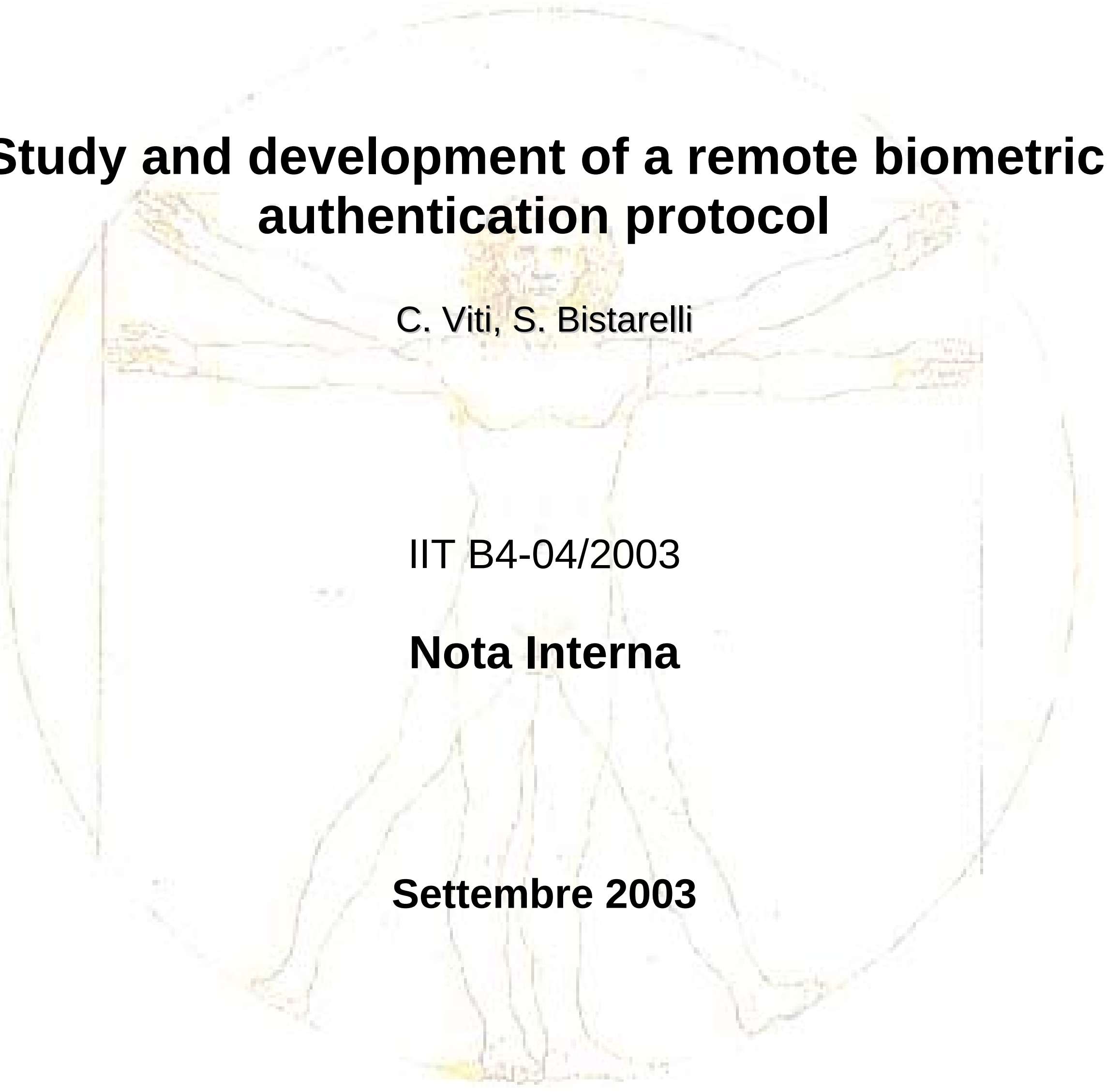
Consiglio Nazionale delle Ricerche



Study and development of a remote biometric authentication protocol

C. Viti, S. Bistarelli

IIT B4-04/2003

Nota Interna

Settembre 2003

Istituto di Informatica e Telematica

Abstract

This paper reports the phases of study and implementation of a remote biometric authentication protocol developed during my internship at the I.i.t. of the C.n.r. in Pisa. Starting from the study of authentication history we had a look from the first system used since the 60ies to the latest technology; this helped us understand how we could realize a demonstration working protocol that could achieve a web remote authentication granting good reliability: to do this we choosed to modify the SSL handshake with biometric tests and we decided to use smart-cards a secure vault for the sensible biometric data involved. In the first chapter you will find a brief definition of authentication and an introduction on how we can achieve it, with a particular focus on new biometric techniques. In the second chapter there's the history of authentication from the very first password system to actual ones: new token and smart card technolgies are longer stressed in order to introduce the reader to the last chapter. In the third chapter you will find the project framework, the development of our implementation choiches and the source code of the demo project.

Index of Contents

1 Introducing authentication

1.1 The reasons why authenticating users is necessary

Actually many enterprise or public networks are projected and built to work on-line with the WWW: that is because the spreading of e-commerce and web technologies among the people, due to technology itself improving and costs decreasing, has led to a subsequent stronger use of IT medias to purport many daily operations previously performed on-site.
Obviously the greater and greater mass of sensible data published on the web or stored on a corporate database has to be protected properly: once operations took place on site and the authentication (recognition we'd better say) of the user was easy; now the service-provider and the service-user (commonly addressed as server and client in web-language) interact never seeing or meeting each other and the problem of trustful, reciprocal recognition is quite huge: privacy and security concerns are strong both for incorporates (whose private data are related to their business activities) and PA (whose sensible data implies strong privacy concerns for the citizens they represent and serve).
Moreover for a big company with many employees the control of their rights over certain data is a strict necessity: easily guessable not all the research-lab database should be browsable through the web nor possibly accessible from all insiders, but it's going to be used and administrated only by few authorized users who must be able to prove doubtlessly their rights to the system daemon before they could interact with it.

1.2 Authentication is not Identification

All of the previous security concerns are authentication problems and they make us aware why authentication is important: it is the process of uniquely associating users to their rights and capabilities in a trustful way developing all possible protocols and mechanisms to determine if the identity they claim is their own one. But through years and technology improving misunderstanding the process of identification with the authentication one is almost common sense: despite this in the subsequent pages we would like to focus on authentication and the next paragraph could help the reader determining the topic of the paper.

1.2.1 Identification and Authentication

Namely the process of identification is a one-to-many match test while authentication is a one-to-one comparison. Even though the perception retrieved by the user could be likely the same (submitting some credentials) these processes involve huge differences referring implementation, protocols, and performances. Authentication could be addressed as an identification subroutine and in some cases this could be real but this is not always true. Obviously the identification processes involve database's queries in a larger measure but the main difference is due to the level of reliability the process must grant to the system administrator. Users to be identified do not claim their identity: the system searches, among a determined group of profiles, the one matching with these credentials and optionally performs authentication, submitting whatever else request to the user. When a user tries to authenticate on a system he will submit two factors: an identity (coded as a username or wired on a token for example) and then some credentials to prove he is the one he pretends to be. The security focus is greater in the second scheme because of the two factors process and due to the fact that a check on determined data could take place in a more accurate way then a long search (resulting not time-consuming) and because this process involves a larger user-training interacting with the system. Of course identification is generally only one part of an embedded security system operating on a workstation or a system: in the following paragraph a description of this subsystem is briefly undertaken.

1.2.2 Authentication, Authorization and Accounting

Authentication is only one part of the security functions usually set up on systems.
The problem of trustful recognition during a e-communication is the first step of any act a user, on behalf of some credentials, can perform within a system. As well as in social interaction the principals shall recognize each other (Authentication), subsequently they would act according their confidence degree or their business relationship (Authorization) and then they should constantly check out how their meeting is evolving and remember what they have been speaking about (Accounting).
So **AAA** is a term for a system to smartly controlling access to computer resources, enforcing policies, auditing usages and providing any relevant information concerning users' actions.

i. *Authentication*.
   An authentication process provides the techniques to certainly identify a user, granting him access only once he has been able to exhibit his unique set of criteria which he previously gained, according a secure protocol, from a principal of the communication.

ii. *Authorization*.
   After users get logged into a system we have to state what type of actions, activities and services the user trustfully authenticated has right to perform. This is the process of enforcing policies within the system providing user the rights they possess according his authenticated identity. This technique allows smarter administration using multiple group-profiles for example.

iii. *Accounting*.
   A set of statistic controls focuses on the amount of system time, data and resources a user is sharing. Through processing these information the system is able to plan and optimise its use, as well as to protect itself from atypical profiles: auditing authenticated users' profiles allows to verify abnormal usage.

The best way to integrate AAA process depends on how authentication is performed, and on the specific implementation, that is the object of next paragraphs: easy guessable no security could be achieved when system's user is not safely and certainly identified.

1.3 Techniques, factors and protocols to achieve authentication

In this paragraph we try to evaluate what kind of peculiarities an authentication scheme should have and how we could achieve these performances to be pursued efficiently. Starting from a list of desirable topics we're then going to determine how we can achieve authentication and what kind of authentication.

1.3.1 The goals for an authentication scheme

We would like to focus our attention on what kind of requirements, utopistic or not, an authentication process should satisfy in order to accomplish as smartly as possible its tasks:

i. Users should be registered only once, no matter how many system they may need to use in order to perform their function. Supposing A-company owns five server and the employee user has to work with all of them to pursue his task: he will be registered only once and a mechanism of trust-chain should be developed through the servers to allow him to be recognized equally by and authenticated on any of those.

ii. The process should be conducted promptly and efficiently: there's no advantage in getting money from an ATM if the authentication protocols take half an hour to let you interact with your bank's server.

iii. Once they are registered users should have to log-on the system, submitting their credentials, interacting with a user-friendly daemon. Both for ease of use and user-guidance the authentication subsystem should be designed with a framework dedicated to user-acceptance and assistance. For example users should authenticate only once per session and have their identity confirmed as long as that session lasts. A system that asks you your credentials every time you turn to the next page on the same web site would be useless and time-consuming, as well as a system pretending users to remember 64 digits alphanumeric passwords

iv. The authentication process has to be effectively able to determine whether prospective users are registered and they really are whom they say to be and not, for example, the wife of an employee trying to get access to the enterprise database to check her husband's dater.

v. The process shouldn't be time or cost consuming above the company enforcing policies: none would realize an authentication system that guarantees absolute perfection in user-authentication but that needs 15 minutes for each user to be granted access and many additional expensive hardware infrastructures to accomplish that.

vi. Soundness: the system should grant, according to the site where is built, consequent soundness and provide adequate protection to the authentication devices. Supposing a strong authentication system is implemented to gain access from the street, through the front door of the directive building, in a company establishment, this system has to be rain-proof, tamper-resistant, temperature independent and tuned to perform authentication even until a certain dirt-degree of the devices.

vii. Security: an authentication system is part of a security system and should grant some guarantees. Supposing the authentication system protects a high confidence archive that is often accessed by many users the availability of the system should be nearly 24/24 unless we accept to deny all access to the data (or grant anyone reading rights) during the periods when the authentication system is down. Supposing the authentication system eventually crashes down more than a reserve method should be implemented to protect the private data as well as not to lock the productive process of the company. Even the integrity of the data managed by the authentication system should never be violated by the system itself unless specific policies of suspension or enrollment are implemented. An authentication system is to be able to provide secure recognition under many circumstances providing recovery protocols even under particularly hard circumstances.

Every resource and every environment requests different authentication protocols: there's not one authentication methodology that fits all kind of data and any kind of system user so that special care should be taken in projecting a determined identification system customizing protocols according to their future use and requested strength.

1.3.2 Many kinds of user-authentication .

These topics should be differently stressed whenever we distinguish between local and remote authentication whether we talk about static, rigid or continuos authentication. We can easily pick it up: the strength of the protocols used depends on what we are defending and from who.

1.3.2.1 Local vs. remote

In a dedicated and protected environment authentication will take place on trusted networks by authorized users to give them access to those resources they can interact with. In this situation you would probably perform physical access to the structure where hosts stand and moreover every authentication session takes place on a machine that you consider trustworthy because it's held under your administrative control. This situation seems (we should always remember that the common idea that evil is out of our enterprise is quite wrong: the 90% of attacks to our IT systems is taken by insiders!) to be much more friendly than the classic remote authentication scheme where you can't trust the link between your host and the other one: that could be a tampered, totally unsecured channel or a network monitored by an eavesdropper trying to steal your credentials to fraudulently use them for his purposes. Outside your organization's area you can't perform any kind of physical access control so that definitely you don't know who's sitting in front of the host contacting you : this is possible within your company where this could be achieved but providing a large amount of money (might be even discarding user-friendliness from your employees' perception, forcing them to many controls and procedures to let them reach their workstation): so the authentication session is exposed to many kind of hacking attacks able to fool your system. This scenario presents many more risks than an intranet or local authentication scheme and still represents a great challenge for security experts to succeed in uniquely and securely granting access only to authorized users. Software devices and protocols should be designed to discard the flaws of the following dangerous situations:

-Communication privacy: according the fact that the communication takes place on third party's channels and lines, that we cannot control, the parts of an authentication session could be eavesdropped and recorded for further fraudulent uses and we would have no perception of this.

-Authenticator's authentication: if we authenticate outside of our intranet towards a server we will be obliged to trust that server, submitting it our credentials, and we will

authenticate on that server but who’s going to authenticate that server towards us and make us aware that is not a fake application trying to steal our profile? The problem is quite huge.
-Authentication channels and services availability: supposing we refer to a third party to authenticate towards some services we use for our business, once the authentication server is down will the service-company allow us to interact with its services or will those services be denied? This could be a great challenge if we consider health-care problems and hospitals’ computer networks. Inside our company we could be able to provide many recovery protocols but outside our control everything could happen that would stop the service-availability or that could interrupt the authentication channel.

1.3.2.2 Static, rigid or continuos authentication

These adjectives don't define three different techniques but three different ways to authenticate users.
In a Static authentication scheme once the user has been certainly recognized the subsequent communication between client and server takes place in clear, without any further control being performed until the session comes to an end.
A rigid authentication scheme provides authentication along a secure channel that allows performing the communication session between the two principals: this technique guarantees a secure channel to exchange sensible data throw it. This channel is kept alive until the session ends.
A Continuos authentication scheme establishes a memoryless secure channel: for any single part of the communication the principals have to prove each other their identity again: this scheme may result slow but affords even stronger security than the previous ones. Practically both client and server never trust each other without a previous evidence of identification had been shown by the other principal: this happens every single time they just exchange one message.
Of course these different typology of authentication implies different protocols to be developed and needs different medias to have these ones implemented.

1.3.2.3 On-line and Off-line authentication

Due to many different protocols implemented through the years authentication could be achieved on-line or off-line. This could seem a useless definition but providing the differences between using web procedures or relying on third parties static factors implies huge differences. In fact, according to the different status of the protocol we could face different kind of dangers: the interaction with live principals allows us to better discard any kind of attack that use old credentials (such as replay attacks exposed in paragraph 2.1.1) but expose our workstation to all the kind of attacks that are performed live such as man in the middle or interleaving ones (see paragraph 2.3.2). On the contrary off-line authentication does avoid all the problems related to networks’ sniffing, eavesdropping and tampering over unsecured channels else making the system more prone to any kind of attacks that relies on no third party live interaction.

1.3.2.4 User or workstation authentication.

Careful readers may have noticed that no emphasis has been posed on user whether workstation authentication but this is a huge problem. Every single protocol that will follow is to be observed through this lens in order to understand whether we are authenticating a user or his workstation. We are used to think about a user and his workstation as one thing but in large environments even a quick coffee-break could result in a dramatically serious threaten. Once a user has successfully authenticated himself for the valuable data stored on a workstation or collected on a server opened working session we should find a way to prove that the user actually accessing them is the same that initiated the transaction; moreover this is to be done without discarding user-friendliness or privacy perception of the user itself. Despite this there’s no larger chance, out of continuos security system (a memoryless one) to safely achieve that: the use of screen locks or the proximity token technology (using devices that unlock workstations only when the authorized users is in the nearby of the machine) might enforce some good defense but most of this attacks success hopes relies on user laziness: once a user felt safe and has left his workstation without locking it the threaten grows strongly.

1.3.3 Different credentials to achieve authentication

After a first review of authentication topics has been developed we should now explain how we could practically grant these features that has been described previously. Particularly we should stress the topic :”With what kind of credentials a user can authenticate himself on a system?”.
That's because many commercial authentication protocols are different according the factor they choose to recognize the users, usually these categories are addressed as:
i.“*what you know*”:

Something that you know and that is supposed to be kept secret or not to be shared with others such as a Personal Identification Number (PIN) or a password;

ii. “*what you have*”:

Something like a key that we solely possess: a token or a smart card for example;

iii. “*what you are*”:

A biometric feature that belongs us: usually divided in behavioural or physical biometrics such as sign verification and keystroke rhythm or thumbprints, our iris shape and colours, our ear shape and whatever else can be measured from our body (even DNA code!).

Each of these factors implies different hardware support and different protocols to achieve the same goal: doubtlessly authenticate users on a IT system. We will now develop a quite wide review of possible solutions according these three different kind of authentication reporting, when it's interesting, examples of working protocols. For each of these protocols we'll focus on topics, caveats and advantages introduced.

1.3.4 Using a single factor in flat authentication schemes

For each kind of single factor in the next chapter we will report a review of used protocols evaluating advantages and disadvantages, examples, security features and ease of usage. Moreover we're going to briefly talk about costs, set up problems and user acceptance because these topics assume relevant importance on the market-place. Before reporting this we would like to stress much on the three categories we have just exposed, trying to understand the nature of each and the inner capabilities it refers. Then we will see how these skills are combined in actual products.

1.3.4.1 “Something you Know”

Protocols using this genre of factor had been the first to be implemented on many systems and are still the most used ones over the WWW. Actually is a really mature field of work and, due to the exclusive use in the past years, is practically integrated in most of the software available on the market. Even though they take advantage of user-acceptance over the average (the IT market was born and bred with them) these systems no longer provide an appropriate degree of protection and their use is going to be dedicated to low-impact protection environment (such as unlocking the pin of a mobile phone) or as a step during a layered [see paragraph X] or multi-factor authentication schemes [see paragraph Y].

1.3.4.2 “Something you have”

The second factor that we can use to authenticate ourselves on a system is something we have and others have not. Tokens as well as keys represent the best example of objects of this class. Devices of this kind has always been used through the years for many different purpose and for protection of private properties: actually they’ve been used to secure IT systems since the end of the 70ies.

1.3.4.3 “Something you are”

Before we could talk about biometric authentication we'd better stress on biometric itself: this topic is not yet so common and many misunderstandings could confuse the reader of this report. So we will briefly focus on this resource, its many actual uses and its possible future many other ones. In fact biometrics it’s not a news but it’s still today a quite obscure field for the majority of the people and divulgating its strengths as well as its weaknesses it’s a determined scientific goal.

1.3.4.3.1 What biometrics are and how they are used.

Biometric is a measure of a person's unique biological feature such as a fingerprint or voice: this measure can be stored in somewhat a template that a third party, that you have previously authorized, is going to use for verifying your identity granting you access to any sensible resource it wants to share with you. Biometrics represent the smartest way to prove your identity (we would say the common way in everyday life..) and basically the most natural; you have nothing to remember nor to forget: you have only to be yourself.
For example having your face scanned by an opportune device could allow you to enter a restricted-area in your bank or check-in in an airport: no queues, no stops, no questions.
Therefore there are many biometrics and not all of them might be well accepted because of their intrusiveness and the high-collaboration degree required from the user.
That's why we have to mind about biometrics' real potentialities: bringing a huge improvement in ease of use and tolerance between users is as important as trading none of the security tips actually available on the market as well as keeping operations' average times close to the actual ones.
Biometrics are actually used mostly for identify or verify users: the logical application field is physical access to restricted areas. Biometrics had been used for this purpose since 80ies to build unmanned access controls. Probably they represent the ideal solution for 24 hours high-volume access control. In the future they could help authorities in implementing covert surveillance systems such as automatic check-in for airports and many other sceneries.

1.3.5 Granting stronger security through multi factor and layered authentication schemes

We usually authenticate ourselves many times a day into our workstations giving proof of who we claim to be. Whatever method we use for each purpose usually only one method is used: in a layered authentication scheme this is no longer true. The aim is to provide more dimension to an otherwise flat security model introducing many different authentication steps to be undertaken in different situations and for different system resources. For example every generic employee of X company has the same software installed on his/her workstation and to have access to it he ought to be able to digit his own private password. To access the research lab folders' tier and related applications, virtually available from all workstations, he has to insert his own USB token into a port of the workstation. If now he wants to remove, modify or edit one of this files he will have to be strongly authenticated through a retinal scan process, proving the system his writing rights over these folders. This plausible scenario addresses a classic example of a layered authentication scheme. Each different layer of the system software is protected, according to data sensibility, with a greater effort by the system. This is what is called a layered authentication system and such a scheme could be adopted for two main reasons:
a)user friendliness: if a super user, who has full rights even on the research files, is working on his workstation in order to browse some white papers on the Web he doesn't need to insert his token and face a retina-scan.
b)stronger security: if a third malicious person wants to harm the company's valuable know how he should know a password, possess the related token, and manage to fool the retinal-scan system: moreover these factors should be stolen all from the same employee and this make it harder to succeed. There is no need to stop at three distinct levels and administrators could choose to implement some tiers they want according to the data protection requirements. Another important point to make is that you may grant authentication from one level to the next by an entirely different form (smart card to password to biometric) or you may choose to use different formats of similar devices between the levels (such as fingerprint and then voice recognition after an iris scanning).
Of course any layered authentication system is a multifactor authentication system (at least from the second tier on) but this does not imply that every multifactor system is a layered one at all. In fact it is possible that your bank provides you, in order to access its on-line financial consultant offices, a pin and a token to be unlocked with. Anytime you enter your bank web-site if you want to talk with your consultant you have to type in your user ID, insert your token and digit your pin to gain access to the reserved section of the site. This is a multifactor authentication scheme but not a layered authentication scheme and it addresses all situations when a stronger authentication is required according to point (b) in layered systems: it's harder to steal more than a credential from a user. In fact multifactor authentication is common practice (as well as layered one) in high security environments where every effort is produced to keep sensible data secured.

2 A review about the state of the art, discussing principal features

2.1 Passwords

Passwords have been the predominant method of authentication in client/server environment and still today they are the most widely used form of authentication: they represent the first and best example of a static security approach. To prove their claimed identity users provide an identifier, a typed in word or phrase (usually referred as an ID) along with a password and once they are logged they could interact with the system freely (according to their system's rights). Password authentication doesn't require additional hardware device since authentication of this type is in general simple and achieved through software operations. Of course password-systems do not require much processing power and they aren't time-consuming.

Issues:

i.Ownership : Password need to be used and owned by a single individual and never shared by a group.

ii.Social acceptance : Users all over the world are already very familiar with passwords and are comfortable with this authentication process. Two main reasons could be listed: the fact that this has been the first authentication scheme ever adopted since the 80ies and the wide-spreading of PINs to operate many electronic products (such as mobile-phones).

iii.Distribution : Whether distributed in hardcopy or electronically the distribution process should grant the maximum degree of protection against disclosure.

iv.Cost : Password are probably the cheapest and easier protocol to implement. Costs may rise when the number of user begins to generate a significant amount of administrative work in issuing/registering users and when special software is required to handle the authentication process efficiently. Remarkably this is a completely software protocol: this means no need for additional hardware or devices.

v.Ease of installation : Relating to the previous lines most systems already have user ID and password facilities built in by the manufacturer to handle this type of authentication.

vi.User operability : the entire solution is implemented in software and the users don't need to worry about any peripheral hardware devices.

vii.Entry : the computer terminal should not display the user's password as it's being entered as well as users should enter it smartly, paying attention not to be observed by others.

viii.Storage : Password are usually stored in large password-files: whenever a hacker could take possess of this file he would be able to manage our system as long as he wants whether to let authorized users out of the system. Passwords must be stored encrypted or in physical separated area which is only accessible by system-components or administrative authorised users: if an eavesdropper manages to get one password he will not be able to get all the other passwords and take control of the system.

ix.Passwords' cache: user friendliness is to be considered as a milestone of every product and imaging to oblige users typing in a password thousand times a day (or even typing in twenty different passwords two times each) would result in a totally annoying system that would stress users over their limits. As is, passwords' cache has been implemented to let users work on a more "human" and kind system but this files (i.e. usually implemented in browsers), sometimes deleted once the workstation is shut down, sometimes not, represent for hackers the vault of every secrets. If a third party could access this file he will be able from then on to masquerade as the user.

x.Lifetime: Password can be used for a lifetime and so they should (or could) be changed at reasonable intervals several times: this is to prevent password to be copied or stolen by an eavesdropper and then used for replay attacks (see paragraph 2.1.1). Obviously administrators cannot urge users to change their password every week but users themselves should remember that the soundness of this security system resides in the password itself: even though frequent changes of the password could bring to write it down on a piece of paper stored, likely, under the keyboard, a password should never being used as far as it will be discovered: the longer we use it, the much probable someone could discover it by repetitive tries, by shoulder-surfing or by social-engineering (paragraph 2.1.1). So users and administrators should agree a reasonable interval (according even to protected data's sensibility) at which they will change their passwords.

xi. Passwords' history: to help the user logging on the system many vendors implement a passwords' history so that if the user forgets the ultimate password he has chosen the previous day he could use the latest one he used to type in until yesterday. This history could remember more than a level of past passwords and could be used, in case of suspected attacks, to track the user knowledge of the passwords previously used by himself. This mechanism is extremely dangerous because it could allow a hacker to keep on misusing our resources even if the fooled users had changed their password.

xii.Password choice: Easily guessable this issue is the most important one: the choice of a simple password or a short default one is hugely deplorable. If in personal networks and home system this issue has relevant but not strict importance for enterprise or public network this issue provide a quite relevant amount of extra administrative work. Administrators could use controls in password choosing or afterwards when passwords has been enrolled yet. In the first scheme the system let the user choose his own password but refuses to enrol it if it's considered to easy according to its parameters (low user acceptance). In the other protocol the system periodically checks all the passwords chosen and used by the users: if some of them are too easy the system could temporarily deny access to those accounts (until a new appropriate password is chosen) or simply notify the need to change the password (this could result in long periods during which many users access their accounts with easy passwords making the system prone to attacks!).

Advantages:

i.Password systems can be implemented entirely within software, avoiding the need for peripheral hardware.

ii.ID/password carried over encrypted channels (for example SSL as described in annex A) is feasible and deployable.

iii.Low realization costs.

iv.Quick and simple protocols allow quick authentication.

v.User-friendly and high user acceptance.

Disadvantages:

i.IDs and passwords sent in clear over the network are becoming more and more prone to "eavesdropping".

ii.Password systems are subject to replay attacks.

iii.Password systems are prone to be forced with password-guessing attacks
iv.The need for ineffective password management and controls (i.e. Re-issue, unlocking, etc.)
v.Lack of user awareness and training could result in them writing down their passwords on sticky papers nearby the workstation or even on the monitor.
vi.Trojan horses can capture IDs and passwords under false pretences.
vii.Password can be inferred through social-engineering.
viii.They can be stolen or observed while being entered.
ix.Password choosing could result in a hard duty for users if strict rules are urged by the administrators.
x.Passwords' cache files make the system open at every risk if violated.
xi.Passwords' history could help hackers accessing the system if a well synchronized mechanism of history-file refresh is not implemented (usually the history-file should be deleted in case of detected attacks).

2.1.1 Attacks vs. Password systems

As lately written in these authentication schemes (i.e. Http Basic Authentication one [3]) since username and password cross the networks unprotected, vocabulary attacks, as well as replay attacks, are quite easy to succeed . A replay attack is the attack that a third party purports, once it somewhat way got the password (eavesdropped on a tampered channel or viewed while being typed in), using the same password as the authorized users: it's called a replay because for the server this is but another legal access. A vocabulary attack is enforced on the fact that the range of alphanumeric codes among which users choose their own passwords is much smaller than the available bit space (this is ought to remembering problems): hackers had collected through the years a set of high probable passwords observing and testing the habits of their colleagues and victims. Doing this results in a vocabulary that makes a brute force attack quicker to succeed using nonsense combinations not first than passwords reported in these lists. Sometimes a password remains the same for long periods and this allows hackers to use many techniques to break into the system. The first step towards systems' break is usually moved by authorized users that choose bad passwords. Whenever we talk of bad password we shall think about those users who never change the password on their hosts using the default ones, or to those users who choose a too short password or information related to their private or social life (i.e. girlfriend or children's names, workmate's nickname, birthday). Hackers could use lots of on-line vocabulary, made of some likely passwords and previously compiled by other hackers, to brutally force our authentication system or they can make some social engineering (investigating and asking our colleagues or friends about our habits and family, or even contacting us, under a masquerade or fake identity, to make us giving them precious information) and achieve particular personal information with which they can try to guess our password and consequently impersonate us. Of course this kind of attacks, though quite rude, has greater chance to succeed than a brute attack trying to submit every possible alphanumeric combination: if users choose bad passwords the hacker has even a greater chance to break into the system. Passwords could even be eavesdropped or sniffed while we are remotely authenticating on the LAN our host to the system's server. Sometimes a password can be easily stolen simply searching among the papers in the nearby of a workstation or exploring the passwords' cache file of the browser. Once a Vocabulary or Brute Force Attack (repetitive attempts with randomly generated password) has succeeded the hacker is able to purport a Replay attack newly using the data he had previously stolen. Easily guessable the biggest security caveat in password protocols is the fact that the data used to authenticate the user is always the same so that once a third party comes in possess of that data it will pretend to impersonate us. The first and easier solution is to change every time the credentials used to prove one's identity and it has been the nonce mechanism.

2.2 The use of nonces: the http protocol example

The http protocol had implemented a nonce mechanism enhancing the security of authentication [3] procedures. From the Basic Authentication scheme it has passed to the Digest Authentication scheme that avoids the biggest and more serious flaw in clear-password schemes. The opportunity to provide whatever mechanism that could change every time the credentials exchanged during the simple challenge between the principals has been definitely huge. The data that every times changes and that is associated with the password itself (that always remains the same) is called a **nonce** (see annex B). That's why even for a protocol like Http, that is not usually referred as a secure one, the Basic Authentication Scheme (a clear text challenge-response password mechanism) has been replaced with a Digest Authentication scheme. Although Digest Access Authentication scheme is not intended to be a complete answer to the need for security in the World Wide Web in fact it provides no encryption of message-content. Like Basic Access Authentication, the Digest scheme is based on a simple challenge-response paradigm but using a nonce value. A valid response contains a checksum (by default, the MD5 checksum) of the username, the password, the given nonce value, the HTTP method, and the requested URI. In this way, the password is never sent in the clear and the hash message transmitted is never the same. The Digest authentication scheme suffers from many known limitations. It is intended as a replacement for Basic authentication and nothing more. It is still a password-based system and (on the server side) suffers from all the same problems of any password system. In particular, as well as in the Basic scheme, no provision is made in this protocol for the initial secure arrangement between user and server to establish the user's password. That's why http has been provided, even though http is never referred as a secure protocol, of Digest Authentication scheme. Basic authentication schemes are now used only in low-protection protocols and represent mostly an identification process then an authentication one. A replay attack against Digest authentication would usually be pointless for a simple GET request since an eavesdropper would already have seen the only document he could obtain with a replay. This is because the URI of the requested document is digested in the client request and the server will only deliver that document. Because this hash (see annex B) is unique for each accessed URI, no other documents can be accessed nor can it not be used from other IP address without detection. The password is also not vulnerable to eavesdropping because of the hashing. By contrast under Basic Authentication once the eavesdropper has the user's password, any document protected by that password is open to him. Thus, for some purposes, it is necessary to protect against replay attacks. A good Digest implementation can do this in various ways. The server created "nonce" value is implementation dependent, but if it contains a digest of the client IP, a time-stamp, the resource ETag, and a private server key (as recommended above) then a replay attack is not simple. An attacker has to persuade the server that the request is coming from a false IP address and must cause the server to deliver the document to an IP address different from the address to which it believes it is sending the document. Moreover an attack can only succeed in the period before the time-stamp expires. Digesting the client IP and time-stamp in the nonce permits an implementation which does not maintain state between transactions (continuos authentication) time stamp, and additional secret information. The system is, however, vulnerable to active attacks such as a man-in-the middle attack. Even with the nonce mechanism this scheme is prone to many attacks as IP spoofing (there's IP spoofing every time that an IP header's Source address field is modified in order to confuse the client's IP protocol and make it sending his replay messages towards a different desired destination in the network as well as make it believing that these messages were sent by another machine than the real one) or all the kind of interleaving attacks (see paragraph2.3.2).

2.3 Symmetric key cryptography

The risk of eavesdropping (and then of replay attacks) can be managed by using each time different information to authenticate and hashing the challenges sent across the network but a much safe way to achieve authentication and secure communication is to share with the other principal a secret key that belongs you both only. In these schemes the connecting party usually sends per each session a random value encrypted (as in digest authentication this technique prevents the shared secret key from being easily discovered and introduces a random variability in the communication) and the capability that the authenticating party has to show is the knowledge of a shared secret password. The realization of this scheme is achieved through symmetric cryptography techniques: both the principals of the communication have to share a secret key that has being formerly exchanged with a secure protocol. The security of this cryptography is guaranteed as long as the key is kept secret, afterwards the method is totally insecure. The authentication protocol could provide one way or two way challenge allowing one or both hosts to authenticate themselves. Supposing host A wants to authenticate itself to host B to access private information B hosts it will first send a request to connect to B claiming its identity. Then B will send A a random value R asking it to prove that it shares the secret key E_k . A is going to encrypt this value and send back B the value $E_k(R)$. B will then use the same key he possesses to verify if the value it has received encrypted is the same it formerly sent to A. If it will be so, A will have proved its rights and will be granted access into the system. This technique is called symmetric key cryptography because, as you can easily understand, $X = D_k(E_k(X))$. Two main problems affects this kind of protocols:
a) the random generator should be a strong one and not a weak one: if the rate of repeated numbers increases the protocols allow hackers to decrypt the key in a much easier way.
b) even though symmetric cryptography is not at all time or power consuming, we have to worry about the real applications of this technique, for example, in e-business. Supposing you want to buy a book from a UK on-line vendor : which advantage would you have in buying it on the web if to have a secure transaction you have to move in Paris to exchange with the seller a symmetric secret key? This might enforce the idea that symmetric cryptography is quite useless on a large scale: this is not properly true as the next paragraph reports. So the greater deal with symmetric cryptography concerns the problem of distributing, storing keys and keeping them secret. [11] In 1976 Diffie and Hellman developed an on-line protocol: the Diffie-Hellman key agreement protocol (also called exponential key agreement) that allows two users to exchange a secret key over an insecure medium without any prior secrets. The protocol has two system parameters ***p*** and ***g***. They are both public and may be used by all the users in a system. Parameter ***p*** is a prime number and parameter ***g*** (usually called a generator) is an integer less than ***p***, with some special relationship based on exponential and logarithmic properties. Suppose Alice and Bob want to agree on a shared secret key using the Diffie-Hellman key agreement protocol they proceed as follows: first, Alice chooses a random private value ***a*** and Bob chooses a random private value ***b***. Both ***a*** and ***b*** are drawn from the set of integers {1, ..., ***p***-2}. Then they derive and exchange their public values using parameters ***p*** and ***g*** and their private values. Finally both Alice and Bob apply some mathematical predetermined operations to

their private value and the other's public one transmitted and gain a common shared secret key ***k***. This protocol depends on the discrete logarithm problem for its security and its built on exponential operations. It assumes that it is computationally infeasible to calculate the shared secret key ***k*** given the two public values when the prime ***p*** is sufficiently large. But the Diffie-Hellman key exchange protocol doesn't prevent from password sniffing or interleaving attacks. Moreover is itself vulnerable to a man-in-the-middle attack: in paragraph 2.3.2 many simple protocols are exposed according to their caveats and their enhancements that are strongly linked with the attacks that attackers had brought them. This will help us understanding how and why these techniques has evolved through time.

2.3.1 Key distribution centers

Since the end of the 70ies the problem of handling encryption keys safely and effectively has been addressed developing a standard for Key Distribution Centers [KDCs]. With KDCs Banks, the first users and developers of distributed symmetric encryption systems, generated temporary encryption keys to be worked for a single transaction between two banks' offices: this defeated the risk of replay attacks and eliminated the high costs of distributing a pair of private keys between every pair of offices. In fact each trusted site has a unique symmetric key (called master key) that it shares with the KDC and that allows the host to talk safely with it. Basically the KDC affords to Host A, who wants to communicate safely with Host B, a temporary secret key to talk with B and a **ticket**, encrypted with Host B master shared key, to present himself to Host B. This ticket is intelligible only for Host B because is encrypted with the secret key that KDC and Host B shares. In this ticket Host B could verify:

I. if the request of communication to the KDC was really sent from host A (thus preventing IP spoofing);
II. Which the temporary encryption key is, thus providing the means for verifying a man in the middle attack;

Now Hosts A and B can easily authenticate each other using the shared temporary secret key generated by the KDC but there's still a big problem to solve: Host A cannot be sure that the key and the ticket it received are genuinely build to talk with Host B and no one can prove him that this is not a masquerade attack of someone trying to impersonate the KDC to break into its system. The problem was first faced in 1978 by Needham and Schroeder that solve it introducing a portion of challenge response in both the Host A-KDC communication session and the Host B-Host A authentication session

III. When Host A requests the KDC for a ticket it sends a nonce encrypted with its master key and when it reads the response it looks for the decremented response value,
IV. When Host B receives the ticket it sends Host A a nonce challenge and asks for a response with the corresponding decremented value: this should help both Hosts to detect a probable replay attack while a time stamp provides certainties about the fact that this ticket is part of a contemporary request and not part of a replay attack.

To achieve this is fundamental that the encryption algorithm used to cipher the parts of the communication is a block cipher and not a stream one: this is because a block cipher relates different parts of the data stream and any part of the resulting ciphertext depend on one another: this causes invalid results, once you've decrypted the cyphertext, if the ciphertext has been changed. Lately Denning and Sacco introduced a time stamp that provides both hosts certainties about the fact that this ticket is part of a contemporary request and not part of a replay attack. They realized that once an attacker has got a couple made of a session key and a ticket, no matter how he got them, he could convince the ticket receiver to authenticate him any time he wants because this one will not be able to understand if that ticket has yet been used by an authorized user: managing the corresponding session key the attacker will be able to perform the challenge-response protocol and foolish the server.

V. The introduction of a time stamp in every ticket dramatically reduces the possibility of a replay attack into a determined time-slice making the attack an on-line one harder to succeed: the server could now easily verify, even though the attacker has a good couple of session key and ticket, if the request is a contemporary or a old one. A stronger defense is so enforced against replay attacks.

Even if symmetric key cryptography has not had such a wide spreading it is correct to report that this technology has in MIT Project Athena's Kerberos protocol the greater example of working product: Kerberos is not only an authentication system but presents many interesting features that can be retrieved in other security protocols, see annex C for details.

2.3.2 Attacks vs. symmetric encryption systems

A quite wide review of attack-techniques can help us making up our mind over many possible hidden flaws in symmetric cryptography authentication protocols.
As previously explained the principals can even implement cryptographic handshake with a nonce mechanism that prevents any replay attack changing every time the same symmetric key. If a user wants to authenticate himself he has to prove to be able to use a cryptographic key he shares ("something he knows"). Both a one-way (only one principal of the communication is authenticated) or a two-way authentication protocol presume to achieve authentication using nonces to encrypt with symmetric keys. The possible main flaws in symmetric key cryptography may consists of protocols' caveats as well as low power random number generator. In the second chance the ciphers could be even more easily attacked: for example a simple two-way protocol is quite weak towards **known plaintext, chosen ciphertext and chosen plaintext attacks [7]** and we're going to explain why the random value should be as random as possible. In the figure below a simple two way protocol is taken.

$$A \rightarrow B : \{Rnd_1\}_{Ek};$$

$$B \rightarrow A : Rnd_1, \{Rnd_2\}_{Ek}$$

$$A \rightarrow B : Rnd_2$$

figure 1, a two-way authentication protocol

This kind of attacks sets up a dictionary of selected pairs to use later. For example a wire-tapping intruder can collect two pairs of ciphertext and plaintext per protocol run because the random numbers are sent both in cipehrtext and plaintext between A and B. This is called a known plaintext attack.
The attack may also be turned in a chosen ciphertext attack : pretending to be A an Eavesdropper sends a ciphertext message (that he has selected) to B that sends him the decrypted value back. E will not be able to complete the protocol but is now able to collect pairs of ciphertext and plaintext starting from a ciphertext that he has chosen. Moreover E could arrange even a chosen plaintext attack having a selected plaintext encrypted by others. Supposing that part of the handshake is submitted on the net as clear text and that the random generator has limited power, E will now be able to fill a table of correspondences and allow this hacker to decrypt the private key. That is why an intruder that can easily eavesdrop messages on the network and fill a table of desired pairs, while authorized users are connecting, once he has built a table with many pairs of random values and encrypted random values, he could be able to authenticate himself in the system in two situations: when a random value he possesses is going to be used or, if the random generator is not so powerful and the cipher is not so tough, he could try to infer the secret key from the many collected couples and obviously, once he had discovered the secret key, log into the system every time he wants to.
A possible solution is the one in figure 2:

$$A \rightarrow B : \{Rnd_1, Rnd_2\}_{Ek}$$

$$B \rightarrow A : \{inv[Rnd_1], Rnd_2\}_{Ek}$$

figure 2, a one-way authentication protocol

Using a partially inverted nonce makes the tampering harder. Although in this protocol no message is sent in cleartext many other attacks, relying on the protocol's implementation caveats, could still break out the security of our system. **Interleaving** or **Man-in-the-middle attacks** interpose the hacker between two authorized principals and use one (or both) of them as an oracle to gain access to private resources. This kind of attacks is not like replay-one: a replay attack can be purported whenever a hackers desires to force our system, interleaving or man-in-the-middle attacks are effective only if performed live and on-line at the same time when A and B authorized users are contacting each other. For example an eavesdropper can perform an oracle session attack as in figure 3: using A as an oracle he will be able to authenticate himself to B and having granted access to private or sensible data.

A *E* B

$E \rightarrow B : \{Rnd_1\}_{Ek}$

$B \rightarrow E : Rnd_1, \{Rnd_2\}_{Ek}$

$E \rightarrow A : \{Rnd_2\}_{Ek}$

$A \rightarrow E : Rnd_2, \{Rnd_3\}_{Ek}$

$E \rightarrow B : Rnd_2$

E → A : “leaving session”

figure 3, an oracle session attack

$A \rightarrow B : Rnd_1$

$B \rightarrow A : \{Rnd_2\}_{Ek}, \{Rnd1\}_{Ek}$

$A \rightarrow B : Rnd_2$

Figure 4, ISO sc27 protocol

This kind of attack can be easily discarded encrypting only on one side of the communication: see for example in figure 5 the ISO sc27 protocol. Even though ISO sc27 is safe towards oracle attacks it still lacks of protection in case of **parallel session attacks**: intruders could still fraudulently try to use a principal against itself as an oracle starting two sessions ,one is fake, with the host itself; see figure 5 for details.

A *E* B

A → E as B : Rnd_1

E → A : Rnd_1

$A \rightarrow E : \{Rnd_2\}_{Ek}, \{Rnd1\}_{Ek}$

E as B → A : $\{Rnd_2\}_{Ek}, \{Rnd1\}_{Ek}$

A → E as B : Rnd_2

E → A : Rnd_2

figure 5, A parallel session attack

We can easily discard even this tampering technique implementing a test over the random number used to open the communication: whether it's the same just used to open another session it will be refused. All the way this is a “patch” and not a real enhancement: the protocol is still susceptible to attacks and that's why lately the use of asymmetric cryptography has been introduced even in authentication protocols.

2.4 Public key cryptography

Few lines above we evaluated that key-exchange costs were too high and practically not low cost feasible in symmetric cryptography. Moreover shared keys do not afford complete reliability as anything that is shared between two can be revealed, even though through transitive trusts, to many other third parties without the consent of one of the secret’s owner. Furthermore shared keys are possessed by two and you cannot doubtlessly say who of them signed a data stream. We are now going to introduce another enciphering method that assures total reliability on the signer’s identity of a data stream, thus providing the meanings for certain authentication, and that might avoid any kind of unsecured key exchange removing the threatens of password sniffing. In fact the basics of public (or asymmetric) key cryptography are the following:

- *two asymmetric (different) keys* ***X*** and ***Y*** that had been generated (according one of the existing dedicated algorithms) to generally perform secure and enciphered communication between two principals.
- One of this key, ***Y***, is made public, published and known to anyone who requests about it and it’s referred as the *public key*.
- One of this key, the *private key* ***X***, to be kept secret.
- The algorithm used to encrypt and decrypt is a two way asymmetric one, this means that $M = d_x(e_y(M)) = d_y(e_x(M))$ but it’s not true that $M = d_x(e_x(M))$ as well as $M = d_y(e_y(M))$.
- When I need to authenticate myself on a system I will send the host administrator my request *enciphered* and *signed* with my private key so that, once he had retrieved in somewhat way my public key from a third trusted party, he can doubtlessly authenticate myself on his system.
- Moreover he will possess evidences of my request because the *digital signing* of the messages will guarantee *non-repudiation*.

Asymmetric-cryptography is much more power-intensive and time consuming then the symmetric one. Then the need of a third party infrastructure arises strongly in order to guarantee the genuineness of public keys: if the other principal of the communication cannot be sure that the public key he has gained it's mine then he will not be able to prove that it's me asking for authentication. Certainly public key cryptography introduces great improvements in security concerns but those will not be available as far as these problems will not be solved. Starting from the pair of keys used in public key cryptography those many new topics introduced in the previous lines are going to be developed in the following paragraphs.

2.4.1 Public and private keys

The couple of keys that operates asymmetric cryptography consists of a private (that we are going to address as ***X***) and a public key (addressed as ***Y***). These keys are interrelated by some special mathematical properties as they are generated together. The base secret (usually represented by a large prime number) is embedded in ***X*** , which is kept strictly secured in our hands. Through performing a special one-way mathematical function we construct from the base secret and from ***X*** the public key ***Y***. Being this function a one-way one thus it's not feasible, but really hardly, for an attacker to deduce ***X*** from ***Y***. Both of these keys can be used for encrypting some data but only the other one will then correctly decrypt the enciphered result. Immediately you can notice that if someone encrypts some data with ***Y*** only the owner of the private key, if this has been kept secured, will be able to decrypt the message. This is a straightforward method to convey to a third party messages that will be intelligible only by that principal whom we are sending them once we had previously used his public key ***Y*** to encipher them. Moreover if entity A distributes its public key ***Y*** to all its trusted recipients it will be able, keeping ***X*** secured, to communicate in a private manner with them, being the messages intelligible only for those who possess ***Y***. Many algorithms has been implemented through the years to generate from a chosen prime, or a couple of, a pair of keys: the most used and famous one is probably the RSA Public Key algorithm. RSA algorithm works as follows:
-Choosing the keys) the pair consists of three parts : a public part ***e***, a private part ***d*** and a shared value ***n***. ***n*** is constructed multiplying two large primes, ***e*** is a random value chosen as meeting some certain mathematical restrictions. ***e*** is computed starting from ***e*** and the large two primes. So the public key ***Y*** consists of ***n*** along with ***e*** ([***n,e***]) and the private key ***X*** consists of [***n,d***]. Once you kept ***d*** secured and you've erased the two primes your keys are secured and ready to be used.
-Encrypting a message) taken a text ***p*** we exponentiate it by the public part ***e*** (example using ***Y*** to encrypt) and take the modulus relative to ***n***. To decrypt the message we repeat the same operation but using ***d*** instead of ***e***.
Though this algorithm should seem quite simple is really hard to force if the number of bits representing ***e,n*** and ***d*** is quite large: if these values are greater enough according to the present CPU computing power it would take too long time to decrypt the messages and the attack will result useless: in 1977 a 416 bit long key seemed unbreakable, actually modern RSA keys are usually more than two thousand bits long.

2.4.2 Authentication achieved through digital signing

Due to the fact that a private key is private we can use public key encryption to achieve user authentication through Digital Signing [DS] techniques: if a user encrypts a message with its own private key ***X*** (obviously assuming that ***X*** is not equal to any other private key ***Z*** and that is not compromised yet) and send it along with the original one he marks that data in a unique way that could afford some important properties for the sender as well as for the receiver. On the sender side the user could demonstrate at any time if someone has tried to manipulate the original message because applying newly his private key to the clear modified message the result of this encrypting procedure will be different from the signed message previously sent with it, thus providing the meanings for proving the counterfeiting attempt. On the receiver side the same property is useful to achieve *non repudiation* (is the term for doubtlessly proving that you and you only said/made/signed this and just this): decrypting the signed message with the sender's public key and comparing it to the cleartext the receiver can prove that the user had wrote down (and moreover signed) that text (according to the special relationship between public key encryption and the key pair). In fact he is the only one to possess the specific private key to determine such a signed result. In a common authentication scheme a server might send a user a nonce with a signing request, the user should sign the nonce with its private key and send it back to the server so that this one has now the possibility to check out with the public key whether its interlocutor is really that claimed user. Careful readers can easily address the two main issues of this technique: keeping the keys secret (but as well as with passwords this is not a news) and trustfully demonstrating that a public key ***Y*** belongs to a certain user. The problem of tying key property with key possession or with some unchangeable identifying data it's the main goal of public key cryptography developers (although not disregarding enciphering algorithms complexity) and the method developed through the years has been the public key certificate.

2.4.3 Public key certificates

The idea of implementing certificates to authenticate a determined public key is attributed to Leon Kornfelder that, as an MIT undergraduate, proposed the use of certificates to provide certain association between names and keys. Kornfelder basic idea was linking on a single document the key owner's name, the (public) key and a trusty digital sign: since then the procedures for building a certificate and the standard formats it should assume had deeply changed. Today certificates usually carry far more information than the key and the user's name, especially, according to some specific standard, they could provide these notices [10] :

- Date of issue
- Expiration date
- Version of the certificate's format
- Indication of types of keys contained
- Owner's name in various formats
- Other properties of the owner (i.e. useful for biometric authentication)
- Owner's right and privileges

Actual standards for public key certificates support a number of extensions but most certificate products omit person data and privilege information from their certificates: the fact is that experts are still debating about the excellent format that a certificate should assume, some arguing that thin certificates (carrying as less information as possible) are best, others arguing that added features or fields are useful.

2.4.3.1 Certificate Standards

Most public key certificates are today based on X.509 standard: the characterizing features of this standard are Abstract Syntax Notation #1 [ASN.1] and distinguished names. A distinguished name is a name containing many several elements (usually hierarchy levels) that make the name identify a particular person in a unique way.
Distinguished name could base their hierarchy of name, for example, on geography:
/C="I"/SP="Pi"/L="Pisa"/PA="via G.Moruzzi 1"/CN="Iit-Cnr"/....

A hierarchic scheme as lately addressed could be useful for ensuring certificates' uniqueness : replicating the DNS servers scheme at each level an authority could provide collision-proof guarantees thus resulting there would be no replication chance all over the

world (as far as all levels perform their task).
In every X.509 certificate the contents of data is arranged according ASN.1 statements: ASN.1 is a standard notation for formally describing structured data to be transmitted over different platform and through different medias (refer to ISO 8824 and 8825 for a full comprehension of this standard).

2.4.3.2 Verifying certificate's genuineness through vertical or horizontal trust schemes

The most urging question related to the use of certificates is who and why signs certificates themselves. As is, no doubt should arise on the use of a certificate if we want to fully exploit public key cryptography's potentialities. The problem of reliably distinguishing between original and bogus certificates could be addressed through a vertical or a horizontal scheme. In the first scheme users authenticate each other, eventually relying on trustable known third users, forming a trust chain, to verify a certificate. In the second scheme a dedicated service entity issued by many users (and certified as trustworthy by their confidence in it) works as a third party within a specific operating realm; at an upper level another entity operate surveying many of these first level authorities and so on in a centralized hierarchy. Anyway the signer is addressed as a Certificate Authority [CA].

The first scenery refers to the case of Pretty Good Privacy [PGP] an encryption software package created to exchange secure messages between users. In fact PGP relies on each user's friends or colleagues to trust and certify his digital sign: everyone on his own creates his certificate and due to the fact that someone else he knows trusts him he could use that certificate to authenticate himself (obviously only to those who could construct a chain of trust leading to its home-made certificate) through that person. The second scenery is related to the youth of Internet when someone had thought that it could have been possible to implement a single central root authority to manage (of course on more than a level) all the certificates but, as a matter of fact, the obvious difficulties to be overtaken, the high costs and the many problems related to lack of standards in the first part of Web life has made of this scheme no more than a theory. Actually most major commercial CAs forms a chain of trust for which they all recognize each other DSs allowing their users to collect a certificate chain proving or not the validity of the requested certificate.

The strength of public key cryptography that allows indirect and off-line authentication is even is major weakness: if PKYs don't provide frequent and fresh information about revocation of some determined certificates many could be struck by attacks leaded with the corresponding compromised pair of keys. Three protocols provides protection towards this eventuality:

1. Certificate revocation lists compiled by CAs reports all the certificates that those authorities for some reason considers to be invalid.

2. On-line verification of a certificate's validity (thus loosing the advantages of indirect and off-line authentication)

3. Prefixed time validity setted as low as possible so that certificates should then be often reissued by a trusted PKY (but this is hardly feasible for end users or little companies because the costs to sustain would probably raise too much relating their profits.)

2.4.4 Certificates and CA authenticating hosts

Digital Certificates are electronic documents that are issued generally by a trusted third party called a Certificate Authority (CA). These certificates contain information about a user, that the CA has verified to be true, concerning his/her/its Public Key and possibly some more related information . In dedicated authentication protocols when an electronic message is sent from the client to the server it is signed with client’s private key: using the corresponding digital certificate the server could authenticate the user is who he claims to be eventually implementing non-repudiation capabilities. Public Key cryptography is the enabling technology for this enhancement to be developed and Public Key Infrastructures [PKIs] are the medias managing keys and certificates, distributing them to users.

Issues:

i. Ease of use : it will probably be quite complicated for the user to handle their certificates and moreover those should be manipulated only by a trusted third party.

ii. Problems with public workstations : certificate use is problematic with public workstations because certificates are not automatically flushed from browser after use proposing the risk of having the workstation (and all its potential users) authenticated with the credentials of a prior user.

iii. Mobility & Portability : certificates must move from system to system for a particular roaming user.

iv. Multiple certificates : it might be difficult to achieve single sign on as far as users are always likely to have to choose among a number of certificates.

v. Administration. In order to manage certificates, the following needs should be considered:
a certificate authority that issues and manages certificates;

vi. Routines to register individuals, and create their keys and certificates;

vii. Routines to revoke certificates (due to key loss/compromise);

viii. Controls over eventually matching keys : even if hugely unlike, being the generating algorithm public, a chance exists that two different users generate the same pair of keys.

ix. A directory service to store and retrieve certificates and revocation lists;

x. Legal issues : the law about certificates has not been clarified e.g. should CAs have to provide Key escrow (back up of private keys) and whether digital signatures will come to be legally binding.

xi. Legacy systems : there will be legacy systems that cannot interact with certificates.

xii. Certificates’ storage : how should certificates be stored for each client? Should large databases be set or personal data carriage using smart cards might be preferred? According the fact that likely users will possess more than a certificate are smart card able to carry over all this information or their memories are still to little for all those?

xiii. Costs : certificates themselves are quite expensive and the whole system of certification is quite complex. The cost of digital signatures is relatively low compared to the other schemes, since DS’s is primarily a software-based solution to authentication. Ongoing maintenance is relatively inexpensive in schemes where users administer their own keys, ranging to more expensive if an entity wants to get a third party CA involved.

xiv. Accuracy : the accuracy of digital signatures is considered high as the methodology and logic incorporated into both of these (DS and Public key cryptography) methods has provided excellent reliability in authenticating their users. The only way to compromise authentication is if someone learns a person's authentication code and has access to the computer holding the private key. This risk, while valid, is relatively low and therefore does not alter the overall accuracy of the authentication method. One final point is that digital signatures often utilize complex mathematical algorithms to ensure accuracy and reliability.

xv.Operability : technology today requires minimal effort on the part of the user. Once the DS capability is "turned on," DS's are applied automatically. Additionally, the speed of authentication using DS's is high due to the fact that the operation of attaching a DS to a message can be performed at the speed of transmission. DS's reside on the hard drive of the user's PC or servers on the network. Therefore, their physical size is not applicable, but their logical size can be significant and should be considered.

xvi.Social acceptance : the general public is not very familiar with DS's nor it does understand how digital signatures work. Until society alters its present reliance on a paper signature, digital signatures will be slow to gain acceptance even though the term sign itself afford for some a sense of reliability.

xvii.Product lifecycle : the digital signature product is in its infancy with constant enhancements being made. Therefore, depending upon a company's information security strategy, many systems could be implemented until the product reaches maturity. One advantage of using digital signatures is that there is no separate hardware required eliminating the need for hardware replacements. Also, the lifespan of an individual digital signature is high since once created, it is limited only by the user or the CA upon generating the keys. DSs can be configured to be valid for life.

xviii.Ease of installation : depending on the complexity of the DS authentication scheme, ease of installation can be relatively easy or difficult. If no Certificate Authority is used, installation is relatively easy. However, the use of a CA introduces additional steps in the installation of DS software and creation of Public and Private keys which decreases the ease of installation (but this scenery addresses only PKYs and not end users).

xix.Non repudiation : DS schemes possess a medium level of non-repudiation. By design, it cannot give absolute assurance on who initiated the transaction since private keys and their passwords can be stolen. Moreover legal constraints are world wide hugely different towards this topic and a scheme valid in a country could be totally deserved in an another thus making really difficult for DS to become a public security standard.

Advantages:

i.Validates the creation of a file by a sender. Recipients need to know that the sender created the file.

ii.Prevents the sender from denying involvement in the creation of a file.

iii.Ensures that only intended recipients are able to read the files.

iv.Guarantees that the file was not altered during transmission.

v.Customer centric solution to base personalization features from authentication.

vi.Wide application use for the future which achieves authentication and encryption using the same technology.

vii.Certificates can be stored on smart cards providing secure physical storage.

viii.Considered in scientific, politic and administrative environments to be an authentication method reliable enough to be likely recognized by legal institutions.

ix.The impelling need for a legal international standard to be invested, as soon as possible, of a commonly recognized value in order to regulate the e-commerce and the IT world is strongly urged by many governments.

x.Industry momentum is growing for digital certificates.

xi.Certificates are attractive because browsers and servers already have some support for them (designed for electronic commerce).

xii.Open software components needed for X.509 PKIs are easily available on the marketplace.

Disadvantages:
i.Complicate for users to install.
ii.Part of the software must be installed on every computer user wishes to work on.
iii.Not easily extra routines must be implemented to make it feasible where users share machines.
iv.High danger of counterfeiting if certificates are home made.

2.5 Authentication Tokens’ technology

Token-based authentication require users to have with them a token and use it according special protocols in order to authenticate themselves on a workstation. There's not only a kind of tokens or one token technology. Tokens are now increasing their market quote and many vendors develop tokens with different capabilities. Tokens can be made of ROM, RAM, and even a CPU and a small operating system. Usually they are built with a strong tamperproof shield. From the simplest token that host only readable data on a magnetic stripe to the later smart-card with embedded CPU and cryptographic capabilities there's a large choice of products that can satisfy user expectations. These devices can be used for applications ranging from secure e-mail to electronic cash and credit cards. They offer physical protection to the keys residing inside them, thereby providing some assurance that these keys have not been maliciously read or modified. The majority of token devices actually use a two-factor authentication protocol but one factor token are still sell on the market and we’ll have a look even at this technology though quite old.
Typically, gaining access to the contents of a tamper-resistant device requires knowledge of a PIN or password; exactly what type of access can be gained with this knowledge is device-dependent as some tamper-resistant devices do not permit certain keys to be exported outside the hardware. This can provide very strong guarantees that these keys cannot be abused: the only way to use these keys is to physically possess that particular device. Of course actually these devices must be able to perform cryptographic functions with their protected keys, since these keys would otherwise be useless, thus arising the framework complexity and production costs as well.

2.5.1 Tokens' actual functions and capabilities

[10] At a first abstract level tokens could be addressed as keys as they practically present the same aspects and so we can easily infer what kind of security properties tokens should afford:

I. A person must physically possess the token in order to use it.
II. A good token is hard to duplicate.
III. A person can lose a token and unintentionally lose access to a critical resource.
IV. People can detect stolen tokens by taking inventory of the tokens they should have in their possession.

Tokens are used in many different fields (even non-computer applications) and their success resides in the fact that they take away the burden of memorizing passwords. Furthermore you can use long and complex alphanumeric passwords with tokens as for them there’s no additional difficulty in carrying short, long or hard to remember

passwords while it does matter for a person. So, even if usually tokens request a four digit PIN to be unlocked, users feel more comfortable in using them and moreover they represent nothing newer than a key to bring along.
Actually active tokens shares the largest part of research interests but passive tokens' market is larger because they are still used (for example for ATMs) in many fields: in the following paragraph we will stress their functionalities. Before referring to particular technology considerations we should explain what kind of capabilities, form factors and functions a general purpose token could have. Actually a token could have a shape varying from a PCMCIA card to a ring with a wire coil embedded on it, as well a magnetic stripe card or a smart card. Lately USB tokens are spreading because of their shallow dimensions and the ease of use due to the concurrent wide spreading of USB technology on computer products. Passive tokens with magnetic stripe, like badges or ATM card for cash-machine operations, are simpler and has relative small memory capabilities. Latest tokens and smart cards provide up to 200K memories (that could be ROM, RAM, or both in proportion), internal CPU, clock and timer, advanced computational and cryptographic functions, tamperproof shield, up to five years lasting battery pack, support for eight digits PIN. If the biggest problem with smart card technology was related to the interaction with existing workstation, although without raising big funds for additional compliant devices, the USB tokens technology could now represents a great chance for tokens security systems' spreading.

2.5.1.1 Passive tokens

Passive tokens are very common and their function, though achieved in many different ways, is quite simple: hold a readable version of a secret. Being passive they are not able to encrypt or hash this secret and they store it on a magnetic stripe, in a wire coil, on ROM chip or in patterns of punched holes. The great advantage of using passive token systems is brought by cheapness of realization but, as far as this should be referred as a security system, the disadvantages largely overweigh this cost consideration. In fact passive tokens are too easy to copy: usually most magnetic stripe cards achieve stronger authentication using a PIN (see paragraph 2.5.1.2.1) to unlock themselves and this is a classic example of a two factors authentication protocol. Even in this case the problem is where to store the PIN but obviously not on the token where an hacker could easily read it.

2.5.1.2 Active tokens

Active tokens afford a higher reliability in user authentication since they don't need to emit their base secret out of the token but they usually use it to perform some other combination of actions to authenticate the owner such as generating a one time password or performing a wide range of cryptographic functions in a challenge-response session with the workstation. Tokens of the first generation (but still in use and still sell all over the world) are used to show a secret, possibly a one-time secret, that the owner should transcribe on a keyboard. A second generation of tokens is plugged directly into a computer or the machine to be used with and they can usually perform a set of more complex operations (obviously they are more expensive). The broad variety of shapes that these tokens could assume starts from the classic card factor form to the latest ring one or the larger cryptocard but every plug-in token shares with others certain security strengths and weaknesses. All of them usually store the valuable data in a tamperproof shield or in an embedded chip so that an attacker could hardly recover it physically breaking the device. Unlike passive tokens they usually contain a keypad, a display and a little independent CPU: this allows them to perform operations on their own and completely rely on them to store the base secret, calculate the password and show in on the display. This means that active tokens are equipped with a software, no matter how simple or secure it might be, that a hacker could break or force to do what he wants it to do. The main problem with active tokens is that they usually require a connection that is not always present on most workstation such as a smart card reader or a PCMCIA slot but the new USB token technology seems to be able to smartly solve this as it was the greater brake for tokens' development.

2.5.1.2.1 Two factor authentication: incorporating a PIN

The basic problem with tokens (as underlined in paragraph 2.5.1.1) is that once you have the token you could easily impersonate the owner and perform every kind of action he has right to perform. So as well as with keys tokens should be kept safely with us but likely it would happen to be stolen they could be secured with another authentication factor that would increase the system security as well as the authentication reliability: usually this is achieved with a PIN. The PIN solution is quite good because people are used to operate PINs with their ATM cards or with their mobile phones. The PIN operability is a quite simple:

I. the token can't be used without the PIN
II. the PIN is usually a four decimal digits password (even if many tokens could use PINs up to eight digits long)
III. almost all tokens lock themselves after three consecutive wrong attempts to use a wrong PIN

The fact that PIN are four digit long only (but this is not always the truth) could help some determined hackers to guess our password with systematic trials and this poses special regard on PIN choosing by its owner (all previous considerations about password choosing are now of great importance). There are three techniques for implementing a PIN and they are all stressed in the following paragraphs.

2.5.1.2.1.1 PIN appended as an external password

This technique is usually implemented for those tokens that have no keypad. The PIN is combined with some other information as the output of a token: for example a one-time password token displays on its screen the generated password 123456, I append my PIN 7890 and submit the complete password (1234567890) to the system. This is usually referred as a "soft PIN". This approach is the only one usable to secure a keypad-less token and in fact is quite used. Obviously these PINs have to reside in somewhat a file on the server-side and this implies that an hacker could virtually break into the system, sniff the PINs' collector file and then steal a token to work with it. Of course this is a sophisticated attack that involves both physical and network fraudulence thus making it hard to succeed and providing users a quite safe security system using low-cost tokens. Plus the PIN, that is always the same, is sent along with the password on the net increasing stealing' possibilities.

2.5.1.2.1.2 PIN as an internal password

In this case the PIN is stored in and only into the token: it works as a password to grant access to the token functions. Being the PIN stored into the token this could provide the token itself the means for detecting guessing attacks and either blocking itself or delaying operations. Two points requires our attention: once the token is blocked the user must bring the token to a system administrator to reactivate it with a new PIN and this could take a long time (or it could be not feasible if a user is actually working abroad for his/her company). Since the software PIN is into the token a high sophisticated attack could manage to have the token saying its own PIN: though this should require great skills it is not impossible to be done.

2.5.1.2.1.3 PIN as part of the base secret

The third approach provides probably the more secure protocol towards PIN attacks. In this case no software PIN is wired or saved into the token neither a copy of the base secret: only a part of the last one (useless because stored incomplete) is saved in the token, the user digits his PIN and this is combined with that to calculate the password. The user could verify if he has typed in the right PIN only once he has submitted (or the token had done that for him) the complete password to the server. No attacker could steal the PIN as it doesn't resides on the token nor it could sniff the base secret as only part of it is saved on the device. A greater challenge for attackers is to verify on server-side whether the PIN was correct and verify his guesses but some could use another technique to PINs' guessing: if one is able to sniff several "good" passwords being transmitted to the server then he could use them to verify his guesses but first he has to copy the PIN software to have a base for his trials. Despite the fact that this is quite difficult to do we have to notice that every single vendor implementing these functions use different protocols, functions and algorithms to achieve this and so, if an attacker is able to fool one vendor's protocol he want be able to automatically break all other vendors' PINs.

2.5.1.2.2 One-time password tokens

[10] As reported in the previous paragraphs password systems are quite difficult to succeed in affording good security reliability because a good password is such that you can't remember or change it every time. Of course this would be impossible to achieve unless a dedicated device operates for users such a computation: this is realized through the use of one-time passwords tokens. In this case the framework is totally different from the nonce mechanism because there the password was always the same and a randomly generated value was used as challenge object to make more difficult the decryption of the password. Now the password itself always changes, according to some pre-determined rules, so there's no way for a hacker, even if he's able to sniff one of these sent in clear over the networks, to newly use it unless he's able to purport a live man in the middle attack and masquerade as the server. In fact when the client will try to log again on the system the password will be absolutely different so even if this hacker is able to break the security system once he won't be able to do this again for free (he will have to put in action a new man in the middle attack but notice that this kind of attacks are quite complex and requires almost 24/24 monitoring of the networks on part of the hacker). The strategies for generating one time passwords are counter-based or clock-based ones. Tokens implementing counter-based technology use a base secret (stored into the token and used only for token's reading operation) combined with a synchronized counter to generate one-time password. Clock-based tokens use instead of a counter a synchronized clock. So every new password combines a base secret strongly secured in the token with some arbitrary value recovered on the token as well. Two paragraphs report the peculiarities of both the implementation techniques. Before we enter the peculiarities of each system we would like to underline that virtually with one token we could authenticate to more than a server selecting for which kind of service the token should generate a password: in fact the base

secret and the counter are low-memory resources and virtually more than a couple of this kind could reside in each token deserving the scenery of multiple tokens to bring along with many multiple keys we do carry over with us yet. Of course this would happen only when a high standardization degree is reached between the different vendors and algorithms producers but trading off part of the security features of tokens related to the richness of different algorithms and computational techniques implemented on various tokens.

2.5.1.2.2.1 Counter-based tokens

This kind of tokens incorporate an internal counter and use it to generate different and fresh password every time the owner of the token needs one. Usually this request is submitted pushing a button on the shield of the token: according to somewhat function the token increments the counter and shows this number on its display. To enhance the security of this protocol the password could be eventually hashed before is sent and the initial values should be inserted, according a strict protocol, only from an authorized administrators for each couple of token and server counters. This frameworks works as long as both the token counter and the server one are synchronized but since the password button could be pressed for error resulting in a mismatch between the two of them some recovery or re-synchronizing system should be implemented. In fact most tokens vendors support their products with a specific recovery systems: double-length password technique (even if many other protocols are in use from specific companies) is provided on the largest part of market products.
With the double-length password technique most tokens assume that a users, when a logging attempt fails, will immediately try to log again: by storing the first password and receiving the second the server could move along its list and verify if it could find a couple of subsequent passwords such the one submitted by the token. With this procedure the server can authenticate the user with a high degree of confidence (because it’s highly unlike that an hacker could on-line guess two consequent and different passwords) and resynchronise with the token setting the counter to the required value.

2.5.1.2.2.2 Clock-based tokens

These products combine an internal clock value with a base secret to generate passwords and authenticate the users. The same huge synchronization problems raised up with counter-based tokens are to be faced even in this framework: but in this case a drift is to be expected due to the transmission time the password takes to cross the network. Moreover a possible additional drift may occur, for example, when the token’s battery is nearly exhausted. So the server checks the password using all the clock values within a determined time window. The setting of this window strongly depends on the environment and the use of the token: in a small secured network where tokens are monitored and frequently replaced a time slice of 30 seconds could be enough for local authentication but for general web applications that ought to authenticate hundreds of users from all over the world, might be using a slow free Web connection, typically the time window is wide up to three minutes. Anyway we should notice that a large window increases the threat of password interception and provides greater chances for man in the middle attacks succeeding.
The problem of resynchronization is quite huge even for these products: authentication servers must maintain clock drift information for each token and update it every time a

successful log-in takes place. If a user doesn't active the token for a long time the drift between this and the server could grow more than the fixed time window. In this case RSA SecurID® servers solve the problem in a counter-based similar way: reject authentication , save the received password and wait for a subsequent password (double-length password): if these password they receive, though out of the time window, are compatible in drift and in delay of arrival server successfully authenticates the token and synchronises is own clock to the new value.

2.5.1.2.3 One-time password tokens issues and characteristics

So this hardware device avoids problems associated with password reusage through a hardware-based authentication protocol and usually, as we noticed, the token has to be unlocked by the user with a PIN or passphrase. Combining this PIN and other various data stored on the token and/or received from the server:
-a) the token itself generates a password that is valid only once;
-b) the server is able to generate on its own the same password performing then a matching operation with the one received.
Obviously any replay attack would result ineffective in this scheme. Proceeding with technologies' overview we take a brief illustration over tokens topic issues.

Issues:

i. Vendor dependency: dependence on vendors to supply physical token cards causes risk of future hardware support without a guaranteed compatible format.
ii. Distribution – distribution of tokens to customers needs to be considered according costs and time of realization.
iii. Cost: cost of implementing a token system has to be considered. One of the additional costs for using token authentication systems is the token itself. It is important to note that there are many instances where user's report lost tokens which lead to additional replacement costs. The costs of the tokens themselves needs be a critical component in a company's evaluation process. Consideration of both the application software implementation costs as well as the token hardware need to be taken into consideration.
iv. Accuracy: the accuracy a token system could reach is considered high but is strongly dependent to the chosen implementation of the synchronism mechanism. If the methodology and logic incorporated into this method provides excellent reliability in authenticating users the product implementing will probably afford a valuable security level .
v. User operability : user operability of token authentication requires that the end user must maintain a piece of hardware. This additional hardware requirement can become inconvenient. Nothing more could be said about this because most people is today used to work with PINs and magnetic cards.
vi. Social acceptance : One disadvantage of the token is that some individuals will not accept the token as a means for authentication due to their fear of losing the device. If the user needs to pay for replacement and replacement cost is high, then the acceptance of such a device will be diminished.
vii. Product lifecycle : three factors need to be taken into account to assess the lifecycle of token authentication. First, the token itself is usually portable. As such, the replacement cost for lost, stolen or misplaced tokens tend to reduce the lifecycle of any such token. Second, token authentication systems are still evolving in many different directions: this is due no more to the youth (token technology is now quite mature if we consider that the first tokens appear on the market in the early 80ies) of this technology but to a lack of a real standardization developing during this years. So new technologies consisting of lower transaction costs and increased features is being developed continuosly in many

different protocols. Companies must consider their strategy and how this fits in with token authentication. If the company desires to have the latest and smartest token authentication scheme available, then the lifecycle of a token scheme is drastically shortened as this is not a stable technology nor a common standard (even though lately all vendors are slowly moving towards (hardware) USB and (software) JavaCard™ technologies. At last, but not at least, the 95% of tokens' market products claims tamperproof shields and erasable memories in case of tampering-attempts: this implies that once the battery of the token is exhausted (and usually this happens after three to five years after vendor's release) it must be returned to the vendor or trashed to be replaced with a new one.

viii.Ease of installation: the process of implementing a token authentication system into the existing computer environment can be time consuming. The process requires setting up a server (a dedicated server), issuing a token to each user, training the user on how employ the authentication process, and setting up the database to maintain the tokens and possibly individuating recovery techniques or reserve procedures in case of token loss .

ix.Non-repudiation: the token scheme could possess, according previously undertaken agreements between the principals, a medium level of non-repudiation. By framework it does not give absolute assurance on who initiated the transaction since tokens and their passwords can be stolen.

x.Administration: as soon as a user loses his rights towards the resources protected by its token the device should be immediately deactivated because the administrator could hardly physically access the token.

Advantages:

i.Ease of use for users as they only have to remember a single PIN to access the token and authenticate themselves to the system.

ii.Ease of management as there is only one token instead of multiple passwords.

iii.Enhanced security as the attacker requires both the PIN and the token to masquerade as the user (two factor protection).

iv.Better accountability as tokens are tangible.

v.They are mobile in comparison to digital signatures.

vi.Mature solutions that have gained widespread market acceptance and deployment.

vii.Consistent with Thin Client approach, no client side software component.

viii. With USB tokens nor hardware component is required on the client side.

ix.Mobility and portability: since security is not tied to the specific machine, users could possibly access with any browser based internet connection correctly profiled to do that.

x.Browser independent technology.

Disadvantages:

i.Client required to carry a token card.

ii.Tokens need to be replaced every 4 years.

iii.Ongoing operation costs to keep track of token cards.

iv.Possible barrier to customer acquisition since client cannot use services until he receives the token card.

v.Longer time to authenticate the identity of the user as numerous steps are required to authenticate the client.

2.5.1.2.4 Smart cards

A smart card is a credit card sized plastic card with a microprocessor chip embedded into the card itself. This kind of tokens can be contactless (RF transmission) or contact cards.

Usually the smart card readers makes electrical contact with various connectors to feed data in and out of the chip. The card is “smart” since it contains its own processor, memory and operating system and so it's able to perform many kind of operations such as reading, writing, encrypting and decrypting, inverting, and so on. The smart card has considerably more abilities than other tokens because of the embedded microchip. Usually used as a substitute for user ID/password systems it achieves a much higher level of security through dedicated communication protocols between the reader and the smart card and between the smart card and the host. Smart card technology has actually spread all over the world and it is used in many application fields. Costs are actually decreasing as well as market’s requests raise and technology improves.

Issues:

i. Administration : central update of rights profiles on smart cards needs to be maintained; administration/issuing authority and secure logistics are necessary to ensure that this system works efficiently.

ii. Mobility and portability : public key certificates and private keys can be utilized by web browsers and other popular software packages but they in some sense identify the workstation rather than the user. The key and certificate data is stored in a proprietary browser storage area and must be export/imported in order to be moved from one workstation to another. With smart cards, the certificate and private key are propriety of the user, and portable, and can be used on multiple workstation, whether they are at work, at home, or on the road. If the lower level software layers support it, they can be used by different software programs from different vendors, on different platforms, such as Windows, Unix and Mac.

iii. Costs : price of implementing and maintaining this type of system compared to that of other token alternatives is expensive. Lost/forgotten smart card replacement costs also need to be taken into consideration.

iv. User operability : User operability of token authentication requires that the end user must maintain a piece of hardware and keep it with him/her.

v. Social acceptance: two opposite factors affects this issue. Since a smart card operates virtually identically to a credit card, the user could perceive this token authentication device as just another piece of plastic inside his wallet. On the other side users are more comfortable with associating ownership with smart cards as they are trained yet in using and protecting physical objects through experience with campus id cards, mag-stripe cards etc.

vi. Product Lifecycle: as smart cards are usually portable, the replacement cost for lost, stolen or misplaced tokens tend to reduce the lifecycle of any such token.

vii. Ease of installation : the process of implementing a smart card system requires setting up the server, issuing a card to each user, training the user on how to employ the authentication process, and eventually setting up a database to maintain the smart cards on your own.

viii. Non-repudiation : the ability to deny, after the fact, that your private key performed a digital signature is called repudiation. If however your private key signing exists only on a single smart card and only you know the pin to that smart card, it is very difficult for others to impersonate your digital signature by using your private key.

Advantages :

i. "Single sign on". Instead of having to enter passwords or token codes for every new service, the user can just insert the card and enter the pin to activate the card and let it perform for him authentication for every subsequent service he would like to sign on.

ii. Key features of the smart card are digital signatures, where the smart card can prove that a particular user signed a given document.

iii.A smart card can contain all kind of data (up to 200K in latest release) needed to personalize networking, such as certificates, personal data, keys, secrets and much more information.
iv.Versatility of possibly combining credit, debit and stored value cards in one convenient platform
v.Multifunctional use possible (access card, time recording ...) Smart cards may carry multiple applications which may, in principle, be added or removed during the card's lifecycle. This can considerably aid the business case for the introduction of card technology, since a single card can be used for multiple functions by multiple organisations.
vi.The processing power of a smart card makes it ideal to mix multiple functions thereby enabling banks to manage and improve their operations at lower costs and offer innovative services
vii.Ability to carry out off-line, on-line and peer-to-peer transactions
viii.Secret key information is stored tamperproof on the card secret key operation is performed directly on the card, therefore no Trojan horses can spy the secret key on the PC.
ix.High security when running cryptographic operations.
x.Rights, profiles and keys are stored with the user (better support of travelling users).
xi.Public Key Infrastructure systems are more secure than password based systems because there is no shared knowledge of the secret. The private key need only be known in one place, rather than two or more. If the one place is on a smart card, and the private key never leaves the smart card, the crucial secret for the system is never in a situation where it is easily compromised. A smart card allows for the private key to be usable and yet never appear on a network or in the host computer system.
xii.Smart cards can enable multi-authentication by accepting a thumbprint on the surface of the card in addition to the PIN in order to unlock the services of the card. Alternatively, a thumbprint template, retina template, or other biometric information can be stored on the card, only to be checked against data obtained from a separate biometric input device.

Disadvantages:
i.Special reading hardware is necessary for users (unless you use USB tokens with smart card capabilities);
ii.Lack of a complete standard infrastructure for smart card reader/writers is often cited as a complaint. Even though lately major computer manufactures are moving towards some kind of standardization they haven’t given much thought to offering a smart card reader as a standard component. Still today much smart cards works with different power-supply voltages, especially referring to proprietors’ functions implemented on the cards. Moreover many companies don’t want to absorb the cost of outfitting computers with smart card readers until the economies of scale drive down their cost. This dissertation is much true regarding smart cards’ programming devices.
iii.Concerns about different groups accessing the information on the card.

2.6 Authenticating through what you are factors: biometrics' use features.

Technically speaking we must consider some relevant aspects of biometric technologies which cannot be undervalued. Obviously new patterns and techniques brings with them new moods and problems: in this paragraph you can find an overview of the main news and difficulties that the use of Biometrics for authentication endeavours.

i. Every time we take a biometric template from a person that sample will never (almost) be exactly equal to any other template of the same person: the data richness included in a biometric sample is hugely high.

ii. So any time you live-acquire a particular biometric feature you must be ready to authenticate someone whose credentials are, for example, for X% (with X<100) similar to the template credentials stored on dedicated device: probably administrators wouldn't feel absolutely safe doing this.

iii. Dedicated recognition and matching techniques has to be implemented to avoid both false-acceptance and false-rejection events. If a too high matching percentage is required the possibility of false rejection (**f**alse**.r**ejection**.r**ate**.**) increases strongly, might be leaving out the system authorized users. If we tolerate a too loose acceptance the risk of intruders getting in the system from the main door (**f**alse**.a**cceptance**.r**ate**.**) is reasonably high. According to our security concerns in the first case the user could address a false rejection as a system-fault or a network error (if its rate is quite spare) and he will try to log again while in the second chance we would log an outsider onto the system granting him a set of rights : the first situation is far more good than the second as far as a false rejection seldom occurs (in the order of one false rejection every thousand trials). See annex ? for details.

iv. Legal constraints are even greater and widely variable: whenever we talk about biometrics we deal with a person's unique characteristics that, as stated by international laws, belong to him/her solely: we must be able to guarantee that this information won't circulate on the internets without strong protection or, in the end, that it won't circulate at all out of a trustworthy host. So the use of biometric technologies regrets the scenery of big distributed (or even central) database on which we store lots of personal data: whatever a hacker penetrate our system we should have contacted a really expensive lawyer yet. Moreover every country has different regulations about the use of private citizens' data for commercial or administrative uses so that it would be difficult to address a standard that could be able to fit every single state in the world (standardization has made fortunes and misfortunes of every single protocol developed on the WWW and of IT projects in general).

v. Nevertheless we should remember that we have "only" no more than ten fingers, one voice, a face, etc. .If we could easily change a pin thousand times during a whole life we could be able to change the fingerprint template no more than nine times so we must assure strong protection of this files basically through data encryption. That's why security concerns assume great importance in our effort and oblige us to take special care of. Likely these measures could not be enough strong to protect us because, for determined hackers, it would be quite easy to steal during a phone-call our voice's features or copy our fingerprints dating with us a fake brunch.

vi. Ease of use and integration capability with actual systems is the first step towards diffusion of biometric-techniques: the intrusiveness of some biometrics, the slow acquisition-time of some others and the probable environmental disturbs during sampling bring many doubts about ease of realization.

vii.As well as granting a user “standard” credentials involves dedicated protocols even the enrollment of a “biometric” user proposes a large amount of problems: due to the new factors considered a biometric enrollment consists of a much longer and complex operation that adds to the old protocols at least some others biometric readings of the physical trait the system is going to verify in the future. From these multiple readings, taken for the sake of accuracy, the system should abstract a definitive template merging with special dedicated averaging functions the live-scanned samples acquired. Then according to some special protocols the template is to be stored where it will reside for future uses.

viii.Of course creating a fake identity and enrolling it on the system for rough future use becomes more difficult than with the basic scheme [userID;password] but nevertheless it is possible to steal someone's credentials from a fingerprint reader for example: that's why we have to deal with the **live-scan** feature. Any time we perform a matching test between a sample and a template we must be sure that this sample comes from a contemporary scanning process and that we aren't victims of a replay attack pursued by an intruder.

2.6.1 Many biometric technologies

In the lines below a brief description of actual biometric technologies is developed and then a comparison scheme is reported where main features are summed up :

- *Hand geometry*
  Self-explicative as the title is we can only add that finger-size and shape are the keys of this characterization. This biometric doesn't allow great precision and the possibility of hand shape changes due to aging or by accidents is quite relevant.

- *Retina scan*
  The blood vessels behind your retina is going to be completely scanned by a low-power infrared reader while you're watching, as still as you can, an image appearing throw the lenses. Tolerance towards this technique, excluding temporary enthusiasms concerning science-fiction, is not so good: costs are remarkably high, ease of use it's remarkably low, integration with existing machines is not simple but accuracy is extremely high, as well as operative times.

- *Iris scan*
  Quicker, easier and less intrusive then retina scan this technique could probably break out in use as soon as costs will decrease because a good level of accuracy is guaranteed.

- *Face scan*
  Most of the doubts linked to this technology concerns aging but even usual changes we bring in our look: hair length and colour, glasses or lenses, beard, etc. or...

- *Voice recognition*
  This technique abstracts the voice sound features that belongs to a person solely. This is achieved having the user read sample phrases. The result of any voice-scan could be strongly affected by environmental situation and/or voice modifications due to illness.

- *Fingerprint*
  Fingerprint scan is usually restrained as a police technique to achieve criminals' recognition: definitely this biometric for ease of use, good accuracy, non-intrusiveness, hardware costs, etc. is the one that will some quickly spread.

- *Signature-style*
  Using a special pen and a dedicated pattern the system analyses signature features as: velocity, speed, pressure moreover than the signature's static shape. Even in signing is an everyday action in social life this technology doesn't seem to guarantee many possible applications.

- *Key-stroke pattern*
  Everybody uses the keyboard according a personal style different in speed, pressure etc. Typing a predetermined string on the keyboard will make the system able to recognize whence the claimed user is the same interacting with it.

- *Ears shape*
  Ear shape is the last bound of U.S. contemporary studies in biometrics and make us pick something up: every single part of human body represents a vault of information all different from other bodies and so every feature could be used in a biometric authentication scheme.

ID type	Strenghts	Weakness	Cultural concerns
Fingerprint scan	good accuracy actually cheap small readers mature technology	a low percentage of population does not have usable prints finger dirtiness	some countries prohibit fingerprint use other than criminal justice
Iris scan	High accuracy Unchanging feature	actually quite slow perceived as intrusive good user cooperation needed	unacceptable for some cultures
Retina scan	Highest accuracy Lowest feature	high intrusive actually slow readings total user cooperation	unacceptable for some cultures
Face scan	Cheap not intrusive	less accurate good user cooperation based on easily and greatly changing feature	unacceptable for some cultures
Facial Thermogram	rated very accurate	not available yet difficult outdoor usage good user cooperation required	?
Voice Recognition	Not intrusive The only biometric capable of telephone Authentication	less accurate affected by sore throats and colds	none
Hand geometry	Not intrusive fast low data storage	less accurate large readers supposed to age changings	none
Signature dynamics	Fast not intrusive user habit in signing	less accurate multiple samples usually required analphabetism	none
Keystroke Pattern	Not intrusive user acceptance	less accurate unusable without a keyboard, affected by much factors	none
Ear's Shape	Not intrusive low data storage User acceptance	not available yet large readers supposed to age and look changings	none

2.6.2 From a classic Authentication Scheme to a Biometric Authentication model.

As we dealt with other medias we are going to take a brief review of topics that outstanding in biometric authentication, underlining advantages, disadvantages and issues. But we must always remember that is not definitely true that physical human characteristics are unique and cannot be borrowed, misplaced, forgotten, stolen, forged or duplicated. This could seem a science-fiction scenery but for example part-replacement surgery improvements could lead to new perspectives actually hidden.

Issues:

i. Physical variance: when using biometric devices some problems may occur due to technical difficulties in measuring and profiling physical characteristics as well as from somewhat variable nature of physical attributes.

ii. Vendor interoperability: a number of different vendors exists currently and most of the hardware and software are incompatible as the technology is not mature and a standard de facto has not been determined yet.

iii. Costs: biometric devices tend to be expensive (at least more than password systems!). Biometric systems does not require much as far as future replacements or maintenance through out the life of the product but they are relatively expensive initially. Other items adding to the cost of implementing a biometric system include the cost to train personnel to use the system, and the initial set-up cost to enrol users in the system.

iv. Accuracy: since biometric technology is based on a unique physical trait of a human being, its accuracy to correctly identify authorized users while rejecting unauthorized ones does not always have to precise. The level of accuracy of biometric systems is based on the setting of the comparison algorithm. The comparison compares how close the digital representation is to the stored template. For many systems this threshold can be adjusted to ensure that virtually no impostors will be accepted or to ensure that virtually no users will be rejected but hardly we could tune the device to a perfect balance. This is probably the outstanding challenge of actual biometric research as it's still quite huge to determine a perfect balanced threshold.

v. Operability: the biometric scheme is very convenient since the user does not need to remember to carry some form of token with them. This high level of convenience is offset by the speed of authentication as being in the lower range. Depending on the number of user templates stored in the database, it may take a while before a match can be placed, thereby decreasing the speed of authentication.

vi. Social acceptance: biometric is viewed as having an high level of social acceptance. This criterion depends on the type of biometric device used as some biometrics are seen as intrusive.

vii. Product lifecycle: as some physical characteristics of a person change over time, routine updates would need to be made to the host server in order that proper levels of authentication are achieved. With respect to the life of the system, most biometrics systems consist of the use of a scanner, a CPU, and other equipment such as video monitor and cameras. These items don't have a relatively long life span because they can be easily (and even accidentally) damaged or corrupted. As such, biometric authentication systems generally don't have a longer life span.

viii. Ease of installation: these devices require enrolling each individual using the biometric authentication method into the system. Some vendors state on their white papers that it takes two minutes to enrol a user into the application. If you are enrolling a significant amount of users, the process can take a significant amount of time but consider how long enrolling a "normal" user, with the pair [userID,password], takes.

ix.Non-repudiation: biometric devices in theory possess a high level of non-repudiation but according to local laws, even if these devices are based on unique characteristics of one's being, a user, could deny that his fingerprint initiated a transaction without further more controls being implemented. We absolutely must not think about biometrics as a non repudiation device!

Advantages:

i.Biometric might offer a high level of assurance in the authentication of users as it's hugely difficult for one to easily and low-cost fake the physical characteristics of another (at least you can’t do this with one software device) but environment and implementation concerns could strongly affects these capabilities.

ii.Easy to use as the user only needs to present himself and needs not remembering data or carrying tokens: this might represent a huge improvement in system user-friendliness.

Disadvantages:

i.Most implementations require special hardware input devices at each workstation

ii.If the physical reader can be bypassed, such that biometric data derived from the scanning can be entered, then the person can be impersonated.

iii.Biometric devices are unreliable under abnormal environment circumstances (i.e. Dirty fingers or High noises in the office).

iv. Biometric devices require special assistance because even if employees fingers are clean the glass of the reader(camera) could be dirt.

v. Not all biometrics afford the same reliability , ease of use and quickness: this is a relevant condition.

vi.A bad tuned device, accepting too much low-rate matching accesses, could result in a disaster for system protection.

vii. Biometric authentication relies on uniqueness of some physical traits but injuries or time-variances could make a template useless: two solution could be implemented and both of them are not low-cost ones.

a) You could newly enrol the user in the system according to the new measurements reading but what if the injury involves only temporary changes? Or think about biometric storage on a smart card or a token: in this case we should replace the device supporting other costs and may be too long times to achieve operability again.

b) You could provide a backdoor mechanism to unlock the system only in these specific cases but such a backdoor could also be exploited by an attacker.

3 Implementing a remote biometric authentication protocol using smart card.

3.1 Biometrics and user authentication: going in deeper.

Our concern is now to look at the use of biometrics technology to determine how secure it might be authenticating users with these techniques and how the users' job functions or roles would impact an authentication process or protocol. Referring even privacy issues we will try to evaluate, not without lighting its limitations and its critics, biometric technology's real potentialities.

3.1.1 Security and privacy

We have to evaluate how much people would trade-off of his privacy to satisfy the needs of authentication: the overall concern for privacy and how unique identifiers, no matter of what kind , will be used (and today this a growing feeling among the people) faces the problems of stronger security in IT operations (and no stronger security could be achieved without doubtless authentication). Nevertheless stronger authentication could be much easily achieved using all three factors forming the authentication trinity. Specifically users recognize biometrics their power in user authentication but relate great concerns about how and where this information will be stored, who can access it, how it can be used, and the real reliability of its usage. Two driving forces promoting the use of biometrics for security today are protection against identity theft and user friendliness. Having stressed yet enough the second topic we could say that concerning the first problematic a telephone number or an address is enough to start someone on the way to stealing an identity. It is a predominant concern for many companies and individuals, especially considering the use of Internet for business continues to experience rapid growth rates. Victims of identity theft know how difficult is to prove that a theft has in fact occurred, as well. A biometrics system would go a long way towards the prevention of identity theft because it is based on something that is specific to one being only. It is unique and can be duplicated hardly. That is not to suggest however that a biometric system is without vulnerabilities. Nevertheless, whether biometrics is or not a security solution in and of itself, biometrics by themselves are not necessarily the problem: weakness and flaws can be introduced into the system largely on how the authentication protocol is implemented. The greater questions that users pose are related not on the use but how, where and why biometrics will be used. Biometrics systems developers should be able to exhaustively answer these doubts addressing exactly: how the biometric data is stored, if it will be archived in a central database or on a smart card under user's control. And if is stored on a database users will probably ask whether it's stored in clear and linked to personal information about the individual or if it's stored anonymously in an encrypted and reverse-engineering-proof way with no link to related personal information. Data segregation of personal and biometric information should strongly apply for biometric applications, especially those storing information in a centralized manner. The public is better educated in technologies today and is aware of the fact that , because of wider and wider use of Internet for business, certain risks to individual privacy occur. Despite Scott McNealy's (Sun Microsystems inc.) famous web quotation "You have zero privacy anyway" the debate over the security and privacy trade-off will continue until the public is satisfied with how implementations of biometric systems affect their private lives and protect their interests. [5] The points that vendors should stress much as focuses of people attention are:

- is this authentication system able to mute itself into an identification system? If this is possible producers should encounter much mistrusts.

- Overt or covert technologies: these systems will be capable of operating without user knowledge and consent or they won't.

- Behavioural or physiological characteristics will be the basis of this systems: behavioural biometrics are considered likely to be changing and are perceived as less intrusive and less strictly personal related.

- These systems could receive or grab the characteristic to be measured: the first kind of products works only with user cooperation, the second ones can acquire user images without the user initiating a sequence.

- The template, as reported before, could be stored in clear or encrypted, on personal support devices or in central database.

Once these topics will be completely developed and the users' acquaintance of these problems will be plain enough biometrics systems will fully encounter the market's approval. Lately in many different biometric technologies (such as fingerprint recognition for example) the biometric templates, not only for storing convenience, are based on elaborations of the sample and reduced to such file formats that makes impossible to reproduce a sample or a reading of the physical character from the template. This enhances not only the portability of such files into tokens but even the privacy perception retrieved by users that feel safer. Unfortunately this is true but at a first sight because this file could be always used to verify and track the users movements on the Web every time he achieves authentication through this specific biometric.

3.1.2 Biometrics and Smart Cards

The combined use of biometrics and smart card sums the advantages of the two technologies enhancing the security of the authentication protocol. This combination raised as a matter of trustful authentication but still more than a security caveat could affect the implementation of this kind of systems and usually, if we try to prevent unauthorised accesses, we could use a three factor authentication protocols that involves even a PIN to primarily unlock the card for the biometric testing. Combining these factors we should had achieved the strongest combination of information needed to provide authentication into a system. But the specific protocol and its implementation could always hide some dangerous flaws. A typical example of a protocol involving all the three factors and concerning fingerprint recognition could be this:

I) insert your smart card in a reader or in the USB port of a workstation

II) enter your secret PIN to unlock the smart card

III) place your finger on the scanner and have the sample compared to the fingerprint template

IV) if the data matches the smart card secured private key could be use in somewhat way, for example encrypting a nonce sent by the host’s application

V) the application can now verify that a certified key obtained from a valid certificate encrypted its nonce and verify, using the public key as well, whether the nonce is the same it has sent.

Although this protocol involves all the three factors of the trinity, a smart card , a pin and a finger none of the readers of this paper would feel safe in using it because we have no information on its implementation, we do not know where the sample is taken and how is sent to the smart card, we don’t know how the PIN is used by the host workstation, we cannot trust the workstation itself and we do not know if the reader has been manipulated by thirds. This demonstrates that it’s not true at all that using more than an authentication factor could lead to strong and certain authentication unless protocols (and their implementations) are strong and secure! So using a smart card at its best we could achieve a safe encrypted storage for the biometric template, addressing much of the privacy concerns exposed in the previous paragraph (but not all of them because we have no certainties that the template will not exit the smart card) and avoiding large on-line databases appealing the attention of all Web’s hackers. Using a biometric factor, might be combined with a PIN, we could grant higher recognition rate. Answering the first question we‘ve posed talking about secure storage and smart card we could report more than a technique to interact with the template: each of these represents different challenges and grants variable security features. In the following three paragraph they are briefly explained.

3.1.2.1 Template on card

In this scheme the smart card works as a secure biometric storage. Whether the card stores a biometric template only or a private key as well the access to the smart card read memory is granted after the recognition of the user addressed as the smart card owner. When a workstation requests access to the smart card , the card emits the pattern and the workstation compares it to the biometric reading collected from the operator: the host decides whether the user could be granted access and tells the smart card if the template and the sample match. This approach appeals to people concerned about biometric privacy: individuals perceive personal control of their biometric data addressing the smart card as a safe container. Moreover no large-scale database is required on system side. These simple advantages are largely overweigh by the flaws of sending the biometric patterns out of the card and having third party, not trusted for the card, decide if the attempting user is the authorised owner. An attacker could easily bypass if the biometric matching relies on software running on the operator’s workstation: replacing that software with a subverted program that told the smart card that the biometric matching test has come to an happy end the intruder could easily access the read-only memory of the smart card without collecting no signature from the operator. A first solution could be addressed, always using the card as a secured box, integrating the smart card and the biometric readers in the same device but even in this case, even though the biometric never enters the host, the problem is only temporarily removed as now we must be sure to believe the reader itself and this is not so easy. At least we achieve an increase of the attacker’s work factor.

3.1.2.2 Match on card

This approach is slightly better because on the workstation side only the biometric reading is required. After the operator submits the card the sample is the card to unlock itself matching inside its body the patterns received. Using its own processing capabilities the smart card decides whether the sample matches the owner’s template closely enough providing the workstation use authorization of its data. This scheme is similar to collecting a PIN and presents the same danger as above: we have no certainty that this biometric reading has been collected through live-scan and, as well as what happens with PINs, there’s the same risk of an attacker’s sniffing the biometric and later using it to unlock the card in a replay attack. Nevertheless the biometric matching procedures places additional demands on the card’s resources: the card must have additional program memory to store the biometric matching procedure and will require additional working storage for performing the match calculations. Easily guessable this implies higher costs of the devices involved, cards themselves as well.

3.1.2.3 System on card

Embedding a biometric reader on a smart card provides all the privacy and security answers (regarding the template and the secret data on the smart card) we are looking forward to but presents more than a realization problems and implies highly rising costs. This time we are obviously speaking of fingerprint biometrics because it’s quite hard thinking about a hand-palm reader embedded on a smart card unless this is a very large one. Since the card accepts samples only by its own built-in reader it addresses the problem of live-scan, biometric sniffing and replay attacks thus the template will never leave the card and neither the host will access the private read only memory of the card without the card providing it explicit permission. On the other side this poses a great challenge for smart card technology because it requires a lot of processing resources plus a good quality fingerprint reader packed into a tiny device. Actually more than a product exists that resumes the previous features (for example Sony Puppy) but none of them has typical smart card body’s measures.

3.1.2.4 Storing the template on the smart card

Of course the use of a smart card could help us making the system-authentication safer but this doesn’t mean that what we are going to store on the smartcard will be safer too. According to some specific vendor’s device there are some dedicated format to contain biometric templates. These templates can be saved in a covert encrypted manner or in clear on this smart cards but the use of certificates could introduce some kind of standardization. In fact some special certificates could be filled in their extensions with the biometric template thus linking the template genuineness to a recognized authority certifying the origin of the file but bringing up again the problem of biometric info related to sensible personal data; this trade off can't be easily overtaken since linking a biometric template to a certificate means linking it to personal related information certified by a central official authority meanwhile storing the template as a single digitally signed (or encrypted) file means cutting off some of the reliability of the genuine origin of this template that a C.A.'s signed certificate doubtlessly assures. In the end it will be necessary to make a wise choice according the protocol specification because actually there's no solution smartly facing the problem.

3.1.3 Using biometrics on unsecured networks: main issues

The use of biometric information introduces new privacy concerns over the huge challenge of granting secure remote authentication over unsecured networks. Today Web growth implies new schemes that can't be easily addressed with old practices but neither with new ones. The growing complexity of the WWW implies on-line 24hrs threatening of valuable data stored on your system and possibly a greater quantity of potential hackers interested in penetrating your network. The use of biometric does not represent in itself the solution to this concerns but introduces new possibilities in implementing a safe authentication protocol. We should wisely consider that new methodologies bring along with them new undiscovered threatens and, as we reported in the previous paragraphs, biometrics is not rid of them.

The main problem of remote authentication is the use of a third party channel that is totally untrustworthy and usually unsecured. No matter what kind of information our IP packet is conveying over the WEB this could be discarded, intercepted and replaced whether strong defences are undertaken on both side of the communication. Many products promise the "construction" of a solid channel via the web to safely interchange information between two points of the web but as the complexity of the security design grows even the possible break points number increase. In a generic biometric remote authentication scheme those are:

I. The biometric reader: someway we must prove that the sample he submits out of its box is genuine and is the result of a live-scan.

II. The link between the biometric reader and the local host: even this link could be eavesdropped and we must take care of this eventuality.

III. The local host workstation is another possible break point.

IV. The channel between the local host and the server presenting all the generic risks of an open channel: interleaving and man in the middle attacks.

V. The device storing the templates, no matter what is, represents a valuable data vault for hackers and it must be properly secured.

VI. If the biometric match test takes place on the local host we must implement a protocol according which the server could trust the local host first and the communication channel as well then.

VII. On client's side is probable the request for server authentication.

Each of these points addresses a wide range of problems and concerns quite huge to solve and thus they represent an harder step to overcome if mixed all together. So step by step we should try to explain how a solution could be reached.

3.2 Web authentication through biometrics : a possible model

To achieve our goal many choices had to be undertaken : in this paragraph each of these is stressed and underlined according its special considerations and regulations. Our protocol is based on SSL and will deploy a fingerprint reading system using smart cards as a personal secure vault for the biometric template. In the following lines the how and why for each choice is developed according our previous considerations.

3.2.1 Using fingerprint biometric systems

Among the many biometric techniques we choose the fingerprint scanning system to implement our protocol. Fingerprint readers focuses on the ridges patterns that characterize in a unique way each finger all over the world: so that there's not one fingerprint equal to another you can collect. In this section we would like to introduce the reasons of our choice and give a more technical reading of many concerns regarding fingerprint scanning systems. These systems represent a good compromise between cost and reliability supported by a quite mature technology. Of course many aspects of fingerprint scanning is to be considered because there's not one reader and one matching algorithm but many packages on the market present solutions with predetermined performances which undermind some specific aspects that we do not want to be underminded at all. In fact starting from the choice of the scanner technology on to the match-test algorithm many differences occurs and could probably affect the protocol development. [6] Scanner readers could be for example of more than a type or technology:

- Optical readers: using a light refracted through a prism and shining on the surface of a glass on which the user poses his finger;
- Silicon readers of two kind: (1) using tactile or thermal techniques based on a sophisticated silicon chip that is activated by pressure or heat or (2) capacitance silicon sensors measuring electric charges and electrical signals from the areas where finger ridges are touching the surface;
- Ultrasound image capture;
- Touchless scanners.

Obviously each reader presents different acquisition times and accuracy, as well as different costs: optical ones represent actually a good choice mediating reliability, usability and cost. Of course once we have a sample image scanned we have to transform this file in a template referring to it for future matching-tests. According the privacy issues developed before the fingerprint template is constructed in a special way so that you can't infer the original image of the fingerprint from the template file and this is really important. In order to have a template to be performed a match-test on we have to choose a special algorithm that create this kind of file starting from the scanned image of a fingerprint. Now the problem is how to characterise this file and decide what are the relevant parameters to be evaluated to address uniquely a fingerprint through a single file.

3.2.1.1 Fingerprint technical issues: creating a template

Fingerprint biometric technology constructs from an image a special file linked to a finger through the use of minutiae. A minutia is a special character of the print ridges patterns and special points: these algorithms place a minutia in every point where a ridge stops, forks or breaks and the result of this digitalization of the image results in a file (that could be txt or a bmp file for example) collecting a variable number of minutiae. In figure ? you can see some minutiae collected from a sample image.

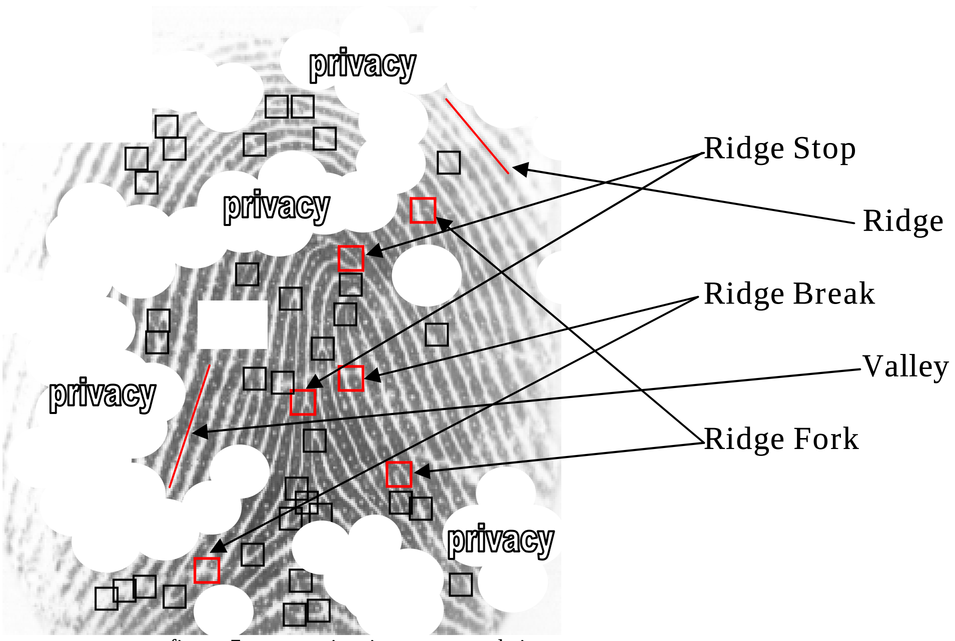


figure 7, some minutiae on a sample image

All the minutiae collected in a file are linked by some special distance and angle relationships and this relationships are the same used in a match-test to perform user authentication. [6] According to recent studies it's hugely difficult to find more than eight equal positioned minutiae in prints from different fingers: now at first glance you can easily infer that the number of minutiae in a fingerprint is extremely high so that using more than forty of them could afford a great degree of reliability for non-critical applications. Of course the minutiae set could be shallower or bigger according to authentication strength required in a special environment. Although there's one more aspect to be considered talking about templates: if you want to store it on a personal device, such as a token or a smart card, you shouldn't create a template bigger than 1kB in fact the space available on smart cards' memory is few and is to be used as smartly as possible.

3.2.2 Smart card as a personal safe fingerprint container

Concerning the privacy regards that we largely stressed in the previous paragraphs we would like to store the template on a secure personal device not to implement a great central database working as a template collector. Thus resulting in user acceptance and giving each one the feeling of controlling his own private data keeping it with himself. Of course this storage device must perform some special operations to fulfil some of the authentication tasks during the protocol communication and it should afford a strong degree of protection for the template. Probably the use of smart card (in the classic or in the USB-token factor) with cryptographic abilities is the best solution to store the template, a public key or a certificate. Many smart cards offers two separate memory areas: one dedicated to private data, that can never leave the card itself, and another to be used for common data. Moreover actual smart cards provide support for both pkcs#11 communication standard and Microsoft© CryptoAPI one making it possible to be used with all browsers for Web encryption applications. Nevertheless user acceptance towards smart card devices is quite good and related to a common trust of good security achieved so that even the public perception towards this kind of devices could be positive.

3.2.2.1 Smart card readers

If we want to use a smart card we surely need an appropriate reader but this could not be true at all as many smart card vendor actually produce devices with USB-token factor allowing this smart card to be used with common workstation supporting USB technology. If this would not be our choice we have to face the problem of purchasing a card reader: on the market is possible to find a large selection of products presenting very different features. Some of them integrate both a scanner and a reader , avoiding the problem of acquiring a fingerprint reader, and have some special licensed software which is able to enrol on the card a template and to purport a matching–test with a live scanned image(this could mean product integration, less hardware to buy and more free space on the desk but even blind trust towards a black box purchased form a third entity that can gives no guarantee). Some others purport only the image reading operation and do not even create the sample from that image. Anyway the huger problem with these readers is to find a protocol that could give certainties about the live-scan of the sample and good performances (if the scanner is integrated) with the matching test. Some packages offers even an open interface with some predetermined functions to customize and develop our own applications using the reader's processing possibilities.

3.2.3 SSL as the basis for hosts communication

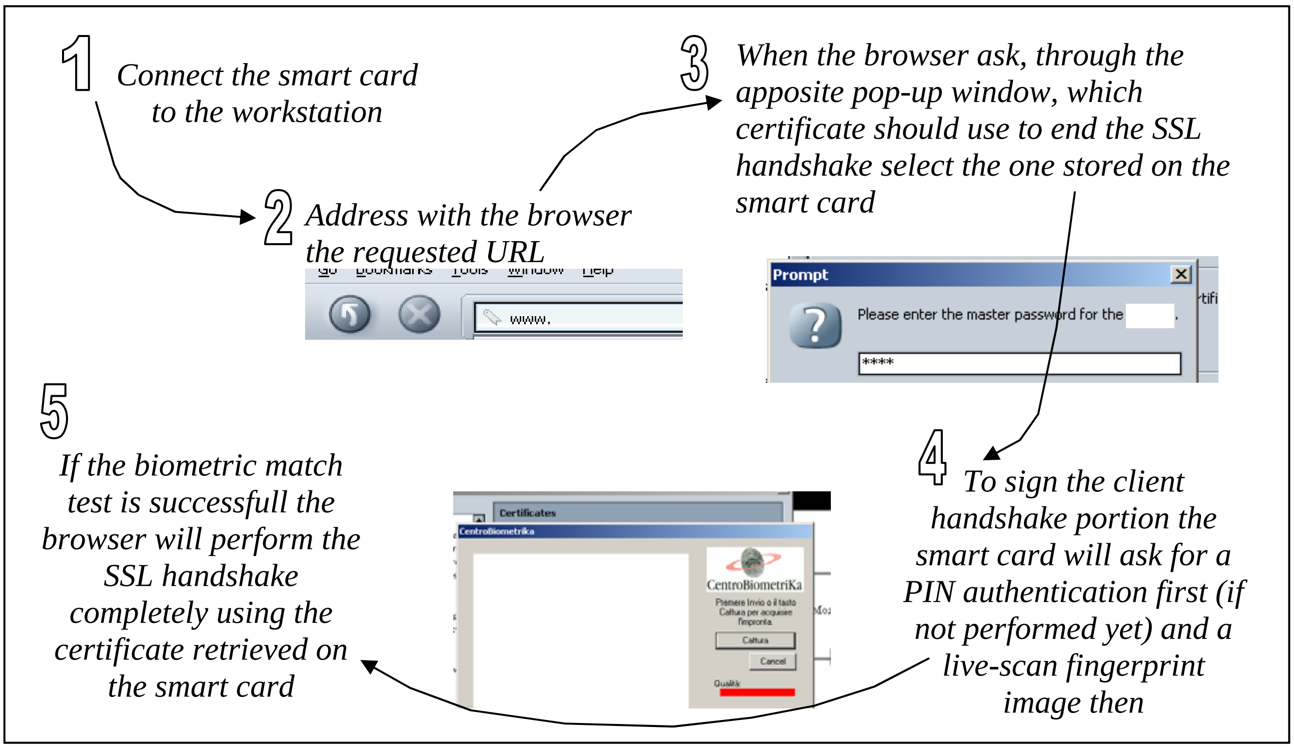
Due to the wide range of solutions actually working on the net concerning authentication we decide that one good solution, according the Japanese "kaizen" theory, is to use the security basics furnished by some of those products modifying ad hoc their protocols to introduce biometric authentication. After an accurate analysis we evaluate that the use of SSL (see annex A) over https web-connections would be the ideal platform to deploy our framework for more than one reason:

1. SSL is for Netscape and Mozilla an open standard (RSA security PKCS#11) whose code is open thus optionally modifiable too.

2. SSL is almost a standard de facto (SSL v2.0, v3.0, TLS): wide diffusion of SSL and SSL product integration within all browsers allows a use of this protocol on the majority of actually working machines.
3. SSL yet supports user authentication through tokens or smartcards thus making our task easier to purport since few changes are needed to introduce biometrics.
4. user acceptance: The impact on user trust of “https” connections is today quite good and this represent a key to win some kind of resistance among potential users.
5. SSL affords by itself a quite strong security system (at least it does if all the option parameters are well set) and so we could possibly achieve higher reliability and tighter protection if our changes are wisely implemented.
6. If we manage to modify SSL we could bypass system and hardware differences due to vendors: approaching the security within SSL every vendor dependency will be bypassed!

3.3 Protocol Specifics

The design of the protocol is quite simple: to achieve secure remote biometric authentication we use over a https connection that is configured to perform client authentication a certificate stored on a smart card. Placing this certificate in the private secured memory area of the smartcard we modify the access-protocol (on browser side) so that the card performs a biometric authentication on to the card itself using first a PIN and then the fingerprint. The fingerprint is required after the correct pin has been dialled since we presume that the sensitiveness of the biometric information has to be unlocked only after a first phase of user recognition had successfully taken place. So that if you rely on a https connection you could rely on this protocol that makes nothing different than having a fingerprint match test perform by the local host in order to fulfil the SSL handshake and authenticate the client to the server. A possible scheme for this protocol is reported in figure 8:


1
Connect the smart card to the workstation
2
Address with the browser the requested URL
www.
3
When the browser ask, through the apposite pop-up window, which certificate should use to end the SSL handshake select the one stored on the smart card
Prompt
Please enter the master password for the
****
4
To sign the client handshake portion the smart card will ask for a PIN authentication first (if not performed yet) and a live-scan fingerprint image then
5
If the biometric match test is successfull the browser will perform the SSL handshake completely using the certificate retrieved on the smart card
Certificates
CentroBiometrika
Cattura
Cancel

Before we would stress on protocol development we have to point out all our choices regarding products, software and hardware we have been working with to implement this protocol. In the following list there are the names of the products used and their main characteristics.

- Smart card device: for testing and development we used an eToken Pro, by Aladdin Knowledge Systems, which is a USB-token factor smart card with 16kb memory. The main advantages of using an eToken is that this is easily usable with all USB-compliance workstations (actually 90%), without any additional hardware, and that is usable both with Microsoft© CryptoAPI technology as well as with RSA Security© PKCS#11 one. In the figure below you can see the aspect of an eToken and read the most important features supported. For further information in the following URL you can find all its sheets and specifications: eToken PRO web page on Aladdin site.

- Onboard cryptographic processors, - PKI Infrastructure: generating private keys, storing digital certificates and digital signatures, - Standard Crypto API connectivity. - On-board PKI generation: private keys are never exposed outside the eToken. Standards and certifications: PKCS#11 v2.01, CAPI (Microsoft Crypto API), Siemens/Infinion APDU commands PC/SC, X.509v3 certificates, SSLv3, IPSec/IKE PRO, - Weight: 5 g, Dimensions: 47 x 16 x 8 mm (1.85 x 0.63 x 0.31 inches)

Figure 9, an Aladdin Knowledge Systems eToken pro

- Fingerprint reader: we used a fingerprint reader with a smart card reader embedded by Centro Biometrika, the product is a CBK304-b. This is released with a software developer kit called CBK WSDK 302® with the interface functions available to customize applications and software. This product presents a USB interface and can be used to read smart-card, enrol templates on a smart card, read a sample, making a match test between a sample and a template and many other things. For further information please refer to the vendor’s web site. We used the proprietary functions of CBK304-b to perform matching test of the template and the sample loading the template stored on the e-Token (using Centro Biometrika’s dedicated functions) : all the biometric operations had been performed on this device.

- All software has been developed on Microsoft© Visual Studio 6.0 and additional knowledge support has been retrieved on Microsoft© Platform SDK (October 2002 release).

3.4 Interfacing SSL both with CriptoAPI and PKCS#11 standard: two different implementations

Not all browsers use the same standard to communicate with smart card: Mozilla® as well as Netscape® use the RSA Security PKCS#11 standard to communicate with smart card and so every device that wants to be used by these browsers has to be compliant with this standard. Microsoft Internet Explorer talks with smart cards using the Cripto Application

Protocol Interface Standard(CriptoAPI, usually shortened in CAPI). Thus resulting in more than a vendor building devices which are able to communicate according both the standards we quote and some others more. Due to this difference we approach the problem of a biometric remote authentication protocol in two separated development: one for PKCS#11 compliant browsers and another for CAPI compliant ones (MS IE v.6.0; 5.0; etc.)

3.4.1 Netscape and PKCS#11 standard

The Public-Key Cryptography Standards are specifications produced by RSA Laboratories in cooperation with secure systems developers worldwide for individuating a standard for the deployment of public-key cryptography. First published in 1991 as a result of meetings with a small group of early adopters of public-key technology, the PKCS documents have become widely referenced and implemented. Contributions from the PKCS series have become part of many formal and de facto standards, including ANSI X9 documents, PKIX, SET, S/MIME, and SSL. Actually PKCS#11 is the communication standard between smart card and not-Win32 application programs. This standard specifies an API, called Cryptoki, to devices which hold cryptographic information and perform cryptographic functions. Cryptoki, short for cryptographic token interface, follows a simple object-based approach, addressing 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 device called a cryptographic token. To achieve this the Criptoki interfaces with a Dynamic Link Library, usually called pkcs11 or similarly, that implements a set of 68 functions able to purport all the functionalities required to run a smart card device according the pkcs#11 standard. All of these functions could be customized and personalized by system developers according that its interface will not be modified. This make it easier to customize some special features and to have some systems performing special functions. For our purpose with make an empty pkcs#11 dynamic link library interface called biopkcs11.dll that poses itself as a filter between the Criptoki and the real pkcs11 Dll : its task is to work as an empty interface every time that the function called by the system is different from the C_Login one. If the function referred is the C_Login our Dll lets the original function fulfil its tasks and, whether its result would be the normal CKR_OK return value, it takes control of the routine prior to giving the control back to the Criptoki performing a biometric match test using a special function called Biomatch. This Biomatch function uses the Centro Biometrika's CBK WSDK 302® and its functions to perform the biometric test. In the following paragraph these choices are completely developed.

We should notice that, due to the ease of use of the biometric interface, the user tolerance towards this protocol should be quite good. The overall authentication times, considering plugging in the e-token, typing in the PIN, placing your finger (the right one if possible) on the glass and having the biomatch performed, do not pass fifteen seconds and most of this time is involved in placing and moving things and not staring at the screen waiting for a reply.

3.4.1.1 Files and code

As we briefly explained above our Dll works as an interface to the real Dll that it uses to purport all the normal tasks. We address the original Criptoki functions, referenced through the original dedicated pointer structure CK_C_FunctionName, implemented by the pkcs#11 standard itself, through the command GetProcAddress that loads at run time the address to refer to execute a function. This command follows the LoadLibrary one that loads in the memory, always at run time, the address of a Dll that is not referenced in the code and that we want to use anyway. So that we use this couple of lines to initialize a pointer to the address of a special function and then, dereferencing this pointer and passing in that line the right parameters, we could use that function even tough the operating system is not using it. Due to the particularity of the pkcs#11 structure all of its functions are addressed through a table-structure in which they are listed. To have our functions called instead of the real ones we had to create a fake structure recalling to our fake functions and so we had to modify the dedicated function C_GetFunctionList according our needs. So inside the body of the new C_GetFunctionList we create our own array addressing our functions through special pointers. Every time that the Criptoki calls our dll this will be able to refer our functions and then using the GetProcAddress to execute the original function with the parameters we have formerly passed it.

Example of a fake interface generic function, the code is developed in C:

```
//.....
CK_PKCS11_FUNCTION_INFO(C_FunctionName)
#ifdef CK_NEED_ARG_LIST
(
 CK_TYPE_ofParameters   nameOfParameters
)
#endif
{
            CK_C_FunctionName addr;
            int ret;
            addr = (CK_C_FunctionName) GetProcAddress(OriginalDll, "C_ FunctionName ");
            ret = (*addr) (pReserved);
            return ret;
}
```

As you can easily notice the return value of our function is exactly the return of the original one just because we want the execution of every procedure to proceed according the pkcs#11 standard. Every function works by this scheme mirroring the real function reported in the pcks11.dll . The C_Login function has been modified in order to have the biometric test performed because no cryptographic functionality could be performed without having this function successfully terminated. The few changes introduced address a call to the biomatch function implemented in the biometric_pkcs.c file. In this file a C function is developed, using the CBK WSDK 302® development kit, to complete the following steps:

1. During the same session and using the same handles this function first checks if a sample could be retrieved on the smartcard (remember that if this function runs the “normal” C_Login has successfully terminated thus meaning that the user had been able to digit the right PIN to unlock it)

2. If possible it stores this template on a memory buffer.

3. After this the research session with the smart card is closed and the Biometric reader CBK304-b is initialized.

4. The model is loaded on the reader’s embedded memory from the host memory.

5. The functions set up for a triple matching test that is performed each time with a live-scan sample. For ease of use and user-acceptance the threshold to a rejection has been fixed in three medium quality rejection thus meaning that for two times the user sample could eventually not match the stored model. Notice that the model is immediately erased from the biometric reader memory after a single match test has been performed. That is to guarantee that no part of the sample will remain in the reader after the test has been performed: the CBK304-b model reading process is a destructive one.

6. After the biomatch ends, doesn't matter if successfully or not, the template in the host memory is immediately deleted and the communication with the biometric reader closed.

7. The proper result is then returned to "our" C_Login function that through the use of some dedicated messageboxes communicates to the user the result of the authentication session.

The two main files of this development are reported in the following pages: the first one, biopkcs11.c, develops the interface pkcs11.dll to the real pkcs11.dll. The second one, called biometrics_pkcs.c, implements the biometric test using the CBK WSDK 302® interface.

3.4.1.1.1 File biopkcs11.c

```
#pragma pack(push, cryptoki, 1)

#include "pkcs11mod.h"

#pragma pack(pop, cryptoki)

#include <windows.h>

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define CK_NEED_ARG_LIST  1
#define CK_PKCS11_FUNCTION_INFO(name) \
extern CK_DEFINE_FUNCTION(CK_RV, name)


CK_FUNCTION_LIST_PTR_PTR ppFunctionList;

extern int BioMatch();

FILE *log=NULL;

char path_real_dll[]="eTpkcs11.dll";

int useful=0;

static CK_FUNCTION_LIST prova;

HINSTANCE OriginalDll = NULL;
```

```
static void
log_error (char * c)
{
        fprintf (log,"BioPkcs11 error in function: %s\n", (c == NULL? "" : c));
}

/*
CK_DEFINE_FUNCTION(CK_RV, C_Initialize)(CK_VOID_PTR pReserved)
{
return 0;
}
*/

/* General-purpose */

/* C_Initialize initializes the Cryptoki library. */
CK_PKCS11_FUNCTION_INFO(C_Initialize)
#ifdef CK_NEED_ARG_LIST
(
CK_VOID_PTR   pInitArgs
CK_C_INITIALIZE_ARGS_PTR
)
#endif
{
        CK_C_Initialize addr;
        int ret;
        if (log == NULL)
        {
                log = fopen("c:\\log_biopkcs11.txt", "w");
        }
        if (OriginalDll == NULL)
        {
                OriginalDll = LoadLibrary(path_real_dll);
                fprintf(log,"---ho caricato con la loadlibrary la dll reale\n");
                if ( OriginalDll == NULL)
                        log_error ("C_Initialize, load_library");
        }
        addr = (CK_C_Initialize) GetProcAddress(OriginalDll, "C_Initialize");
        ret = (*addr) (pInitArgs);
        return ret;
}

/* C_Finalize indicates that an application is done with the
* Cryptoki library. */
CK_PKCS11_FUNCTION_INFO(C_Finalize)
#ifdef CK_NEED_ARG_LIST
(
 CK_VOID_PTR   pReserved
)
#endif
{
        CK_C_Finalize addr;
        int ret;
        addr = (CK_C_Finalize) GetProcAddress(OriginalDll, "C_Finalize");
        ret = (*addr) (pReserved);
        return ret;
}

/* C_GetInfo returns general information about Cryptoki. */
CK_PKCS11_FUNCTION_INFO(C_GetInfo)
#ifdef CK_NEED_ARG_LIST
```

```
(
 CK_INFO_PTR   pInfo  /* location that receives information */
)
#endif
{
                    CK_C_GetInfo addr;
                    int ret;
                    addr = (CK_C_GetInfo) GetProcAddress(OriginalDll, "C_GetInfo");
                    ret = (*addr) (pInfo);
                    return ret;
}

/* C_GetFunctionList returns the function list. */
CK_PKCS11_FUNCTION_INFO(C_GetFunctionList)
#ifdef CK_NEED_ARG_LIST
(
CK_FUNCTION_LIST_PTR_PTR ppFunctionList   /* receives pointer to
                                           * function list */
)
#endif
{
                    if (useful == 0)
                    {
                                        prova.version.major = 2;
                                        prova.version.minor = 01;

                                        prova.C_Initialize = C_Initialize;
                                        prova.C_Finalize = C_Finalize;
                                        prova.C_GetInfo = C_GetInfo;
                                        prova.C_GetFunctionList = C_GetFunctionList;
                                        prova.C_GetSlotList = C_GetSlotList;
                                        prova.C_GetSlotInfo = C_GetSlotInfo;
                                        prova.C_GetTokenInfo = C_GetTokenInfo;
                                        prova.C_GetMechanismList = C_GetMechanismList;
                                        prova.C_GetMechanismInfo = C_GetMechanismInfo;
                                        prova.C_InitToken = C_InitToken;
                                        prova.C_InitPIN = C_InitPIN;
                                        prova.C_SetPIN = C_SetPIN;
                                        prova.C_OpenSession = C_OpenSession;
                                        prova.C_CloseSession = C_CloseSession;
                                        prova.C_CloseAllSessions = C_CloseAllSessions;
                                        prova.C_GetSessionInfo = C_GetSessionInfo;
                                        prova.C_GetOperationState = C_GetOperationState;
                                        prova.C_SetOperationState = C_SetOperationState;
                                        prova.C_Login = C_Login;
                                        prova.C_Logout = C_Logout;
                                        prova.C_CreateObject = C_CreateObject;
                                        prova.C_CopyObject = C_CopyObject;
                                        prova.C_DestroyObject = C_DestroyObject;
                                        prova.C_GetObjectSize = C_GetObjectSize;
                                        prova.C_GetAttributeValue = C_GetAttributeValue;
                                        prova.C_SetAttributeValue = C_SetAttributeValue;
                                        prova.C_FindObjectsInit = C_FindObjectsInit;
                                        prova.C_FindObjects = C_FindObjects;
                                        prova.C_FindObjectsFinal = C_FindObjectsFinal;
                                        prova.C_EncryptInit = C_EncryptInit;
                                        prova.C_Encrypt = C_Encrypt;
                                        prova.C_EncryptUpdate = C_EncryptUpdate;
                                        prova.C_EncryptFinal = C_EncryptFinal;
                                        prova.C_DecryptInit = C_DecryptInit;
                                        prova.C_Decrypt = C_Decrypt;
                                        prova.C_DecryptUpdate = C_DecryptUpdate;
                                        prova.C_DecryptFinal = C_DecryptFinal;
```

```
                prova.C_DigestInit = C_DigestInit;
                prova.C_Digest = C_Digest;
                prova.C_DigestUpdate = C_DigestUpdate;
                prova.C_DigestKey = C_DigestKey;
                prova.C_DigestFinal = C_DigestFinal;
                prova.C_SignInit = C_SignInit;
                prova.C_Sign = C_Sign;
                prova.C_SignUpdate = C_SignUpdate;
                prova.C_SignFinal = C_SignFinal;
                prova.C_SignRecoverInit = C_SignRecoverInit;
                prova.C_SignRecover = C_SignRecover;
                prova.C_VerifyInit = C_VerifyInit;
                prova.C_Verify = C_Verify;
                prova.C_VerifyUpdate = C_VerifyUpdate;
                prova.C_VerifyFinal = C_VerifyFinal;
                prova.C_VerifyRecoverInit = C_VerifyRecoverInit;
                prova.C_VerifyRecover = C_VerifyRecover;
                prova.C_DigestEncryptUpdate = C_DigestEncryptUpdate;
                prova.C_DecryptDigestUpdate = C_DecryptDigestUpdate;
                prova.C_SignEncryptUpdate = C_SignEncryptUpdate;
                prova.C_DecryptVerifyUpdate = C_DecryptVerifyUpdate;
                prova.C_GenerateKey = C_GenerateKey;
                prova.C_GenerateKeyPair = C_GenerateKeyPair;
                prova.C_WrapKey = C_WrapKey;
                prova.C_UnwrapKey = C_UnwrapKey;
                prova.C_DeriveKey = C_DeriveKey;
                prova.C_SeedRandom = C_SeedRandom;
                prova.C_GenerateRandom = C_GenerateRandom;
                prova.C_GetFunctionStatus = C_GetFunctionStatus;
                prova.C_CancelFunction = C_CancelFunction;
                prova.C_WaitForSlotEvent = C_WaitForSlotEvent;

                *ppFunctionList = &prova;
                useful= 1;
        }
        if (OriginalDll == NULL)
        {
                OriginalDll = LoadLibrary(path_real_dll);
                assert(OriginalDll != NULL);
        }
        return CKR_OK;
}

/* Slot and token management */

/* C_GetSlotList obtains a list of slots in the system. */
CK_PKCS11_FUNCTION_INFO(C_GetSlotList)
#ifdef CK_NEED_ARG_LIST
(
 CK_BBOOL              tokenPresent,  /* only slots with tokens? */
 CK_SLOT_ID_PTR pSlotList,             /* receives array of slot IDs */
 CK_ULONG_PTR          pulCount        /* receives number of slots */
)
#endif
{
        CK_C_GetSlotList addr;
        int ret;
        addr = (CK_C_GetSlotList) GetProcAddress(OriginalDll, "C_GetSlotList");
        ret = (*addr) (tokenPresent,pSlotList,pulCount);
        return ret;
}

/* C_GetSlotInfo obtains information about a particular slot in
```

```
* the system. */
CK_PKCS11_FUNCTION_INFO(C_GetSlotInfo)
#ifdef CK_NEED_ARG_LIST
(
 CK_SLOT_ID              slotID,  /* the ID of the slot */
 CK_SLOT_INFO_PTR        pInfo    /* receives the slot information */
)
#endif
{
            CK_C_GetSlotInfo addr;
            int ret;
            addr = (CK_C_GetSlotInfo) GetProcAddress(OriginalDll, "C_GetSlotInfo");
            ret = (*addr) (slotID, pInfo);
            return ret;
}

/* C_GetTokenInfo obtains information about a particular token
* in the system. */
CK_PKCS11_FUNCTION_INFO(C_GetTokenInfo)
#ifdef CK_NEED_ARG_LIST
(
 CK_SLOT_ID              slotID,          /* ID of the token's slot */
 CK_TOKEN_INFO_PTR       pInfo            /* receives the token information */
)
#endif
{
            CK_C_GetTokenInfo addr;
            int ret;
            addr = (CK_C_GetTokenInfo) GetProcAddress(OriginalDll, "C_GetTokenInfo");
            ret = (*addr) (slotID, pInfo);
            return ret;
}

/* C_GetMechanismList obtains a list of mechanism types
* supported by a token. */
CK_PKCS11_FUNCTION_INFO(C_GetMechanismList)
#ifdef CK_NEED_ARG_LIST
(
 CK_SLOT_ID                       slotID,                 /* ID of token's slot */
 CK_MECHANISM_TYPE_PTR            pMechanismList,         /* gets mech. array */
 CK_ULONG_PTR                     pulCount                /* gets # of mechs. */
)
#endif
{
            CK_C_GetMechanismList addr;
            int ret;
            addr = (CK_C_GetMechanismList) GetProcAddress(OriginalDll, "C_GetMechanismList");
            ret = (*addr)(slotID, pMechanismList,pulCount);
            return ret;
}


/* C_GetMechanismInfo obtains information about a particular
* mechanism possibly supported by a token. */
CK_PKCS11_FUNCTION_INFO(C_GetMechanismInfo)
#ifdef CK_NEED_ARG_LIST
(
 CK_SLOT_ID                       slotID,     /* ID of the token's slot */
 CK_MECHANISM_TYPE                type,       /* type of mechanism */
 CK_MECHANISM_INFO_PTR            pInfo       /* receives mechanism info */
)
#endif
{
```

```
                CK_C_GetMechanismInfo addr;
                int ret;
                addr = (CK_C_GetMechanismInfo) GetProcAddress(OriginalDll, "C_GetMechanismInfo");
                ret = (*addr)(slotID, type, pInfo);
                return ret;
}

/* C_InitToken initializes a token. */
CK_PKCS11_FUNCTION_INFO(C_InitToken)
#ifdef CK_NEED_ARG_LIST
(
 CK_SLOT_ID              slotID,          /* ID of the token's slot */
 CK_CHAR_PTR             pPin,            /* the SO's initial PIN */
 CK_ULONG                ulPinLen,        /* length in bytes of the PIN */
 CK_CHAR_PTR             pLabel           /* 32-byte token label (blank padded) */
)
#endif
{
                CK_C_InitToken addr;
                int ret;
                addr = (CK_C_InitToken) GetProcAddress(OriginalDll, "C_InitToken");
                ret = (*addr)(slotID, pPin,ulPinLen, pLabel);
                return ret;
}

/* C_InitPIN initializes the normal user's PIN. */
CK_PKCS11_FUNCTION_INFO(C_InitPIN)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE        hSession,  /* the session's handle */
 CK_CHAR_PTR              pPin,            /* the normal user's PIN */
 CK_ULONG                 ulPinLen   /* length in bytes of the PIN */
)
#endif
{
                CK_C_InitPIN addr;
                int ret;
                addr = (CK_C_InitPIN) GetProcAddress(OriginalDll, "C_InitPIN");
                ret = (*addr)(hSession, pPin, ulPinLen);
                return ret;
}

/* C_SetPIN modifies the PIN of the user who is logged in. */
CK_PKCS11_FUNCTION_INFO(C_SetPIN)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE      hSession,  /* the session's handle */
 CK_CHAR_PTR            pOldPin,   /* the old PIN */
 CK_ULONG               ulOldLen,  /* length of the old PIN */
 CK_CHAR_PTR            pNewPin,   /* the new PIN */
 CK_ULONG               ulNewLen   /* length of the new PIN */
)
#endif
{
                CK_C_SetPIN addr;
                int ret;
                addr = (CK_C_SetPIN) GetProcAddress(OriginalDll, "C_SetPIN");
                ret = (*addr)(hSession, pOldPin,ulOldLen, pNewPin, ulNewLen);
                return ret;
}


/* Session management */
/* C_OpenSession opens a session between an application and a
* token. */
CK_PKCS11_FUNCTION_INFO(C_OpenSession)
#ifdef CK_NEED_ARG_LIST
(
 CK_SLOT_ID                    slotID,          /* the slot's ID */
 CK_FLAGS                      flags,           /* from CK_SESSION_INFO */
 CK_VOID_PTR                   pApplication,    /* passed to callback */
 CK_NOTIFY                     Notify,          /* callback function */
 CK_SESSION_HANDLE_PTR        phSession        /* gets session handle */
)
#endif
{
        CK_C_OpenSession addr;
        int ret;
        addr = (CK_C_OpenSession) GetProcAddress(OriginalDll, "C_OpenSession");
        ret = (*addr)(slotID, flags,pApplication,Notify, phSession);
        return ret;
}

/* C_CloseSession closes a session between an application and a
* token. */
CK_PKCS11_FUNCTION_INFO(C_CloseSession)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE hSession  /* the session's handle */
)
#endif
{
        CK_C_CloseSession addr;
        int ret;
        addr = (CK_C_CloseSession) GetProcAddress(OriginalDll, "C_CloseSession");
        ret = (*addr)(hSession);
        return ret;
}

/* C_CloseAllSessions closes all sessions with a token. */
CK_PKCS11_FUNCTION_INFO(C_CloseAllSessions)
#ifdef CK_NEED_ARG_LIST
(
 CK_SLOT_ID          slotID          /* the token's slot */
)
#endif
{
        CK_C_CloseAllSessions addr;
        int ret;
        addr = (CK_C_CloseAllSessions) GetProcAddress(OriginalDll, "C_CloseAllSessions");
        ret = (*addr)(slotID);
        return ret;
}

/* C_GetSessionInfo obtains information about the session. */
CK_PKCS11_FUNCTION_INFO(C_GetSessionInfo)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE     hSession,      /* the session's handle */
 CK_SESSION_INFO_PTR   pInfo          /* receives session info */
)
#endif
{
        CK_C_GetSessionInfo addr;
        int ret;
```

```
                addr = (CK_C_GetSessionInfo) GetProcAddress(OriginalDll, "C_GetSessionInfo");
                ret = (*addr)(hSession, pInfo);
                return ret;
}

/* C_GetOperationState obtains the state of the cryptographic operation
* in a session. */
CK_PKCS11_FUNCTION_INFO(C_GetOperationState)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE      hSession,                         /* session's handle */
 CK_BYTE_PTR              pOperationState,                /* gets state */
 CK_ULONG_PTR             pulOperationStateLen  /* gets state length */
)
#endif
{
                CK_C_GetOperationState addr;
                int ret;
                addr = (CK_C_GetOperationState) GetProcAddress(OriginalDll, "C_GetOperationState");
                ret = (*addr)(hSession, pOperationState,pulOperationStateLen);
                return ret;
}

/* C_SetOperationState restores the state of the cryptographic
* operation in a session. */
CK_PKCS11_FUNCTION_INFO(C_SetOperationState)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE      hSession,                                  /* session's handle */
 CK_BYTE_PTR             pOperationState,                         /* holds state */
 CK_ULONG                ulOperationStateLen,                     /* holds state length */
 CK_OBJECT_HANDLE       hEncryptionKey,                           /* en/decryption key */
 CK_OBJECT_HANDLE       hAuthenticationKey                        /* sign/verify key */
)
#endif
{
                CK_C_SetOperationState addr;
                int ret;
                addr = (CK_C_SetOperationState) GetProcAddress(OriginalDll, "C_SetOperationState");
                ret = (*addr)(hSession, pOperationState,ulOperationStateLen,hEncryptionKey, hAuthenticationKey);
                return ret;
}

/* C_Login logs a user into a token. */
CK_PKCS11_FUNCTION_INFO(C_Login)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE      hSession,  /* the session's handle */
 CK_USER_TYPE           userType,  /* the user type */
 CK_CHAR_PTR            pPin,          /* the user's PIN */
 CK_ULONG               ulPinLen   /* the length of the PIN */
)
#endif
{
                CK_C_Login addr;
                int ret;
                addr = (CK_C_Login) GetProcAddress(OriginalDll, "C_Login");
                ret = (*addr)(hSession, userType,pPin,ulPinLen);
                if ( ret == CKR_OK)
                ret = BioMatch(hSession, prova);
                if (ret != 1)
                {
                                MessageBox(NULL, "Authentication Failed!", "biomatch error", MB_ICONERROR);
```

```
                        return CKR_PIN_INCORRECT;
                }
                else MessageBox(NULL, "Biomatch OK", "clicca per proseguire", MB_ICONINFORMATION |
                MB_OK);
                return CKR_OK;
}

/* C_Logout logs a user out from a token. */
CK_PKCS11_FUNCTION_INFO(C_Logout)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE hSession  /* the session's handle */
)
#endif
{
                CK_C_Logout addr;
                int ret;
                addr = (CK_C_Logout) GetProcAddress(OriginalDll, "C_Logout");
                ret = (*addr)(hSession);
                return ret;

}


/* Object management */

/* C_CreateObject creates a new object. */
CK_PKCS11_FUNCTION_INFO(C_CreateObject)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE      hSession,        /* the session's handle */
 CK_ATTRIBUTE_PTR       pTemplate,       /* the object's template */
 CK_ULONG               ulCount,         /* attributes in template */
 CK_OBJECT_HANDLE_PTR phObject           /* gets new object's handle. */
)
#endif
{
                CK_C_CreateObject addr;
                int ret;
                addr = (CK_C_CreateObject) GetProcAddress(OriginalDll, "C_CreateObject");
                ret = (*addr)(hSession, pTemplate,ulCount,phObject);
                return ret;
}

/* C_CopyObject copies an object, creating a new object for the
* copy. */
CK_PKCS11_FUNCTION_INFO(C_CopyObject)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE        hSession,    /* the session's handle */
 CK_OBJECT_HANDLE         hObject,          /* the object's handle */
 CK_ATTRIBUTE_PTR         pTemplate,   /* template for new object */
 CK_ULONG                 ulCount,          /* attributes in template */
 CK_OBJECT_HANDLE_PTR phNewObject  /* receives handle of copy */
)
#endif
{
                CK_C_CopyObject addr;
                int ret;
                addr = (CK_C_CopyObject) GetProcAddress(OriginalDll, "C_CopyObject");
                ret = (*addr)(hSession, hObject,pTemplate, ulCount,phNewObject);
                return ret;
}
```

```
/* C_DestroyObject destroys an object. */
CK_PKCS11_FUNCTION_INFO(C_DestroyObject)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE hSession,  /* the session's handle */
 CK_OBJECT_HANDLE  hObject                    /* the object's handle */
)
#endif
{
                CK_C_DestroyObject addr;
                int ret;
                addr = (CK_C_DestroyObject) GetProcAddress(OriginalDll, "C_DestroyObject");
                ret = (*addr)(hSession, hObject);
                return ret;
}

/* C_GetObjectSize gets the size of an object in bytes. */
CK_PKCS11_FUNCTION_INFO(C_GetObjectSize)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE      hSession,        /* the session's handle */
 CK_OBJECT_HANDLE       hObject,         /* the object's handle */
 CK_ULONG_PTR           pulSize          /* receives size of object */
)
#endif
{
                CK_C_GetObjectSize addr;
                int ret;
                addr = (CK_C_GetObjectSize) GetProcAddress(OriginalDll, "C_GetObjectSize");
                ret = (*addr)(hSession, hObject,pulSize);
                return ret;
}

/* C_GetAttributeValue obtains the value of one or more object
* attributes. */
CK_PKCS11_FUNCTION_INFO(C_GetAttributeValue)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE      hSession,        /* the session's handle */
 CK_OBJECT_HANDLE       hObject,         /* the object's handle */
 CK_ATTRIBUTE_PTR       pTemplate,       /* specifies attrs; gets vals */
 CK_ULONG               ulCount          /* attributes in template */
)
#endif
{
                CK_C_GetAttributeValue addr;
                int ret;
                addr = (CK_C_GetAttributeValue) GetProcAddress(OriginalDll, "C_GetAttributeValue");
                ret = (*addr)(hSession, hObject,pTemplate, ulCount);
                return ret;
}

/* C_SetAttributeValue modifies the value of one or more object
* attributes */
CK_PKCS11_FUNCTION_INFO(C_SetAttributeValue)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE       hSession,       /* the session's handle */
 CK_OBJECT_HANDLE       hObject,         /* the object's handle */
 CK_ATTRIBUTE_PTR       pTemplate,       /* specifies attrs and values */
 CK_ULONG               ulCount          /* attributes in template */
)
```

```
#endif
{
		CK_C_SetAttributeValue addr;
		int ret;
		addr = (CK_C_SetAttributeValue) GetProcAddress(OriginalDll, "C_SetAttributeValue");
		ret = (*addr)(hSession, hObject,pTemplate, ulCount);
		return ret;
}

/* C_FindObjectsInit initializes a search for token and session
* objects that match a template. */
CK_PKCS11_FUNCTION_INFO(C_FindObjectsInit)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE      hSession,       /* the session's handle */
 CK_ATTRIBUTE_PTR       pTemplate,      /* attribute values to match */
 CK_ULONG               ulCount         /* attrs in search template */
)
#endif
{
		CK_C_FindObjectsInit addr;
		int ret;
		addr = (CK_C_FindObjectsInit) GetProcAddress(OriginalDll, "C_FindObjectsInit");
		ret = (*addr)(hSession, pTemplate,ulCount);
		return ret;
}

/* C_FindObjects continues a search for token and session
* objects that match a template, obtaining additional object
* handles. */
CK_PKCS11_FUNCTION_INFO(C_FindObjects)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE      hSession,               /* session's handle */
 CK_OBJECT_HANDLE_PTR phObject,                 /* gets obj. handles */
 CK_ULONG               ulMaxObjectCount,       /* max handles to get */
 CK_ULONG_PTR           pulObjectCount          /* actual # returned */
)
#endif
{
		CK_C_FindObjects addr;
		int ret;
		addr = (CK_C_FindObjects) GetProcAddress(OriginalDll, "C_FindObjects");
		ret = (*addr)(hSession, phObject,ulMaxObjectCount, pulObjectCount);
		return ret;
}

/* C_FindObjectsFinal finishes a search for token and session
* objects. */
CK_PKCS11_FUNCTION_INFO(C_FindObjectsFinal)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE hSession  /* the session's handle */
)
#endif
{
		CK_C_FindObjectsFinal addr;
		int ret;
		addr = (CK_C_FindObjectsFinal) GetProcAddress(OriginalDll, "C_FindObjectsFinal");
		ret = (*addr)(hSession);
		return ret;
}
```

```
/* Encryption and decryption */

/* C_EncryptInit initializes an encryption operation. */
CK_PKCS11_FUNCTION_INFO(C_EncryptInit)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE hSession,            /* the session's handle */
 CK_MECHANISM_PTR  pMechanism,          /* the encryption mechanism */
 CK_OBJECT_HANDLE  hKey                 /* handle of encryption key */
)
#endif
{
                CK_C_EncryptInit addr;
                int ret;
                addr = (CK_C_EncryptInit) GetProcAddress(OriginalDll, "C_EncryptInit");
                ret = (*addr)(hSession, pMechanism,hKey);
                return ret;
}

/* C_Encrypt encrypts single-part data. */
CK_PKCS11_FUNCTION_INFO(C_Encrypt)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE      hSession,                               /* session's handle */
 CK_BYTE_PTR            pData,                                  /* the plaintext data */
 CK_ULONG               ulDataLen,                              /* bytes of plaintext */
 CK_BYTE_PTR            pEncryptedData,                         /* gets ciphertext */
 CK_ULONG_PTR           pulEncryptedDataLen                     /* gets c-text size */
)
#endif
{

                CK_C_Encrypt addr;
                int ret;
                addr = (CK_C_Encrypt) GetProcAddress(OriginalDll, "C_Encrypt");
                ret = (*addr)(hSession, pData, ulDataLen,pEncryptedData, pulEncryptedDataLen);
                return ret;
}

/* C_EncryptUpdate continues a multiple-part encryption
* operation. */
CK_PKCS11_FUNCTION_INFO(C_EncryptUpdate)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE       hSession,                  /* session's handle */
 CK_BYTE_PTR             pPart,                     /* the plaintext data */
 CK_ULONG                ulPartLen,                 /* plaintext data len */
 CK_BYTE_PTR             pEncryptedPart,            /* gets ciphertext */
 CK_ULONG_PTR            pulEncryptedPartLen       /* gets c-text size */
)
#endif
{
                CK_C_EncryptUpdate addr;
                int ret;
                addr = (CK_C_EncryptUpdate) GetProcAddress(OriginalDll, "C_EncryptUpdate");
                ret = (*addr)(hSession, pPart,ulPartLen,pEncryptedPart,pulEncryptedPartLen );
                return ret;
}

/* C_EncryptFinal finishes a multiple-part encryption
* operation. */
CK_PKCS11_FUNCTION_INFO(C_EncryptFinal)
```

```
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE      hSession,                              /* session handle */
 CK_BYTE_PTR            pLastEncryptedPart,                    /* last c-text */
 CK_ULONG_PTR           pulLastEncryptedPartLen                /* gets last size */
)
#endif
{
                CK_C_EncryptFinal addr;
                int ret;
                addr = (CK_C_EncryptFinal) GetProcAddress(OriginalDll, "C_EncryptFinal");
                ret = (*addr)(hSession, pLastEncryptedPart,pulLastEncryptedPartLen);
                return ret;
}

/* C_DecryptInit initializes a decryption operation. */
CK_PKCS11_FUNCTION_INFO(C_DecryptInit)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE hSession,            /* the session's handle */
 CK_MECHANISM_PTR  pMechanism,          /* the decryption mechanism */
 CK_OBJECT_HANDLE  hKey                 /* handle of decryption key */
)
#endif
{
                CK_C_DecryptInit addr;
                int ret;
                addr = (CK_C_DecryptInit) GetProcAddress(OriginalDll, "C_DecryptInit");

                ret = (*addr)(hSession, pMechanism,hKey);
                return ret;

}

/* C_Decrypt decrypts encrypted data in a single part. */
CK_PKCS11_FUNCTION_INFO(C_Decrypt)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE      hSession,                  /* session's handle */
 CK_BYTE_PTR              pEncryptedData,          /* ciphertext */
 CK_ULONG                 ulEncryptedDataLen,      /* ciphertext length */
 CK_BYTE_PTR              pData,                   /* gets plaintext */
 CK_ULONG_PTR             pulDataLen               /* gets p-text size */
)
#endif
{
                CK_C_Decrypt addr;
                int ret;
                addr = (CK_C_Decrypt) GetProcAddress(OriginalDll, "C_Decrypt");
                ret = (*addr)(hSession, pEncryptedData,ulEncryptedDataLen, pData,pulDataLen);
                return ret;
}

/* C_DecryptUpdate continues a multiple-part decryption
* operation. */
CK_PKCS11_FUNCTION_INFO(C_DecryptUpdate)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE      hSession,                              /* session's handle */
 CK_BYTE_PTR              pEncryptedPart,                      /* encrypted data */
 CK_ULONG                 ulEncryptedPartLen,                  /* input length */
 CK_BYTE_PTR              pPart,                               /* gets plaintext */
 CK_ULONG_PTR             pulPartLen                           /* p-text size */
)
#endif
{
                CK_C_DecryptUpdate addr;
                int ret;
                addr = (CK_C_DecryptUpdate) GetProcAddress(OriginalDll, "C_DecryptUpdate");
                ret = (*addr)(hSession, pEncryptedPart,ulEncryptedPartLen, pPart,pulPartLen);
                return ret;
}

/* C_DecryptFinal finishes a multiple-part decryption
* operation. */
CK_PKCS11_FUNCTION_INFO(C_DecryptFinal)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE      hSession,            /* the session's handle */
 CK_BYTE_PTR              pLastPart,         /* gets plaintext */
 CK_ULONG_PTR             pulLastPartLen  /* p-text size */
)
#endif
{
                int ret;
                CK_C_DecryptFinal addr = (CK_C_DecryptFinal) GetProcAddress(OriginalDll, "C_DecryptFinal");
                ret = (*addr)(hSession, pLastPart,pulLastPartLen);
                return ret;
}


/* Message digesting */

/* C_DigestInit initializes a message-digesting operation. */
CK_PKCS11_FUNCTION_INFO(C_DigestInit)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE hSession,               /* the session's handle */
 CK_MECHANISM_PTR  pMechanism          /* the digesting mechanism */
)
#endif
{
                int ret;
                CK_C_DigestInit addr = (CK_C_DigestInit) GetProcAddress(OriginalDll, "C_DigestInit");
                ret = (*addr)(hSession, pMechanism);
                return ret;
}

/* C_Digest digests data in a single part. */
CK_PKCS11_FUNCTION_INFO(C_Digest)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE      hSession,          /* the session's handle */
 CK_BYTE_PTR              pData,           /* data to be digested */
 CK_ULONG                 ulDataLen,     /* bytes of data to digest */
 CK_BYTE_PTR              pDigest,        /* gets the message digest */
 CK_ULONG_PTR             pulDigestLen  /* gets digest length */
)
#endif
{
                int ret;
                CK_C_Digest addr = (CK_C_Digest) GetProcAddress(OriginalDll, "C_Digest");
                ret = (*addr)(hSession, pData,ulDataLen,pDigest, pulDigestLen);
                return ret;
}
```

```
/* C_DigestUpdate continues a multiple-part message-digesting
* operation. */
CK_PKCS11_FUNCTION_INFO(C_DigestUpdate)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE      hSession,          /* the session's handle */
 CK_BYTE_PTR                pPart,             /* data to be digested */
 CK_ULONG                   ulPartLen      /* bytes of data to be digested */
)
#endif
{
		int ret;
		CK_C_DigestUpdate addr = (CK_C_DigestUpdate) GetProcAddress(OriginalDll, "C_DigestUpdate");
		ret = (*addr)(hSession, pPart,ulPartLen);
		return ret;
}

/* C_DigestKey continues a multi-part message-digesting
* operation, by digesting the value of a secret key as part of
* the data already digested. */
CK_PKCS11_FUNCTION_INFO(C_DigestKey)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE hSession,               /* the session's handle */
 CK_OBJECT_HANDLE  hKey                     /* secret key to digest */
)
#endif
{
		int ret;
		CK_C_DigestKey addr = (CK_C_DigestKey) GetProcAddress(OriginalDll, "C_DigestKey");
		ret = (*addr)(hSession, hKey);
		return ret;
}

/* C_DigestFinal finishes a multiple-part message-digesting
* operation. */
CK_PKCS11_FUNCTION_INFO(C_DigestFinal)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE       hSession,         /* the session's handle */
 CK_BYTE_PTR                pDigest,         /* gets the message digest */
 CK_ULONG_PTR               pulDigestLen  /* gets byte count of digest */
)
#endif
{
		int ret;
		CK_C_DigestFinal addr = (CK_C_DigestFinal) GetProcAddress(OriginalDll, "C_DigestFinal");
		ret = (*addr)(hSession, pDigest,pulDigestLen);
		return ret;
}


/* Signing and MACing */

/* C_SignInit initializes a signature (private key encryption)
* operation, where the signature is (will be) an appendix to
* the data, and plaintext cannot be recovered from the
*signature. */
CK_PKCS11_FUNCTION_INFO(C_SignInit)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE hSession,              /* the session's handle */
 CK_MECHANISM_PTR  pMechanism,         /* the signature mechanism */
 CK_OBJECT_HANDLE  hKey                     /* handle of signature key */
)
#endif
{
                int ret;
                CK_C_SignInit addr = (CK_C_SignInit) GetProcAddress(OriginalDll, "C_SignInit");
                ret = (*addr)(hSession, pMechanism,hKey);
                return ret;
}

/* C_Sign signs (encrypts with private key) data in a single
* part, where the signature is (will be) an appendix to the
* data, and plaintext cannot be recovered from the signature. */
CK_PKCS11_FUNCTION_INFO(C_Sign)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE      hSession,                    /* the session's handle */
 CK_BYTE_PTR              pData,                     /* the data to sign */
 CK_ULONG                 ulDataLen,                 /* count of bytes to sign */
 CK_BYTE_PTR              pSignature,                /* gets the signature */
 CK_ULONG_PTR             pulSignatureLen            /* gets signature length */
)
#endif
{
                int ret;
                CK_C_Sign addr = (CK_C_Sign) GetProcAddress(OriginalDll, "C_Sign");
                ret = (*addr)(hSession, pData,ulDataLen,pSignature, pulSignatureLen);
                return ret;
}

/* C_SignUpdate continues a multiple-part signature operation,
* where the signature is (will be) an appendix to the data,
* and plaintext cannot be recovered from the signature. */
CK_PKCS11_FUNCTION_INFO(C_SignUpdate)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE       hSession,  /* the session's handle */
 CK_BYTE_PTR              pPart,          /* the data to sign */
 CK_ULONG                 ulPartLen  /* count of bytes to sign */
)
#endif
{
                int ret;
                CK_C_SignUpdate addr = (CK_C_SignUpdate) GetProcAddress(OriginalDll, "C_SignUpdate");
                ret = (*addr)(hSession, pPart,ulPartLen);
                return ret;
}

/* C_SignFinal finishes a multiple-part signature operation,
* returning the signature. */
CK_PKCS11_FUNCTION_INFO(C_SignFinal)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE      hSession,                    /* the session's handle */
 CK_BYTE_PTR              pSignature,                /* gets the signature */
 CK_ULONG_PTR             pulSignatureLen            /* gets signature length */
)
#endif
{
                int ret;
                CK_C_SignFinal addr = (CK_C_SignFinal) GetProcAddress(OriginalDll, "C_SignFinal");
                ret = (*addr)(hSession, pSignature,pulSignatureLen);
                return ret;
```

```
}

/* C_SignRecoverInit initializes a signature operation, where
* the data can be recovered from the signature. */
CK_PKCS11_FUNCTION_INFO(C_SignRecoverInit)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE hSession,          /* the session's handle */
 CK_MECHANISM_PTR  pMechanism,        /* the signature mechanism */
 CK_OBJECT_HANDLE  hKey               /* handle of the signature key */
)
#endif
{
        int ret;
        CK_C_SignRecoverInit addr = (CK_C_SignRecoverInit) GetProcAddress(OriginalDll,
        "C_SignRecoverInit");

        ret = (*addr)(hSession, pMechanism,hKey);
        return ret;
}

/* C_SignRecover signs data in a single operation, where the
* data can be recovered from the signature. */
CK_PKCS11_FUNCTION_INFO(C_SignRecover)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE      hSession,              /* the session's handle */
 CK_BYTE_PTR              pData,               /* the data to sign */
 CK_ULONG                 ulDataLen,           /* count of bytes to sign */
 CK_BYTE_PTR              pSignature,          /* gets the signature */
 CK_ULONG_PTR             pulSignatureLen      /* gets signature length */
)
#endif
{
        int ret;
        CK_C_SignRecover addr = (CK_C_SignRecover) GetProcAddress(OriginalDll, "C_SignRecover");
        ret = (*addr)(hSession, pData,ulDataLen, pSignature,pulSignatureLen);
        return ret;
}


/* Verifying signatures and MACs */

/* C_VerifyInit initializes a verification operation, where the
* signature is an appendix to the data, and plaintext cannot
*  cannot be recovered from the signature (e.g. DSA). */
CK_PKCS11_FUNCTION_INFO(C_VerifyInit)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE hSession,          /* the session's handle */
 CK_MECHANISM_PTR  pMechanism,        /* the verification mechanism */
 CK_OBJECT_HANDLE  hKey               /* verification key */
)
#endif
{
        int ret;
        CK_C_VerifyInit addr = (CK_C_VerifyInit) GetProcAddress(OriginalDll, "C_VerifyInit");
        ret = (*addr)(hSession, pMechanism,hKey);
        return ret;
}

/* C_Verify verifies a signature in a single-part operation,
* where the signature is an appendix to the data, and plaintext
* cannot be recovered from the signature. */
CK_PKCS11_FUNCTION_INFO(C_Verify)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE      hSession,           /* the session's handle */
 CK_BYTE_PTR              pData,            /* signed data */
 CK_ULONG                 ulDataLen,        /* length of signed data */
 CK_BYTE_PTR              pSignature,       /* signature */
 CK_ULONG                 ulSignatureLen  /* signature length*/
)
#endif
{
        int ret;
        CK_C_Verify addr = (CK_C_Verify) GetProcAddress(OriginalDll, "C_Verify");
        ret = (*addr)(hSession, pData,ulDataLen,pSignature,ulSignatureLen);
        return ret;
}

/* C_VerifyUpdate continues a multiple-part verification
* operation, where the signature is an appendix to the data,
* and plaintext cannot be recovered from the signature. */
CK_PKCS11_FUNCTION_INFO(C_VerifyUpdate)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE      hSession,  /* the session's handle */
 CK_BYTE_PTR              pPart,            /* signed data */
 CK_ULONG                 ulPartLen  /* length of signed data */
)
#endif
{
        int ret;
        CK_C_VerifyUpdate addr = (CK_C_VerifyUpdate) GetProcAddress(OriginalDll, "C_VerifyUpdate");
        ret = (*addr)(hSession, pPart,ulPartLen);
        return ret;
}

/* C_VerifyFinal finishes a multiple-part verification
* operation, checking the signature. */
CK_PKCS11_FUNCTION_INFO(C_VerifyFinal)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE      hSession,           /* the session's handle */
 CK_BYTE_PTR              pSignature,       /* signature to verify */
 CK_ULONG                 ulSignatureLen  /* signature length */
)
#endif
{
        int ret;
        CK_C_VerifyFinal addr = (CK_C_VerifyFinal) GetProcAddress(OriginalDll, "C_VerifyFinal");
        ret = (*addr)(hSession, pSignature,ulSignatureLen);
        return ret;
}

/* C_VerifyRecoverInit initializes a signature verification
* operation, where the data is recovered from the signature. */
CK_PKCS11_FUNCTION_INFO(C_VerifyRecoverInit)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE hSession,             /* the session's handle */
 CK_MECHANISM_PTR  pMechanism,        /* the verification mechanism */
 CK_OBJECT_HANDLE  hKey                  /* verification key */
)
#endif
```

```
{
        int ret;
        CK_C_VerifyRecoverInit addr = (CK_C_VerifyRecoverInit) GetProcAddress(OriginalDll,
        "C_VerifyRecoverInit");
        ret = (*addr)(hSession, pMechanism,hKey);
        return ret;
}

/* C_VerifyRecover verifies a signature in a single-part
* operation, where the data is recovered from the signature. */
CK_PKCS11_FUNCTION_INFO(C_VerifyRecover)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE    hSession,          /* the session's handle */
 CK_BYTE_PTR          pSignature,        /* signature to verify */
 CK_ULONG             ulSignatureLen,    /* signature length */
 CK_BYTE_PTR          pData,             /* gets signed data */
 CK_ULONG_PTR         pulDataLen         /* gets signed data len */
)
#endif
{
        int ret;
        CK_C_VerifyRecover addr = (CK_C_VerifyRecover) GetProcAddress(OriginalDll,
        "C_VerifyRecover");

        ret = (*addr)(hSession, pSignature,ulSignatureLen,pData,pulDataLen);
        return ret;
}


/* Dual-function cryptographic operations */

/* C_DigestEncryptUpdate continues a multiple-part digesting
* and encryption operation. */
CK_PKCS11_FUNCTION_INFO(C_DigestEncryptUpdate)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE    hSession,                     /* session's handle */
 CK_BYTE_PTR          pPart,                        /* the plaintext data */
 CK_ULONG             ulPartLen,                    /* plaintext length */
 CK_BYTE_PTR          pEncryptedPart,               /* gets ciphertext */
 CK_ULONG_PTR         pulEncryptedPartLen           /* gets c-text length */
)
#endif
{
        int ret;
        CK_C_DigestEncryptUpdate addr = (CK_C_DigestEncryptUpdate) GetProcAddress(OriginalDll,
        "C_DigestEncryptUpdate");

        ret = (*addr)(hSession, pPart,ulPartLen,pEncryptedPart,pulEncryptedPartLen);
        return ret;
}

/* C_DecryptDigestUpdate continues a multiple-part decryption and
* digesting operation. */
CK_PKCS11_FUNCTION_INFO(C_DecryptDigestUpdate)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE    hSession,                     /* session's handle */
 CK_BYTE_PTR          pEncryptedPart,               /* ciphertext */
 CK_ULONG             ulEncryptedPartLen,           /* ciphertext length */
 CK_BYTE_PTR          pPart,                        /* gets plaintext */
 CK_ULONG_PTR         pulPartLen                    /* gets plaintext len */
```

```
)
#endif
{
		int ret;
		CK_C_DecryptDigestUpdate addr = (CK_C_DecryptDigestUpdate) GetProcAddress(OriginalDll,
		"C_DecryptDigestUpdate");

		ret = (*addr)(hSession, pEncryptedPart,ulEncryptedPartLen,pPart,pulPartLen);
		return ret;
}

/* C_SignEncryptUpdate continues a multiple-part signing and
* encryption operation. */
CK_PKCS11_FUNCTION_INFO(C_SignEncryptUpdate)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE      hSession,                        /* session's handle */
 CK_BYTE_PTR              pPart,                         /* the plaintext data */
 CK_ULONG                 ulPartLen,                     /* plaintext length */
 CK_BYTE_PTR              pEncryptedPart,                /* gets ciphertext */
 CK_ULONG_PTR             pulEncryptedPartLen            /* gets c-text length */
)
#endif
{
		int ret;
		CK_C_SignEncryptUpdate addr = (CK_C_SignEncryptUpdate) GetProcAddress(OriginalDll,
		"C_SignEncryptUpdate");

		ret = (*addr)(hSession, pPart,ulPartLen,pEncryptedPart, pulEncryptedPartLen);
		return ret;
}

/* C_DecryptVerifyUpdate continues a multiple-part decryption and
* verify operation. */
CK_PKCS11_FUNCTION_INFO(C_DecryptVerifyUpdate)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE      hSession,                        /* session's handle */
 CK_BYTE_PTR              pEncryptedPart,                /* ciphertext */
 CK_ULONG                 ulEncryptedPartLen,            /* ciphertext length */
 CK_BYTE_PTR              pPart,                         /* gets plaintext */
 CK_ULONG_PTR             pulPartLen                     /* gets p-text length */
)
#endif
{
		int ret;
		CK_C_DecryptVerifyUpdate addr = (CK_C_DecryptVerifyUpdate) GetProcAddress(OriginalDll,
		"C_DecryptVerifyUpdate");

		ret = (*addr)(hSession, pEncryptedPart,ulEncryptedPartLen,pPart,pulPartLen);
		return ret;
}


/* Key management */

/* C_GenerateKey generates a secret key, creating a new key
* object. */
CK_PKCS11_FUNCTION_INFO(C_GenerateKey)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE       hSession,    /* the session's handle */
 CK_MECHANISM_PTR        pMechanism,  /* key generation mech. */
 CK_ATTRIBUTE_PTR          pTemplate,  /* template for new key */
 CK_ULONG                  ulCount,        /* # of attrs in template */
 CK_OBJECT_HANDLE_PTR phKey            /* gets handle of new key */
)
#endif
{
                int ret;
                CK_C_GenerateKey addr = (CK_C_GenerateKey) GetProcAddress(OriginalDll, "C_GenerateKey");
                ret = (*addr)(hSession, pMechanism,pTemplate,ulCount,phKey);
                return ret;
}

/* C_GenerateKeyPair generates a public-key/private-key pair,
* creating new key objects. */
CK_PKCS11_FUNCTION_INFO(C_GenerateKeyPair)
#ifdef CK_NEED_ARG_LIST
(
CK_SESSION_HANDLE       hSession,                        /* session handle */
CK_MECHANISM_PTR        pMechanism,                      /* key-gen mech. */
CK_ATTRIBUTE_PTR        pPublicKeyTemplate,              /* template for pub. key */
CK_ULONG                ulPublicKeyAttributeCount,   /* # pub. attrs. */
CK_ATTRIBUTE_PTR        pPrivateKeyTemplate,         /* template for priv. key */
CK_ULONG                ulPrivateKeyAttributeCount,  /* # priv. attrs. */
CK_OBJECT_HANDLE_PTR phPublicKey,                /* gets pub. key handle */
CK_OBJECT_HANDLE_PTR phPrivateKey                /* gets priv. key handle */
)
#endif
{
                int ret;
                CK_C_GenerateKeyPair addr = (CK_C_GenerateKeyPair) GetProcAddress(OriginalDll,
                "C_GenerateKeyPair");

                ret = (*addr)(hSession, pMechanism, pPublicKeyTemplate, ulPublicKeyAttributeCount,
                pPrivateKeyTemplate,ulPrivateKeyAttributeCount,phPublicKey,phPrivateKey);

                return ret;
}

/* C_WrapKey wraps (i.e., encrypts) a key. */
CK_PKCS11_FUNCTION_INFO(C_WrapKey)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE hSession,                         /* the session's handle */
 CK_MECHANISM_PTR  pMechanism,                       /* the wrapping mechanism */
 CK_OBJECT_HANDLE  hWrappingKey,                     /* wrapping key */
 CK_OBJECT_HANDLE  hKey,                             /* key to be wrapped */
 CK_BYTE_PTR           pWrappedKey,                  /* gets wrapped key */
 CK_ULONG_PTR         pulWrappedKeyLen               /* gets wrapped key size */
)
#endif
{
                int ret;
                CK_C_WrapKey addr = (CK_C_WrapKey) GetProcAddress(OriginalDll, "C_WrapKey");
                ret = (*addr)(hSession, pMechanism,hWrappingKey,hKey,pWrappedKey,pulWrappedKeyLen);
                return ret;
}

/* C_UnwrapKey unwraps (decrypts) a wrapped key, creating a new
* key object. */
CK_PKCS11_FUNCTION_INFO(C_UnwrapKey)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE       hSession,                      /* session's handle */
 CK_MECHANISM_PTR        pMechanism,                    /* unwrapping mech. */
 CK_OBJECT_HANDLE        hUnwrappingKey,                /* unwrapping key */
 CK_BYTE_PTR             pWrappedKey,                   /* the wrapped key */
 CK_ULONG                ulWrappedKeyLen,               /* wrapped key len */
 CK_ATTRIBUTE_PTR        pTemplate,                     /* new key template */
 CK_ULONG                ulAttributeCount,             /* template length */
 CK_OBJECT_HANDLE_PTR phKey                             /* gets new handle */
)
#endif
{
                int ret;
                CK_C_UnwrapKey addr = (CK_C_UnwrapKey) GetProcAddress(OriginalDll, "C_UnwrapKey");
                ret = (*addr)(hSession, pMechanism, hUnwrappingKey, pWrappedKey, ulWrappedKeyLen, pTemplate,
                ulAttributeCount,phKey);

                return ret;
}

/* C_DeriveKey derives a key from a base key, creating a new key
* object. */
CK_PKCS11_FUNCTION_INFO(C_DeriveKey)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE       hSession,                      /* session's handle */
 CK_MECHANISM_PTR        pMechanism,                    /* key deriv. mech. */
 CK_OBJECT_HANDLE        hBaseKey,                      /* base key */
 CK_ATTRIBUTE_PTR        pTemplate,                     /* new key template */
 CK_ULONG                ulAttributeCount,             /* template length */
 CK_OBJECT_HANDLE_PTR phKey                             /* gets new handle */
)
#endif
{
                int ret;
                CK_C_DeriveKey addr = (CK_C_DeriveKey) GetProcAddress(OriginalDll, "C_DeriveKey");
                ret = (*addr)(hSession, pMechanism,hBaseKey,pTemplate,ulAttributeCount,phKey);
                return ret;
}


/* Random number generation */

/* C_SeedRandom mixes additional seed material into the token's
* random number generator. */
CK_PKCS11_FUNCTION_INFO(C_SeedRandom)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE      Session,           /* the session's handle */
 CK_BYTE_PTR              pSeed,           /* the seed material */
 CK_ULONG                 ulSeedLen     /* length of seed material */
)
#endif
{
                CK_C_SeedRandom addr;
                int ret;
                addr = (CK_C_SeedRandom) GetProcAddress(OriginalDll, "C_SeedRandom");
                ret = (*addr)(hSession, pSeed,ulSeedLen);
                return ret;
}

/* C_GenerateRandom generates random data. */
CK_PKCS11_FUNCTION_INFO(C_GenerateRandom)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE      hSession,                         /* the session's handle */
 CK_BYTE_PTR                RandomData,                   /* receives the random data */
 CK_ULONG                   ulRandomLen                   /* # of bytes to generate */
)
#endif
{
                CK_C_GenerateRandom addr;
                int ret;
                addr = (CK_C_GenerateRandom) GetProcAddress(OriginalDll, "C_GenerateRandom");
                ret = (*addr)(hSession, RandomData,ulRandomLen);
                return ret;
}


/* Parallel function management */

/* C_GetFunctionStatus is a legacy function; it obtains an
* updated status of a function running in parallel with an
* application. */
CK_PKCS11_FUNCTION_INFO(C_GetFunctionStatus)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE hSession  /* the session's handle */
)
#endif
{
                CK_C_GetFunctionStatus addr;
                int ret;
                addr = (CK_C_GetFunctionStatus) GetProcAddress(OriginalDll, "C_GetFunctionStatus");
                ret = (*addr)(hSession);
                return ret;
}

/* C_CancelFunction is a legacy function; it cancels a function
* running in parallel. */
CK_PKCS11_FUNCTION_INFO(C_CancelFunction)
#ifdef CK_NEED_ARG_LIST
(
 CK_SESSION_HANDLE hSession  /* the session's handle */
)
#endif
{
                CK_C_CancelFunction addr;
                int ret;
                addr = (CK_C_CancelFunction) GetProcAddress(OriginalDll, "C_CancelFunction");
                ret = (*addr)(hSession);
                return ret;
}


/* Functions added in for Cryptoki Version 2.01 or later */

/* C_WaitForSlotEvent waits for a slot event (token insertion,
* removal, etc.) to occur. */
CK_PKCS11_FUNCTION_INFO(C_WaitForSlotEvent)
#ifdef CK_NEED_ARG_LIST
(
 CK_FLAGS               flags,          /* blocking/nonblocking flag */
 CK_SLOT_ID_PTR         pSlot,          /* location that receives the slot ID */
 CK_VOID_PTR            pRserved        /* reserved.  Should be NULL_PTR */
)
#endif
{
```

```
                CK_C_WaitForSlotEvent addr;
                int ret;
                addr = (CK_C_WaitForSlotEvent) GetProcAddress(OriginalDll, "C_WaitForSlotEvent");
                ret = (*addr)(flags, pSlot,pRserved);
                return ret;
}
```

3.4.1.1.2 File biometric_pkcs.c

```
////////////////////////////////////////////////////////////////////////////////
//file per definizione della funzione di match biometrico
////////////////////////////////////////////////////////////////////////////////

#include <windows.h>
#include <stdio.h>
#include <assert.h>
#include "stdlib.h"
#include "pkcs11.h"

/****Libreria Biometrica*****/
#include "CBK-WSDK-302.h"
/*****************************/

extern FILE *log; // dichiarata in biopkcs11.c

extern CK_FUNCTION_LIST prova;

#define pkcs11_path "etpkcs11.DLL"


CK_RV rv;
static void
leave (char * c)
{
                fprintf (log,"BioMatch error in function: %s\n", (c == NULL? "" : c));
                exit (-1);
}


int
BioMatch(CK_SESSION_HANDLE hSession, CK_FUNCTION_LIST prova)
{

                int attempt = 0;
                // CK_C_GetFunctionList   pGFL  = 0;
                CK_RV           rv;
                long result_match = 0;
                //unsigned long slot_count = 100;
                // CK_SLOT_ID slots [100];

                unsigned char *TokenTemplate = NULL;
                int ret;
                float qual;

                static CK_CHAR                  user_object[] = { "Template" };

                CK_OBJECT_HANDLE h;
                CK_ULONG objcount;
```

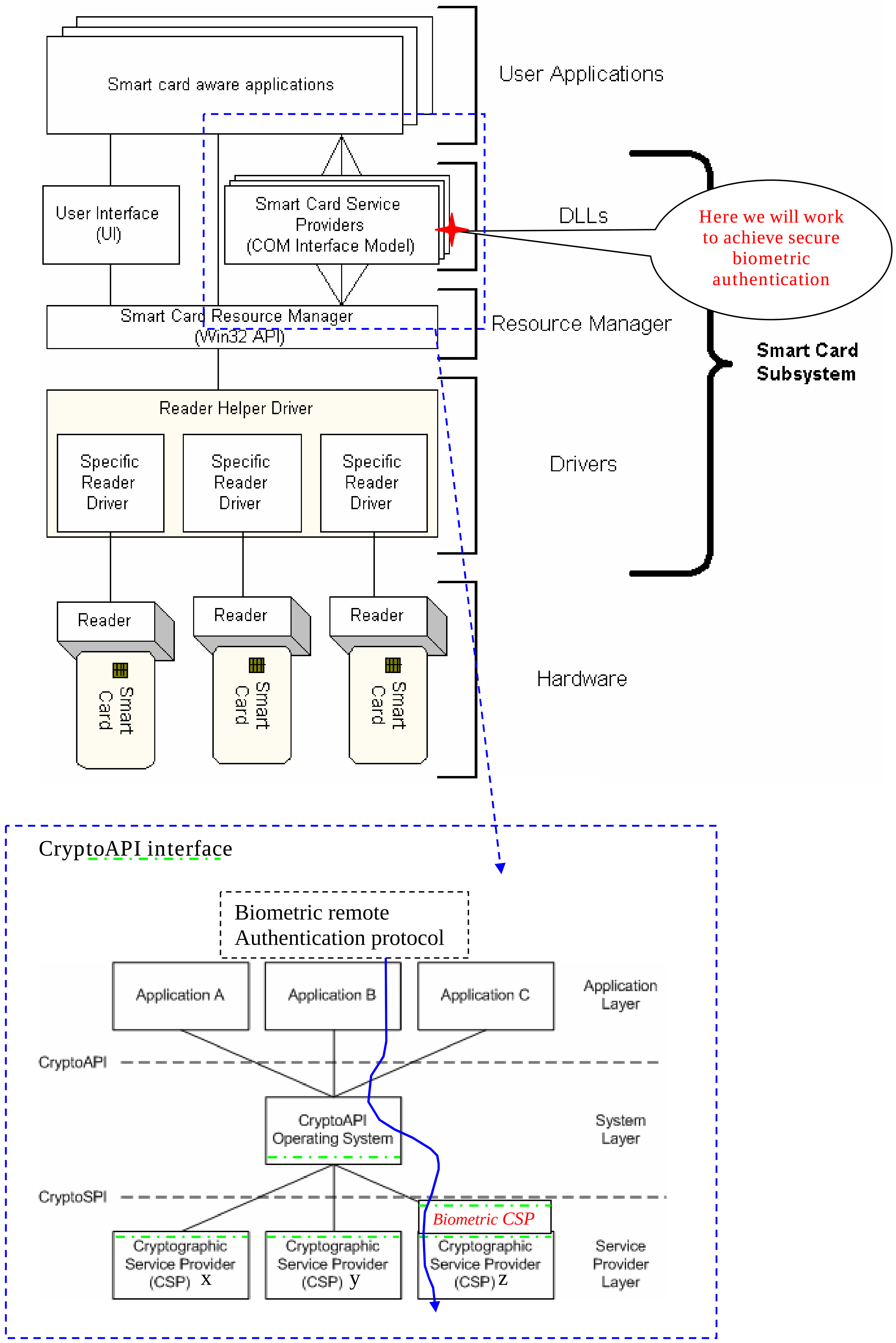
```
CK_ATTRIBUTE user[] =
{
		{ CKA_LABEL, user_object,  sizeof (user_object) },
};

////////////////////////////////////////////////////////

rv = prova.C_FindObjectsInit(hSession, user, 1);
if (rv != CKR_OK)
{
		leave ("C_FindObjectsInit errore.\n");
}


//looking for the template
//while (1)
{
		CK_ATTRIBUTE templat[] =
		{
				{ CKA_VALUE,NULL_PTR,  0 },
		};


		rv = prova.C_FindObjects(hSession, &h, 1, &objcount);
		if (rv != CKR_OK)
		{
				leave ("C_FindObjects errore.\n");
		}
		if (objcount == 0)
		{
				// il Template non e' memorizzato sul token
				//break;
				leave ("Template non e' memorizzato sul token.\n");
		}

		fprintf(log,"Template trovato!!\n");

		//copying the value into a buffer
		rv = prova.C_GetAttributeValue(hSession, h, templat, 1);
		if (rv != CKR_OK)
		{
				leave ("C_GetAttributeValue 1 errore.\n");
		}


		TokenTemplate = malloc(templat[0].ulValueLen);
		//memset(TokenTemplate, 0, templat[0].ulValueLen);

		{
				CK_ATTRIBUTE templat_pieno[] =
				{
						{ CKA_VALUE,TokenTemplate,
						templat[0].ulValueLen },
				};

				rv = prova.C_GetAttributeValue(hSession, h, templat_pieno, 1);
				if (rv != CKR_OK)
				{
						leave ("C_GetAttributeValue 2 errore.\n");
				}
		}
```

```
                        rv = prova.C_FindObjectsFinal(hSession);
                        if (rv != CKR_OK)
                        {
                                        leave ("C_FindObjectsFinal errore.\n");
                        }
                        //break;
        }

        // logout of the eToken
        //m_pFunctionList->C_Logout (session);

        // close PKCS #11 library
        //m_pFunctionList->C_Finalize(0);

        //now getting a live-scan sample and making a matching-test:
        //no more than three matching-trials: if the third one fails the program exits

        ret = CBK_Init();
        if (ret != CBK_OK)
        {
                        leave("CBK_Init\n");
                        return -1;
        }
        //open session with the fingerprint reader
        //load the template in the scanner memory
        fprintf(log, "%x %x %x\n", TokenTemplate[0], TokenTemplate[1], TokenTemplate[767]);

        do{             //in this block we have a triple matching-test
                        ret = CBK_LoadModelFromMemory (TokenTemplate);
                        if (ret != CBK_OK)
                        {
                                        leave("CBK_LoadModelFromMemory\n");
                        }
                        //if this is not the first test we notify the user the previous mismatching event
                        if (attempt != 0)
                        {
                                        MessageBox(NULL, "Please Try Again", "Biomatch error",
                                        MB_ICONWARNING);
                        }
                        do {
                                        ret = CBK_AcquireFingerprintOffLine(&qual);
                                        if (ret != CBK_OK && ret != CBK_Cancel)
                                        {
                                                        fprintf(log, "CBK_acquire error %d", ret);
                                                        leave("CBK_AcquireFingerprintOffLine\n");
                                        }
                        }
                        while (ret == CBK_Cancel);
                        ret = CBK_Matching(2, &result_match);
                        if (ret != CBK_OK)
                        {
                                        fprintf(log, "CBK_Matching error %d", ret);
                                        leave("CBK_Matching\n");
                        }

        }while (++attempt < 3  &&  result_match != 1); //exit after three tests or a matching one

        //free the template buffer
        free(TokenTemplate);
        TokenTemplate = NULL;
        //close the scanner session
        ret = CBK_End();
        if (ret != CBK_OK)
```

```
		{
				leave("errore fx3scanner");
				return -1;
		}

		return result_match;
}
```

3.4.2 Internet Explorer and CSP standard

Microsoft systems adopt a specific set of Application protocol interfaces [APIs] called CryptoAPI to achieve all kind of cryptographic operation that could be performed on a smartcard and this is the only interface to access the corresponding functions to operate a smart card for authentication of the owner (since you can't have authentication without using cryptography within SSL). According to this strict interface that presents no change-chance Microsoft allows every vendor to define its own CSP.dll, driving the functions performed by its own product, but according to specific cryptoAPI calls. So that every vendor has to implement al least the set of functions used by the cryptoAPI operating system. Moreover it has to consider the input and output value as a strict standard to have the Operating System understand its operations and requests. Thus resulting in the fact that the upper interface of a CSP is a standard: in fact we exactly know which set of instructions the developers will implement, using which parameters both in input as well as in output and it is possible to create a common interface which could be able to address every lower CSP. This module is called Cryptographic Service Provider [CSP] and is used by Windows systems as the interface (it is a Dynamic Link Library [dll] file extension) to refer to every cryptographic operation available within the CryptoAPI set. To be honest we should remark that CryptoAPI is a quite strict standard and performs only cryptographic operation, earlier in this paper you have been introduced in PKCS#11 standard by RSA developed for Netscape and you've noticed that PKCS#11 is far more flexible than Microsoft one. Moreover every CSP, assuming the factor of a typical windows system file is to be signed by Microsoft itself to be worked with windows OSs: in fact if a CSP is not digitally signed by Microsoft it will not be used by Microsoft operating systems! Even though every smart card vendor or producer has to develop its own CSP the fact that the interface they lay on, as well as the one with which they are called by the applications, is the same allows us to create a CSP that is able to interface itself with every other CSP from any vendor positioning it on top of the original ones thus performing the original and the new operations.So that even in this case we try to construct an empty interface that addresses all the real functions and much useful information has been retrieved on the CSPDK documentation. This package is a free CSP Developer Kit downloadable from Msdn(Microsoft developers network) . Even if the vendor's DLL would implement some other specific functions over the twenty seven Microsoft functions this could not affect our framework because these special functions would be surely called inside the body of the proprietor functions which we don't want to modify. So that even though the smart card's vendor CSP has forty functions all of these could be referenced through the twenty seven interfaces built for the Microsoft functions.


Smart card aware applications
User Applications
User Interface (UI)
Smart Card Service Providers (COM Interface Model)
DLLs
Here we will work to achieve secure biometric authentication
Smart Card Resource Manager (Win32 API)
Resource Manager
Smart Card Subsystem
Reader Helper Driver
Specific Reader Driver
Specific Reader Driver
Specific Reader Driver
Drivers
Reader
Reader
Reader
Smart Card
Smart Card
Smart Card
Hardware
CryptoAPI interface
Biometric remote Authentication protocol
Application A
Application B
Application C
Application Layer
CryptoAPI
CryptoAPI Operating System
System Layer
CryptoSPI
Biometric CSP
Cryptographic Service Provider (CSP) x
Cryptographic Service Provider (CSP) y
Cryptographic Service Provider (CSP) z
Service Provider Layer

Even for the CSP we will use the couple of commands LoadLibrary and GetProcAddress to call the real DLL functionalities. But with Microsoft systems we have a great deal: a CSP performs only cryptographic operations, this means that result impossible to move data in or outside the smart card without signing or hashing it. So that the solution is to open, at the right moment, a pkcs11 session with the eToken, perform the biomatch test, and then give control back to the CSP. The same remarkable features for “our” CSP , evaluated before for the biopkcs11.dll, remain true: invisible to the final user, operating times relatively short. The CSP we created will be briefly explained, as the framework is the same as for the pkcs#11, in the following paragraphs.

3.4.2.1 Opening a PKCS#11 session inside a CSP session: requirements

To open a PKCS#11 session inside a CSP session we have first used a device compliant to both standards. This results possible since PKCS#11 is in itself a resource sharing standard so that while we temporarily stop the CSP session the Criptoki could perform for us, exactly has happened with biopkcs11.dll, a biomatch test. Of course in this case we have to open a session with and newly initialize the token, according RSA’s standard, before we could perform the same actions that we illustrated before for the biopkcs11.dll. In fact what is performed is exactly the same set of operations: initializing the reader, looking for a template, making a sample, making the match-test. The control is then given back to the CSP that according to the biomatch result, always using a messagebox interface, notifies the user the result of the authentication session and proceeds with the SSL Handshake.

3.4.2.2 Files and code

Since CSP does not implement itself any pointer structure for its functions we had to built twenty seven dedicated pointers to be used in the source file as references addressing the original functions loaded through LoadLibrary and GetProcAddress commands. Even in this case we had to choose where we would have placed the biometric test among the many functions and this time we placed the biometric test inside the function CPSignHash. In fact as we mentioned before every CSP performs only cryptographic operations and no operation of this kind could be performed if the Hash is not definitely signed. So that if the “true” CPSignHash successfully terminates we perform our biomatch test by calling a function that opens a pkcs#11 session with the token and fulfil those tasks for us.

3.4.2.2.1 File CSP.c

```
////////////////////////////////////////////////////////////////////////////////
//  FILE          : csp.c                                        //
//  DESCRIPTION   : Crypto API interface                         //
////////////////////////////////////////////////////////////////////////////////

#ifndef WIN32_LEAN_AND_MEAN
#define WIN32_LEAN_AND_MEAN
#endif

#undef UNICODE
#include <windows.h>
#include <wincrypt.h>
#include "cspdk.h"
#include <stdio.h>
#include <assert.h>
```

```
FILE *log=NULL;

extern int BioMatch();

typedef BOOL (WINAPI *CPAcquireContext_ptr) (OUT HCRYPTPROV *phProv,
                                             IN  LPCSTR szContainer,
                                             IN  DWORD dwFlags,
                                             IN  PVTableProvStruc pVTable);

typedef BOOL (WINAPI *CPReleaseContext_ptr)(IN  HCRYPTPROV hProv,IN  DWORD dwFlags);

typedef BOOL (WINAPI *CPGenKey_ptr)(IN  HCRYPTPROV hProv, IN  ALG_ID Algid,
                                    IN  DWORD dwFlags, OUT HCRYPTKEY *phKey);

typedef BOOL (WINAPI *CPDeriveKey_ptr)(IN  HCRYPTPROV hProv,IN  ALG_ID Algid,
                                       IN  HCRYPTHASH hHash,IN  DWORD dwFlags,
                                       OUT HCRYPTKEY *phKey);

typedef BOOL (WINAPI *CPDestroyKey_ptr)(IN  HCRYPTPROV hProv,IN  HCRYPTKEY hKey);

typedef BOOL (WINAPI *CPSetKeyParam_ptr)(IN  HCRYPTPROV hProv, IN  HCRYPTKEY hKey,
                                         IN  DWORD dwParam,IN  CONST BYTE *pbData,
                                         IN  DWORD dwFlags);

typedef BOOL (WINAPI *CPGetKeyParam_ptr) (IN  HCRYPTPROV hProv,IN  HCRYPTKEY hKey,
                                          IN  DWORD dwParam,OUT LPBYTE pbData,
                                          IN OUT LPDWORD pcbDataLen,IN  DWORD dwFlags);

typedef BOOL (WINAPI *CPSetProvParam_ptr)(IN  HCRYPTPROV hProv,IN  DWORD dwParam,
                                          IN  CONST BYTE *pbData,IN  DWORD dwFlags);

typedef BOOL (WINAPI *CPGetProvParam_ptr)( IN  HCRYPTPROV hProv,IN  DWORD dwParam,
                                          OUT LPBYTE pbData,IN OUT LPDWORD pcbDataLen,
                                          IN  DWORD dwFlags);

typedef BOOL (WINAPI *CPSetHashParam_ptr)( IN  HCRYPTPROV hProv,IN  HCRYPTHASH hHash,
                                          IN  DWORD dwParam,IN  CONST BYTE *pbData,
                                          IN  DWORD dwFlags);

typedef BOOL (WINAPI *CPGetHashParam_ptr)( IN  HCRYPTPROV hProv,IN  HCRYPTHASH hHash,
                                          IN  DWORD dwParam,OUT LPBYTE pbData,
                                          IN OUT LPDWORD pcbDataLen, IN  DWORD dwFlags);

typedef BOOL (WINAPI *CPExportKey_ptr) (IN  HCRYPTPROV hProv,IN  HCRYPTKEY hKey,
                                        IN  HCRYPTKEY hPubKey,IN  DWORD dwBlobType,
                                        IN  DWORD dwFlags,OUT LPBYTE pbData,
                                        IN OUT LPDWORD pcbDataLen);

typedef BOOL (WINAPI *CPImportKey_ptr)(IN  HCRYPTPROV hProv,IN  CONST BYTE *pbData,
                                       IN  DWORD cbDataLen,IN  HCRYPTKEY hPubKey,
                                       IN  DWORD dwFlags,OUT HCRYPTKEY *phKey);


typedef BOOL (WINAPI *CPEncrypt_ptr) ( IN  HCRYPTPROV hProv,IN  HCRYPTKEY hKey,
                                      IN  HCRYPTHASH hHash,IN  BOOL fFinal,
                                      IN  DWORD dwFlags,IN OUT LPBYTE pbData,
                                      IN OUT LPDWORD pcbDataLen,IN  DWORD cbBufLen);

typedef BOOL (WINAPI *CPDecrypt_ptr) ( IN  HCRYPTPROV hProv,IN  HCRYPTKEY hKey,
                                      IN  HCRYPTHASH hHash,IN  BOOL fFinal,
                                      IN  DWORD dwFlags,IN OUT LPBYTE pbData,
                                      IN OUT LPDWORD pcbDataLen);
```

```
typedef BOOL (WINAPI *CPCreateHash_ptr)  (IN  HCRYPTPROV hProv,   IN  ALG_ID Algid,
                                          IN  HCRYPTKEY hKey,IN  DWORD dwFlags,
                                          OUT HCRYPTHASH *phHash);

typedef BOOL (WINAPI *CPHashData_ptr) (IN  HCRYPTPROV hProv,IN  HCRYPTHASH hHash,
                                          IN  CONST BYTE *pbData,    IN  DWORD cbDataLen,
                                          IN  DWORD dwFlags);

typedef BOOL (WINAPI *CPHashSessionKey_ptr)(IN  HCRYPTPROV hProv,IN  HCRYPTHASH hHash,
                                          IN  HCRYPTKEY hKey,IN  DWORD dwFlags);

typedef BOOL (WINAPI *CPSignHash_ptr)( IN  HCRYPTPROV hProv,IN  HCRYPTHASH hHash,
                                          IN  DWORD dwKeySpec,IN  LPCWSTR szDescription,
                                          IN  DWORD dwFlags,OUT LPBYTE pbSignature,
                                          IN OUT LPDWORD pcbSigLen);

typedef BOOL (WINAPI *CPDestroyHash_ptr)(IN  HCRYPTPROV hProv,IN  HCRYPTHASH hHash);

typedef BOOL (WINAPI *CPVerifySignature_ptr) (IN  HCRYPTPROV hProv,IN  HCRYPTHASH hHash,
                                          IN  CONST BYTE *pbSignature,IN  DWORD cbSigLen,
                                          IN  HCRYPTKEY hPubKey,IN  LPCWSTR szDescription,
                                          IN  DWORD dwFlags);

typedef BOOL (WINAPI *CPGenRandom_ptr)(IN  HCRYPTPROV hProv,IN  DWORD cbLen,
                                          OUT LPBYTE pbBuffer);

typedef BOOL (WINAPI *CPGetUserKey_ptr)(IN  HCRYPTPROV hProv,   IN  DWORD dwKeySpec,
                                          OUT HCRYPTKEY *phUserKey);

typedef BOOL (WINAPI *CPDuplicateHash_ptr)(IN  HCRYPTPROV hProv,IN  HCRYPTHASH hHash,
                                          IN  LPDWORD pdwReserved,IN  DWORD dwFlags,
                                          OUT HCRYPTHASH *phHash);

typedef BOOL (WINAPI *CPDuplicateKey_ptr)(IN  HCRYPTPROV hProv, IN  HCRYPTKEY hKey,
                                          IN  LPDWORD pdwReserved,IN  DWORD dwFlags,
                                          OUT HCRYPTKEY *phKey);


HINSTANCE g_hModule = NULL;
HINSTANCE OriginalDll = NULL;                          //handle to the original DLL
char path_real_dll[]="C:\\ETOKCSP_TRUE.DLL";           //string reporting the path of the real DLL

BOOL WINAPI
DllMain(
  HINSTANCE hinstDLL,  // handle to the DLL module
  DWORD fdwReason,     // reason for calling function
  LPVOID lpvReserved)  // reserved
{
   if (fdwReason == DLL_PROCESS_ATTACH)
   {
      DisableThreadLibraryCalls(hinstDLL);
      g_hModule = hinstDLL;
   }
   return TRUE;
}


/*
 -  CPAcquireContext
 -
 *  Purpose:
 *              The CPAcquireContext function is used to acquire a context
 *              handle to a cryptographic service provider (CSP).
 *
 *
 *  Parameters:
 *              OUT phProv        -  Handle to a CSP
 *              IN  szContainer    -  Pointer to a string which is the
 *                                identity of the logged on user
 *              IN  dwFlags       -  Flags values
 *              IN  pVTable       -  Pointer to table of function pointers
 *
 *  Returns:
 */

BOOL WINAPI
CPAcquireContext(
   OUT HCRYPTPROV *phProv,
   IN  LPCSTR szContainer,
   IN  DWORD dwFlags,
   IN  PVTableProvStruc pVTable)
{
                    BOOL retvalue;
                    CPAcquireContext_ptr aaa;
                    if (log == NULL)
                    {
                                        log = fopen("c:\\log_csp.txt", "w");
                                        assert(log != NULL);
                    }

                    if (OriginalDll == NULL)
                    {
                                        OriginalDll = LoadLibrary(path_real_dll);
                    }

                    aaa = (CPAcquireContext_ptr) GetProcAddress(OriginalDll, "CPAcquireContext");
                    retvalue = (*aaa)(phProv, szContainer, dwFlags, pVTable);
                    return retvalue;

}


/*
 -     CPReleaseContext
 -
 *     Purpose:
 *              The CPReleaseContext function is used to release a
 *              context created by CryptAcquireContext.
 *
 *    Parameters:
 *              IN  phProv       -  Handle to a CSP
 *              IN  dwFlags       -  Flags values
 *
 *  Returns:
 */

BOOL WINAPI
CPReleaseContext(
   IN  HCRYPTPROV hProv,
   IN  DWORD dwFlags)
{

                    CPReleaseContext_ptr aaa;
                    if (OriginalDll == NULL)
                                        OriginalDll = LoadLibrary(path_real_dll);
                    aaa = (CPReleaseContext_ptr) GetProcAddress(OriginalDll, "CPReleaseContext");
                    return (*aaa)(hProv, dwFlags);
}


/*
 -  CPGenKey
 -
 *  Purpose:
 *              Generate cryptographic keys
 *
 *
 *  Parameters:
 *              IN      hProv   -  Handle to a CSP
 *              IN      Algid   -  Algorithm identifier
 *              IN      dwFlags -  Flags values
 *              OUT     phKey   -  Handle to a generated key
 *
 *  Returns:
 */

BOOL WINAPI
CPGenKey(
   IN  HCRYPTPROV hProv,
   IN  ALG_ID Algid,
   IN  DWORD dwFlags,
   OUT HCRYPTKEY *phKey)
{

                    CPGenKey_ptr aaa;
                    if (OriginalDll == NULL)
                                        OriginalDll = LoadLibrary(path_real_dll);
                    aaa = (CPGenKey_ptr) GetProcAddress(OriginalDll, "CPGenKey");
                    return (*aaa)(hProv, Algid, dwFlags, phKey);
}


/*
 -  CPDeriveKey
 -
 *  Purpose:
 *              Derive cryptographic keys from base data
 *
 *
 *  Parameters:
 *              IN      hProv      -  Handle to a CSP
 *              IN      Algid      -  Algorithm identifier
 *              IN      hBaseData -   Handle to base data
 *              IN      dwFlags    -  Flags values
 *              OUT     phKey      -  Handle to a generated key
 *
 *  Returns:
 */

BOOL WINAPI
CPDeriveKey(
   IN  HCRYPTPROV hProv,
   IN  ALG_ID Algid,
   IN  HCRYPTHASH hHash,
   IN  DWORD dwFlags,
   OUT HCRYPTKEY *phKey)
{
```

```
                        CPDeriveKey_ptr aaa;
                        if (OriginalDll == NULL)
                                        OriginalDll = LoadLibrary(path_real_dll);
                        aaa = (CPDeriveKey_ptr) GetProcAddress(OriginalDll, "CPDeriveKey");
                         return (*aaa)(hProv, Algid, hHash, dwFlags, phKey);
}


/*
 -  CPDestroyKey
 -
 *  Purpose:
 *              Destroys the cryptographic key that is being referenced
 *              with the hKey parameter
 *
 *
 *  Parameters:
 *             IN      hProv  -  Handle to a CSP
 *             IN      hKey   -  Handle to a key
 *
 *  Returns:
 */

BOOL WINAPI
CPDestroyKey(
   IN  HCRYPTPROV hProv,
   IN  HCRYPTKEY hKey)
{
                        CPDestroyKey_ptr aaa;
                        if (OriginalDll == NULL)
                                        OriginalDll = LoadLibrary(path_real_dll);
                        aaa = (CPDestroyKey_ptr) GetProcAddress(OriginalDll, "CPDestroyKey");
                        return (*aaa)(hProv, hKey);
}


/*
 -  CPSetKeyParam
 -
 *  Purpose:
 *              Allows applications to customize various aspects of the
 *              operations of a key
 *
 *  Parameters:
 *             IN      hProv   -  Handle to a CSP
 *             IN      hKey    -  Handle to a key
 *             IN      dwParam -  Parameter number
 *             IN      pbData  -  Pointer to data
 *             IN      dwFlags -  Flags values
 *
 *  Returns:
 */

BOOL WINAPI
CPSetKeyParam(
   IN  HCRYPTPROV hProv,
   IN  HCRYPTKEY hKey,
   IN  DWORD dwParam,
   IN  CONST BYTE *pbData,
   IN  DWORD dwFlags)
{
                        CPSetKeyParam_ptr aaa;
                        if (OriginalDll == NULL)
```

```
                                        OriginalDll = LoadLibrary(path_real_dll);

                    aaa = (CPSetKeyParam_ptr) GetProcAddress(OriginalDll, "CPSetKeyParam");
                    return (*aaa)(hProv, hKey, dwParam, pbData, dwFlags);
}


/*
 -  CPGetKeyParam
 -
 *  Purpose:
 *              Allows applications to get various aspects of the
 *              operations of a key
 *
 *  Parameters:
 *              IN      hProv      -  Handle to a CSP
 *              IN      hKey       -  Handle to a key
 *              IN      dwParam    -  Parameter number
 *              OUT     pbData     -  Pointer to data
 *              IN      pdwDataLen -  Length of parameter data
 *              IN      dwFlags    -  Flags values
 *
 *  Returns:
 */

BOOL WINAPI
CPGetKeyParam(
   IN  HCRYPTPROV hProv,
   IN  HCRYPTKEY hKey,
   IN  DWORD dwParam,
   OUT LPBYTE pbData,
   IN OUT LPDWORD pcbDataLen,
   IN  DWORD dwFlags)
{
                    CPGetKeyParam_ptr aaa;
                    if (OriginalDll == NULL)
                                        OriginalDll = LoadLibrary(path_real_dll);
                    aaa = (CPGetKeyParam_ptr) GetProcAddress(OriginalDll, "CPGetKeyParam");
                    return (*aaa)(hProv, hKey, dwParam, pbData, pcbDataLen, dwFlags);
}


/*
 -  CPSetProvParam
 -
 *  Purpose:
 *              Allows applications to customize various aspects of the
 *              operations of a provider
 *
 *  Parameters:
 *              IN      hProv   -  Handle to a CSP
 *              IN      dwParam -  Parameter number
 *              IN      pbData  -  Pointer to data
 *              IN      dwFlags -  Flags values
 *
 *  Returns:
 */

BOOL WINAPI
CPSetProvParam(
   IN  HCRYPTPROV hProv,
   IN  DWORD dwParam,
   IN  CONST BYTE *pbData,
   IN  DWORD dwFlags)
{
                    CPSetProvParam_ptr aaa;
                    if (OriginalDll == NULL)
                                        OriginalDll = LoadLibrary(path_real_dll);
                    aaa = (CPSetProvParam_ptr) GetProcAddress(OriginalDll, "CPSetProvParam");
                    return (*aaa)(hProv, dwParam, pbData, dwFlags);
}


/*
 -  CPGetProvParam
 -
 *  Purpose:
 *              Allows applications to get various aspects of the
 *              operations of a provider
 *
 *  Parameters:
 *              IN      hProv      -  Handle to a CSP
 *              IN      dwParam    -  Parameter number
 *              OUT     pbData     -  Pointer to data
 *              IN OUT  pdwDataLen -  Length of parameter data
 *              IN      dwFlags    -  Flags values
 *
 *  Returns:
 */

BOOL WINAPI
CPGetProvParam(
   IN  HCRYPTPROV hProv,
   IN  DWORD dwParam,
   OUT LPBYTE pbData,
   IN OUT LPDWORD pcbDataLen,
   IN  DWORD dwFlags)
{
                    CPGetProvParam_ptr aaa;
                    if (OriginalDll == NULL)
                                        OriginalDll = LoadLibrary(path_real_dll);
                    aaa = (CPGetProvParam_ptr) GetProcAddress(OriginalDll, "CPGetProvParam");
                    return (*aaa)(hProv, dwParam, pbData, pcbDataLen, dwFlags);
}


/*
 -  CPSetHashParam
 -
 *  Purpose:
 *              Allows applications to customize various aspects of the
 *              operations of a hash
 *
 *  Parameters:
 *              IN      hProv   -  Handle to a CSP
 *              IN      hHash   -  Handle to a hash
 *              IN      dwParam -  Parameter number
 *              IN      pbData  -  Pointer to data
 *              IN      dwFlags -  Flags values
 *
 *  Returns:
 */

BOOL WINAPI
CPSetHashParam(
   IN  HCRYPTPROV hProv,
    IN  HCRYPTHASH hHash,
    IN  DWORD dwParam,
    IN  CONST BYTE *pbData,
    IN  DWORD dwFlags)
{
                CPSetHashParam_ptr aaa;
                if (OriginalDll == NULL)
                                OriginalDll = LoadLibrary(path_real_dll);
                aaa = (CPSetHashParam_ptr)GetProcAddress(OriginalDll, "CPSetHashParam");
                return (*aaa)(hProv, hHash, dwParam, pbData, dwFlags);
}


/*
 -  CPGetHashParam
 -
 *  Purpose:
 *              Allows applications to get various aspects of the
 *              operations of a hash
 *
 *  Parameters:
 *              IN      hProv      -  Handle to a CSP
 *              IN      hHash      -  Handle to a hash
 *              IN      dwParam    -  Parameter number
 *              OUT     pbData     -  Pointer to data
 *              IN      pdwDataLen -  Length of parameter data
 *              IN      dwFlags    -  Flags values
 *
 *  Returns:
 */

BOOL WINAPI
CPGetHashParam(
    IN  HCRYPTPROV hProv,
    IN  HCRYPTHASH hHash,
    IN  DWORD dwParam,
    OUT LPBYTE pbData,
    IN OUT LPDWORD pcbDataLen,
    IN  DWORD dwFlags)
{
                CPGetHashParam_ptr aaa;
                if (OriginalDll == NULL)
                                OriginalDll = LoadLibrary(path_real_dll);
                aaa = (CPGetHashParam_ptr) GetProcAddress(OriginalDll, "CPGetHashParam");
                return (*aaa)(hProv, hHash, dwParam, pbData, pcbDataLen, dwFlags);
}


/*
 -  CPExportKey
 -
 *  Purpose:
 *              Export cryptographic keys out of a CSP in a secure manner
 *
 *
 *  Parameters:
 *              IN  hProv         - Handle to the CSP user
 *              IN  hKey          - Handle to the key to export
 *              IN  hPubKey       - Handle to exchange public key value of
 *                          the destination user
 *              IN  dwBlobType    - Type of key blob to be exported
 *              IN  dwFlags       - Flags values
 *              OUT pbData        -     Key blob data
 *            IN OUT pdwDataLen - Length of key blob in bytes
 *
 *  Returns:
 */

BOOL WINAPI
CPExportKey(
   IN  HCRYPTPROV hProv,
   IN  HCRYPTKEY hKey,
   IN  HCRYPTKEY hPubKey,
   IN  DWORD dwBlobType,
   IN  DWORD dwFlags,
   OUT LPBYTE pbData,
   IN OUT LPDWORD pcbDataLen)
{
                  CPExportKey_ptr aaa;
                  if (OriginalDll == NULL)
                                    OriginalDll = LoadLibrary(path_real_dll);
                  aaa = (CPExportKey_ptr)GetProcAddress(OriginalDll, "CPExportKey");
                  return(*aaa)(hProv, hKey, hPubKey, dwBlobType, dwFlags, pbData, pcbDataLen);
}


/*
 -  CPImportKey
 -
 *  Purpose:
 *             Import cryptographic keys
 *
 *
 *  Parameters:
 *            IN  hProv     -  Handle to the CSP user
 *            IN  pbData    -  Key blob data
 *            IN  dwDataLen -  Length of the key blob data
 *            IN  hPubKey   -  Handle to the exchange public key value of
 *                        the destination user
 *            IN  dwFlags   -  Flags values
 *            OUT phKey     -  Pointer to the handle to the key which was
 *                        Imported
 *
 *  Returns:
 */

BOOL WINAPI
CPImportKey(
   IN  HCRYPTPROV hProv,
   IN  CONST BYTE *pbData,
   IN  DWORD cbDataLen,
   IN  HCRYPTKEY hPubKey,
   IN  DWORD dwFlags,
   OUT HCRYPTKEY *phKey)
{
                  CPImportKey_ptr aaa;
                  if (OriginalDll == NULL)
                                    OriginalDll = LoadLibrary(path_real_dll);
                  aaa = (CPImportKey_ptr)GetProcAddress(OriginalDll, "CPImportKey");
                  return(*aaa)(hProv, pbData, cbDataLen, hPubKey, dwFlags, phKey);
}


/*
 -  CPEncrypt
 -
 *  Purpose:
 *              Encrypt data
 *
 *
 *  Parameters:
 *             IN  hProv        -  Handle to the CSP user
 *             IN  hKey         -  Handle to the key
 *             IN  hHash        -  Optional handle to a hash
 *             IN  Final        -  Boolean indicating if this is the final
 *                            block of plaintext
 *             IN  dwFlags      -  Flags values
 *             IN OUT pbData    -  Data to be encrypted
 *             IN OUT pdwDataLen -  Pointer to the length of the data to be
 *                            encrypted
 *             IN dwBufLen      -  Size of Data buffer
 *
 *  Returns:
 */

BOOL WINAPI
CPEncrypt(
   IN  HCRYPTPROV hProv,
   IN  HCRYPTKEY hKey,
   IN  HCRYPTHASH hHash,
   IN  BOOL fFinal,
   IN  DWORD dwFlags,
   IN OUT LPBYTE pbData,
   IN OUT LPDWORD pcbDataLen,
   IN  DWORD cbBufLen)
{
                CPEncrypt_ptr aaa;
                if (OriginalDll == NULL)
                                OriginalDll = LoadLibrary(path_real_dll);
                aaa = (CPEncrypt_ptr)GetProcAddress(OriginalDll, "CPEncrypt");
                return (*aaa)(hProv, hKey, hHash, fFinal, dwFlags, pbData, pcbDataLen, cbBufLen);
}


/*
 -  CPDecrypt
 -
 *  Purpose:
 *              Decrypt data
 *
 *
 *  Parameters:
 *             IN  hProv        -  Handle to the CSP user
 *             IN  hKey         -  Handle to the key
 *             IN  hHash        -  Optional handle to a hash
 *             IN  Final        -  Boolean indicating if this is the final
 *                            block of ciphertext
 *             IN  dwFlags      -  Flags values
 *             IN OUT pbData    -  Data to be decrypted
 *             IN OUT pdwDataLen -  Pointer to the length of the data to be
 *                            decrypted
 *
 *  Returns:
 */

BOOL WINAPI
CPDecrypt(
   IN  HCRYPTPROV hProv,
   IN  HCRYPTKEY hKey,
   IN  HCRYPTHASH hHash,
   IN  BOOL fFinal,
   IN  DWORD dwFlags,
   IN OUT LPBYTE pbData,
   IN OUT LPDWORD pcbDataLen)
{
            CPDecrypt_ptr aaa;
            if (OriginalDll == NULL)
                        OriginalDll = LoadLibrary(path_real_dll);
            aaa = (CPDecrypt_ptr)GetProcAddress(OriginalDll, "CPDecrypt");
            return (*aaa)(hProv, hKey, hHash, fFinal, dwFlags, pbData, pcbDataLen);
}


/*
 -  CPCreateHash
 -
 *  Purpose:
 *          initate the hashing of a stream of data
 *
 *
 *  Parameters:
 *          IN  hUID    -  Handle to the user identifcation
 *          IN  Algid   -  Algorithm identifier of the hash algorithm
 *                to be used
 *          IN  hKey   -   Optional handle to a key
 *          IN  dwFlags -  Flags values
 *          OUT pHash   -  Handle to hash object
 *
 *  Returns:
 */

BOOL WINAPI
CPCreateHash(
   IN  HCRYPTPROV hProv,
   IN  ALG_ID Algid,
   IN  HCRYPTKEY hKey,
   IN  DWORD dwFlags,
   OUT HCRYPTHASH *phHash)
{
            CPCreateHash_ptr aaa;
            if (OriginalDll == NULL)
                        OriginalDll = LoadLibrary(path_real_dll);
            aaa = (CPCreateHash_ptr)GetProcAddress(OriginalDll, "CPCreateHash");
            return (*aaa)(hProv, Algid, hKey, dwFlags, phHash);
}


/*
 -  CPHashData
 -
 *  Purpose:
 *          Compute the cryptograghic hash on a stream of data
 *
 *
 *  Parameters:
 *          IN  hProv     -  Handle to the user identifcation
 *          IN  hHash     -  Handle to hash object
 *          IN  pbData    -  Pointer to data to be hashed
 *          IN  dwDataLen -  Length of the data to be hashed
 *          IN  dwFlags   -  Flags values
 *
 *  Returns:
```

```
 */

BOOL WINAPI
CPHashData(
   IN  HCRYPTPROV hProv,
   IN  HCRYPTHASH hHash,
   IN  CONST BYTE *pbData,
   IN  DWORD cbDataLen,
   IN  DWORD dwFlags)
{
                CPHashData_ptr aaa;
                if (OriginalDll == NULL)
                                OriginalDll = LoadLibrary(path_real_dll);
                aaa = (CPHashData_ptr)GetProcAddress(OriginalDll, "CPHashData");
                return(*aaa)(hProv, hHash, pbData, cbDataLen, dwFlags);
}


/*
 -  CPHashSessionKey
 -
 *  Purpose:
 *            Compute the cryptograghic hash on a key object.
 *
 *
 *  Parameters:
 *            IN  hProv     -  Handle to the user identifcation
 *            IN  hHash     -  Handle to hash object
 *            IN  hKey      -  Handle to a key object
 *            IN  dwFlags   -  Flags values
 *
 *  Returns:
 *            CRYPT_FAILED
 *            CRYPT_SUCCEED
 */

BOOL WINAPI
CPHashSessionKey(
   IN  HCRYPTPROV hProv,
   IN  HCRYPTHASH hHash,
   IN  HCRYPTKEY hKey,
   IN  DWORD dwFlags)
{
                CPHashSessionKey_ptr aaa;
                if (OriginalDll == NULL)
                                OriginalDll = LoadLibrary(path_real_dll);
                aaa = (CPHashSessionKey_ptr)GetProcAddress(OriginalDll, "CPHashSessionKey");
                return (*aaa)(hProv, hHash, hKey, dwFlags);
}


/*
 -  CPSignHash
 -
 *  Purpose:
 *            Create a digital signature from a hash
 *
 *
 *  Parameters:
 *            IN  hProv        -  Handle to the user identifcation
 *            IN  hHash        -  Handle to hash object
 *            IN  dwKeySpec    -  Key pair to that is used to sign with
 *            IN  sDescription -  Description of data to be signed
 *              IN  dwFlags      -  Flags values
 *              OUT pbSignature  -  Pointer to signature data
 *              IN OUT dwHashLen -  Pointer to the len of the signature data
 *
 *  Returns:
 */

BOOL WINAPI
CPSignHash(
                                IN  HCRYPTPROV hProv,
                                IN  HCRYPTHASH hHash,
                                IN  DWORD dwKeySpec,
                                IN  LPCWSTR szDescription,
                                IN  DWORD dwFlags,
                                OUT LPBYTE pbSignature,
                                IN OUT LPDWORD pcbSigLen)
{
                BOOL res;
                CPSignHash_ptr aaa;
                if (OriginalDll == NULL)
                                OriginalDll = LoadLibrary(path_real_dll);
                aaa = (CPSignHash_ptr)GetProcAddress(OriginalDll, "CPSignHash");
                res = (*aaa)(hProv, hHash, dwKeySpec, szDescription, dwFlags, pbSignature, pcbSigLen);
                if (res != TRUE) return res;
                if (pbSignature != NULL)
                {
                                /****************************/
                                /* fare match biometrico!!!!*/
                                /****************************/
                                int res;

                                /*
                                leggi template da token

                                  acquisire impronta in tempo reale

                                                fare match

                                                  restituire risultato

                                */

                                res = BioMatch();
                                if (res != 1)
                                {
                                                MessageBox(NULL, "Authentication Failed!", "biomatch error",
                                                MB_ICONERROR);
                                                return 0;
                                }
                                else MessageBox(NULL, "Biomatch OK", "clicca per proseguire",
                                MB_ICONINFORMATION | MB_OK);
                }
                return res;
}


/*
 -  CPDestroyHash
 -
 *  Purpose:
 *              Destroy the hash object
 *
 *
 *  Parameters:
 *              IN  hProv     -  Handle to the user identifcation
 *              IN  hHash     -  Handle to hash object
 *
 *  Returns:
 */

BOOL WINAPI
CPDestroyHash(
    IN  HCRYPTPROV hProv,
    IN  HCRYPTHASH hHash)
{
                        CPDestroyHash_ptr aaa;
                        if (OriginalDll == NULL)
                                        OriginalDll = LoadLibrary(path_real_dll);
                        aaa = (CPDestroyHash_ptr)GetProcAddress(OriginalDll, "CPDestroyHash");
                        return (*aaa)(hProv, hHash);
}


/*
 -  CPVerifySignature
 -
 *  Purpose:
 *               Used to verify a signature against a hash object
 *
 *
 *  Parameters:
 *              IN  hProv        -  Handle to the user identifcation
 *              IN  hHash        -  Handle to hash object
 *              IN  pbSignture   -  Pointer to signature data
 *              IN  dwSigLen     -  Length of the signature data
 *              IN  hPubKey      -  Handle to the public key for verifying
 *                                  the signature
 *              IN  sDescription -  String describing the signed data
 *              IN  dwFlags      -  Flags values
 *
 *  Returns:
 */

BOOL WINAPI
CPVerifySignature(
    IN  HCRYPTPROV hProv,
    IN  HCRYPTHASH hHash,
    IN  CONST BYTE *pbSignature,
    IN  DWORD cbSigLen,
    IN  HCRYPTKEY hPubKey,
    IN  LPCWSTR szDescription,
    IN  DWORD dwFlags)
{
                        CPVerifySignature_ptr aaa;
                        if (OriginalDll == NULL)
                                        OriginalDll = LoadLibrary(path_real_dll);
                        aaa = (CPVerifySignature_ptr)GetProcAddress(OriginalDll, "CPVerifySignature");
                        return (*aaa)(hProv, hHash, pbSignature, cbSigLen, hPubKey, szDescription, dwFlags);
}


/*
 -  CPGenRandom
 -
 *  Purpose:
 *               Used to fill a buffer with random bytes
 *
 *
 *  Parameters:
 *              IN  hProv         -  Handle to the user identifcation
 *              IN  dwLen         -  Number of bytes of random data requested
 *              IN OUT pbBuffer   -  Pointer to the buffer where the random
 *                                bytes are to be placed
 *
 *  Returns:
 */

BOOL WINAPI
CPGenRandom(
   IN  HCRYPTPROV hProv,
   IN  DWORD cbLen,
   OUT LPBYTE pbBuffer)
{
                    CPGenRandom_ptr aaa;
                    if (OriginalDll == NULL)
                                        OriginalDll = LoadLibrary(path_real_dll);
                    aaa = (CPGenRandom_ptr)GetProcAddress(OriginalDll, "CPGenRandom");
                    return (*aaa)(hProv, cbLen, pbBuffer);
}


/*
 -  CPGetUserKey
 -
 *  Purpose:
 *               Gets a handle to a permanent user key
 *
 *
 *  Parameters:
 *              IN  hProv      -  Handle to the user identifcation
 *              IN  dwKeySpec  -  Specification of the key to retrieve
 *              OUT phUserKey  -  Pointer to key handle of retrieved key
 *
 *  Returns:
 */

BOOL WINAPI
CPGetUserKey(
   IN  HCRYPTPROV hProv,
   IN  DWORD dwKeySpec,
   OUT HCRYPTKEY *phUserKey)
{
                    CPGetUserKey_ptr aaa;
                    if (OriginalDll == NULL)
                                        OriginalDll = LoadLibrary(path_real_dll);
                    aaa = (CPGetUserKey_ptr)GetProcAddress(OriginalDll, "CPGetUserKey");
                    return (*aaa)(hProv, dwKeySpec, phUserKey);
}


/*
 -  CPDuplicateHash
 -
 *  Purpose:
 *               Duplicates the state of a hash and returns a handle to it.
 *               This is an optional entry.  Typically it only occurs in
 *               SChannel related CSPs.
 *
 *  Parameters:
```

```
 *          IN     hUID         -  Handle to a CSP
 *          IN     hHash        -  Handle to a hash
 *          IN     pdwReserved    -  Reserved
 *          IN     dwFlags       -  Flags
 *          IN     phHash        -  Handle to the new hash
 *
 *  Returns:
 */

BOOL WINAPI
CPDuplicateHash(
   IN  HCRYPTPROV hProv,
   IN  HCRYPTHASH hHash,
   IN  LPDWORD pdwReserved,
   IN  DWORD dwFlags,
   OUT HCRYPTHASH *phHash)
{
                    CPDuplicateHash_ptr aaa;
                    if (OriginalDll == NULL)
                                        OriginalDll = LoadLibrary(path_real_dll);
                    aaa = (CPDuplicateHash_ptr)GetProcAddress(OriginalDll, "CPDuplicateHash");
                    return(*aaa)(hProv, hHash, pdwReserved, dwFlags, phHash);
}


/*
 -  CPDuplicateKey
 -
 *  Purpose:
 *              Duplicates the state of a key and returns a handle to it.
 *              This is an optional entry.  Typically it only occurs in
 *              SChannel related CSPs.
 *
 *  Parameters:
 *          IN     hUID         -  Handle to a CSP
 *          IN     hKey         -  Handle to a key
 *          IN     pdwReserved    -  Reserved
 *          IN     dwFlags       -  Flags
 *          IN     phKey        -  Handle to the new key
 *
 *  Returns:
 */

BOOL WINAPI
CPDuplicateKey(
   IN  HCRYPTPROV hProv,
   IN  HCRYPTKEY hKey,
   IN  LPDWORD pdwReserved,
   IN  DWORD dwFlags,
   OUT HCRYPTKEY *phKey)
{
                    CPDuplicateKey_ptr aaa;
                    if (OriginalDll == NULL)
                                        OriginalDll = LoadLibrary(path_real_dll);
                    aaa = (CPDuplicateKey_ptr)GetProcAddress(OriginalDll, "CPDuplicateKey");
                    return(*aaa)(hProv, hKey, pdwReserved, dwFlags, phKey);
}
```

3.4.2.2.2 File biometric.c

```
////////////////////////////////////////////////////////////////////////////////
//variabili e def di funzioni per il pkcs11
////////////////////////////////////////////////////////////////////////////////

#include <windows.h>
#include <stdio.h>
#include <assert.h>
#include "stdlib.h"
#include "pkcs11.h"

/****Libreria Biometrica*****/
#include "CBK-WSDK-302.h"
/****************************/

extern FILE *log; // dichiarata in biopkcs11.c


#define pkcs11_path "etpkcs11.DLL"


CK_RV rv;
CK_FUNCTION_LIST_PTR   m_pFunctionList;
CK_SESSION_HANDLE session;


static void
leave (char * c)
{
  exit (-1);
}



////////////////////////////////////////////////////////////////////////////////
//                                  fine variabili e definizioni funzioni per il pkcs11
////////////////////////////////////////////////////////////////////////////////



int
BioMatch()
{

                int attempt = 0;
                CK_C_GetFunctionList   pGFL  = 0;
                CK_RV           rv;
                long result_match = 0;
                unsigned long slot_count = 100;
                CK_SLOT_ID slots[100];

                unsigned char *TokenTemplate = NULL;
                int ret;
                float qual;

                static CK_CHAR user_object[] = { "Template" };

                CK_OBJECT_HANDLE h;
                CK_ULONG objcount;

                CK_ATTRIBUTE user[] =
                {
                                { CKA_LABEL,user_object,sizeof (user_object) },
                };
```

```
// Load the library
HINSTANCE hLib = LoadLibrary(pkcs11_path);
if (hLib == NULL)
{
                leave ("Cannot load PKCS 11 DLL.");
}

// Find the entry point.
pGFL = (CK_C_GetFunctionList) GetProcAddress(hLib, "C_GetFunctionList");
if (pGFL == NULL)
{
                leave ("Cannot find GetFunctionList().");
}

rv = pGFL(&m_pFunctionList);
if(rv != CKR_OK)
{
                leave ("Can't get function list. \n");
}

// Initialize to PKCS #11 library
//
rv = m_pFunctionList->C_Initialize (0);
if (CKR_OK != rv )//m_pFunctionList->C_Initialize (0))
{
                leave ("C_Initialize failed...\n");
}

fprintf (log,"Init eToken with basic data. \n");

// get all the occupied slots
//
if (CKR_OK != m_pFunctionList->C_GetSlotList (TRUE, slots, &slot_count))
{
                leave ("C_GetSlotList failed...\n");
}

if (slot_count < 1)
{
                leave ("No eToken is available.\n");
}

// open a read/write session on the eToken so we can write
// information to it
//

if (CKR_OK != m_pFunctionList->C_OpenSession (slots[0],(CKF_SERIAL_SESSION |
CKF_RW_SESSION), 0, 0, &session))
{
                leave ("C_OpenSession failed...\n");
}

rv = m_pFunctionList->C_FindObjectsInit(session, user, 1);
if (rv != CKR_OK)
{
                leave ("C_FindObjectsInit errore.\n");
}

//looking for the template
//while (1)
{
		CK_ATTRIBUTE templat[] =
		{
			{ CKA_VALUE,NULL_PTR, 0 },
		};

		rv = m_pFunctionList->C_FindObjects(session, &h, 1, &objcount);
		if (rv != CKR_OK)
		{
			leave ("C_FindObjects errore.\n");
		}
		if (objcount == 0)
		{
			leave ("Template non e' memorizzato sul token.\n");
		}
		fprintf(log,"Template trovato!!\n");

		//copying the value into a buffer
		rv = m_pFunctionList->C_GetAttributeValue(session, h, templat, 1);
		if (rv != CKR_OK)
		{
			leave ("C_GetAttributeValue 1 errore.\n");
		}

		TokenTemplate = malloc(templat[0].ulValueLen);

		{
			CK_ATTRIBUTE templat_pieno[] =
			{
				{ CKA_VALUE,TokenTemplate, templat[0].ulValueLen
				},
			};

			rv = m_pFunctionList->C_GetAttributeValue(session, h, templat_pieno,
								1);
			if (rv != CKR_OK)
			{
				leave ("C_GetAttributeValue 2 errore.\n");
			}
		}

		rv = m_pFunctionList->C_FindObjectsFinal(session);
		if (rv != CKR_OK)
		{
			leave ("C_FindObjectsFinal errore.\n");
		}
	}

	// logout of the eToken
	m_pFunctionList->C_Logout (session);

	// close PKCS #11 library
	m_pFunctionList->C_Finalize(0);

	//now getting a live-scan sample and making a matching-test:
	//no more than three matching-trials: if the third one fails the program exits

	ret = CBK_Init();
	if (ret != CBK_OK)
	{
		leave("CBK_Init\n");
		return -1;
	}
```

```
		//open session with the fingerprint reader

		//load the template in the scanner memory
		do{		//in this block we have a triple matching-test

				ret = CBK_LoadModelFromMemory (TokenTemplate);
				if (ret != CBK_OK)
				{
						leave("CBK_LoadModelFromMemory\n");
				}

				//if this is not the first test we notify the user the previous mismatching event
				if (attempt != 0)
				{
						MessageBox(NULL, "Please Try Again", "Biomatch error",
						MB_ICONWARNING);
				}
				do {
						ret = CBK_AcquireFingerprintOffLine(&qual);

						if (ret != CBK_OK && ret != CBK_Cancel)
						{
								fprintf(log, "CBK_acquire error %d", ret);
								leave("CBK_AcquireFingerprintOffLine\n");
						}
				}
				while (ret == CBK_Cancel);


				ret = CBK_Matching(2, &result_match);
				if (ret != CBK_OK)
				{
						fprintf(log, "CBK_Matching error %d", ret);
						leave("CBK_Matching\n");
				}

		}while (++attempt < 3  &&  result_match != 1); //exit after three tests or a matching one

		//free the template buffer
		free(TokenTemplate);
		TokenTemplate = NULL;

		//close the scanner session
		ret = CBK_End();
		if (ret != CBK_OK)
		{
				leave("errore fx3scanner");
				return -1;
		}

		return result_match;
}
```

4 References

[1] Kohl & Neuman - RFC 1510

[2] Netscape© , http://developer.netscape.com/docs/manuals/security/sslin/contents.htm

[3] Franks et al. - RFC2617

[4] Richard Duncan - An Overview of Different Authentication Methods and Protocols

[5] Many authors – SANS Institute information security reading room: authentication

[6] Simo Huopio – Biometric identification

[7] E. Cuijpers – A design for a safe internet communication channel with the help of smart cards

[8] Kurt Seifried - WWW Authentication

[9] Freier et al. - The SSL protocol version 3.0

[10] Richard E. Smith – "Authentication (from passwords to public keys)"; Addison-Wesley, October 2001

[11] RSA Security Laboratories on-line CryptoFAQ review

Annex A

Secure Socket Layer [SSL]: Granting a secure communication channel with Netscape© protocol.

Secure Socket Layer protocol is almost a standard de facto for authenticated and encrypted communication on the Web. It represents the core of the IETF's Transport Layer Security protocol (acronym TLS): a middlewear layer that operates between transport layer and the OSI model's upper ones . Basically SSL uses public and symmetric key cryptography to develop relevant capabilities concerning network communication: server authentication, client authentication, encrypted connection. SSL achieves that both the client and the server share symmetric keys, so that they can exchange messages in a trusted channel, using public cryptography techniques and it represents one of the best examples of rigid security protocol.

Pretending to be as flexible as possible the SSL protocol consists of many cipher suites (DES, DSA, KEA, MD5, RC2, RC4, RSA, SHA-1, SKIPJACK, Triple-DES), and of two sub-protocols, the SSL record and the SSL handshake ones. That is because not all SSL versions support all the same ciphers.

The primary goal of the SSL Protocol is to provide privacy and reliability between two communicating applications. The protocol is composed of two layers. At the lowest level, layered on top of some reliable transport protocol (e.g., TCP), is the SSL Record Protocol. The SSL Record Protocol is used for encapsulation of various higher level protocols. One such encapsulated protocol, the SSL Handshake Protocol, allows the server and client to authenticate each other and to negotiate an encryption algorithm and cryptographic keys before the application protocol transmits or receives its first byte of data. One advantage of SSL is that it is application-protocol independent in fact a higher level protocol can lay on top of the SSL Protocol transparently. The SSL protocol provides rigid connection security that has three basic properties:

- The connection is private. Encryption is used after an initial handshake to define a secret key. Symmetric cryptography is used for data encryption (i.e. DES, RC4);
- The peer's identity can be authenticated using asymmetric, or public key, cryptography (i.e. RSA, DSS, etc.);
- The connection is reliable. Message transport includes a message integrity check using a keyed MAC. Secure hash functions (i.e., SHA, MD5, etc.) are used for MAC computations.

The SSL Handshake Protocol.

SSL takes messages to be transmitted, fragments the data into manageable blocks, optionally compresses the data, applies a MAC, encrypts, and transmits the result. Received data is decrypted, verified, decompressed, and reassembled, then delivered to higher level clients. Let's focus on the Handshake protocol, operating on top of the SSL Record Layer, that is the one providing principals' authentication. An SSL session is stateful. It is the responsibility of the SSL Handshake protocol to coordinate the states of the client and server, thereby allowing the protocol state machines of each to operate consistently, despite the fact that the state is not exactly parallel. An SSL session may include multiple secure connections; in addition, parties may have multiple simultaneous sessions. When a SSL client and server first start communicating, they agree on a protocol version, select cryptographic algorithms, optionally authenticate each other, and use public-key encryption techniques to generate shared secrets. These processes are performed in the handshake protocol, which can be summarized as follows: Following the hello messages, the server will send its certificate, if it is to be authenticated.

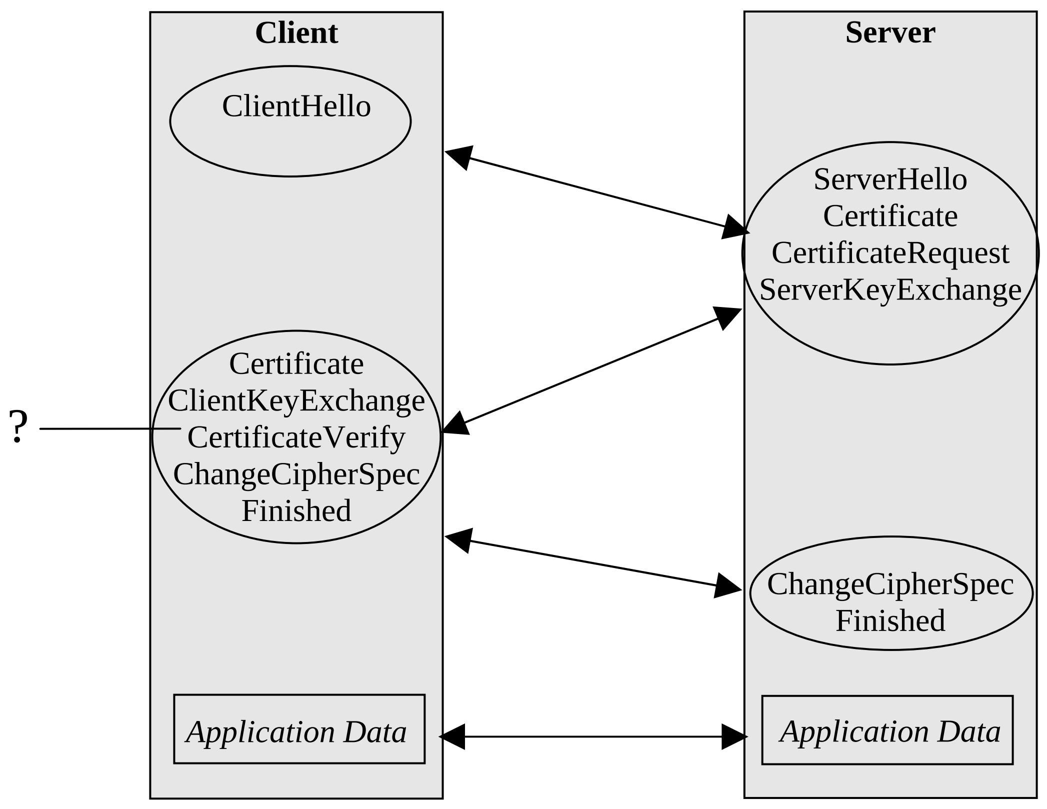
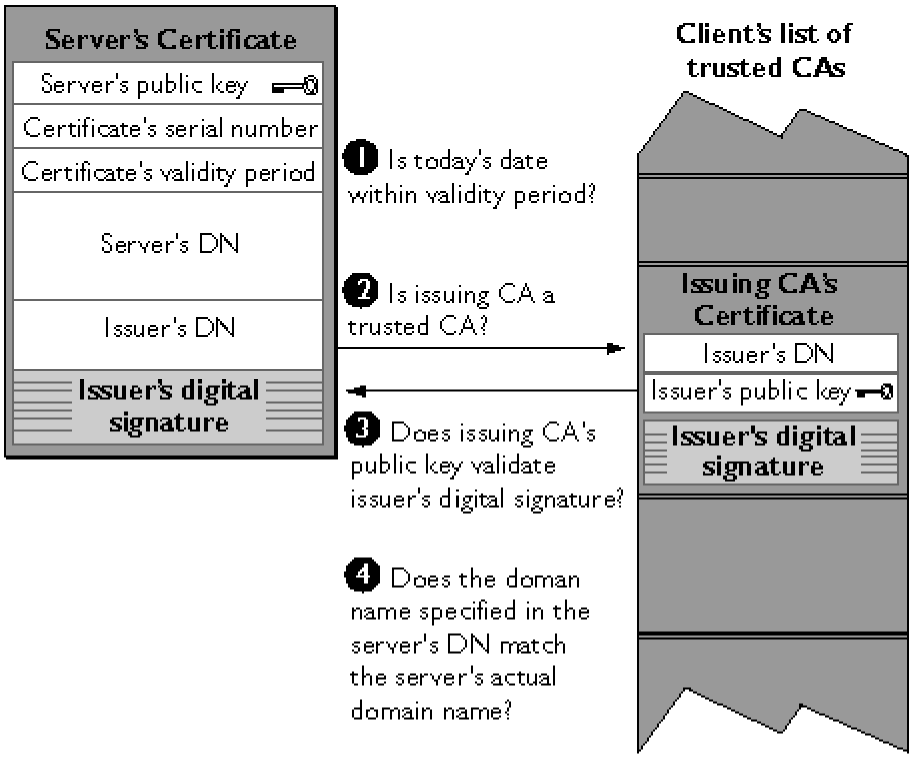


figure 10, the SSL Handshake protocol

- The client sends a client hello message (to which the server must respond with a server hello message, or else a fatal error will occur and the connection will fail). The client hello and server hello are used to establish security enhancement capabilities between client and server. The client hello (and server hello) establish the following attributes: Protocol Version, Session ID, Cipher Suite, and Compression Method. Additionally, two random values are generated and exchanged: ClientHello.random and ServerHello.random.

- The server sends back a server hello message (that it's equal to the client's one) and, if requested to authenticate, its own certificate. Nevertheless, if necessary, it requests the client's certificate; Additionally, a server key exchange message could be sent, if it is required (e.g. if their server has no certificate, or if its certificate is for signing only). If the server is authenticated, it may request a certificate from the client, if that is appropriate to the cipher suite selected. Now the server will send the server hello done message, indicating that the hello-message phase of the handshake is complete. The server will then wait for a client response.

- The client tries to authenticate the server through the scheme in picture A.2; the Client needs a positive answer to all four question in figure A.2. at step2 the Client could even authenticate a Server whose certificate is guaranteed throw a certificate chain ending with a trusted CA; at step 4 it takes care of any possible Man-In-The-Middle attack;

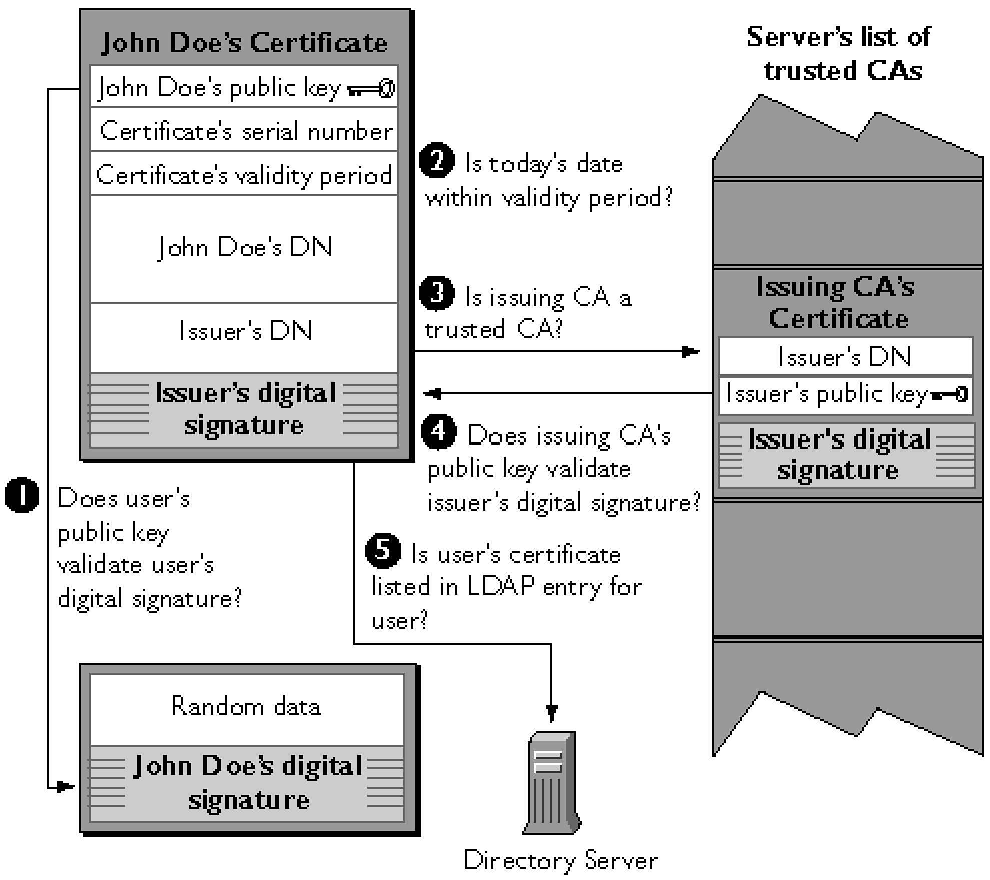

Server's Certificate
Server's public key
Certificate's serial number
Certificate's validity period
Server's DN
Issuer's DN
Issuer's digital signature
1 Is today's date within validity period?
2 Is issuing CA a trusted CA?
3 Does issuing CA's public key validate issuer's digital signature?
4 Does the doman name specified in the server's DN match the server's actual domain name?
Client's list of trusted CAs
Issuing CA's Certificate
Issuer's DN
Issuer's public key
Issuer's digital signature

Netscape© ,http://developer.netscape.com/docs/manuals/security/sslin/contents.htm

- After all previous checks has been successfully performed the client authenticates the server and sends it:

  1. If the server has sent a certificate request Message, the client must send either the certificate message or a no_certificate alert. The client key exchange message is now sent, and the content of that message will depend on the public key algorithm selected between the client hello and the server hello. If the client has sent a certificate with signing ability, a digitally-signed certificate verify message is sent to explicitly verify the certificate;
  2. a premaster secret generated according to the cipher suite selected for the communication encrypted with server's public key;
  3. optionally (to perform client's authentication) a piece of random data, unique to this handshake and known by both the principals, digitally signed by the client, as well as its certificate;

- If the server requests client's authentication it attempts throw a similar scheme as in server's authentication, to certainly identify the client (see picture A.3);


John Doe's Certificate
John Doe's public key
Certificate's serial number
Certificate's validity period
John Doe's DN
Issuer's DN
Issuer's digital signature
Server's list of trusted CAs
2 Is today's date within validity period?
3 Is issuing CA a trusted CA?
Issuing CA's Certificate
Issuer's DN
Issuer's public key
Issuer's digital signature
4 Does issuing CA's public key validate issuer's digital signature?
1 Does user's public key validate user's digital signature?
5 Is user's certificate listed in LDAP entry for user?
Random data
John Doe's digital signature
Directory Server

Netscape© ,http://developer.netscape.com/docs/manuals/security/sslin/contents.htm

In steps 1-4 the server behaves exactly as the client does but in step 5 it optionally checks whether the certificate claimed by the client, even if validated by a trusted CA, is the same as the one the server once stored in the LDAP for future checks or if that certificate has been revoked. If this control ends the client is authenticated by the server;

- After authentication the Server could eventually proceed through authorization process according client's rights;
- The server decrypts the pre-master secret and then generates the master secret.
- On the client-side the same steps are performed to obtain the master secret.
- Master secret is used by both the principals to elaborate the symmetric session keys use to encrypt, decrypt and verify the integrity of any message further exchanged.
- The client sends the server a message reporting that from now on all the messages will be encrypted with the session key and notifies the server that client handshake has come to an end (even this message is encrypted)
- The same takes place on server-side.
- SSL handshake is now complete and SSL session has begun: the principals will now communicate using the session key to validate and encrypt data they send each other.

Take care and do not rely on SSL blindly: in late 2000 has been demonstrated that TLS (and SSL v3.0) could suffer a special kind of man in the middle attack but if the packages are well configured the workstation will notify the user this occurring danger, using special messages, avoiding him to dive into the hackers's web.

Annex B

Nonce, digests and hash functions: a brief explanation.

A **nonce** is a randomly generated value used by an host during a communication. The utility of using random numbers resides in the fact that previously, when workstation computing power was not so high as is today, to break a cipher or a cryptographic function you would have used special hardware machines wiring some algorithms used to break other algorithms. Due to (relative) low power computing CPUs on the market the introduction in the algorithms of some mechanism that could not be reproduced via hardware would had significantly slowed down the breaking process of the ciphers. Since it's impossible to hardware simulate a random value generator (no hardware machine could produce stochastic random values) hackers did have to purport their tries via software and that resulted much slower than via a hardware simulator. Because of this the use of nonce appeared as a good mechanism to prevent passwords to be always the same and to avoid hackers discovering the actual nonce(password) during just one communication session. Actually increasing CPU possibilities (and software evolution towards more sophisticated forms of security) had led to completely changed sceneries even if for some years the common answer to increasing processing power has been a proportional increase of the bit space generated by the nonce digit quantities (more digits you submit, the greater the bit-space of the nonce become, the more difficult breaking the cipher will get).

A **hash** function H is a transformation that takes an input m and returns a fixed-size string, which is called the hash value h (that is, h = H(m)). [11] Hash functions with just this property have a variety of general computational uses, but if employed in security and cryptography, the hash functions are usually chosen to have some additional properties.
The basic requirements for a cryptographic hash function are as follows:

1. The input can be of any length.
2. The output has a fixed length.
3. H(x) is relatively easy to compute for any given x.
4. H(x) is one-way.
5. H(x) is collision-free.

A hash function H is said to be one-way if it is hard to invert, where "hard to invert" means that given a hash value h, it is computationally infeasible to find some input x such that H(x) = h. If, given a message x, it is computationally infeasible to find a message y not equal to x such that H(x) = H(y), then H is said to be a *weakly collision-free* hash function. A *strongly collision-free* hash function H is one for which it is computationally infeasible to find any two messages x and y such that H(x) = H(y). An **hash** function is a (almost) one-way cryptographic function that produces from a stream of bit code another (almost) unique shorter stream of bit code called a **digest**. In fact, the job of the hash function is to take a message of arbitrary length and shrink it down to a fixed length.

The use of the almost preposition is necessary because an hash function is a mathematic elaboration of some data and even if it could be has hardly as possible to discover the cipher there's always a time space, as long as you like, in which using a quite wide review of hashed password, you could infer the function's operations: <u>this does not represent a big problem if this time is estimated to be longer than the lifecycle of the information hashed in the digest!</u> Even the digest produced with these functions should all be different one from another but being these values represented on shorter bit sequences it is possible, though unlikely, that two different bit codes result in the same hash. In fact even a single bit changed in a sequence cause deep changes in the digest but this could not always address different sources as some hash functions, for example, produce a 16 bit digest from a 64 bit entry. When two messages hash to the same message digest it is called a collision; the collision-free properties of hash functions are a necessary security requirement for most digital signature schemes. A hash function is secure if it is very time consuming, if at all possible, to figure out the original message given its digest. However, there is an attack called the **birthday attack** that relies on the fact that it is easier to find two messages that hash to the same value than to find a message that hashes to a particular value. Its name arises from the fact that for a group of 23 or more people the probability that two or more people share the same birthday is better than 50%.

Annex C

The MIT Project Athena's protocol: Kerberos

The creators of Kerberos framework, Miller and Neumann, started to design this protocol according to Needham-Schroeder one but integrating even Denning and Sacco's work. Kerberos emphasizes strongly on user authentication addressing even the specific workstation that a user could operate with. Many versions of this protocol has been released by the MIT: version 5 is the latest update but more than an open one is available from MIT's Web-sites. Kerberos address directly the problem of user authentication: although it's possible to use Kerberos authentication server to generate any single service tickets this is not the best solution as the *master key* (the secret shared between a host and the Kerberos KDC) is required to be typed in every time that a user interacts with the Authentication Server [AS]. But the majority of people use several services when working on its workstation and it would be quite some dangerous more than quite boring typing in several times a day the same password to contact the AS. Certainly it would be extremely dangerous to store that secret somewhere on the workstation as well. To eliminate these problems Kerberos uses the master key for as a short time as possible to achieve another set of shared secret keys that could last for all one session providing the workstation the meanings to communicate with another Kerberos server (the *Ticket Granting Service* [TGS]) which is able to grant access tickets for many services without utterly typing in the master key again. So the main issue of Kerberos authentication protocols is represented by its KDC server that provides a particular separated multifunctional structure. Within Kerberos there's an *Authentication Server* [AS] that works similarly than Needham-Schroeder's KDC but providing tickets to be used not with a particular service but with another Kerberos server, the Ticket Granting Service [TGS], that is the one deputed to release tickets for some specific services. Basically another level of indirection has been introduced from Project Athena's team to achieve user friendliness as well as stronger authentication guarantees: the specifics of this protocol are explained in the following lines. When a user wants to authenticate himself for a specific service, or for more than only one, he submits to the AS a *KRB_AS_REQ message* that reports:

- user identity information
- the name of the desired service's server
- a nonce (according Needham-Schroeder challenge response portion)
- a validity period individuating a time window in which the *ticket-granting-ticket* [TGT] that the AS is going to send back will be active
- A workstation identifier to lately let Kerberos verify if a determined workstation is authorized to use a certain ticket

The AS replies with a *KRB_AS_REP message* containing two different portions: the first one is encrypted with user's master key and contains:

- The TGS's name to contact to access the desired service
- The same transmitted validity period
- The decremented nonce
- The *ticketing key* that will allow the user to communicate with the TGS

Another portion of the message is represented by the Ticket-Granting-Ticket [TGT] that is a special ticket that the user will present to a second server, the TGS, using the ticketing key he retrieved in the *KRB_AS_REP message*, to achieve the last and effective ticket that will grant him access to the required service. This TGT, encrypted with the TGS's master key by the AS, contains:

- The user name of the user that will contact the TGS
- The TGS's name that the user will ask for
- The validity period to verify if the received TGT is still (or yet) active
- The ticketing key with which authenticate the user.
- A workstation identifier to verify if the TGT comes from an authorized machine or not.

Once the user has received this TGT he should authenticate himself to the Ticket Granting Service and this is achieved with a *KRB_TGS_REQ message* sent to the TGS. This message is written, starting from the *KRB_AS_REP message*, according simple rules and contains:

- The user's *authenticator*, that is the couple of his username and a timestamp encrypted with the ticketing key
- The name of the server he wants to contact
- A validity period to be set for this new effective ticket
- A nonce
- At last he includes the TGT as received from the AS.

As soon as the TGS receives the user's request verifies the authenticator's authenticity, checks if the request is coming from an authorized workstation and the validity period of the TGT against the current time and the validity period the user's workstation has requested for this new ticket. If all this checks has positive ending the TGS generates a *session key* to be used between the user and the requested service's server. So the TGS construct a *KRB_TGS_REP message* consisting of two main parts: a ticket to be used with the requested server and some validation data for the user. The ticket for the server, encrypted with server's master key, consists of:

- User name of the user
- server name
- A validity period for this ticket
- A session key to authenticate the user
- A workstation identifier to check out if the request comes from an authorized machine

The other part of the message, encrypted with the ticketing key, consists of:

- The name of the server to contact
- The validity period of the relative ticket
- A session key to authenticate to the server
- A workstation identifier
- The decremented nonce.

According to this protocol, typing in one per session his master key, a user can access a quantity of services in a secure way but to be worked with, kerberos requires a good clock synchronization between the hosts and its KDC (and this is not a simple or low cost problem) and moreover all the services that are presumed to be accessed with Kerberos must be adapted to work with this protocol and behave according its requirement, in one word they must be *Kerberized*. Of course in the past years, as Kerberos is today a quite old technology, the MIT has provided many releases of kerberized products to work with but this shall not lead us to some misunderstandings: Kerberos is still today a working protocol, even if with some changes, and even Windows2000 adopted a modified version of it for some specific purposes.

Acknowledgements

I would like to thank those who granted me the opportunity to spend my very first professional experience in such a hi-tech context next to such professionals. For the University of Pisa I must mention my tutor Prof. Graziano Frosini and D.ssa Angela Dini, the person who much cared and worked for this stage to take place. At the I.i.t. I met people with great ideas and wysdom for their work who has spent some of their time after me: I would like to thank D.ssa Anna Vaccarelli, the Si’s team leader, for her kindness and because she definetely decided for this stage to take place; I would like to thank Dott. Luca Bechelli and Dott. Stefano Frassi as they are the ones that spent with me much time patiently trying to answer to all the questions that a non professional has. I have to thank all the other members of the Si Team but above all I must mention the person who has waited, cared for and looked after me during those months, writing me e-mails from all over the world and helping me having a look on the future of systems security: Dott. Stefano Bistarelli, my tutor at the I.i.t. . I’ve arrived as a non professional and I’ve leaved as a non professional but if one day I would be able to become such I know that I have to wholeheartedly thank you!