office yields c' and $h^{N+1}([c';V]_{k_i})$ where c' is completely unrelated to any challenge c currently used or later to be used by any of the verifiers. Hence $h^{N+1}([c';V]_{k_i})$ gives no valid information to conduct a fake authentication, and cannot be used to extract valid authentication response from a prover.

6 Using one-time signatures

A drawback of the second protocol is that the adversary can (disregarding PIN protection for the moment) repeatedly start authentication runs with a prover to extract future responses.

At first sight, this appears to be a fundamental characteristic of software-only verification. Because the adversary is allowed to see data stored by the

verifier under attack, the adversary knows at least as much as the verifier does. Therefore, the adversary can do whatever the verifier can, and run fake authentication exchanges with a prover independently. If the protocol is deterministic (which the previous protocol is in between requests for new (randomised) authentication credit from the back office, the adversary can pre-compute valid authentication responses to future verifier challenges.

To prevent this, future challenges or responses should be random given the current state of both the prover and the verifier. Observer, however, that if only the prover randomises the future, the given scenario is not prevented. We conclude that the verifier must randomise its challenges. In this section we present such a protocol, based on Merkle's one-time signatures [Mer89] and the protocol of Wu and Sung [WS96].

The protocol is mainly of theoretical interest (the size of the signature or the number of hashes that need to be computed for a signature are quite large), but does show that querying the prover for future responses can be prevented in the software-only setting.

6.1 One-time signatures

Merkle's one-time signatures (including Winternitz optimisation) [Mer89] work as follows.

Fix a value z. This will limit the number of times a value will be hashed in the one-time signature algorithm. For simplicity assume that z is a power of two (and hence $^2\log z$ is an integer[7]).

Fix a constant k, that limits the maximal length of a message (including padding to be described below) to be signed to be $k \log z$ bits. Finally fix a one-way hash function h (with block length ℓ).

The prover generates a vector $\vec{r}$ of k random strings r_i of length ℓ each, computes the one-time verification key

$$V(\vec{r}) = h^z(r_1); \ldots ; h^z(r_k)$$

and sends this to the verifier.

To sign a message m, the message is first padded with the binary representation of the number of 0's in m. Let c be this number, and let $m' = m;c$, which is no longer than $k \log z$ bits. Split m' in k parts m'_i of $\log z$ bits each, and compute

$$S(m, \vec{r}) = h^{m_1}(r_1); \ldots ; h^{m_k}(r_k) \ .$$

This is the one-time signature of m (based on the secret $\vec{r}$).

To verify this signature $\vec{s} = S(m, \vec{r})$, the verifier

- computes m' given m, and splits m' into k parts m'_i of $\log z$ bits each,
- splits $\vec{s}$ into k strings s_i, each of length ℓ bits,
- splits $\vec{v} = V(\vec{r})$ also into k strings v_i, each of length ℓ bits, and
- verifies that for all i, $h^{z-m_i}(s_i) = v_i$.

[7] In the remainder of this paper, log will denote the logarithm base 2.

6.2 The Protocol

The basic idea of the protocol is the following. A verifier stores, for each user i, a tuple containing the public one-time verification key. This verification key is usually obtained (with a valid one-time signature) from the prover in each new authentication run. For a new user, the verification key is encrypted by the prover using his secret key, and verified by the backoffice using the master key.

For authentication, the verifier sends a random challenge to the prover. The prover computes a new random vector and a new corresponding one-time verification key and signs these together with the random challenge received from the verifier using the one-time signature scheme. The signature, together with the new verification key are sent back to the verifier. Using the old verification key the response is checked, and if valid, authentication is successful, and the new verification key replaces the old key in the tuple for user i.

Because the prover has only limited memory available, it cannot keep the secret vector $\vec{r}$. Therefore, this vector is kept by the verifier encrypted under the secret key of prover i, and is sent and renewed with every authentication run. This is similar to the scheme used by the protocol in Sect. 5.

Each entity stores the following data.

- the prover (with identity i): stores a diversified symmetric key k_i.
- the verifier: storing for each i a tuple $\langle \{\vec{r}\}_{k_i}, V(\vec{r})\rangle$.
- the central and secure back office system with access to the master key k_M, from which k_i can be derived.

The authentication protocol now runs as follows.

1. The verifier requests the identity from the prover.
2. The prover sends his identity i.
3. The verifier checks to see if it owns a tuple for i.
   (a) If so, continue with step 4.
   (b) Otherwise, request from the prover a new tuple, to be encrypted by the symmetric key k_i.
   (c) The prover sends $\{\langle i, \{\vec{r}\}_{k_i}, V(\vec{r})\rangle\}_{k_i}$.
   (d) Authenticate the back office and set up an integrity preserving communication channel with it, using a random session key. Forward the response from the prover to the backoffice.
   (e) The central back office computes $k_i = \{i\}_{k_M}$, decrypts $\{\langle i, \{\vec{r}\}_{k_i}, V(\vec{r})\rangle\}_{k_i}$, checks i and sends $\langle \{\vec{r}\}_{k_i}, V(\vec{r})\rangle$ back to the verifier.
4. The verifier selects the tuple, generates a random challenge c and sends $\{\vec{r}\}_{k_i}$ together with c to the prover.
5. The prover decrypts $\{\vec{r}\}_{k_i}$, generates a new random vector r', computes $V(\vec{r'})$ and $\{\vec{r'}\}_{k_i}$ and sends

$$V(\vec{r'}); \{\vec{r'}\}_{k_i}; S(h(c; V(\vec{r'}); \{\vec{r'}\}_{k_i}), \vec{r})$$

   to the verifier.

6. The verifier receives $\vec{v}' = V(\vec{r}')$, $\vec{e}' = \{\vec{r}'\}_{k_i}$ and $\vec{s} = S(h(c; V(\vec{r}'); \{\vec{r}'\}_{k_i}), \vec{r})$. It then verifies that $\vec{s}$ is a valid signature on $h(c; \vec{v}'; \vec{e}')$ using the value of c it generated in the challenge step and using the current value of $V(\vec{r})$ stored in the tuple. If so, authentication was successful, and the new tuple for i becomes $\langle \vec{e}', \vec{v}' \rangle$. If not, authentication failed and the tuple for user i is discarded.

Next we fix the size of the messages exchanged and the size of the one-time signature data. Let ℓ be the block length of h (e.g., 128 bits for MD5). The one-time signatures will be generated over the output of this hash function. Hence the length of the message over which a one-time signature must be generated equals $\ell + \log \ell$ bits (e.g., 135 bits in the case of MD5). Next fix z, the maximal number of times a block will be hashed to compute a signature. This also fixes the number of blocks $k = \lceil (\ell + \log \ell) / \log z \rceil$ and the size of the one-time signature $k\ell$. We see there is a tradeoff between the maximal total number of hashes $k(z-1)$ to compute a one-time signature and the size of the one-time signature $k\ell$. The total number of hashes is minimised for smallest possible z, i.e. $z = 2$. For MD5, we then get $z = 2$, $k = 135$, 135 maximal number of hashes, and $135 \times 128 = 17280$ bits signature length.

6.3 Security

We split the proof of security in two parts. We first consider the authentication protocol between prover and verifier. Next we consider the registration protocol to obtain a fresh tuple with a verification key for a user i with the aid of the central back office.

Between Prover and Verifier. To prevent replay attacks, fresh challenges must be sent in each run of the protocol. This is indeed the case. Moreover, the verification key changes after each authentication run, also rendering previous responses invalid.

Passive eavesdropping, including looking inside the verifier reveals i, c, $\{\vec{r}\}_{k_i}$, $V(\vec{r})$, and $S(h(c; V(\vec{r}'); \{\vec{r}'\}_{k_i}), \vec{r})$ for various values of $\vec{r}$. By assumption on the strength of DES and the one-wayness of h, neither $\{\vec{r}\}_{k_i}$ nor $V(\vec{r})$ reveals information on $\vec{r}$ and hence it is impossible to compute $S(h(c; x), \vec{r})$ for any value of x.

Actively querying the prover may reveal $\{\vec{r}'\}_{k_i}$ as well as $S(c; V(\vec{r}'); \{\vec{r}'\}_{k_i}, \vec{r})$ for sequences of $\vec{r}'$ and challenges c chosen by the adversary. However, only one signature $S(h(c; V(\vec{r}'); \{\vec{r}'\}_{k_i}), \vec{r})$ can be obtained for a particular $\vec{r}'$ and a c of choice. Later use of this signature is prevented by the fact that the verifier picks his own challenge c' which is unequal to c with very high probability (depending on the bitlength of the challenge).

Between Verifier and Back Office. To get a new tuple for user i, the verifier requests the encrypted tuple $\{\langle i, \{\vec{r}\}_{k_i}, V(\vec{r})\rangle\}_{k_i}$ from the prover and forwards

this to the backoffice. The backoffice decrypts this using k_i derived from k_M and checks i in the message. Only a principal knowing k_i can encrypt a message containing i using k_i. By assumption, this is only prover i (or the backoffice). Finally, the backoffice sends $\langle \{\vec{r}\}_{k_i}, V(\vec{r}) \rangle$ back to the verifier. Because the channel between verifier and backoffice is authenticated and integrity preserving, the verifier receives this tuple in good order and can be convinced this tuple corresponds to prover i.

6.4 Discussion

Wu and Sung [WS96] developed a similar protocol for authenticating passwords over an insecure channel. There are two problems with their protocol. First of all, their protocol requires the prover (viz. a smart card) to store a rather large amount of data for the generation of one-time signatures. Moreover, this data is different for each verifier to which the prover is registered. This conflicts with our requirement that the protocol should use little prover memory.

Secondly, the protocol of Wu and Sung does not protect against the active attack outlined above, where the adversary repeatedly queries a prover to obtain valid future authentication responses. In the first phase of their protocol they establish a new (authentic) one-time verification key which includes a value $Y_t = f^z(T)$ for some secret T picked by the prover. In the second phase, a challenge response protocol is run where

- the verifier challenges the prover with a random value c, for $0 < c < z$,
- the prover computes and responds with $T_c = f^c(T)$ using its secret T, and
- the verifier checks that $f^{z-c}(T_c)$ equals Y_t.

Unfortunately, an adversary can spoof to be a verifier, and repeatedly engage in authentication runs where he

- collects future one-time verification keys from the prover, and
- collects, for each one-time verification key, the response for challenge $c = 1$ (i.e. $f^1(T)$) which can be used later to fool the real verifier issuing an arbitrary challenge $c' \geq 1$.

7 Applications

The protocols can be used in several environments. For instance, using smart cards, they can be used to login on computer-terminals or grant access to buildings. In this case, the smart card is directly inserted into the verifying terminal (and there is no network, in the strict sense, between prover and verifier).

They can also be used for authentication over a distance, e.g. authentication at a web server. In this case the smart card is inserted in a terminal owned by the smart card owner, which is connected over the Internet to the server running the verification software. Here we consider the server to be the 'terminal' doing the verification; the terminal in which the smart card is inserted is transparent

and merely serves to connect the smart card to the network. Our protocols give a clear advantage over the traditional approach. Verifying the high volume of authentication responses using relatively slow SAMs proves to be a performance bottleneck. Faster Host Security Modules (HSMs) could be employed, but these are much more expensive.

Applications of the protocols is not restricted to smart card systems only. Currently, there is huge interest in using Personal Digital Assistants (PDAs) – like the Palm Pilot, the Visor or Windows CE based machines – as personal security tokens. Helme *et al.* [HSK99], for instance, use PDAs to support offline delegation of access rights to a file repository. By design, the delegation certificates should be small enough to be transmitted verbally (e.g. over the phone), in their case 256 bits. To achieve this, they use elliptic curve based public key cryptography, with the PDA storing the user's private key. Using our first two protocols, the keys and certificates could be made even smaller.

8 Conclusions & Further Research

We have presented three protocols for authentication using symmetric key cryptography by public verification, where the verifier uses public 'verification keys' and thus can safely be implemented entirely in software. The protocols require occasional connection to a central server to obtain a verification key or authentication credit, which must be stored for several users locally. The first two protocols are practical; the third protocol is mainly of theoretical interest.

Many applications do not necessarily require full authentication of an individual, but instead may require authentication on certain attributes (e.g. age, sex, etc.). More fine-grained protocols for this type of authentication need to be developed as well, as they enhance the privacy of the user by revealing no more information to the verifier than necessary for the application.

References

[AB97] Allen, C., and Barr, W. J. (Eds.). *Smart Cards: seizing strategic business opportunities.* McGraw-Hill, New York, 1997.

[AB96] Anderson, R. J., and Bezuidenhoudt, S. J. On the reliability of electronic payment systems. *IEEE Trans. on Softw. Eng.* **22**, 5 (1996), 294–301.

[ANS86] ANSI X9.19. *American National Standard - Financial institution retail message authentication.* ASC X9 Secretariat - American Bankers Association, 1986.

[BGH+92] Bird, R., Gopal, I., Herzberg, A., Janson, P., Kutten, S., Molva, R., and Yung, M. Systematic design of a family of attack-resistant authentication protocols. Tech. rep., IBM Raleigh, Watson & Zurich Laboratories, 1992.

[DM98] Dreifus, H., and Monk, J. T. *Smart Cards: A Guide to building and managing smart card applications.* J. Wiley, New York, 1998.

[FIP77] FIPS 46. Data encryption standard. Tech. Rep. NBS FIPS PUB 46, National Bureau of Standards, U.S. Department of Commerce, 1977.

[FIP95] FIPS180-1. Secure hash standard. Tech. Rep. NIST FIPS PUB 180-1, National Institute of Standards and Technology, U.S. Department of Commerce, 1995.

[GB96] GOLDWASSER, S., AND BELLARE, M. Lecture notes on cryptography. MIT lecture notes, 1996.

[HMNS98] HALLER, N., METZ, C., NESSER, P., AND STRAW, M. RFC 2289: A one-time password system, 1998.

[HSK99] HELME, A., AND STABELL-KULØ, T. Offline delegation. In *8th USENIX Sec. Symp.* (Washington, D.C., USA, 1999), USENIX, pp. ??–??

[KK99] KÖMMERLING, O., AND KUHN, M. G. Design principles for tamper-resistant smartcard processors. In *1st USENIX Worksh. on Smartcard Tech.* (Chicago, IL, 1999), USENIX, pp. 9–20.

[Lam81] LAMPORT, L. Password authentication with insecure communication. *Comm. ACM* **24**, 11 (1981), 770–772.

[Mer89] MERKLE, R. C. A certified digital signature. In *CRYPTO '89* (Santa Barbara, CA, USA, 1989), G. Brassard (Ed.), LNCS 435, Springer-Verlag, pp. 218–238.

[Riv92] RIVEST, R. RFC 1321: The MD5 message-digest algorithm, 1992.

[Sch96] SCHNEIER, B. *Applied Cryptography: Protocols, Algorithms and Source Code in C (2nd edition).* John Wiley & Sons, New York, 1996.

[WS96] WU, T.-C., AND SUNG, H.-S. Authenticating passwords over an insecure channel. *Comput. & Security* **15**, 5 (1996), 431–439.