USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

The following paper was originally published in the

USENIX Workshop on Smartcard Technology

Chicago, Illinois, USA, May 10–11, 1999

Mutual Authentication with Smart Cards

Bastiaan Bakker
Delft University of Technology

For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738
Email: office@usenix.org WWW: http://www.usenix.org

Mutual Authentication with Smart Cards

Bastiaan Bakker <Bastiaan.Bakker@Lifeline.nl>
Delft University of Technology, the Netherlands

Abstract

The World Wide Web has become the de facto interface for consumer oriented electronic commerce. So far the interaction between consumers and merchants is mostly limited to providing information about products and credit card based payments for mail orders. This is largely due to the lack of security currently available for commercial transactions. At the moment the only security mechanism present in most browsers is the Secure Socket Layer (SSL) which is limited to authentication and encryption of the HTTP session. It does not aim to secure transactions.

This report describes the design of a new three party authentication and key distribution protocol to serve as a foundation for WWW based transactions. Instead of having a radically new design it is derived from KryptoKnight protocol family developed at IBM. An important design consideration has been that it can be implemented with existing smart card technology. Specifically the Dutch Chipper and ChipKnip cards have been examined for their applicability. The result is an ABK(t) type protocol that runs with any card that supports either the ISO7816 *internal authenticate* command or the En726 *read stamped* or *protected read* instructions.

Secondly a prototype has been implemented in Java that can run in either the Java Development Kit or the Netscape or HotJava browser. Though Java was not designed for implementing hardware drivers it has proven perfectly suitable for communication with smart cards. Also it has effectively demonstrated its cross platform capabilities over multiple operating systems: except for a small native library to talk to the RS232 port the same code runs on Win32, Linux and the NCD network computer.

1. Introduction

The subject of this graduation assignment is to design and prototype a mutual authentication system for WAN-based Client/Server applications. Implementation of the system shall be based on common Internet technology. More specifically the applications are assumed to be WWW enabled, with a Java capable WWW browser on the client side and an HTTP server on the server side.

The system shall provide the basic elements for the initialization of a secure channel from client to server: authentication and (session) key negotiation. Properties of the channel it self, such as encryption, message integrity, and non-repudiation, etc., are beyond the scope of this assignment.

Some countries, notably the USA and France pose restrictions on the export of strong encryption technology. In order to accommodate unrestricted development in these countries the authentication system shall be built only with cryptography that may be exported legally.

Since the primary users of the system are consumers ease of use is very important. Consumer oriented systems for securing transactions that rely on soft keys instead of hardware tokens, such as the IPay system have proven to be difficult to use and administer. Hardware tokens on the other hand store the keys and the algorithms that may use them internally, safe from harm by careless users or hard disk crashes. Therefore this assignment will investigate the possibilities to use hardware tokens as security providers. Particularly smart cards will be examined because they are designed for this purpose. Moreover since the introduction of two nation wide electronic purse systems, the Chipper and the ChipKnip, by all Dutch consumer banks the majority of Dutch consumers owns one ore more smart cards.

For reasons of cost and availability it is attractive to be able to reuse these existing smart cards to authenticate clients and servers to each other. Typically the server is not the card issuer or owner of part of the card and therefore has no knowledge of the keys used by the smart card of the client. One option would be to install a security module containing all relevant keys at each server. This approach is used for the aforementioned ChipKnip and Chipper electronic purse implementations. This was motivated largely by the restriction that the purse transactions could be performed off line, without intervention of a central system. However in case of transactions on the Internet offline processing is not required. When considering the security and management issues generated by massive deployment of security modules, using a central security server is a preferable alternative. Consequently the authentication protocol features three parties: a client (the consumer), a server (the service provider) and a trusted third party (the smart card issuer, e.g. a bank).

The core of the system is a new cryptographic protocol that can be implemented with smart cards. It shall provide:

- Source authentication. The protocol should return an identifier of the peer party. This identifier may be persistent, such as an account or membership number or a transient, valid only for the duration of the session. In case of persistent IDs the trusted third party guarantees that the presented IDs belong to the communicating party. The protocol shall fail to complete if either party is not authenticated.
- Key negotiation. If authentication is successful, the protocol shall return a shared secret to both communicating parties. This secret can be used as a session key to provide message authentication, message integrity and optionally encryption.

1.1. Elements

Within an authentication system one can identify several elements. These include the users of the system of course, others are:

- Identificators. These are the objects the users (both clients and servers) use to prove their identity. They contain an identifier for their owner and a mechanism to authenticate this identity. In this assignment the application of smart cards as identificators will be investigated.
- A Trusted Third Party (TTP). The issuer of the identificators. It is trusted by both client and server parties, hence its name.
- Semi Trusted Computing Base. The to be secured application runs in this environment. The semi trusted computing base is trusted not to compromise the security of a single session of the application, but not trusted enough to protect long term secrets used for identification.

1.1.1. Requirements on the Identificators

In physical life people prove their identity with an identification paper, for example a passport. In 'computer life' we'll need a similar *identificator*.

Passports and alike contain three parts:

- An organization wide identifier: a social security number, a name, age, birthplace, birthday tuple, etc. This information is used within the 'organization' to uniquely refer to a person.
- Biometric identification information: photographs, age, height, eye color, etc. This data set should unique describe anyone bearing a passport and thereby link a passport to one unique individual.
- Physical authentication proofs: watermarks, uncopiable prints, special paper, etc. They should prove the passport is genuine and actually issued by the claimed authority (and thereby prove that this authority attests to the identity of the bearer).

So passports rely on biometric verification and the unfeasibility to physically forge them. Biometric verification relies on the trusted verificator (e.g. a customs officer) to be present at the same place as the identificator (passport): the biometric properties are public, their security lies in their unforgeability.
Since we are looking for a distributed protocol the peer party cannot know whether properties are actually measured or just pretended to be measured.

Challenge Response Identificators

Instead of relying on publicly known properties that may be forged, the identifying properties should be kept secret. You prove your identity by showing you know a unique secret that no other user knows, without disclosing it. A common method for this is a challenge/response protocol: party A sends a random number (the challenge) to party B, whose identity has to be verified. B returns a response calculated from the challenge and its unique secret key. The calculation has the property that it is infeasible to determine the key from the challenge response pairs. Nor is it possible to deduce new pairs from known ones. Verification of the response is only possible for parties knowing the secret used in the calculation. They can reproduce the challenge/response calculation and verify the response given by the authenticating party.

Physical, electrical and functional properties of smart cards are standardized in ISO standard 7816 ([ISO89], [ISO92] and [ISO93]). The majority of currently deployed cards implement 7816 parts 1 to 3 and most application level commands in part 4. The latter part includes some standardized commands to perform challenge/response calculations. The actual choice and implementation of a particular algorithm is left to the card manufacturer.
Minimally we require the card to contain an identifier and a key that are unique for the owner of the card and maintained by the Trusted Third Party (TTP). Examples of the identifiers are (bank) account numbers, social security numbers, membership numbers, etc.
Secondly we require the card to offer some challenge response mechanism using the unique key that can prove the validity of the ID supplied by the card.

1.1.2. Requirements on the Trusted Third Party

The Trusted Third Party is the organization issuing the smart cards or at least controlling the part of the card used for the protocol in case of Multi Function Cards

(MFCs). It has knowledge of the persistent keys used in the protocol. As the name suggests, the client and server trust the TTP to provide them accurate information. Also they trust the TTP to issue identificators that have sufficient security provisions to keep the persistent keys secret.

1.1.3. Requirements on the Semi Trusted Computing Base

A user may consider his or her computer and web browser to be a semi trusted computing base. People trust their computer and browser to communicate with the service provider, that is they trust it not to compromise the communication, for example to relay it to another party, to manipulate it, etc.

However Internet enabled computers are compromised on a daily basis all over the world, so one cannot rule out the possibility a hacker gains control over the web browser or even the whole computer. An important feature of the authentication system has to be that in such a case compromise will be incidental and not total: once the security breach in the semi trusted base has been fixed compromise should no longer be possible. Note that this is different from a breach in the security of the trusted computing base: once a smart card has leaked its secret nothing can be done to restore the security of that card.

1.2. Standardized Cryptographic Features for Smart Cards

Both the ISO 7816 and En726 standards only include commands that use secret key cryptography. Consequently their authentication mechanisms require that both parties (the card and the application) know the authentication key.

ISO 7816 specifies a single command by which a card can authenticate it self to the outside world: the *internal authenticate* command. This is a straightforward challenge/response mechanism with challenges and responses of typically eight bytes (this is not mandatory though). Commonly this command is implemented with DES encryption in Electronic CodeBook (ECB) mode. Conversely *the external authenticate* command allows the outside world to authenticate it self to the card.

ISO 7816 also specifies mechanisms for providing data integrity and confidentiality, labeled Secure Messaging. These do not seem to be implemented widely though, so they will not be discussed here.

Additionally the En726 standard offers methods for authentication and concealment of data on the card. Elementary files can be given access conditions, which specify whether and how the files may be read and/or updated. Among the conditions are “ALW” (for always), “NEV” (for never), “PRO” (for protected) and “ENC” (for enciphered). The “PRO” access condition mandates that a cryptogram shall be sent along with the data read from or written to the card. This cryptogram is the result of a keyed MAC over the data and a challenge. The “ENC” access condition is an extension of the “PRO” condition. It mandates that all data shall be enciphered as well as protected by a cryptogram.

1.3. Key Management of Some Smart Card Systems

Electronic purses like Chipper and ChipKnip are designed for off line use: the customer should be able to pay the merchant without the need for a connection to a central server of a bank. This means only two parties are involved: just the card of the customer and the Payment Terminal System (PTS). Generally the part of the terminal that provides the security (contains all keys, etc) called the Secure Application Module (SAM), is a smart card as well.

Secret key based authentication requires both parties to share a common secret. The simplest method to arrange this is to have a single organization wide key stored in all smart cards. In closed systems deployed within a single organization, authentication with a single key is still commonly used. An example is the Closed Electronic Purse on the Studenten ChipKaart of 1995/96 [Hoekstra97].

For larger systems such as a nation wide intersector electronic purse a single global authentication key is not a feasible solution: breaking the security of a single smart card, that is obtaining the key, compromises the security of the entire purse application. Failure of a single entity in the system results in failure of the security of the entire system.

To remedy this electronic purses like Chipper and ChipKnip feature a unique authentication key per purse card. Obtaining the key of one purse card does not help you to forge other purse cards. After a couple of fraudulent purchases with a forged version of the cracked card the bank will notice the fraud (because more money was ‘downloaded’ from the card than was previously uploaded to it) and black list the card. The solution is only partial however: since every Payment Terminal System (PTS) should be able to authenticate every purse it should know all authentication keys. It is infeasible to store every key in a big database located in the PTS; we would be talking about something like 5 million entries replicated in at least 50.000 databases. Instead the authentication key of a purse is determined by key transformation: the bank has a single master key that serves as a key encrypting key or key generating key over the ID number of the card: $K_{purse} = E_{Km}(ID_{purse})$ or $K_{purse} = MAC_{Km}(ID_{purse})$. Now the SAM only needs

to know the master key from which it can deduce the purse key for every purse it has to authenticate. The purse itself contains only its own K_{purse}. Of course if this master key extracted from a SAM somehow, the entire system still is compromised completely.

2. A New Protocol for Authentication & Key Distribution

The KryptoKnight protocol family provides three party based authentication and key distribution built on exportable symmetric cryptography. It has most of the properties we desire. Nevertheless it has not specifically been designed for implementation with existing smart cards. For example it assumes party A and B know their respective keys K_A and K_B. For smart cards this is not true: neither smart card users nor applications are allowed to know the keys stored in the card. The operating system of the card only permits applications to initiate certain commands that use the key, but these are all executed within the secured environment of the card itself.

This chapter describes the design of a three party authentication and key distribution protocol suitable for smart cards, based on the KryptoKnight protocol designs. More specifically the ABK variant of the KryptoKnight protocol will be adapted to a smart card compatible ABK(t) type protocol, named KLOMP (KryptoKnight-based Lightweight Open Mutual authentication Protocol).

2.1. Assumptions and Constraints

1. Only the server shall contact the trusted third party.
2. Assumptions about the identificators used by the client and the server shall be minimized.
3. The protocol shall be compatible with contemporary smart card technology.
4. The protocol shall be suitable for a Wide Area Network (WAN) environment.
5. All cryptographic algorithms shall be legally exportable (out of the United State and other countries that put restrictions on export of cryptography).
6. The protocol shall be stateless with respect to the trusted third party.
7. The trusted party shall not need to give any response before successful authentication of both client and server. Specifically it shall not disclose any information about the client to the server before both client and server are authenticated and the client has given its consent.

Ad 1) In a typical Internet based client/server system the number of clients is much larger than the number of servers. Furthermore the relation between a trusted party and the application servers is more static than between the trusted party and the clients. Therefore both from a security and a topological viewpoint it is wise to limit access to the trusted party to the application servers.

Ad 2) Minimizing the assumptions about identificators maximizes the reusability of the protocol.

Ad 3) The protocol is specifically aimed at letting smart cards provide the security for it. It shall at least be implementable with ISO7816 or En726 compliant cards.

Ad 4) The protocol is aimed at use in an Internet environment, which certainly is a Wide Area Network.

Ad 5) Authentication and key distribution by them selves are exportable functions. In many cases they suffice and confidentiality by encryption is not needed for the subsequent transactions. With careful design it is possible to build a system that is both secure and does not rely on export restricted cryptography

Ad 6) A stateless protocol considerably reduces the amount of administration to be kept by the trusted party. Keeping the server of the trusted party simple is important:

- compared to the other parties, the authentication server has to handle many requests: whereas the client has to perform only one authentication, the application server has to perform as many authentications as there are clients connecting and the trusted third party has to serve requests from all the applications servers it serves.
- a simple server is more robust than a complex one.
- a simple server is more likely to be secure than a complex one.

Ad 7) To ensure the privacy of users any information about them should be disclosed (to the application server) only after the user agrees to it, which means that both the user and the server should be authenticated first. Also for security it better not to have to send encrypted messages (tickets) to unauthenticated principals: it allows attackers to collect cipher texts which might help breaking the cryptographic protocols.

2.2. Generating MACs with smart cards

The KryptoKnight protocol needs to calculate authentication codes over messages longer than 32 bytes. Smart cards may not support MAC generation over messages of that length. They often offer only limited support for generation of authentication codes over messages. Some methods are:

- ISO7816 Internal Authenticate
- En726 Read Stamped / Protected Read of freely writeable field
- En726 Read Stamped / Protected Read of read only field
- CBC-MAC based of one of the above methods.

2.2.1. ISO7816 Internal Authenticate command

The ISO7816 Internal Authenticate command returns a keyed hash of a short message (the challenge) in order to authenticate the card to the outside world. This keyed hash can also be used to generate an authentication code over an arbitrary message. Since the challenge that the Internal Authenticate command accepts must have a limited length (typically 8 bytes), the message first has to be compressed using a secure hash algorithm. The compressed message is then fed to the Internal Authenticate command. In order to ensure freshness of the resulting Message Authentication Code, a random challenge number should be included in the message before compression. To put it in a formula:
$MAC_K(M) = InternAuth_K(H(R // M))$, where M is the message, K the authentication key and R a random number to ensure freshness.
Although this method does yield a MAC generating algorithm, the compression of the message before the Internal Authenticate command does weaken it. For example in order to find two messages that collide (generate the same MAC) an attacker does not need to have access to the Internal Authenticate command. She only needs to find collisions in the secure hash algorithm. Any collisions found will then occur with any smart card regardless of the key.

2.2.2. En726 Read Stamped of Freely Writeable Field

The smart card supports the En726 *Read Stamped* or *Protected Read* methods and has a freely writeable file that can be read with (one of) these methods. This offers the best way to perform MAC calculation on the card: first write the message on the card and then read it back with the *Read Stamped* or *Protected Read* method. The only limitation is that the message has to fit into the file.

2.2.3. En726 Read Stamped of Read Only Field

In this case the smart card supports the *Read Stamped* or *Protected Read* methods but none of the files on the card is freely writeable. This gives the same opportunities as the *Internal Authenticate* command: an 8-byte challenge that yields an 8-byte response. The only difference is that the contents of the file have to be sent to the verificator along with the response or else the verificator will not be able to reproduce the MAC calculation.

2.2.4. CBC-MAC

All of the above methods use an authentication code generating command only once. Depending on the speed of the card and the time constraints on the protocol this may be the maximum as well. But if one is permitted multiple invocations, either of the methods mentioned above can be used to build a CBC MAC, analogous to the ANSI X9.9 keyed MAC algorithm.
The Read Stamped over Writeable Field and the CBC-MAC method generate MACs over the entire message, the others only over a compressed representation of the message, which is less secure, so the first are preferred. But since they might not always be available or feasible the protocol will be designed to work with the other two methods as well.

2.3. Challenges, responses and authentication proofs

From the previous paragraph follows that minimally available is a challenge/response mechanism that accepts fixed length short challenges (of 8 bytes in most cases) and returns responses of the same length. In other words where KryptoKnight calculates MACs with K_A or K_B on complete messages, the new protocol first has to compress these messages. With a secure hash function the message can be mapped to a fixed length challenge that appears pseudo random. For example let $C_A = H(N_A, N_B, B)$ and $C_B = H(N_A, N_B, A)$.
However this introduces a possible weakness: the space of possible challenges is much smaller. When the space is small enough it might be feasible for an attacker that has already has collected correct challenge/response pairs to try to find a N_A (or N_B) that yields a known challenge by brute force. For example if the challenge is 64 bits big and the attacker has collected 2^{16} challenge/response pairs, every one in 2^{48} nonces yields a challenge for which the response is known. With the currently available computing resources this is feasible by 'brute force'. This attack is possible because the search for a suitable nonce can be performed off line. In order to limit this off line search the protocol may impose a time window outside of which the challenge is not valid. A window that is small enough renders a brute force attack infeasible: once a suitable challenge has been found the transaction has already expired. The challenges become:
$C_A = H(N_A, N_B, T, B)$ and $C_B = H(N_A, N_B, T, A)$, where T is a time stamp.

2.4. Order and Direction of the Data Flow

The order in which the three parties communicate is largely dictated by the constraints put on the protocol: the limitation that the protocol should be stateless with respect to the TTP (constraint 6) implies that it shall be contacted only once during a protocol run. The contactor shall be the server (see constraint 1). So the protocol flow includes B→K→B.
Since A has to be notified about success or failure of the protocol B subsequently sends a message to A. And since the ID of A has to be known before contacting the protocol flow has to be at least A→B→K→B→A. From constraint 7 follows that both the client and server shall generate their authentication proofs before contacting the TTP. This in turn implies that the challenges the client and server identificator have to respond to cannot include a nonce from the TTP. Instead it has to be substituted by a time stamp. In order to establish challenges that contain nonces of both A and B and the time stamp, the protocol flow has to start with a message from B to A that contains B's ID and its nonce N_B. If the protocol will be initiated by A it has to send a (possibly empty) 'trigger' message to B before that. This results in the following protocol flow: A→B→A→B→K→B→A.

All in all the desired protocol can be called an ABK(t) protocol, using the naming convention of the KryptoKnight documentation. (The t after the K for the KDC = TTP, indicates the substitution of a timestamp for a nonce from the TTP.) Unfortunately the KryptoKnight family does not include a protocol of this type, so a new one has to be developed.

2.5. Session Key Distribution

The method of key distribution is also influenced by the constraints on the protocol. The KryptoKnight protocols all let the TTP generate the session key and distribute it with tickets to both the client (A) and server (B). As discussed before this has the drawback that an attacker may collect any number of tickets for an attack on the encryption algorithm without any authentication.
In the new algorithm a different distribution mechanism is proposed: party A generates the session key, transports it encrypted (via party B) to the TTP, which decrypts the key and finally sends it to party B reencrypted with a key known to B. So in this case the session key is transported (from A to B) rather than distributed (from the TTP to A and B).
A and B generate cryptographic proofs before contacting the trusted third party. In the same effort the session key can be generated as well: party A generates session key K_{AB} with the same parameters as proof P_{AK} it sends to the TTP. Since these parameters are either publicly sent to the TTP or known by the TTP (in case of the key transforming key K_A), the TTP can reproduce the session key without further knowledge.

2.6. Protocol Description

The considerations in the previous paragraphs lead to the protocol flows depicted in figure 1.

Step 1 might seem strange since it does not include any protocol data. Its purpose is to trigger the start of the authentication protocol and need not be explicitly performed: the protocol might be triggered implicitly when the client requests a secured action within the application protocol.

In step 2 the server replies with an identifier for itself (B), a nonce (N_B) and a time stamp (T). T substitutes for a nonce N_K of the TTP because the TTP may contacted only once, after the challenge/responses have been performed at the client and server. The time stamp does not need to be very precise: it sets but a time window in which the authentication session has to be performed. It is checked by the TTP, so the clocks of the server and the TTP must have a time skew less than the allowed time window. A time window in the order of 5 to 10 minutes seems reasonable: it is small enough not to compromise the security of the protocol and wide enough to avoid the difficulties of tightly synchronized the two clocks.

Steps 3 and 4 are split up in three parts: In step 3a the client calculates a challenge C_A to send to the identificator. This challenge should be a secure hash of N_A, N_B, T and B, where N_A is a nonce generated by the client. Proposed is $C_A = MAC_T(N_A // N_B // B)$.
In step 3b the client collects the data from the identificator: the ID of the client (A), the type of the identificator (I_A), the response on the challenge (K_{A*}), and optionally any extra data used in the calculation of the response (D_A). Rather than having the TTP generate and distribute the session key, response K_{A*} will be used as a temporary key for the generation of session key K_{AB}. The choice for key generating function *f()* depends on whether party B is allowed to learn challenge/response pairs of A's smart card. In that case *f(x)=x* will suffice else an (optionally salted) hash function has to be used. The authentication proof P_{AK} is calculated from A, N_A, N_B and K_{AB} using formula $P_{AK} = MAC_{AB}(N_A // N_B // A)$. Using K_{AB} instead of K_{A*} makes the proof from A for the TTP identical to the proof from A for B: $P_{AK} = P_{BK}$.
In step 3c the client sends all parameters (A, N_A, I_A, D_A, P_{AK}, B, N_B, T) to party B.

The server (party B) similarly generates a challenge (C_B) and retrieves the corresponding response (K_{B*}) from its smart card. However the server does not need to generate a session key nor should its authentication proof be verifiable by the client. Therefore it can directly apply response K_{B*} in its calculation of P_{BK}. Since the server does not know K_{AB} yet it cannot verify P_{AB}. Instead the server sends all parameters (A, N_A, I_A, D_A, P_{AK}, B, N_B, I_B, D_B, P_{BK}, T) to the trusted third party.

In step 5 the TTP verifies all parameters received from B:

- ❄ is time stamp T valid?
- ❄ is A a valid and active client ID?
- ❄ is B a valid and active server ID?
- ❄ corresponds proof value P_{AK} to the value yielded with genuine key of the client K_A?
- ❄ corresponds proof value P_{BK} to the value yielded with genuine key of the client K_B?

In case the authentication request passed verification, the TTP calculates the session key and encrypts it for the server using key encrypting function *g()*: T_{BA} = *g(*K_{B*}, K_{AB}, ...*)*. if A may learn K_{B*} the function may be very simple again: *g(*x, y, ...*)* = x ⊕ y. The TTP sends back the result, the ticket T_{BA}, to the server B.

In step 6 the server decrypts the session key K_{AB}. Now B is able to verify proof P_{AK}, because P_{AK} is calculated from secret K_{AB} and the public values N_A, N_B and A. If the verification either client A or the trusted third party are imposters or the communication has been tampered with. If verification is successful B generates proof P_{BA} for client A and sends it to A. Finally A verifies proof P_{BA}.

2.7. Improvements on the protocol

The basic KLOMP protocol has been designed to be implementable with any challenge/response capable identificator. With some identificators the protocol can be enhanced for better security. At the cost of some extra calculations also some potential weak spots can be eliminated.

2.7.1. Collection of challenge / response pairs

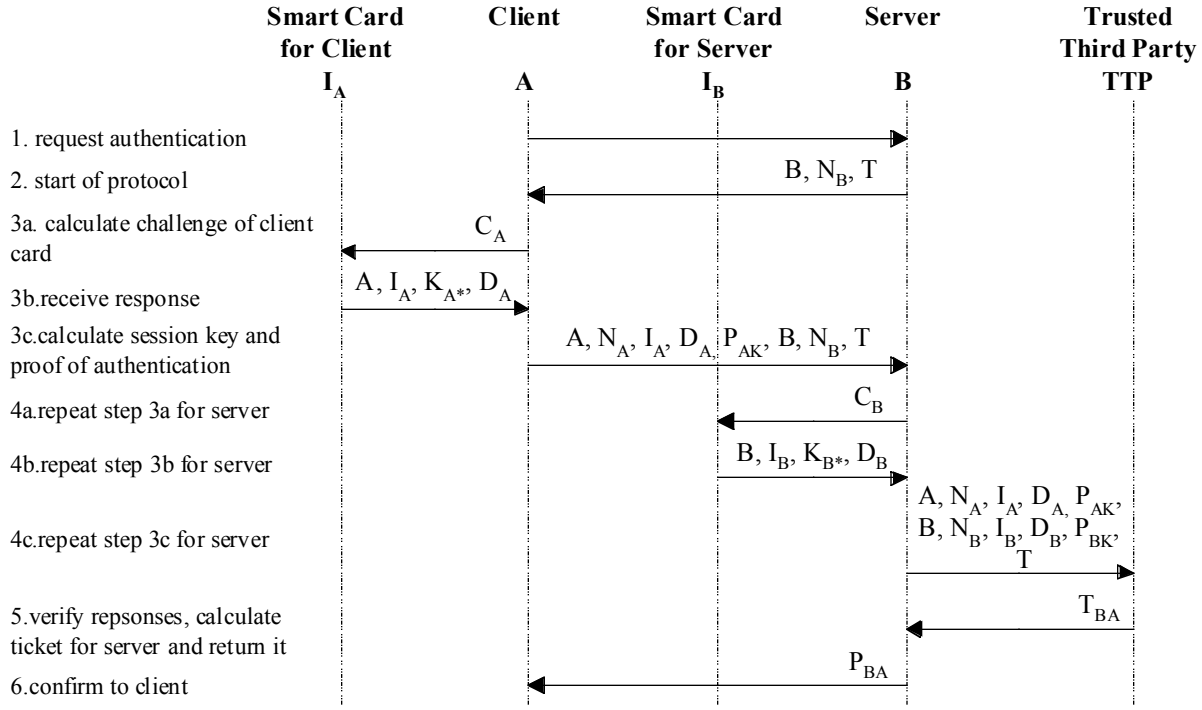

Figure 1: Data flow of the KLOMP protocol

$C_A = H(N_A, N_B, T, B)$ $\quad$ $C_B = H(N_A, N_B, T, A)$

$K_{A*} = MAC_A(C_A, D_A)$ $\quad$ $K_{B*} = MAC_B(C_B, D_B)$

$K_{AB} = f(K_{A*}, ...)$

$P_{AK} = P_{AB} = MAC_{AB}(N_A // N_B // A)$ $\quad$ $P_{BK} = MAC_{B*}(N_A // N_B // B)$

$T_{BA} = g(K_{B*}, K_{AB}, ...)$ $\quad$ $P_{BA} = MAC_{AB}(N_A // N_B)$

The current protocol allows servers to collect challenge/response pairs of the identificator of the client. Conversely clients can collect pairs of the identificator of the server if they eavesdrop on the connection between the server and the trusted third party. The reason for this weakness is obviously the simplicity of the key encrypting and key generating function applied in the protocol. To remedy this let:
$K_{AB} = H(K_{A*} // N_A)$ and $T_{BA} = H(K_{B*} // N_B) \oplus K_{AB}$

2.7.2. Clients searching for suitable C_A's

If a rogue client knows some challenge/response pairs of the client identificator it may try to find a challenge he knows by trying different client nonces (N_A's). Of course the search must be completed before the time stamp T expires. In any case this attack can be prevented if the client already has to commit to a specific N_A in step 1. A way to accomplish this is to let the client send a concealed commitment to the server in step 1: it has to send a verification parameter $V_A=H(N_A, S_A)$. Here S_A denotes the *salt* used to randomize the hash, analogous to the salt in encrypted UNIX passwords. The salt is a public value, meaning that the client sends it together with V_A to the server. The inclusion of the salt in the hash increases the input address space of the hash beyond the point where the server can deduce N_A through a dictionary attack. So the secure hash and the salt prevent the server to deduce the challenge of the client before committing to the value of its own nonce N_B. Yet in step 3c the server can verify that the client already has chosen its nonce before it knew the server nonce, thereby eliminating the search for a suitable nonce.

2.7.3. Time Window of Session Key Compromise

The session key K_{AB} may be used not only to protect the integrity and origin of transactions between A and B but also to encrypt their communication. If the information exchanged between A and B has to stay confidential even after the session has ended, K_{AB} must remain secret as well. Since K_{AB} is calculated solely from long term key K_A and data susceptible to eavesdropping, compromise of K_A will enable an attacker to recalculate all session keys, no matter how long ago the corresponding sessions took place (provided that he recorded those sessions of course). The same applies to the compromise of K_B, since it forms the basis for the transport key for K_{AB}. In case of key distribution protocols implemented completely in software, like KryptoKnight, the best we can hope for is that K_A nor K_B are ever revealed to other parties: there is no solution for this problem.

However with hardware security tokens like smart cards the situation may be much better. Compromise can be divided in two levels now:

1. An attacker obtains access to the smart card and particularly to the challenge/response generating command: this means that the attacker can obtain responses over arbitrary challenges.
2. An attacker discovers the key (K_A or K_B) used by the challenge/response algorithm on the card.

The second level equals the compromise of the key in an implementation in software only: the attacker can recover any session key. Luckily level 2 compromises are very unlikely to occur: smart cards are specifically designed to withstand any attempt (both logically and physically) to illegitimately retrieve the keys they contain.
Level 1 compromises are far more likely to happen, in fact any stolen or lost card may be abused for it. Luckily these compromises are also less severe. The key authentication and distribution protocol does not take advantage of the different properties of level 1 and level 2 compromises: in both cases all session keys can be recovered.
Depending on the implementation of the challenge/response mechanism on the smart cards, the vulnerability of old session keys may be eliminated for level 1 attacks. The challenge/response mechanism has to be based on a reversible function, such as DES rather than on a keyed one way function. The ANSI X9.9 MAC generation algorithm used in the IBM Multi Function Chipcards is an example. It essentially is DES run in CBC mode.
This algorithm has an 'inverse', that calculates a challenge from a message, a key and the corresponding Message authentication Code. In other words: let M be a message, K be a key, C be a challenge and R be the MAC of M under key K and challenge C (so $R = MAC_K(C, M)$) then $C = MAC^{-1}_K (M, R)$. With an inverse MAC one can take advantage of the fact that the

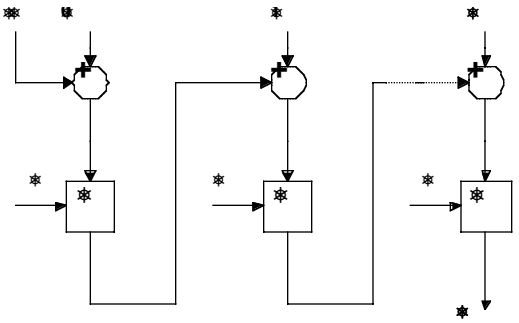

Figure 2: ANSI X9.9 MAC algorithm

TTP has full access to the keys in the identificators, but users of the identificators do not: a user can encrypt a short message by simply having the identificator perform a MAC calculation with the message used as challenge. The TTP decrypts the message by applying the inverse MAC. The holder of the identificator cannot, because the identificator restricts the application of the key it holds to the forward MAC calculation.

To strengthen the authentication protocol with reversible MAC's the following formulas have to be used for the notarization keys K_{A*} and K_{B*}:

$K_{A*} = MAC_A(C_A, D_A) + MAC^{-1}{}_A(N_A, D_A) = MAC_A(C_A, D_A) \oplus R_A$

$K_{B*} = MAC_B(C_B, D_B) + MAC^{-1}{}_B(N_B, D_B) = MAC_B(C_B, D_B) \oplus R_B$

where $N_A = MAC_A(R_A, D_A)$ and $N_B = MAC_B(R_B, D_B)$

Now the notarization keys are offset with the randomly chosen values R_A and R_B respectively. The TTP can decode these random values because they are encoded in the nonces N_A and N_B with a MAC calculation. Since a smart card does not offer a method to derive R_A or R_B from N_A and N_B an attacker cannot calculate the notarization keys and consequently the session key.

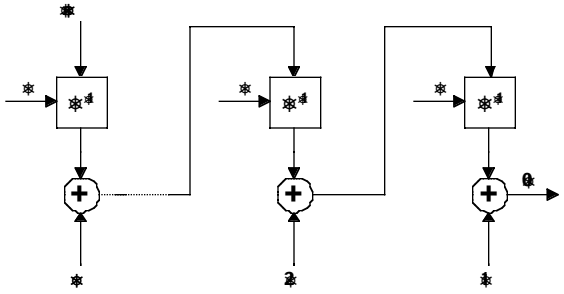

Figure 3: 'inverse' of X9.9 MAC

The TTP, on the other may derive the random values by applying the inverse MAC algorithm.

3. Prototype Implementation

In order to test the applicability and usability of the authentication protocol in actual WWW browser environments a prototype implementation has been built. Minimal goals for the implementation were Java 1.1 compatibility with support for at least one RS232 connected smart card reader and a commonly available smart card. For the latter, the ZeelandKaart and the Chipper, both incarnations of the IBM Multi Function Card were chosen.

The prototype implementation has been split into two parts: an implementation of the authentication and key transportation protocol and an implementation of a smart card access library. The intention was to be able to plug in specific identificators in the authentication protocol, including ones not based on smart cards.

Flexibility was an important design consideration. Adding support for new smart card interfaces or smart cards has to be easy. Also extending the protocol for extra security or new features must not break compatibility.

3.1. Smart Card Interfacing

At the start of this project no Java API for accessing smart cards was available so one was developed as part of the project. The Smart Card API is layered analogous to the ISO 7816 standards:

- Hardware layer: provides uniform access to different smart card interface devices.
- Datalink layer: implements the T=0, T=1, etc datalink protocols described in ISO 7816-3
- Transport layer: provides multiple concurrent smart card communication channels for datalink layers that support it (T=1).
- Application layer: implements the application commands standardized in ISO 7816-4 and prEN726-3

In order to maximize portability the bridge between native code and Java was put in the hardware layer. This meant the development of a native RS232 port access library that can be linked by the Java Runtime through the Java Native Interface (JNI). This library has been implemented for the Win32 and Linux platform with the JDK 1.0, JDK 1.1 and Netscape 3 & 4 Java Runtime Environments. Support for the NCD Explora network computer was added in pure Java.

On top of the RS232 layer drivers have been built for the Towitoko ChipDrive (see [Towitoko97]) and the DumbMouse smart card interface [BillSF96].

For the complete API and JavaDoc documentation, see [Bakker98].

3.2. Smart Card based Identificators

As part of this investigation several smart cards have been examined for their applicability as identificators in the authentication protocol: the Studenten Chip Kaart (SCK) 1995/96, the Zeeland Kaart, the ChipKnip, the Postbank Chipper and the Studenten Chip Kaart 1997/98 supplied to students at the Technical University of Delft.

Except for the ChipKnip, which is a Bull CP8 Transac CC 60 payment card, all cards are based on the IBM Multi Function Card OS that implements (a subset of) the ETSI En726 smart card application layer. (All cards offer an ISO7816-4 command set, the ChipKnip over datalink layer T=0 and the others over T=1).

The ChipKnip turned out not to be compatible with KLOMP since it provides authentication only within the *proof of debit* step of a payment transaction, not separately [BeaNet96]. The other cards provide either a 'protected read' or En726 'read stamped' method, which both include ANSI X9.9 MACs. With these methods improved KLOMP can be implemented. Experiments proved the MAC implementation on the Zeeland Kaart to be insecure however.

3.3. Password based Identificators

For testing purposes also a password based challenge/response identificator has been built. It can be considered a software emulation of a smart card that implements the ISO7816 internal authenticate command. The password-based identificator simply asks the user for a user ID and password to perform MAC calculations with. In this case the cryptographic functions are processed in the semi trusted computing base instead of in a trusted computing base. Of course the consequence is that compromise of the semi trusted computing base fully voids the security of the identificator.

3.4. Protocol Implementation

A prototype of KLOMP using password-based identificators has been written for the Java 1.1 platform. Using smart card based identificators was not possible, because no smart cards were available of which the keys were known. Figure 4 shows a global class diagram of the prototype. At the TTP the identificator factory builds identificators by looking up users and passwords in a PostgreSQL connected through JDBC. The ClientAuthenticator, the ServerAuthenticator and TrustedThirdParty communicate through Java Remote Method Invocation (RMI). Java 1.1 includes a class for generation of MD5 message digests, so this algorithm has been applied for all secure hashes. The prototype is available at [Bakker98-2].

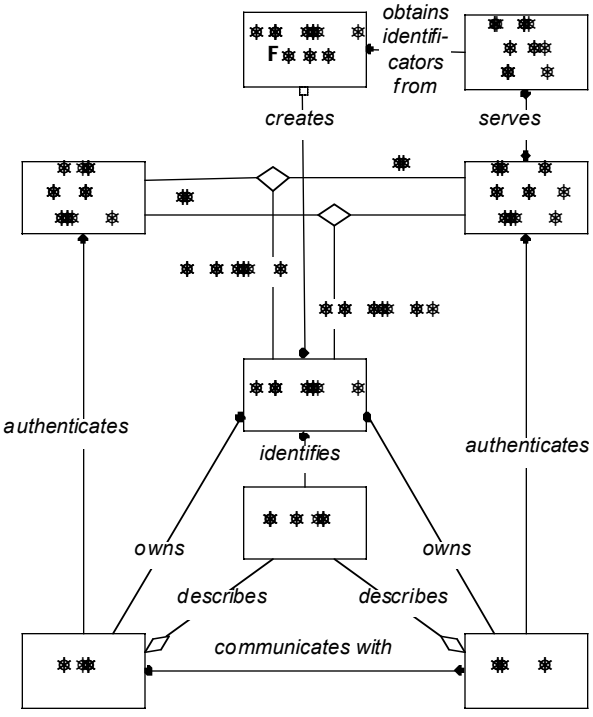

Figure 4: Class Diagram of the Protocol Implementation

3.5. Running the Prototype

Since the prototype protocol implementation is written in pure Java, so it should run with any Java 1.1 compliant runtime. (The prototype does connect to a pure Java UserdID/Password identificator by default, therefore it does not need a native RS232 access library). The server also needs the Java Generic Library (version 3.0) and the TrustedThirdParty needs a JDBC driver, in this case one for PostgreSQL. A PostgreSQL database containing the user table has to be accessible for the TTP.

A small client has been written that repeatedly initiates the authentication protocol and times the duration for completion. The measurements of a typical run of this applet are shown below:

Pass	1	2	3	4
Time	5.542s	0.314s	0.337s	0.432s

Here the applet was running the protocol on a Pentium II 233 with Red Hat Linux 4.0 and JDK 1.1.1v2 generated the above log. The first run of the protocol takes clearly much longer than the other ones. This is due to the initialization of the RMI channels between the parties: it causes loading of many Java RMI classes into the runtime. Also the Java serialization mechanism has to perform elaborate hash calculations for every class serialized. These timings should not be considered an indication of the speed of the protocol in a typical application environment: the JDK 1.1.1 for Linux is much slower than the average Java implementation, for example it does not contain a Just In Time (JIT) compiler. The above test runs were performed with all three processes (client, server and TTP process) and the PostgreSQL database on the same machine. Several other test runs have been performed with these parts running on separate machines running different operating systems. Besides Linux the system was tested on Sparc Solaris 2.5.1 and WindowsNT 4.0. Other than speed no functional differences were observed with the different configurations. Also the PostgreSQL JDBC driver did not have any problems in accessing

the database remotely. All in all the binary code portability of the Java prototype implementation scored 100%.

3.6. Application of the Smart Card API

Separately from the protocol prototype, the smart card access code has been applied in a project for the Landelijk Instituut voor Sociale Verzekeringen (LISV). In this demonstration system people can register and login to the HTTP based application by entering their Chipper or ChipKnip card. A Java applet reads the account number and a password from the card and sends it to the web server for simple password based authentication. The applet takes less than half a second to read the card and send the logon request. Furthermore the applet couples to (Netscape) JavaScript. At the registration page a Javascript routine uses this coupling to read name and address information from the Chipper card and fill the registration request form with it.

4. Conclusions

The ISO7816 standards for smart cards provide basic functionality for authentication through the internal authenticate command. Together with the storage of an ID this command allows us to build challenge/response identificators. It has been demonstrated that with such identificators it's possible to build an intrinsically secure three party authentication protocol (called KLOMP). Furthermore KLOMP imposes few requirements on the trusted party, e.g. no administration of sessions is needed for the authentication, no information has to be returned before both client and server have been authenticated. Also KLOMP separates authentication from privacy through strong encryption, so the system does not suffer from cryptography export regulations like those imposed by the US. The incorporation of encryption of short messages with 'inverse' MAC's strongly enhances the security of the protocol to a point where session keys are still secure even the smart card is stolen or other wise abused. This gives hardware-based identificators like smart cards a definite advantage over mechanisms that rely solely on software.

Currently the majority smart cards used in Holland are either a version of the IBM Multi Function Card (the "Chipper") or a ChipKnip. This is mainly a consequence of the massive scale at which the joined banks and the PostBank introduced the ChipKnip and the Chipper to the public. Though both are ISO7816 compliant cards, neither base their security on the internal authenticate command (even though the MFC at least implements it). The MFC does provide two other challenge/response based commands that can be used as a substitution. The MFC therefore is compatible with the proposed authentication protocol. Unfortunately the ChipKnip cannot be adapted to the protocol because it lacks a transactionless authentication method.

The authentication protocol has been successfully implemented in Java. It has been tested on the Linux, Solaris and Windows32 platforms, demonstrating the cross platform capabilities of Java. The time measurements performed with the relatively slow Linux JDK 1.1.1 indicate that the protocol does not require an unacceptably long time to complete.

Bibliography

[Bakker98] "Smart Card Access Library v1.1", Bastiaan Bakker 1998, http://speeltuin.lifeline.nl/~bastiaan/smartcardapi.html

[Bakker98-2] "Klomp Prototype Implementation v1.0", Bastiaan Bakker 1998, http://speeltuin.lifeline.nl/~bastiaan/klompproto.html

[BeaNet96] "ChipKnip Terminal Specifications", BeaNet BV 1996

[BillSF96] "Everything About Chipcards", Bill SF, magazine 't Klaphek, issue 2, 1996

[Bird92] "Systematic Design of a Family of Attack-Resistant Authentication Protocols", R. Bird, I. Gopal, A. Herzberg, P. Janson, S. Kutten, R. Molva, M. Yung, IBM 1992

[Bird93] "The KryptoKnight Family of Light-Weight Protocols for Authentication and Key Distribution", R. Bird, I. Gopal, A. Herzberg, P. Janson, S. Kutten, R. Molva, M. Yung, IBM 1993

[ETSI93] "Requirements for IC cards and terminals for telecommunication use, part 3", prEN726-3 version 14, ETSI STC9, Valbonne Cedex 1993

[Hoekstra97] "Design and implementation of the ISI-3 authentication protocol", Arjen Hoekstra, ISCIT 1997

[ISO89] "Identification cards – Integrated circuit(s) card with contacts, part 3", ISO/IEC 7816-3, Geneva 1989

[ISO92] "Identification cards – Integrated circuit(s) card with contacts, part 3, amendement 1", ISO/IEC 7816-3 Amendement 1, Geneva 1992

[ISO93] "Identification cards – Integrated circuit(s) card with contacts, part 4", ISO/IEC 7816-4, Geneva 1993

[IBM96]	"The IBM MultiFunction Card Programmer's Reference v3.5", IBM Germany Development Laboratory, Boeblingen 1996. CONFIDENTIAL
[JavaSoft97a]	"Java 1.1 API Documentation", JavaSoft 1997
[Schneier96]	"Applied Cryptography, second edition", Bruce Schneier, John Wiley & Sons 1996, ISBN 0-481-11709-9
[Towitoko97]	"RS232 protocoll description for ChipDrive & KartenZwerg", Towitoko Electronics 1997

Notational conventions

X // Y	X concatenated with Y
X ⊕ Y	X exclusive ORed with Y
K_X	Party X's master secret (= a long term key shared with the TTP)
K_{X*}	Party X's key transforming key (= a derived key shared with the TTP for the duration of a session)
K_{XY}	Secret session key shared by X and Y
$E_X(M)$	Encryption of M under key K_X
$E^{-1}_X(M)$	Decryption of M under key K_X
$H(M_1,..,M_n)$	a secure hash of M_1, ... , M_n
$MAC_X(M)$	Message Authentication Code for message M under key K_X
MAC_X (C,M)	Message Authentication Code for message M under key K_X with challenge C. The algorithm by which the challenge is inserted in the MAC is either unknown or undetermined.
MAC^{-1}_X (R,M)	Inverse MAC for message M under key K_X with response R: $MAC_X(C,M) = R \Leftrightarrow MAC^{-1}_X(R,M) = C$
N_X	a nonce generated by X
C_X	a challenge for I_X
I_X	The identificator owned by X
D_X	Data used by I_X to calculate MACs
M_X	a MAC calculated by I_X based upon challenge C_X
P_{XY}	Proof of authentication by X for verification at Y
T_{XY}	a ticket for X containing the encrypted session key K_{XY}
A	The client
B	The server
KDC	The Key Distribution Center (= the Trusted Third Party)
T	a timestamp

Abbreviations

ABK(t)	Three party authentication protocol in which a time stamp is used to substitute a nonce for party K.
CA	Certificate Authority
CBC	Cipher Block Chaining (mode)
CHV	Card Holder Verification (number)
ECB	Electronic Code Book (mode)
DES	Data Encryption Standard
DSS	Digital Signature Standard
ETSI	European Telecommunication Standards Institute
HTTP	Hyper Text Transfer Protocol
ICC	Integrated Circuit Card
ICV	Initial Chaining Vector
IEP	Intersector Electronic Purse
JDK	Java Development Kit
JVM	Java Virtual Machine
JNI	Java Native Interface
KDC	Key Distribution Center
KEK	Key Encrypting Key
KET	Key Expiration Time
KGK	Key Generating Key
KTK	Key Transforming Key
MD5	Message Digest number 5
MFC	Multi Function (Chip)Card
NC	Network Computer
PIN	Personal Identification Number
PTS	Payment Terminal System
RMI	Remote Method Invocation
SAM	Secure Application Module
SCM	Secure Cryptographic Module
SHA	Secure Hash Algorithm
SSL	Secure Sockets Layer
TTP	Trusted Third Party