Applicability of Smart Cards to Network User Authentication

Marjan Krajewski, Jr., John C. Chipchak,
David A. Chodorow, Jonathan T. Trostle
The MITRE Corporation

ABSTRACT: This paper addresses security issues associated with authenticating users to system services in distributed information systems. Its focus is the presentation of an approach for augmenting the Kerberos network user identification and authentication protocol through the integration of emerging smart card technology. Our conclusions indicate that the security of Kerberos-based network authentication can be substantially enhanced by employing a smart card as a trusted desktop coprocessor.

© 1994 The USENIX Association,

1. Introduction

Two critical aspects of information system security are the application of access controls based on a user's authorizations and the creation of an audit trail based on a user's actions.[1] Both are dependent on the accurate authentication of users to guard against the threat of intruders masquerading as valid users. Traditionally, a user is authenticated to a host upon presentation of a valid combination of user ID and password. In a distributed processing environment, a user often needs to access resources located at multiple servers from multiple workstations interconnected via a communications network. Authentication to each host accessed is crucial, but presenting separate user ID/password pairs can be both unwieldy and unsecure. What is needed is a mechanism that requires users to identify and authenticate themselves once and then transparently performs all further user identification and authentication.

Although much work has and is being done in this area, a general solution suitable for truly hostile environments (i.e., those subject to attacks against both workstations/servers and the network) does not yet exist. Some identification and authentication protocols, designed for use in military environments where the network is physically protected from intruders and the users are trusted, do not use any form of encryption and can be easily defeated by any one of a number of commercially available network protocol analyzers capable of intercepting network transmissions. Other protocols protect against the threat of network eavesdropping through the use of various forms of encryption but still assume that workstations and servers are physically protected (e.g., by individual user ownership/control). The covert introduction of a Trojan Horse program into these workstations can easily "break" the authentication mechanism. Many organizations could greatly ease the many problems associated with password management and the threat from masquerading on their increasingly distributed information systems with an approach that was secure from both a workstation/server and a network perspective.

The Kerberos protocol possesses many advantages as a basis for this capability. Originally developed to provide user authentication for the distributed open computing environment of MIT's Project Athena, Kerberos is growing signifi-

cantly in popularity (it has been adopted by the Open Software Foundation and Unix International and is being offered in several commercial products). It is based on the use of a trusted authentication server to provide "tickets" for presentation to other system servers and uses conventional (secret) key encryption to protect against network eavesdropping and masquerading.

2. Kerberos Overview

Kerberos version 4 was developed at MIT to protect the Project Athena network[2]; Kerberos version 5 is intended for a wider variety of environments and has additional security features.[3] Version 5 has recently been approved as a proposed standard and released as an Internet Request for Comments (RFC 1510, September 1993).

In Kerberos, the entities that interact in authentication exchanges are called principals; these principals are either users that will access the network via workstations or network services running on a host. Each principal receives a secret Data Encryption Standard (DES) encryption key upon registration. Hosts within a local group (referred to as a realm) are served by a Kerberos authentication server (KAS) and a ticket granting server (TGS), which are typically co-located.

The authentication of a user's client application to a particular network service requires three steps; the first step occurs during login. The user enters his or her user ID into the workstation; this user ID is sent to the KAS. The KAS sends back a ticket granting server ticket (TGS ticket), containing the server name, the server realm, and an encrypted part. The encrypted part includes a session key, client identification information, ticket lifetime information, and several optional fields. The encrypted part of the ticket is encrypted in the secret key of the server (the TGS in this case). The KAS also sends back an encrypted response message containing the session key included with the ticket (the TGS session key in this case) along with client identification information and other fields. The response is encrypted in the secret key of the requesting user. Upon receiving the encrypted KAS response, the user is prompted for his/her password; this and other information are used to recreate the user's secret key. The user's secret key is then used to decrypt the KAS response. At this point, the user's workstation holds a TGS ticket and the user is considered "logged in." When needed, these credentials can be used to obtain tickets for network services from the TGS.

To request a ticket for a network service, the client application must contact the TGS. In Kerberos, the authentication of a client to a server requires the client to send both a ticket (granted by the KAS) and an authenticator (created by the

client) to the server. The purpose of the authenticator is to prove that the requesting client is the one to whom the ticket was issued (to prevent replay attacks). The authenticator contains client identification information, a timestamp, checksum, and, optionally, a subkey field and is encrypted in the session key contained within the associated ticket. Upon receiving the TGS ticket and authenticator, the TGS first decrypts the encrypted part of the ticket with its secret key, then uses the session key contained within the ticket to decrypt the authenticator. The TGS then checks if the checksum in the authenticator is correct and if the client identification information matches the same fields in the ticket. Also, the timestamp field is checked to determine if the authenticator is fresh or is a potential replay. If the timestamp is more than five minutes old, the authenticator is rejected. The server also maintains a record of all authenticators that have been received in the last five minutes; if the timestamp and client names match the same fields in a previously recorded authenticator, then the authenticator is rejected. If the credentials pass the various checks, the TGS issues a ticket for the desired application server along witn a response containing the application server session key and other fields encrypted in the TGS session key. The client uses the TGS session key to decrypt the TGS response and obtain the application server session key. Both the application server session key and associated ticket are then stored for later use.

The authentication of the client to an application server involves the creation of an authenticator (encrypting it with the application session key). The application server ticket and authenticator are sent to the application server. The application server uses its secret key to decrypt the ticket and perform the various authentication checks as previously described for the TGS. If the client credentials pass these various checks, the application session is allowed to begin.

Some of the additional features present in Kerberos version 5 that are not implemented in Kerberos version 4 include the server caching of received authenticators for a five-minute period, the modularization of the encryption functions (to facilitate replacement of DES encryption with other types of encryption), integrity protection of plaintext (obtained by embedding a checksum before encryption), use of ASN.1 syntax for encoding fields in messages, flexible ticket lifetimes, authentication forwarding (allowing a client access rights based on a ticket issued to another client), use of a sub-session key to protect a given application session (because the session key is valid for the lifetime of a ticket, some issues arise from reusing a session key for several different application sessions), and options for a challenge/response protocol (to be used as an alternative means for protection against replay attacks).

3. Kerberos Login Vulnerabilities

Kerberos has been analyzed from a general security perspective.[4] A significant vulnerability involves its manipulation of the user's Kerberos password and session keys within the workstation, thereby making it vulnerable to Trojan Horse threats that could capture such data for later use by an intruder. Another vulnerability involves the threat of repeated attacks at the intruder's leisure following interception of the initial response from the KAS. This response contains the ticket granting server session key and is encrypted using the user's relatively weak password-derived secret key. A third vulnerability involves the inherent weakness of depending solely on a single factor (i.e., a password) for the initial user authentication. Passwords can be easily borrowed or stolen. These vulnerabilities are depicted in Figure 1.

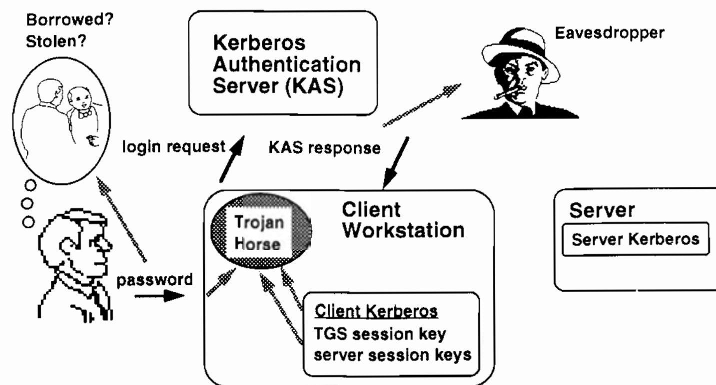

Figure 1. Kerberos login vulnerabilities.

Advances in encryption and smart card technology have reached the point where significant amounts of information can be stored and processed within the card itself.[5] When this technology is combined with user-unique information such as a password or personal identification number (PIN) that is useless except when processed by the appropriate smart card, the security provided by Kerberos can be greatly improved.

Ongoing work in this area has involved the use of magnetic stripe or memory cards to hold the user's password/secret key and the use of time-varying smart cards to provide a unique, perishable secret key known only to the KAS and the user. When combined with a PIN, these approaches address two of the vulnerabilities mentioned above (i.e., the interception of the initial response from the KAS and the dependence on a single authentication factor). However, they do not provide a wholly satisfactory solution for the third vulnerability (i.e., data cap-

ture by a Trojan Horse), since decrypted Kerberos session keys remain within the workstation.

4. Augmentation Concept

The ability of smart cards to provide an independent storage and processing environment allows the migration of a user's secret key and the sensitive cryptographic processing associated with it from an untrusted workstation to a trusted device that is always under the physical control/protection of the user (see Figure 2). The user key stored on the card would itself be encrypted in a key derived from a password. In this way, neither possession of the card alone nor knowledge of the password alone would be sufficient to allow an imposter to masquerade as the authorized user. Encryption and decryption operations and the storage of unencrypted authentication information would occur only within the trusted environment of the smart card, and workstation software would only be allowed access to session keys in encrypted form. With this approach, the smart card forms a *personal trusted extension* of the Kerberos processing environment. This concept was first proposed by Krajewski.[6]

In addition to enhancing security, it is important to maintain interoperability with existing Kerberos implementations in order to allow augmented Kerberos components to be gradually introduced into an operational environment as time and resources permit. This constraint mandates that neither the Kerberos authentication server (KAS) nor the server Kerberos implementations be affected in any way. It also mandates that the client Kerberos software be modified in a way that results in the least impact to the existing code. The initial concept was modified [7] to require only that selected cryptographic routines in client Kerberos be replaced with smart card driver software. The driver software, together with the attached smart card, emulated the original code with one difference—all session

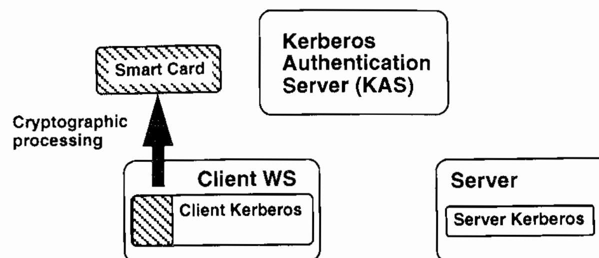

Figure 2. Kerberos augmentation concept.

keys presented to the smart card driver were returned to the calling routine in encrypted form rather than unencrypted form. By doing this, the remaining client Kerberos software is tricked into "believing" that it is handling red (unencrypted) session keys when, in fact, it is handling only black (encrypted) session keys.

Issues that quickly arise in attempting to implement this concept include those related to feasibility, security, and performance. Regarding feasibility, it is not obvious how well current smart card technology can support the required functional partitioning between inboard and outboard elements. Regarding security, it is not obvious how well current smart card technology can provide a trusted environment and protect sensitive information. Regarding performance, it is not obvious how severely the use of an outboard microprocessor will impact response time. The details of and lessons learned from a prototype built to answer these questions are the focus of the remainder of this paper.

5. Prototype Environment

The Kerberos smart card prototype was developed using the Kerberos version 5 beta 1 release of the software. The prototype environment consisted of both prototype components and development components. The prototype components consisted of a Sun SPARCstation™ IPC workstation, which served as the Kerberos authentication server/ticket granting server (KAS/TGS); a Sun 3/50 workstation, which was used as the Kerberos client workstation, a SOTA Electronics Inc. OmegaCard™ Reader connected to the RS-232-C serial port of the client; and a SOTA OmegaCard™. Both of the workstations ran SunOS release 4.1.1 and had the Kerberized telnet daemon running for both client and server modes of operation. The development components consisted of an IBM PC clone with an OmegaCard™ Reader, an OmegaCard™, and development software. The development software included the SOTA development applications and routines supported by the Archimedes Software Inc. C-8051 software—a C-cross compiler and an 8051 assembler. An OmegaCard™ is a smart card with an embedded 8051 microcontroller; 128 bytes of random access memory (RAM); 4 kilobytes of masked read-only memory (ROM), which contains a small operating system; 8 kilobytes of electrically erasable programmable read-only memory (EEPROM), which stores the user application and data; and serial input/output (I/O) to interface the smart card to the outside world (see Figure 3). The OmegaCard™ Reader provides power, clock, and an RS-232-C interface for the OmegaCard™. The OmegaCard™ will henceforth be referred to simply as the smart card.

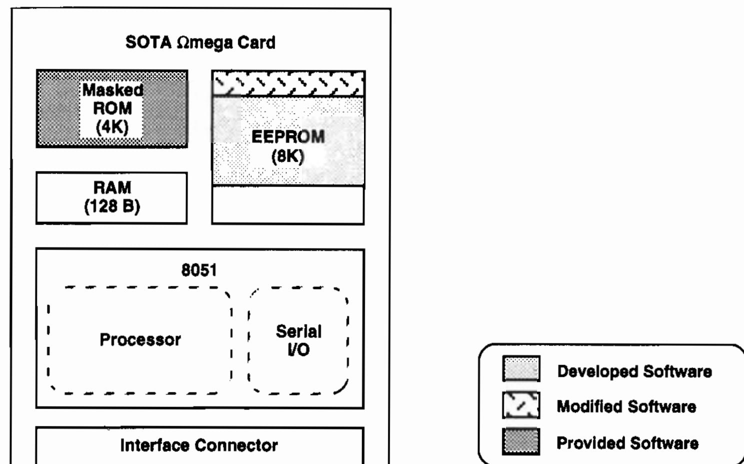

Figure 3. Smart card hardware architecture.

6. Kerberos Client Modifications

In developing the prototype, we confined our modifications of client Kerberos to files within the DES directory. Thus, the prototype can, in principle, be readily modified to work with a version of Kerberos that uses a cryptographic algorithm other than DES. The prototype demonstrates kinit and Kerberized telnet. Due to resource constraints, mutual authentication was not implemented (i.e., the prototype only performs telnet client authentication), nor have we made the modifications to allow private or safe messages to be exchanged. We feel it would not be difficult to modify the prototype to include these features, but it would require client modifications outside of the DES directory.

The program kinit is responsible for getting the ticket granting ticket and the ticket granting server session key from the Kerberos authentication server (see Figure 4). In the prototype, as in regular Kerberos, the principal name is first sent to the authentication server, which responds with a message containing a TGS ticket and an encrypted string. The encrypted string contains the TGS session key in bytes 31 through 38 and is encrypted in the user key. The user then enters his or her password into the workstation, and this password is hashed into a DES key by the mit_des_string2key function. At this point, modifications to the mit_des_string2key function open and configure the serial port line attached to the smart card for exclusive access (thereby preventing other processes on the workstation from accessing the smart card between the exclusive open system call and the close of the serial port when kinit exits). In addition, the modified mit_des_string2key function sends reset, login, and prompt commands to the smart card to initialize it. Next, the modified

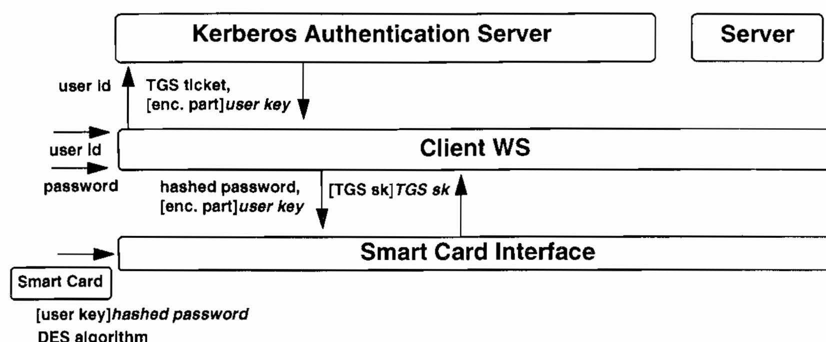

Figure 4. Kinit.

mit_des_string2key function sends the hashed password key to the smart card to decrypt the user key.

The mit_des_cbc_encrypt function and the mit_des_ecb_encrypt function are the Kerberos cryptographic functions. We removed the mit_des_ecb_encrypt function since its work is performed by the smart card. We modified the mit_des_cbc_encrypt function to transfer the encrypted string to the smart card. For decryption operations, the message is returned in decrypted form except for the portion containing the session key (bytes 31 through 38), which is encrypted in the TGS session key. Upon the return of the decrypted message, the client workstation believes it has an unencrypted session key and handles it as it would any such key.

The main problem encountered in the prototype implementation of kinit was that the client computes a checksum over the decrypted string containing the TGS session key and compares it to the checksum that was computed over the same string by the authentication server and included in the message prior to encryption. These values are no longer equal in the prototype because the TGS session key on the client is encrypted. Our interim solution was to remove the checksum calculation code from the client. This introduces a vulnerability in that a malicious Trojan Horse could change the order of bytes in the message sent to the smart card and cause the session key to be returned in the clear. A more general solution, therefore, would be to compute and validate the checksum on the smart card prior to returning any decrypted information.

The sequence of operations to establish a Kerberized telnet session is as follows (see Figure 5). First, the client creates an authenticator (intended to counter the replay threat) for the TGS ticket. Next, using the modified mit_des_cbc_encrypt function, the TGS session key (in encrypted form) is sent down to the smart card. Then, the authenticator is sent to the smart card and returned to the client, in encrypted form. The client sends the encrypted authenticator and TGS ticket to the

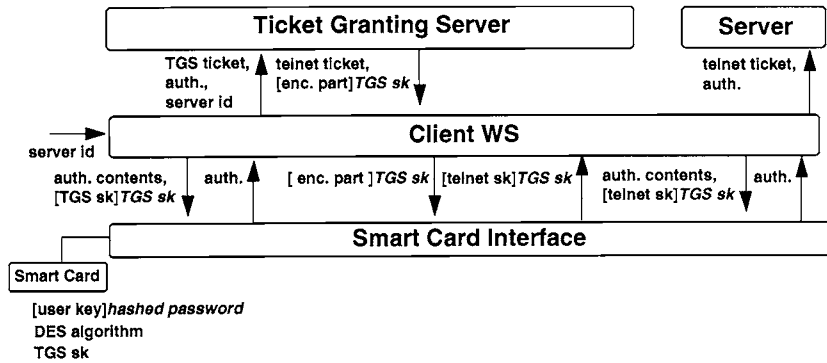

Figure 5. Kerberized telnet.

TGS; the TGS responds with the appropriate telnet ticket and a string encrypted with the TGS session key. As in the case of kinit, the telnet session key is in bytes 31 through 38 of this encrypted string. Using the modified mit_des_cbc_encrypt function, the client sends the encrypted string to the smart card and receives the decrypted result (again, as with kinit, the telnet session key is encrypted in the TGS session key before being sent back). The client then creates an authenticator for the telnet ticket and sends the telnet session key (encrypted) and the authenticator to the smart card. The encrypted authenticator is returned to the client and sent with the telnet ticket to the desired server. Once the telnet server receives the telnet ticket and the encrypted authenticator, it decrypts them, performs the necessary authentication checks, and, if successful, allows the telnet session to begin.

The main problems encountered in developing the modified telnet implementation were twofold and, as was the case with kinit, were due to the fact that the keys on the client are in encrypted form whereas the Kerberos code expects them to be in the clear. One such problem results from the fact that the Kerberos version 5 beta 1 code uses the keys themselves as the DES algorithm cipher block chaining (CBC) mode initialization vectors. We thus found it necessary to have the smart card use the decrypted keys as the initialization vectors rather than obtain them from the client. We also encountered a problem resulting from the client's use of the session key to create a sub-session key to include in the authenticator. The client Kerberos routines check the parity of the session key, and since this key is encrypted on the client, it often has bad parity, thereby causing an error to be flagged. In the prototype implementation, we simply removed the error flagging code. A more general solution is to make a copy of the encrypted session key on the client and allow this copy to be "fixed up" by the mit_des_fixup_key_parity function.

7. *Smart Card Implementation*

The smart card is programmed to be an encryption service provider for the encryption of Kerberos authenticators and the decryption of Kerberos KAS/TGS responses. It implements the DES CBC mode, performing an encryption or decryption operation on 8-byte blocks of data using a DES key and an initialization vector. The software architecture of the smart card consists of the operating system, I/O routines, interface routines, the control application, and DES algorithm (see Figure 6). The operating system and I/O routines are provided in the smart card development system. Assembler interface routines were modified to allow Archimedes version 4.0 C-8051 to make calls to the operating system to read messages into the smart card, write messages out of the smart card, and write bytes to EEPROM. Both the control application and the DES algorithm are written in C.

The control application implements the control protocol to communicate with client Kerberos and performs the requested encryption service on the input data. There are two additional functions in support of the encryption service: decrypting the user key for decryption of the first KAS message and downloading a session key for encryption of authenticators. The control protocol has been implemented such that each request message sent to or from the smart card has a corresponding response message (acknowledgment) returned before another message may be sent. This serves to keep the client Kerberos and the smart card message exchanges tightly coupled and helped in smart card software debugging. There are some smart card initialization procedures regarding card reset, card login, and execution of the control application. Once the smart card control applica-

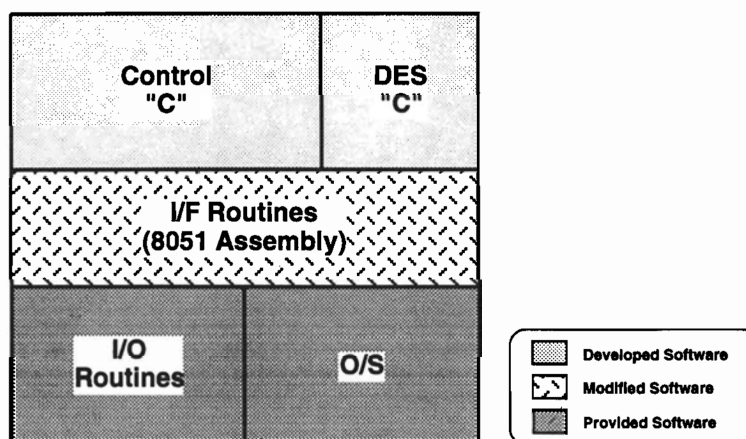

Figure 6. Smart card software architecture.

tion is running, all messages sent to the smart card are delivered to the control application.

In support of KAS/TGS response decryption, the following operations are performed. To process the initial KAS response, it is first necessary to decrypt the Kerberos user key (stored in encrypted form in EEPROM). To do this, client Kerberos sends a hash of the user password to the smart card to be used as a DES key. The smart card decrypts the user key using this hashed password key. Client Kerberos then sends the KAS response to the smart card, one block at a time in a Download Block message. The last block is sent in a Decrypt message, notifying the smart card to start the decryption process. The smart card assumes that the first response to be decrypted following initialization comes from the KAS. This response is encrypted in the user key and contains the TGS session key. Subsequent responses are assumed to come from the TGS. These responses are encrypted in the TGS session key and contain server session keys. Decrypted blocks are sent back in a Return message. The last decrypted block is sent back in an End message. All decrypted session keys are re-encrypted in the TGS session key before being sent back. The decrypted user key is destroyed after the first response is processed. Thereafter, the smart card uses the stored TGS session key for all decryptions.

For subsequent support of the TGS authenticator encryption, the following operations are performed. The appropriate session key is sent to the smart card in the Send Session Key message. The session key is decrypted with the TGS session key and stored in memory. Next, the TGS authenticator is sent down to the smart card one block at a time in a Download Block message. The last block is sent as an Encrypt message, notifying the smart card to start the encryption process. The TGS authenticator is then encrypted in the session key previously downloaded. As each block is encrypted, it is sent back to the client Kerberos in a Return message. The last block is sent in an End message.

Given that the smart card was not required to act as a bulk encryptor for message data streams, the implementation of the DES algorithm on the smart card was driven by memory constraints rather than encryption rate, with the understanding that higher encryption rates could be achieved by trading space for time. We chose a straightforward implementation of the DES algorithm[8] and attempted to fit it into the available memory. The problem was not EEPROM for code, but RAM space for working storage. Ultimately, we were not totally successful and were forced to use EEPROM to hold the key schedule and other intermediate results.

8. Lessons Learned

The clearest lesson learned from the prototype implementation is that a suitable smart card must provide at least 256 bytes of RAM for application use. The card used in the prototype came with 128 bytes of RAM, a significant portion of which was reserved for operating system use. In the prototype implementation, we were thus forced to use EEPROM to augment the available RAM.

Using EEPROM for working storage has a variety of implications. In the prototype, encryption/decryption of a single 8-byte block takes approximately 5.3 seconds, 4.6 seconds of which is spent simply writing data into EEPROM. Performing a standard telnet login first requires the encryption of an authenticator to be associated with the ticket granting ticket (14 blocks), followed by a decryption of the response from the TGS to obtain the telnet server session key (23 blocks), followed by an encryption of an authenticator to be associated with the desired telnet server ticket (14 blocks). In the prototype, this process requires approximately 270 seconds. This represents an obvious performance issue. There is also the less obvious issue of security. In using EEPROM in this manner, there is a significant window of time in which key material is in non-volatile memory. Should the card be removed from the reader during an operation, the key material could be recovered. Finally, there is a reliability issue, since EEPROM supports only a finite number of write cycles (usually in the tens of thousands). Clearly, use of a smart card possessing several hundred bytes of RAM is criticaal for any practical implementation.

The decryption of the encrypted portion of the KAS/TGS response messages by the smart card presents a vulnerability whereby blocks of data may be transposed by a Trojan Horse prior to being sent to the card and the decrypted session key obtained from the decrypted field to which it was moved. This vulnerability can be mitigated if the smart card verified the message checksum following decryption and prior to returning any decrypted fields. This was not done in the prototype due to the limited amount of memory available.

Another vulnerability associated with the prototype implementation is due primarily to the flexible programming environment of the particular smart card used. The postulated attack involves a workstation-resident Trojan Horse surreptitiously downloading to the smart card modified software containing an embedded Trojan Horse whose purpose is to transmit unencrypted smart card data back to the workstation. The prototype smart card will require a restart and execute integrity tests, but the modified software's header block can be set such that the new code will pass those tests and execute with the Trojan Horse in place. For the smart card used in the prototype, placing the encrypted user key immediately after the

header block (i.e., prior to any application code) provides partial protection against this attack in that the user key would be destroyed during any overwrite attempt or the card would fail its startup integrity tests (in either case rendering the smart card useless for future logins and allowing the tampering to be detected). A more secure solution requires the use of a smart card that prevents downloading of software from the workstation (e.g., one in which the application code is in masked ROM).

9. Summary and Conclusions

The essential concept underlying this work involves exploiting the ability of smart cards to provide an independent storage and processing environment to enable the sensitive cryptographic processing associated with Kerberos network user authentication to occur only within a trusted device that is always under the physical control/protection of the user. This improves system security in three significant ways. First, it requires a user to provide both something he or she possesses (i.e., a smart card) as well as something he or she knows (i.e., a password). Either item alone is useless. This greatly reduces the risk from password borrowing or theft. Second, it allows the initial message from the authentication server to be encrypted in a truly random key (i.e., the user key need not be derived from a password). A cryptographic attack on this message must therefore assume that the entire keyspace is available for use. This greatly reduces the risk from network eavesdropping. Finally, it ensures that security-critical cryptographic data is encrypted while on the user's workstation. A malicious Trojan Horse program can obtain no sensitive information. This greatly reduces the risk from such programs.

The prototype implementation described in this paper has proven that the concept is feasible. The primary lessons learned indicate that performance and security are major factors in the selection of a suitable smart card. The smart card used in the proof-of-concept prototype contained 8 kilobytes of EEPROM and 128 bytes of RAM. This amount of memory was found to be marginal for the application. In particular, intermediate computational results that would normally be stored in RAM had to be stored in EEPROM, dramatically increasing response time and potentially compromising security. A smart card possessing a minimum of 256 bytes of RAM appears necessary. In addition, the smart card used in the prototype is potentially vulnerable to an attack whereby EEPROM-resident applications software is overwritten with malicious software from the workstation. Use of a smart card possessing protected areas that cannot be viewed or altered from

the workstation also appears necessary. Should these conditions be met, and current technology indicates that they can, this work has shown that a highly secure network user authentication mechanism can be constructed.

References

1. National Computer Security Center, "Department of Defense Trusted Computer System Evaluation Criteria," DOD Standard 5200.28-STD, December 1985.
2. Steiner, J., Neuman, C., and Schiller, J., "Kerberos: An Authentication Service for Open Network Systems," USENIX Conference, 1988.
3. Kohl, John T., "The Evolution of the Kerberos Authentication Service," Spring EurOpen Conference, 1991.
4. Bellovin, S. M. and Merritt, M., "Limitations of the Kerberos Authentication System," *Computer Communications Review*, October 1990.
5. Smart Card Industry Association and Personal Identification Newsletter, CardTech Conference Proceedings.
6. Krajewski, M., "Concept for a Smart Card Kerberos," 15th National Computer Security Conference, October 1992.
7. Krajewski, M., "Smart Card Augmentation of Kerberos," Privacy and Security Research Group Workshop on Network and Distributed System Security, February 1993.
8. Press, W. H., Flannery, B. P., Teulosky, S. A., and Vetterling, W. T., *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, 1986, pp. 218–220.