University Halle-Wittenberg
Institute of Computer Science

Using Hardware Tokens for Authentication of Multiple Client Application Making Single Sign-On Possible
Sandro Wefel, Paul Molitor

Using Hardware Tokens for Authentication of Multiple Client Application Making Single Sign-On Possible

Sandro Wefel, Paul Molitor

User authentication in most systems is done by the principle: registration with unique user name and authentication by presenting or using a secret, e.g., a password or a private cryptographic key, respectively. To obtain a trustworthy method, combinations of hardware token with user certificates and keys secured by a PIN, a fingerprint or other biometrical characteristics have to be applied. In order to further increase consumer acceptance with respect to hardware tokens which have to be carried hardware tokens have to be provided with added values.

This paper describes an approach which allows different applications to use one single common hardware token for authentication. All these applications use the same interface for authentication of the user to the different service providers. It is shown that this approach works with widely used standard software and plugins. Furthermore the paper addresses single sign-on. We show how to extend our approach so that hardware tokens can be used for authentication in different applications after the PIN is put in once during the login to the operating system. Finally, we dwell on special methods to protect vital information in the computer’s main storage against attacks.

1 Introduction

Smartcards and other forms of hardware tokens allow the user to keep information, e.g., private keys and passwords, in safe custody. They are especially suitable for being used in the context of public key systems [29, 18] when provided with interfaces to call methods requiring vital information. Such hardware tokens ensure that the private keys and passwords never leave the token under any circumstances and thus enhance reliability to a large extent. A user has to authenticate himself to the token. He shows his legitimation by sending a biometrical or a secret information to the token. This process of authentication to the hardware token is called Card-Holder-Verification (CHV), a notion originating from smartcards [26].

After authentication of the user to the token, the token offers methods to read protected information or to use signature or other cryptographic operations. These operations can be executed by the processor which is integrated on the token once authentication to the hardware token has succeeded. Accordingly, it is ensured that one of the criteria for user authentication to appli-
cations, namely "something you have" [10], cannot be duplicated. Thus, misuse of legitimation information is hard when stored in hardware tokens. Protection against a misuse after token loss should be given by CHV. If CHV is done by PIN input, the PIN has to be secured in the manner a password is secured.

In the following sections we focus on hardware tokens secured by PINs which offer methods for being used in single sign-on environments. Such hardware tokens should fulfill several requirements.

- **Certification of the token owner:** A token has to provide methods allowing the token owner to prove that he is the person he pretends to be. A token described by the RSA PKCS#15 standard [7] may hold an X.509 certificate [6] beside the private and public key objects. Such a certificate certifies the affiliation of the token owner, described by a unique item, to the public key [16]. For this a trusted third party, the Certificate Agency (CA), certifies this affiliation. Such a token offers user authentication methods by the following principle: A provider offering sensitive information may check whether the communication partner is really the person he pretends to be. He asks the certificate of the user and, if the certificate check holds, makes a challenge against the private key which belongs to the public key of the certificate. Only the user which holds the token with the non-copyable private key can accomplish this challenge. This approach assumes a traceable chain from the user certificate to the certificate of an authority which the provider trusts.

- **User authentication to applications:** In addition to the authentication to the local operating system authentication to server-client applications has to be supported. In order to enhance reliability secured channels should be used here although they require additional computational capabilities. Most established methods to build a cryptographical secured connection between a client and the service provider are hybrid cryptographical methods, where a random symmetric key is generated and exchanged by an asymmetric method. To use asymmetric cryptography both sides present their certificates and if they are accepted, the exchange of the symmetric key is started. Another method is the Diffie-Hellman key exchange [27] where both communication parties sign the exchanged information using their private keys and the other side checks the signature with the aid of the user certificate. The methods used in software often base on the SSL/TLS protocol [31, 28]. In this protocol only the presence of the server certificate is necessarily required. However, the protocol offers the possibility to ask for and check the client certificate, too. Thus, secured communication
channels can be used for user authentication. A secured channel does not require further authentication information. Thus, secure connection authentication using mechanisms based on passwords like plain, login, digest-md5, cram-md5 or shared login systems, e.g., gssapi, are not required any more.

- **Authentication and authorization:** Applications using passwords or similar methods often combine authentication and authorization. Only authorized users possess username-password combinations. User certificates are however used only for authentication. Authorization has to be done by another method after authentication has succeeded. This drawback of user certificates is due to the fact that a user holds only one certificate in general, but can have different roles with respect to the application. As different roles cannot be handled by the same certificate other methods have to be provided by the hardware token.

The libraries used in server and client applications for establishing secured connections often are the same or resemble each other. For instance, open source software and many other applications use the OpenSSL libraries \[5\] on client and server side for establishing secured connections. Based on this observation, Section 2 presents an approach allowing user authentication of some standard software applications by client certificate authentication. The user is authenticated by using his certificate as the client certificate during establishing a secured connection, followed by a challenge-response check with the associated private key. For this purpose we looked for client and server applications to which the approach can be applied without having to considerably modify the corresponding programs. For client applications, we looked for software with particular emphasis on storing the user information in the hardware token to ensure the reliability so that the vital information for authentication cannot be copied.

Unfortunately, many users use one and only one easy to keep in mind password for authentication in all their environments. Certificate based authentication in combination with hardware tokens seems to be more secure because the knowledge of the PIN, which acts as a password, is more or less useless for authentication if one does not possess the token. However, it is assumed that a user also uses an easy number combination as PIN if the PIN has to be put in very often during a session, e.g., for each application he calls. So an attacker who plans to steal the token can easily obtain the PIN by glimpsing at its input. Thus, the number of PIN inputs should be reduced for enhancing reliability. This requirement leads us to single sign-on. Note that lowering the number of times the PIN has to be put in can also lead to a better consumer acceptance of using hardware tokens. In Section 3 we present a practical approach in which the PIN has to
be put in only once, namely during the login to the operation system. This approach however requires that the PIN is stored in the main storage of the local computer. Thus, methods have to be provided to protect the PIN.

The paper closes with some remarks on the complete system.

2 Using hardware token for authentication

Hardware tokens can be applied for authentication to local services, i.e., services located on the local computer, and to network applications. Since the implementation of the RSA Security Inc. PKCS#11 Cryptographic Token Interface [8] neither extra software nor special tokens for the different types of applications are necessary. PKCS#11 is a standard which specifies an application programming interface (API), called Cryptoki, to devices which hold cryptographic information and perform cryptographic functions. It addresses the goals of technology independence (any kind of device) and resource sharing (multiple applications accessing multiple devices), presenting to applications a common, logical view of the hardware token.

Figure 1 shows the common use of an hardware token. Several applications access the token by the PKCS#11 interface, provided by a computer library. The PKCS#11 interface accesses the
token via a hardware interface, e.g., a smartcard reader or an USB connector.

For an all-purpose hardware token based authentication in typically network environments certification and supply of the required application specific keys have to be provided by the token. Another possibility is to use different certificates for different applications. However, this alternative wastes a lot of the token’s memory and has the problem that generation and distribution of the certificates are complicate. Thus, we are looking for an approach which goes without several certificates but with a single one.

To attack this problem, we have met the challenge of implementing a usable and practicable solution for a representative cross section of software services. The services so chosen should cover the most needed requirements for authentication in daily routine:

- restricted access to web services via web browsers
- fetch and submit emails
- secure terminal applications, Secure Shell
- user login to the operating system.

The first three requirements come from typical network applications. The latter is a local authentication of the user to the local operating system.

A sensible use of certificate based user authentication with a hardware token in network applications requires encrypted communication. The required user certificate is exchanged during the establishment of the encrypted communication and, as already described above, can be used for authentication of the user. Login to the operating system is somewhat different, because data transfer does not play a role.

While Secure Shell has its own protocol, the first two applications mentioned above use SSL/TLS communication as secure communication. In the following subsections, we take a more detailed look at authentication in these different settings.

2.1 User Authentication

We discuss SSL/TLS authentication, Secure Shell authentication, and local authentication.

2.1.1 SSL/TLS-Authentication

Many applications, which require a secured transport channel, use the "SSL Protocol Version 3.0" or its extension described in the "TLS Protocol Version 1.0" [13, 12]. Libraries, e.g., the widely
used free OpenSSL library, implement the methods required by this standard. The used handshake protocol of these standards allows user authentication during the connection is established. This authentication is made by the client by presenting a valid user certificate.

The handshake protocol used in SSL and TLS is shown in Fig. 2. The client always initiates the protocol. The lines marked by * are optional. Thus, a non-anonymous server may request a certificate from the client in its ServerHello message.

![SSL/TLS handshake protocol](image)

If no appropriate certificate is available, the client may send a no_certificate. In this case, the server can either decide that the client is accepted and let the authentication to be checked by the application itself or abort the communication with the client. In the case that the client holds an appropriate certificate, it shows it after having received the server’s hello done message. A certificate verify message is generated for proving that the client possesses the right secret key. For this purpose, the client signs a handshake message with its secret key and submits it together with its certificate to the server (Clientfinished). The server checks this signature by applying the public key which is stored in the certificate. In the next step, the server checks the validity of the certificate as certificates can be revoked before their validity periods expire. The validation method applied is application specific. The server may hold a list of valid Certificate Agencies which can be used to check the certificate or mandate an Online Certificate Status Protocol Server (OSCP) to check the validation of the client’s certificate [22].
2.1.2 Secure Shell Authentication

The secure shell (SSH) provides different types of user authentication [33, 32]. Public key authentication methods are one of them.

In “public key” authentication the client sends a signature created by its secret key. The server must check that the key is a valid authenticator and that the signature is valid. If both hold, authorization is granted by the server, although additional authentications may be claimed by the server. If one of both checks fails authorization is rejected. The server may decide to either abort the communication with the client or offer other authentication methods, e.g. password authentication [32].

2.1.3 Local Authentication

Local authentication is the notion we use to refer to authentication of a user to a service running on a local computer to which the token is directly connected, i.e., there is no intermediate network. In this setting public key authentication as described above can be used: the certificate stored on the token is firstly verified, then a challenge is sent to the token. The token signs the challenge with the secret key and sends the response back to the asking service. The service can verify the response with the public key in the certificate and decide the acceptance. In contrast to network authentication there is no need for a key exchange. However, the methods of authentication do not differ from each other.

2.2 User authorization by certificated data

All methods described above are based on the public key mechanism. They serve as authentication tool. However, users can possess several different roles with respect to an application. Thus, authentication cannot directly imply authorization. The certificate should describe a unique user. For this purpose, in addition to the authentication step further authorization steps are required where the information from the user certificate may be used.

If the server application does not provide additional methods of authorization, the information obtained from the certificate must be sufficient to decide the access. For this purpose additional user information could be included in the certificate using X.509 certificate extensions. Special require directives could decide the authorization. Figure 3 exemplarily shows the approach. It shows a line from a webserver configuration using the OpenSSL libraries for secure communication which lets a user with one of the IDs ”user1”, ”user2” or ”user3” from the ”Institute ABC” access the restricted resource.
An other way is to use a separate (Sub-)CA for each service which certifies the user key for this service, if access for the user is permitted. Each server only accepts certificates from its appropriate CA. This approach is not well usable in combination with hardware tokens, because the memory of hardware tokens is usually limited to a few kilobytes, e.g., 32 kilobyte, so that only precious few certificates can be stored on the token.

An approach of using additional steps for authorization, e.g., inquiries of both local and central databases, is much more adequate. The information gathered from the certificate or the certificate itself could be used as key for a database request. For instance, a central LDAP server which stores the user entries may be used. Such an entry combines unique user identification information, e.g., the unique distinguished name of the certificate subject and the authorization information. After user authentication the server may use the certificate to call the LDAP server. If an entry exists on the LDAP server, authorization with the obtained information can be decided. Of course, it is not necessary to store the whole certificate – this would result in efficiency problems in terms of the LDAP server. A unique attribute (UID) in the certificate is adequate to request the entry from the database.

2.3 A working infrastructure

As described in Section 1, authentication using public keys together with certificates should not automatically lead to the authorization of the users with respect to an application, in general.

Dynamic queries of authorization of already authenticated users are important when distributed network based applications are used by a workgroup. There is evidence that storing the authorization information into a central database is advantageous as central saving leads to minor administration efforts for user management in workgroups with shared resources. For this reason we focus only on solutions which allow to combine certificate based authentication and user authorization by using the certificates for retrieving authorization information from central databases.

In the following we discuss how to apply our approach to the service applications.¹

¹The server applications mentioned in Section 2 require that adequate client applications are available. As
2.3.1 Restricted access to web services

The most common way to restrict access to a web service is the HTTP Basic or Digest Authentication. With this method a user authenticates to a web service by presenting the proper combination of a username and a password [14]. This method requires that server and client keep a shared secret, the user’s password.

The most common client applications accessing a web service are web browsers which allow to access web sites in the world wide web. Most of them allow password authentication methods. A disadvantage of password authentication is the required transmission of the password from the client to the server. To avoid clear text password transmission digest authentication methods should be applied. However, not all client and server applications do support this kind of authentication methods. For this reason, most servers communicate with the clients over channel secured by SSL/TLS during the authentication. Such a channel allows SSL/TLS client authentication as already described in Section 2.1, so that user authentication can be installed at this point.

We have applied our approach to the Apache webserver [1] in combination with the SSL/TLS module **modssl** which implements SSL/TLS handshake. The optional request for client certificates during the handshake can be enabled for scopes of the web server which require user authentication. Applications, which run behind the web server, e.g., chats or forums, obtain the client certificate from the web server after successful authentication.

A drawback of this straightforward approach is the fact that the SSL/TLS module authenticates a user only by verification of the presented client certificate. Classifying users with valid certificates in a more granular way, i.e., restricting the access to only certain but not all users with valid certificates, can be done by means of static filters or local databases which hold a static list of allowed certificate subjects. However, this is only feasible for isolated web servers but not for web servers offering services in a network environment. As the Apache web server only allows access to a central user database like an LDAP-server but not in combination with certificate based authentication, we had to look for appropriate third party modules:

- One such third party module is **mod_authz_ldap** [20] which allows the extraction of information from a certificate. These information are used to generate a highly configurable query to a LDAP server to obtain the role of the user to the web server [20].

- Another approach consists of combining the APACHE web server with DACS (Distributed...
Access Control System) [2]. DACS requires that user authentication is already done by the SSL/TLS module and needs the information from the client certificate. It offers a very wide range of different authorization methods. The information can be obtained from local or central databases, e.g., SQL or LDAP servers.

On client side a web browser is required which allows the presentation of a client certificate during the SSL/TLS handshake. In combination with hardware tokens the client should be able to delegate this task to the token by using a PKCS#11 interface. In our prototyping, we use the Firefox web browser which works very well in connection with an adequate PKCS#11 library and Apache based web services.

2.3.2 Fetch and submit emails

The SMTP (Simple Mail Transfer Protocol) [17] describes the electronic mail transport from one user to the mailbox of another user. Reading received emails is carried out by calling a web service which accesses the corresponding mailbox, in general. To fetch received emails from the mailbox stored on a server to the client computer, POP3 (Post Office Protocol version 3) or IMAP (Internet Message Access Protocol) are used [21, 11]. Both allow connections using secure channels.

Submitting an email to the network requires mail transport agents (MTA). The first MTA, i.e., the server which is first to receive the mail from the mail user agent (MUA), mostly requires an authentication in order to avoid abuse of the mail system by foreign users. In this authentication step, it is checked whether the user belongs to the group of legitimate persons, e.g., by checking whether the email address of the user belongs to a certain domain. A detailed identification of the user is not performed, in general.

User identification could be done by using SMTP-AUTH which extends SMTP to include an authentication step through which the client effectively logs in to the mail server during the process of sending mail, i.e., a database of authorized users is required. However, if the first MTA server also allows identification of a legitimate user by certificate based authentication\(^2\), there is no need for using SMTP-AUTH. After the MUA has presented a valid user certificate during the SSL/TLS handshake, the MTA allows the initial transfer of the email to the mail network [15].

Unlike mail transport both protocols for fetching emails, POP3 and IMAP, require the unique identification of the user. This is essential because a mailbox server offers its service to more than one user and a user is allowed to only access his own mailbox. For this reason, the user has to

---

\(^2\) There are several MTA applications which allow certificate based authentication. In our prototyping, we use postfix.
be uniquely identified by a unique username or email address during authentication. POP3 and IMAP explicitly provide secured channels using TLS/SSL [24], however not for authentication purposes. They apply authentication mechanisms like LOGIN or SASL [11]. Nevertheless we could build an email mailbox system which allows certificate based authentication. We use the POP3/IMAP server **dovecot** [3], which allows SSL/TLS secured connections. The server could be configured such that the client certificate is explicitly requested during the SSL/TLS handshake. The certificate allows extraction of data allowing mapping of the users to their mailboxes by means of a central LDAP server.

However POP3 and IMAP require authentication in the protocol itself and do not cover the underlying SSL/TLS authentication. Though **dovecot** defines a new authentication method **anonymous** which allows the connection without user specific authentication information, most email clients are not aware of this feature. For this reason we have configured the mail server in such a way that it accepts any password and other authentication information during POP3 and IMAP authentication, once user authentication is done during SSL/TLS connection establishment.\(^3\)

On client side the email program Thunderbird is an adequate user agent for submitting emails using SMTP and fetching emails from a POP3/IMAP server in combination with certificate based authentication. Thunderbird can access an hardware token through the PKCS#11 interface.

### 2.3.3 Secure terminal applications, secure shell

One of the authentication methods included in the secure shell protocol is public key authentication (see Section 2.1.2). Most server and client applications allow the use of public user keys for authentication. By slight extensions most of these client applications can access hardware tokens to obtain the user keys.

In our prototyping, we have slightly extended the **OpenSC** token interface so that **OpenSSH** client and server applications can access hardware tokens. In this extension we have included an agent which stores the PIN in a secure manner. Thus it could be used for SSO.

In order to allow certificate based authentication, **OpenSSH** has to be extended so that the user public key is obtained from the user certificate and the server checks the authorization by the presented certificate data instead of comparing the user public RSA or DSS key with a stored certificate data.

\(^3\)**dovecot** also allows usage of Pluggable Authentication Modules (PAM) for authentication. The obtained authentication information from the POP3 or IMAP authentication phase (passwords or tickets) are presented to the PAM module which returns "access permitted" or "access denied". In our configuration, each SSL/TLS connection requires a valid client certificate. If the certificate is valid, module **pam\_allow** which always permit access is called. In this way we could use certificate based authentication together with POP3 and IMAP.
bunch of allowed public keys. There are some suitable patches [25] which can be used for this purpose. In particular, they allow the servers to obtain certificates and certificate revocation lists from a central database, e.g., an LDAP server. Thus, OpenSSH can use the same mechanisms for certificate check as SSL/TLS does which reduces administration efforts.

On client side, the same patch allows OpenSSH to access the hardware token in combination with certificate based user authentication by means of the PKCS#11 interface so that a wide range of hardware tokens can be used.

2.3.4 User login to the operating system

The mechanisms to login to the operating system by means of hardware token highly depend on the operating system used.

- Microsoft Windows allows login by means of a hardware token if an adequate CSP (Crypto Service Provider) for the token is available and installed. Without going into details, there are several CSP libraries and [4] describes a project for mapping the CSP API to a PKCS#11 interface library. Both mechanisms are alike.

- Other operating systems, e.g., Unix, Linux, and MacOS X, allow the usage of a Pluggable Authentication Modules (PAM) to log in to the operating system and for other authentication purposes, e.g., to turn-off a screensaver. The PKCS#11 PAM Login Tools offers a module which can be used for authentication to hardware token with the PKCS#11 interface [30].

After user authentication the module maps the user to a system account. This can be done by using information from the user certificate, e.g., CN (Common Name), UID, or email address, in combination with local or central databases.4

2.4 Our prototyping system

Each of the applications we have discussed in the previous section allows certificate based authentication with hardware tokens and the storage of authorization information in a central database. This leads to a system which works with one single hardware token for all authentication purposes. Figure 4 shows our prototyping system.

A user who wants to access the operating system firstly has to authenticate to the operating system by presenting his hardware token. This token contains a secret key to which operations can be applied. It possesses a certificate signed by the CA. The certificate together with other

4We have extended the LDAP mapper for providing secure and failsafe connections to an LDAP server to obtain the user data from LDAP entries [9].
user information can be stored in the backend LDAP server which can be accessed by the server applications.

The access control module PAM and the servers need the CA certificate to verify user certificates. It should be stored in the system or must be retrievable with applicable methods from the CA. If the CA certificate is locally stored, the applications can verify user certificates without network connection.

The user authorization can be done by means of a local database in the case of “offline” systems. For network systems, storing the information in a central database is more suitable. For instance, an LDAP server, which collects user data (login, UID, email address, service authorization information and — for the old style login — passwords) and certificates, can be used to retrieve the authorization information.

For online usage, a connection to the LDAP server is required. However, the information stored in the LDAP server are only read and not modified during authentication steps. Even if the PIN is changed, the information stored in the LDAP server is not modified as the PIN is contained in the hardware token. For this reason, the data from an LDAP server can be mirrored to proxy servers or an isolated LDAP server which could be used within offline systems.

Note that the system is scalable and easy to administrate. First experiments demonstrate that it is an adequate solution for network authentication purposes.

3 Single-Sign-On

Figure 5 sketches the common use of a hardware token with respect to an end user system. Each application accesses the token by an interface. As already mentioned in Section 2, one of
the mostly used standards is the RSA Security Inc. PKCS#11 Cryptographic Token Interface Standard with Cryptoki as API. This software interface hides hardware details to a large extent. Of course, hardware tokens can also be accessed by using middleware with low-level interfaces, e.g., SCard API based on the Personal Computer/SmartCard specification which has been defined by Microsoft for using smart cards in its operating systems. Such low-level interfaces are quite complex and can be difficult even for experienced programmers. A further problem with these low-level interfaces arises in settings in which multiple applications try to access the token at the same time. For lack of appropriate control mechanisms, either no application has access to the information stored on the token or all applications can grab the vital data even if another user called them. This problem does not arise when using PKCS#11 as it encapsulates each running application in a session so that each application has to authenticate itself to the hardware token. For sake of reliability, most programmers use PKCS#11.\footnote{In most cases, Cryptoki is used as shared library – as adumbrated in Fig. 5 where the box labeled “PKCS#11 Interface” symbolizes the shared library – although linking applications directly to Cryptoki is much more safe. The dynamic approach presumes that the PKCS#11 library is protected against replacement by a malicious library, e.g., one which steals PINs. We will dwell on this aspect later on.}

To access private objects stored on a hardware token, an application which uses Cryptoki has to be firstly authenticated to the token. This is usually done by asking the user for a PIN. Thus the user has to put in his PIN every time an application which has to access the private objects of the token is called. This may be a nuisance. To avoid multiple inputs of a PIN by one and the same user, each user has to be encapsulated in a user session. That way separated from other
users, each user has to put in his PIN only once during a session. After acceptance of the PIN, the hardware token can be used by every application of the said user session without calling back the user.

In the following, we discuss in detail such a token based Single-Sign-On (SSO) approach. Note that the approach presumes that only trustworthy applications in a secure system are allowed to use the token in this way. That is, security risks are involved. They have to be minimized with adequate actions (see Section 3.2).

3.1 An approach to SSO using hardware token

Let us review in detail how Cryptoki is commonly used by an application: (a) The PKCS#11 library is opened and tokens are searched for by scanning the appropriate slots. (b) An application session to the token is started which allows the application for reading the public objects of the token. (c) The user has to be authenticated to the token before the application may access private objects. (d) The logout procedure is called, and (e) the application session as well as the libraries are closed. After the application session has been opened, operations processing public objects stored on the token can be performed. Private objects however are locked until the application calls the login procedure and the user puts in his PIN. Calling the logout procedure makes the private objects inaccessible once again. During an application session the login and logout procedures can be alternately called several times.

Figure 6 shows an outline of a sample program written in C which uses the Cryptoki API. In this sample program, the token is used to check whether the user has the right combination of a not self-signed certificate and a private key. The Cryptoki session of the application provides a logical connection between the application and the hardware token. After the application session has been opened, the application is permitted to access only public objects. To read the private key and gain access to the private objects, the user must be authenticated. This is done by the \texttt{C\_Login()} call which asks for the user’s PIN. After authentication of the user, the application can use the private key and the private objects as often as required until \texttt{C\_Logout()} is called.

To adapt the approach to SSO, the steps 4 to 9 are the interesting one. In Step 4, the login procedure \texttt{C\_Login()} is called. This procedure asks the user for authenticating himself. The PIN is stored in the system’s main memory. In Step 9, the user is logged out and the PIN is removed from the system’s main memory by overwriting it with 0’s. After having called procedure \texttt{C\_Logout()} 6

---

6 Most parameters of function calls and checkings of return values are suppressed for sake of readability.

7 There may be more than one PIN allowing for access to different PIN domains on a token with different private objects. This is not a problem as the find operation only finds the objects to which the user has access.
the PIN is not available to the application any more. Notice that even during the time period between the user’s login and logout (lines 4 to 9) only the corresponding application session is permitted to read private objects on the token. Each other opened application session requires an extra authentication of the user, even if the application is called by one and the same user.

In order to arrive at a SSO approach, a transparent layer between the applications and the PKCS#11 interface has to be introduced. This layer plays the role of a proxy which connects the applications to the PKCS#11 interface. Figure 7 and Figure 8 show our approach which is an extension of the one shown in Fig. 5. We aimed at not having to modify the applications (e.g., Firefox) themselves.

Two different methods have been taken into account by us. The first one we discussed is characterized by a persistent connection between the intermediate layer and the PKCS#11 interface once the user login has been successfully performed. The second one establishes an extra connection for each application.
3.1.1 Persistent connection

As already explained, the user has to log in to the operating system via PAM which asks the user for his PIN and uses objects on the token. After the login has been successfully performed, a common PAM closes the connection to the PKCS#11 interface as the authentication of the user is completed. We have modified our PAM so that an interface application $\alpha$ is started after the login has been successfully performed and the PIN is transferred from the PAM to this new interface application (see Fig. 7 (a)). Application $\alpha$ now opens a session to the token and establishes a persistent connection to the PKCS#11 interface. Each (other) application, which uses a dynamic library to call the Cryptoki functions, connects to a new library $\beta$, which also offers a Cryptoki interface dynamic library (see Fig. 7 (b)). This library is connected to the new interface application $\alpha$. In this way, once $\alpha$ has established a persistent connection to the PKCS#11 interface for a user all the applications of this user can access the private objects on the token by using this connection.\(^8\)

A drawback of the approach is the fact that interface application $\alpha$ has to find out to which user an application belongs to, i.e., it has to undertake the session management, as only applications belonging to an already authenticated user can access the private objects on the token. Implementing such a session management system is quite difficult. For this reason, we have not

\(^8\)Of course, if an application must not be modified it nevertheless pops up a dialog asking the user for the PIN (although it has already access to the private objects of the token via $\beta$ and $\alpha$). The user can confirm the dialog without pin input as $\alpha$ and $\beta$ acts as proxy server. Usually, it is not a great deal of work to modify an application such that the login box is suppressed.
followed up this first approach.

3.1.2 Application specific connections

Another way to arrive at SSO is to use an agent which stores the PIN and is asked when a process needs to authorize and has to present the PIN. In Fig. 8, box \(\alpha\) can play the role of such a Cryptoki agent which stores the user’s PIN. When an application has to authenticate to the token, the Cryptoki agent \(\alpha\) is asked for the PIN by the appropriate procedure of the Cryptoki interface dynamic library \(\beta\).

An application now connects to \(\beta\), which connects itself to the original PKCS#11 interface (see Fig. 8 (c_2)). All library calls but \(C\_login()\) are forwarded to the original interface without modification. If \(C\_login()\) is called, \(\beta\) asks the Cryptoki agent \(\alpha\) for the PIN and inserts the PIN returned by \(\alpha\) into the parameter list of \(C\_login()\).\(^9\) Thus each application has its own connection and slot, token and session handling can be done as usual, i.e., as it would be done if the original Cryptoki library would be used.

The biggest drawback of this second approach is that the PIN has to be stored in the system’s main memory as the PIN has to be internally used every time an application has to authenticate to the token which should be performed without further user interaction. We will briefly discuss this problem in Section 3.2. Another problem is the usage of different PIN domains in different applications. The latter problem can be solved by identifying the login process of the user appli-

\(^9\)As in the first approach, a user application nevertheless pops up a dialog asking the user for the PIN, if the user applications must not be modified. As above, the user can confirm the dialog without pin input.
cation and, if necessary, to ask the user for the new PIN. However, these problems go beyond the scope of this paper.

We have successfully implemented a test system which provides the functionality of this second approach (see Fig. 8). For this purpose we have extended the PAM PKCS#11 library so that a Cryptoki agent is started after a successful login with a given PIN. Among others, Firefox, Thunderbird, and a SSH agent have been used as user applications in the test system. Note that the SSH agent has a similar functionality as a Cryptoki agent. It stores the decrypted secret key which is used every time a SSH client has to authenticate. Of course, the SSH agent can connect directly to the token (dashed line in Fig. 8) as the PIN is stored by the SSH agent itself. However, the SSH agent can also use the Cryptoki interface and act like the other applications which use the Cryptoki interface.

An advantage of our approach we have not yet discussed results from the fact that in our system the PIN is verified during the login process to the operating system. Operating systems give a feedback if the user puts in a wrong PIN. In this way, the user is prevented from putting in the wrong PIN many times such that the hardware token would be locked and had to be unlocked by a special unlock PIN (PUK).

3.2 Security considerations

Using a hardware token for client authentication is a high secure method provided that the mechanism to log in to the token, e. g., by putting in a PIN or using biometrical techniques, are secure, too, and that there are no security bugs in the client software. Thus, the PIN has to be transferred in a secure way to the token and may not be stolen during input. Using the input panel of class two reader\textsuperscript{10} for PIN input and direct insertion of the PIN into the data transfer to the card would ensure this. If there is no other input mechanism as using the keyboard of a computer, then the PIN could be stolen by key loggers or other spy software in the system.

But even if the PIN has been stolen, the attacker cannot take advantage of the PIN until he also got the hardware token or had control of the system with the right inserted token as he cannot extract the private keys from the token. Stealing the hardware token without knowledge of the PIN does not help much, too. Using wrong PINs with a token several times locks the token. Thus, the security of a system using PIN based hardware tokens is much more higher than security of password secured systems.

\textsuperscript{10}There are several classes of card readers. Class two readers have a keypad for secure PIN entry. The keypad cannot be accessed by the computer.
If one identifies a token secured system as infected by a key logger virus or if there are other signs of compromised PINs, the user can choose from two alternatives: he could change the PIN or the certificate of the card could be revoked. Revocation would make the token unusable to all systems, which check the certificate online, beginning from the time of revocation. After a new certificate has been issued and stored on the token, the token can be used again.

In combination with our SSO solution we have to investigate how to ensure that the PIN stored in the system's main memory by the Cryptoki agent after the user has successfully logged in to the operation system cannot be used by unauthorized application. To get this point under control, the system has to check whether the source of the connection is an authorized application when a connection to the interface library $\beta$ is done. Authorized applications could be defined by an administration using software signatures or checksums as footprint. The agent could check the footprint and ensure that only defined applications are accepted. Another method could be checking the path of the calling program only.\textsuperscript{11}

Another important item directly concerns the storage of the PIN in the system's main memory. This location has to be protected against malicious attacker programs. As the PIN is eventually required in its clear non encrypted form, the PIN has to be stored in memory in a way, that it can be read from the agent. To ensure that not a scan of the whole main memory would show the PIN in clear form, the agent could generate a key, encrypt the PIN with this key and decrypt it when necessary. By this action, it is rather difficult to get knowledge of the PIN as the key has to be located and extracted, too.

The agent should detect the removal of the hardware token. When the token is unplugged, the PIN should be deleted from the system's main memory. In addition the user could be logged out by the operating system or a screensaver could be launched which requires the input of token and PIN to reconnect to the user session.

To ensure that a stolen PIN cannot be used to generate user signed messages, authentication and signature certificates should be split such that there is a user certificate for authentication purposes and another user certificate for signature purposes. As only the PIN for authentication is copied into the system's main memory by the agent – the PIN for signature purposes could be put in by using an input panel of class two reader and transferred directly to the hardware token – it is quite difficult for an attacker to steal both PINs. This action makes the system more secure without loosing the SSO capabilities and may make it uninteresting to attack the system.

\textsuperscript{11}All these actions are not appropriate if attackers have administrative privileges. However, system administrators have much easier ways to steal PINs, e.g., by installing a key logger.
Eventually, in SSO environments one can expect qualitatively good digit combinations as PINs from the users because a PIN has to be put in only once during a user session. Long combinations make it even hard to catch a glimpse of the right combination.

Securing the system as described should lead to an SSO environment without (much more) higher security risk than the risk of a system in which the PIN is asked for every time a Cryptoki session is started.

4 Résumé and Conclusions

The approach described in this paper allows to use one and the same personalized hardware token for authentication to several systems. The systems themselves do not need to be connected to a central server which undertakes the task of authenticating the users. In fact each system could authenticate a user by using the user’s personalized hardware token which is plugged into the system. Above all the different local systems could apply different methods to decide whether a user can be accepted. The authorization information could be stored in a database either located in the local environment or in a central database, e.g., an LDAP server could be used which results in a complete system with local authentication and central user management. The programs of the applications need not to be modified in order to be used in this setting.

The system can be upgraded in order to provide SSO without a central login or ticketing system, which is required by other SSO systems, e.g., kerberos [19, 23]. Modifications to the network clients and server software are minimal, if at all necessary. The patches used, e.g., the X509 patch for OpenSSH, are widespread well known regular upgrades. Only the extended token login application which provides the SSO agent is a new system software which has to be installed on the client. This whole SSO system runs on client side and could coexist with other systems.

Some problems arise with respect to the expiration date of certificates from the used CAs. If certificates are not checked online but offline by the system, it has to be ensured that the user certificates are renewed with the new CA key early enough, i.e., long before expiration of the old CA key (and use overlapping periods of validness time). Short-term updates of the revocation lists are necessary, too.

References


[7] **PKCS #15: Cryptographic Token Information Format Standard.**

[8] **PKCS #11: Cryptographic Token Interface Standard.**


Information about the Author(s)

Sandro Wefel
Institut für Informatik
Universität Halle
06099 Halle, Germany
E-Mail: sandro.wefel@informatik.uni-halle.de
WWW: http://www.informatik.uni-halle.de/ti/mitarbeiter/wefel

Paul Molitor
Institut für Informatik
Universität Halle
06099 Halle, Germany
E-Mail: paul.molitor@informatik.uni-halle.de
WWW: http://www.informatik.uni-halle.de/ti/mitarbeiter/molitor