MOC via TOC Using a Mobile Agent Framework

Stefano Bistarelli[1,2], Stefano Frassi[2] and Anna Vaccarelli[2]

[1] Dipartimento di Scienze, Università "G. D'Annunzio" di Chieti-Pescara, Italy
bista@sci.unich.it

[2] Istituto di Informatica e Telematica, CNR, Pisa, Italy
{stefano.bistarelli,stefano.frassi,anna.vaccarelli}@iit.cnr.it

Abstract. A novel protocol is proposed to address the problem of user authentication to smartcards using biometric authentication instead of the usual PIN. The protocol emulates expensive *Match On Card* (MOC) smartcards, which can compute a biometric match onboard, by using cheap *Template on Card* (TOC) smartcards, which only store a biometric template. The biometric match is performed by a module running on the user's workstation, authenticated by a mobile agent coming from a reliable server. The protocol uses today's cryptographic tokens without requiring any HW/SW modifications.

1 Introduction

Smartcards are currently used as a secure and tamper-proof device to store sensitive information such as digital certificates and private keys. Access to smartcards has historically been regulated by a trivial means of authentication: the Personal Identification Number (PIN). A user gains access to a card if he/she enters the right PIN. Experience shows that PINs are weak secrets in the sense that they are often poorly chosen, and that they are easy to forget.

Biometric technologies have been proposed to strengthen authentication mechanisms in general by matching a stored biometric template to a live biometric template [1,2]. In the case of authentication *to* smartcards, intuition imposes the match to be performed by the smartcard chip. However, this is not always possible because of the complexity of biometric information such as fingerprints or iris scans, and because of the still limited computational resources offered by currently available smartcards.

In general, three strategies of biometric authentication can be identified.

Template on Card (TOC). The biometric template is stored on a hardware security module (smartcard or USB token). It must be retrieved and transmitted to a different system that matches it to the live template acquired from the user by special scanners. Cheap memory-cards with no or small operating systems are generally sufficient for this purpose.

Match on Card (MOC). The biometric template is stored on a hardware security module, which also performs the matching with the live template. Therefore, a microprocessor smartcard is necessary, which must contain an operating system running a suitable match application.

System on Card (SOC). This is a combination of the two previous technologies. The biometric template is stored on a hardware security module, which also performs the matching with the live template, and includes the biometric scanner to acquire, select, and process the live template.

Clearly, the third of the strategies sketched out above is the best in terms of security as everything takes place on card. Embedding a biometric reader on a smartcard offers all the privacy and security solutions but, unfortunately, it is expensive and presents more than one realization problem.

The benefits derived from MOC cards are valuable in themselves: using its own processing capabilities the smartcard decides if the live template matches the stored template closely enough to grant access to its private data. Nevertheless this scheme presents a danger: we have no certainty that a biometric reading has been collected through live-scan and there is the risk of an attacker's sniffing the biometric and later using it to unlock the card in a replay attack.

In the present setting, how can we implement biometric authentication on smartcards that are already commercially available?

We address this issue by developing a novel protocol that employs inexpensive TOC cards as if they were MOC cards and that counterbalances the MOC technology's drawbacks; the requirements of the present work are to employ common crypto smartcards without modifying the code inside them and without asking the user directly for the PIN.

This paper is organized in the following way: Section 2 illustrates the security problems using TOC for authenticating a user to a smartcard. Section 3 sketches out the adopted solution. The protocol is introduced in Section 4, while implementation details are described in Section 5. Section 6 illustrates the solved/unsolved security problems. Finally, Section 7 concludes the paper and proposes possible future works.

2 Security Problems Using TOC Technology

Before describing the protocol, we would like to explain some problems related to the use of TOC technology for authenticating a user to a smartcard (SC). There are several points of attack in the use of TOC technology without securing the data transmission between the biometric device, the smartcard reader and the local host that carries out the biometric match. Consequently, we have to consider some aspects before designing our secure protocol.

The idea of using TOC technology to authenticate a user to a SC (without security concerns) is:

1. A cryptographic application asks the user to authenticate himself to the SC via a specific API call.
2. Verification Module reads the biometric template from the SC.
3. A real time template is acquired from the user using a biometric scanner.
4. A biometric match between the two templates is performed on the local host.

5. If the biometric match is successful then the actual PIN is submitted to the card to unlock the crypto chip.

A diagram of the protocol is illustrated in Fig. 1.

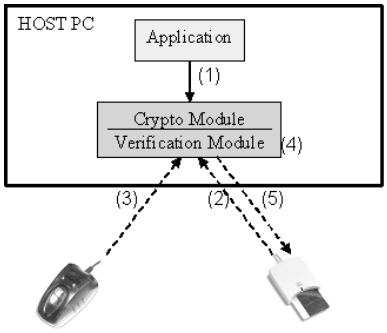

Fig. 1. A TOC Protocol

The major problems of the above protocol are:

- The enrollment template stored inside the smartcard could be eavesdropped at step 2.
- The real time template could be eavesdropped at step 3.
- The smartcard doesn't trust the module performing the match at step 4.
- Where to store the secret login data (PIN) used to actually log the user into the cryptographic chip (we don't want to ask the user for it).

The first two points are critical if it is possible for an attacker to use the smartcard (after he/she has stolen it) for a replay attack sending again the eavesdropped data directly to the verification module. In this case, there is no security mechanism to verify that the biometric verification data are derived from an actual live presentation to the biometric sensor. To solve these problems we could encrypt the data exchanged between the devices and the crypto library or find a way to trust the module that acquires the live template.

SC is unable to authenticate the verification module. Maybe using a kind of challenge-response protocol: the module requests a random to the SC and this is returned encrypted with a shared secret key. Now the dilemma is where to store the key on the host.

The only method to unlock the private area of the chip is to supply the exact PIN to the SC. If the PIN is correct then the SC trusts the module that has performed the local biometric match.

Therefore, the crucial point is the last one: where to store the secret PIN[3]. An obvious method is to store the PIN inside the compiled Crypto Module. This

[3] In a previous work [3], a similar problem has been investigated developing a comparable protocol. In that case, the user was asked directly for the PIN and there was the need to install a piece of code into the smartcard to carry out the protocol.

is not a good solution because a malicious user might do reverse engineering on the library and find the secret. Every place inside the user's file system is not secure if a malicious program has manipulated the host, so the safest place is inside a protected remote Server.

3 The Adopted Solution

Now the problem to solve is how the remote Server can send critical data to an unknown remote host. This is like a black box and the server does not know if a malicious entity is running on that client.

A mutual authentication by establishing a SSL connection between the client and the server is a good solution, but like the PIN, there is always the recurrent problem of where to store the certificate/private key of the client (we don't want to use another smartcard [11] and we want to avoid asking the user for another PIN to unblock this private key).

We chose to adopt another solution: using a Mobile Agent framework. If the server cannot trust applications running on the client, it will trust the code that it launches to the client: a mobile agent.

A remote agent, launched from the secure server, will try to authenticate the module that executes the biometric match on the client; if the result of the authentication is positive then the server sends the secret PIN to the client via a previously opened secure connection.

The chosen framework was **SeMoA** [6] (Secure Mobile Agents). It is a runtime environment for Java-based mobile agents in development at the Fraunhofer Institute for Computer Graphics, with its main focus on security.

A Mobile Agent is a software entity that is not bound to the host where it begins execution, but has the unique ability to travel across a network and perform tasks on machines that provide agent-hosting capability. Unlike remote procedure calls, where a process invokes procedures of a remote host, process migration allows executable code to travel autonomously and to interact with the hosting machine's resources, including other mobile agents. Therefore, a Mobile Agent framework has to cope with various security threats [8]: malicious agents might try to break into the server in order to harm other agents or to gain unauthorized system access. A malicious host could tamper with agents. Agents might be sniffed while they are transferred over the network.

Many open source agent development frameworks are available on the internet: we decided to adopt SeMoA because it focuses on security and tries to solve the above-mentioned problems.

4 Protocol Description

This section presents the protocol. It illustrates the interactions among the main entities (details will be introduced in section 5). Description of the entities:

- Client: the user's workstation

- Application: the user application that requires the access to the smartcard through the Crypto Module (for instance a digital signature application).
- Crypto/Verification Module: the main entity used to access the smartcard and to perform the local biometric match (it implements the client-side protocol).
- SeMoA Framework: the runtime environment for the Mobile Agent coming from the Server.

– Server: the secure host where the secret login data is contained
  - SeMoA Framework: the runtime environment for the Service implementing the protocol.
  - MOC Service: the Service which accepts connections and implements the protocol.
  - MOC Agent: the Mobile Agent that is launched by Moc Service, gets to the Client and comes back with the result of the authentication.

A diagram can be defined as in Fig. 2.

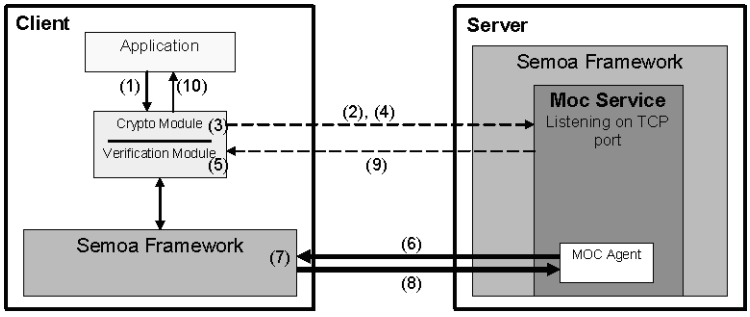

Fig. 2. the Protocol

1. User application requires access to the private space of the smartcard through a particular Crypto API call.
2. The Crypto Module (CM) opens an encrypted connection to the Moc Service ($MocS$) running on the Server ($MocS$ is authenticated via SSL).
3. After the connection has been established, CM generates a large Random value and stores it inside itself (value used by the mobile agent at step 7).
4. Finally, CM sends the Random and the smartcard Serial Number to $MocS$: if the subsequent controls succeed, CM will receive the secret login data at step 9, otherwise $MocS$ will close the connection.
5. In the meantime, the Verification Module executes a biometric match between the template stored inside the smartcard and the live acquired one.
6. $MocS$ stores the Random previously received within a MOC Mobile Agent and launches it to the Client address over a new encrypted channel; after this, $MocS$ begins to wait for the return of the Agent.

7. Now the MOC Agent is migrated on the client:
   - It tests the validity of the modules residing on the user's workstation checking their digital signature.
   - It checks that the random inside itself has the same value as the random contained in CM.
   - It ensures that the biometric match executed at step 5 is successful.

   If all the previous controls are positive then we can trust the CM module that has started the protocol.
8. The Agent comes back to the Server and returns the result to $MocS$.
9. If the result is positive then $MocS$ sends, on the same connection opened at step 2, the secret login data (PIN)[4] correlated to the Serial Number received at step 4. Otherwise, it closes the connection with the client.
10. In the last step, CM unlocks the private area using the received PIN and confirms to the user application the success of the Crypto API call made at step 1.

5 Implementation

The protocol and all the entities have been developed and deployed on a Windows 2000 Professional workstation, so some details are particular to this architecture. As regards the hardware, the biometric scanner employed is an FX2000 produced by Biometrika srl [12], while the smartcard used is a Cyberflex e-gate produced by Schlumberger [13]. (As we will see later, it is possible to employ any kind of biometric device or smartcard without modifying the protocol by only changing the respective library.)

The principal technology employed, besides SeMoA, is the **PKCS#11** standard [7], which has been used as the Crypto module. We have chosen this solution because this is the most widespread de-facto standard in today's cryptographic tokens. The PKCS#11 standard specifies an API, called "Cryptoki" (cryptographic token interface), to interface the devices that hold cryptographic information and that perform cryptographic functions. The Cryptoki is important because it isolates an application from the details of the cryptographic device.

The standard employed to perform all the required biometric operations is **BioAPI** [4]. This API is intended to provide a high-level generic biometric authentication model, covering the basic functions of Enrollment, Verification, and Identification.

Another technology we have used to implement this protocol has been the **Java Native Interface** (JNI) [9]. This was used to exchange data between the MOC Mobile Agent (which runs in a Java virtual machine) and the dlls (which are native libraries) at step 7 of the protocol.

[4] The PIN can be delivered directly by the Agent at the end of step 7 (only if all the checks are positive). We have avoided this solution because even though it is the fastest, it is the least safe too: the agent might be tampered with by a malicious entity, to extract the PIN.

Figure 3 describes the protocol in more detail. The previous client's Crypto-Verification module has been separated in four different dynamic link libraries (dll):

1. PKCS#11 module: this is the library provided by the smartcard manufacturer. This dll permits user authentication to the smartcard using the normal PIN. Therefore it is possible to switch from a smartcard brand to another by only changing this module.
2. Crypto Wrapper: this is a dll wrapper to the PKCS#11 module; all the API calls are proxied to the manufacturer's dll except for the C_Login function: this is the point where the client-side protocol is implemented.
3. Verification module: it performs the local biometric match via the BioAPI library. Also in this case it is possible to use another Biometric device by only changing the Biometric Service Provider (BSP) dll.
4. JNI stub: this module is used by the Java Mobile Agent to access the Crypto Wrapper and the Verification module. It works with JNI.

Inside the server there is a certification authority (CA) which is used to issue certificates for the users. The CA also issues Attribute Certificates containing the enrolled biometric template [1]; they are stored in the smartcards along with the x509 user's certificates.

Every client's dll which performs the protocol is digitally signed [5] by the Server's private key, so that the components can mutually authenticate each other and the Mobile Agent can check their validity.

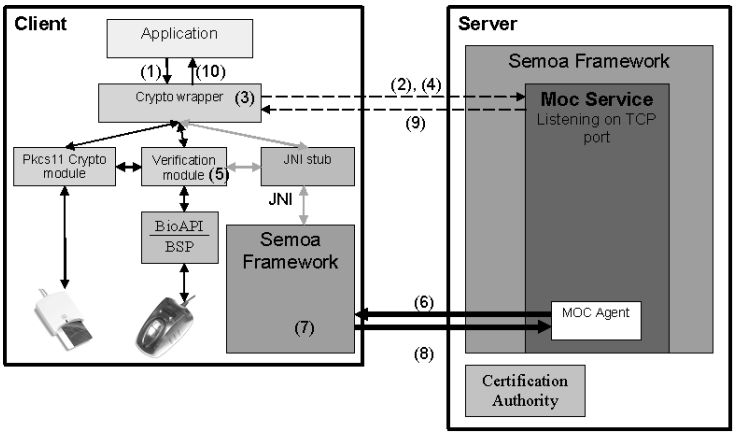

Fig. 3. the detailed Protocol

5.1 Detailed Protocol

1. The user application requires access to the smartcard via a C_Login call. (The application needs, for instance, to use the user's private key stored in

the smartcard). The call corresponds to a C_Login(NULL), where NULL means that biometric authentication is requested (no PIN is given).
2. The Crypto Wrapper (CW) opens a SSL connection to the Moc Service ($MocS$) running on the Server. Naturally, an encrypted connection is used to avoid sniffing the data when the secret login PIN is sent over the channel. We use SSL server authentication to check the server's identity. (NO SSL client authentication, because there would be the recurrent problem of where to store the client's private key.)
3. After the connection has been established, the CW generates a large random value and stores it inside a Shared Data Section [10]: all the processes that will use this module, will access the same variable. In this way, we can detect possible malicious modules that try to start the protocol. (The only module that is permitted to start the protocol is the CW). The random will be checked by the mobile agent at step 7.
4. Finally, the CW sends the Random and the smartcard Serial Number[5] to $MocS$: if the subsequent controls succeed, the CW will receive the secret login data at step 9, otherwise $MocS$ will close the SSL connection.
5. In the meantime, the Verification Module (VM) executes a biometric match between the template stored in the smartcard and the live acquired one. VM reads the Attribute Certificate stored inside the smartcard, verifies its validity and extracts the biometric template. Then, VM acquires the live template from the scanner via the BioAPI module only if the BSP's digital signature is correctly verified.
6. $MocS$ generates a MOC Mobile Agent, signs it, and launches it to the Client address (the Random previously received has been stored inside the Agent); after this, $MocS$ begins to wait for the return of the Agent. Also in this case, the agent is sent over an encrypted channel.
7. Now the MOC Agent is migrated on the client. The SeMoA environment checks the digital signature of the Agent to see if it comes from the trusted Server; if so, then:
   - The Agent tests the validity of the dlls checking their digital signature (the dlls reside on the user's workstation at a precise path).
   - Using Java Native Interface, it checks that the random inside itself has the same value as the random contained in the CW. The Agent reads the random value using a new function created in the CW.
   - Using Java Native Interface, it ensures that the biometric match executed at step 5 is successful.

   The check of the random value has been employed to verify that the correct CW has started the protocol. If the random is different, it means that a malicious entity is trying to deceive the Server to steal the secret PINs.
   The digital signature of the modules is checked to ensure that only trusted dlls are carrying out the protocol. Finally, the last check verifies that the proper user is accessing the smartcard. If the above checks are positive only then we trust the CW module that has started the SSL connection.

[5] There is a DataBase installed inside the Server which contains the corresponding unique PIN for every serial number. The serial is used at step 9.

8. The Agent comes back to the Server and returns the result to $MocS$.
9. If the result is positive then $MocS$ sends, on the same SSL connection opened at step 2, the secret login data (PIN) correlated to the Serial Number received at step 4. Otherwise, it closes the connection with the client.
10. In the last step, CW unlocks the private area using the received PIN (it performs a C_Login(PIN) calling the PKCS#11 module) and confirms to the user application the success of the C_Login call made at step 1.

6 Security Problems Resolved

If the attacker *has not stolen* the user's smartcard, he could try to get possession of the PINs residing in the safe Server:

- If no smartcard is inserted, the protocol will not start.
- If a wrong or malicious dll dealing with the communication with the SC is installed, the Mobile Agent will notice it.
- If a different module from CW tries to connect to $MocS$, the Mobile Agent will notice it using the random comparison.
- If he is using a brand new smartcard, with a proper serial number, the server will not return the PIN: the malicious user is not able to pass the local biometric match. (Inside the smartcard there is not an Attribute Certificate containing the fingerprint template issued by the Server's CA).

If the attacker *has stolen* the user's smartcard and the right dlls are installed:

- he could try to change the template stored in the SC: he cannot do this, because the biometric template is contained in an Attribute Certificate signed by the Server CA.
- Even if a biometric template has been previously sniffed, it is not possible to inject it within the Verification Module: before the VM acquires a template from the biometric device, it verifies that the Biometric Service Provider dll is the trusted one via digital signature (no more replay attacks).

6.1 The Two Feasible Attacks

The possible attacks to the implemented protocol concern how the PIN is transmitted in the final step, and the malicious host threats [8] in a mobile agent framework. In the first case, the PIN could be sniffed if the channel between the SC reader and the host is not protected. If the attacker has stolen the user's SC, he could bypass all the protocol and use only the manufacturer PKCS#11 library. We assume that this is not possible because we rely on the smartcard producer's Crypto Module (a trusted path between the SC and the host should be employed).

The other way to attack this protocol is modifying the SeMoA framework and/or the Java Virtual Machine on the client's workstation. This is a common problem in the Mobile Agent Systems field [8]. In our case, an attacker succeeds

if he is able to alter the Mobile Agent's return value with a positive result even if the checks on the client are negative. This problem can be solved by employing a mutual authentication protocol between the client/server SeMoA environments, using a secure trusted hardware: it would be used to store the framework private key and to check the validity of the agent runtime environment.

7 Conclusions

Modern, inexpensive TOC smartcards cannot compute a biometric match like MOC smartcards. We have developed a protocol, which simulates the MOC strategy through the use of TOC cards. In practice, the actual match is delegated to a module of the card host after an authentication performed by a mobile agent coming from a secure server.

The design we have presented has been fully implemented using the SeMoA framework, which provides an open source mobile agent system, and through two de-facto standards such as PKCS#11 and BioAPI, respectively used to communicate with the crypto smartcard and to interact with the biometric functions. The use of these standards lead to an implementation where any smartcard and any biometric device can be used.

Potential future works will concern addressing the issues described in Section 6.1 (local PIN sniffing and malicious host attack), and adapting the protocol in other areas where the entity performing the biometric match is not trusted.

References

1. L. Bechelli, S. Bistarelli, F. Martinelli, M. Petrocchi, and A. Vaccarelli. Integrating biometric techniques with electronic signature for remote authentication. *ERCIM News*, (49), 2002.
2. L. Bechelli, S. Bistarelli, and A. Vaccarelli. Biometrics authentication with smartcard. *Technical Report 08-2002*, CNR, IIT, Pisa, 2002.
3. G.Bella, S. Bistarelli, and F. Martinelli. Biometrics to Enhance Smartcard Security (Simulating MOC using TOC). *Proc. 11th International Workshop on Security Protocols*, Cambridge, England, 2-4 April 2003.
4. BioAPI Consortium. BioAPI Specification Version 1.1. http://www.bioapi.org
5. Authenticode, *http://msdn.microsoft.com/workshop/security/authcode/signing.asp*
6. V. Roth and M. Jalali. Concepts and Architecture of a Security-centric Mobile Agent Server. *IEEE Proceedings of 5th International Symposium on Autonomous Decentralized Systems (ISADS01)*, pages 435-442, Dallas, Texas, March 2001.
7. RSA Laboratories. PKCS#11-cryptographic token interface standard.
8. E. Bierman and E. Cloete. Classification of Malicious Host Threats in Mobile Agent Computing. *Proceedings of SAICSIT 2002*, pages 141-148, 2002.
9. Java Native Interface, *http://java.sun.com/docs/books/jni/index.html*
10. How To Share Data Between Different Mappings of a DLL. Microsoft KB 125677
11. U. Waldmann, D. Scheuermann and C. Eckert. Protected transmission of biometric user authentication data for oncard-matching. *Proceedings of SAC 2004*, pages 425-430.
12. Biometrika srl, *http://www.biometrika.it*
13. Schlumberger, *http://www.axalto.com*