Co-funded by the Erasmus+ Programme of the European Union

InCyT Training Module

Information Security for Employees **Unit 2 Week 3** **Unit name: Introduction to Cryptography**	
Learning Activities	❑ **Webinar** ❑ **Exercises** ❑ **Workshop** ❑ **Presentation** ❑ **Other**
Learning Responsible	Claudio Fornaro
Learning Activity Goals	1. Cryptography 2. Cryptoanalysis 3. One time pad algorithm 4. Symmetric algorithms 5. Asymmetric algorithms 6. Hash functions 7. Signatures & public key infrastructures 8. Cryptographic Tokens 9. Challenge-response mechanism
Skills to be Gained	• Basic knowledge about cryptography algorithms and infrastructures Technical Skill 2 - TS2
Duration of Learning activity	1 week
Learning requirements	***What is needed?*** • No previous specific knowledge is required

Co-funded by the
Erasmus+ Programme
of the European Union

Brief information about the module

Cryptography pertains to methodologies for hiding information from disclosure, both during a communication and for archival purposes. Its objectives are achieved with the design and implementation of protocols that prevent an unauthorized third party from retrieving the information. In a secure communication, the adversary third party cannot access the information transmitted, in the case of archiving the third party cannot trace the content of what has been archived.

In Cryptography, the third party is always a malicious entity that aims to retrieve information that it must not have access to, whether for privacy, trade secret or national security purposes. The data confidentiality and integrity, the authentication of the parties involved and the non-repudiation of what the data source has transmitted or stored are the fundamental principles of modern cryptography.

After an introductory part, where the fundamental concepts are described, we present a progression of basic cryptographic techniques, starting from the *One Time Pad Algorithm* and then building up for considering the two cryptographic families of algorithms: Symmetric Algorithms and Asymmetric (or Public Key) Algorithms with a comparison of them. The role of the Hash Functions is also discussed to introduce Digital Signatures. A description of the key concept of Public Key Infrastructures is presented in depth with some details on the standard mechanisms and formats. To conclude, some information about Cryptographic Tokens and the Challenge-Response Mechanism is provided.

Content of the Learning Material

Introduction

Cryptography is the mechanism by which the content of a message is hidden from other entities. In other words encryption deals with encrypting messages, i.e. taking a plaintext, using an algorithm and generally a key, making the content of the message unintelligible.

Cryptanalysis studies how to decrypt the message without knowing the key. In an attack, cryptanalysis consists in trying to trace the content of the message without having the key. Cryptographers and cryptanalysts are actually the same people: who writes the cryptographic algorithms also analyzes them to find their weaknesses or validate their robustness. As always, when dealing with security, the idea of being able to deal with protection issues without knowing how attack techniques work is not very effective.

Information in messages is generally redundant, in a formal sense, messages generally use a much higher number of bits than that really necessary to carry the information. **The redundancy of information facilitates cryptanalysis**: redundancy in the information means that the messages can show a structure that makes it easier to trace the content of the message, as not all messages are equally likely. If we think of a text represented with normal ASCII characters in byte we know that the combinations of bits that can represent meaningful messages are relatively few compared to all possible combinations. The true messages that can be represented with a certain number of bytes are very few. This makes it possible to identify valid messages by discarding those that do not carry true information. All this vastly helps the activity of the cryptanalyst, so the purpose of the encryption algorithm is also to hide this redundancy to better hiding the information.

Example: Encoding *a sequence* of the four colors BLUE, GREEN, RED and YELLOW.

These can be coded:

- with 2 bits (00, 01, 10, 11)
- as 4 different uppercase letters
- encrypting the color names

In the first case, the minimum number of entities (bits) is used, by substituting each color name with the corresponding 2 bits will conceal the original sequence, but not the order as there is a correspondence between COLOR and the 2 bits. In the second case, as each letter uses 8 bits in the ASCII code, 6 bits are redundant for every color → 75% redundancy and we achieve the same grade of hiding as the 2 bits. We still cannot associate colors and values.

Consider the following very simple cipher, the so-called *Caesar cipher*, which does nothing but scroll the letters in the alphabet by a certain number of positions. For example, if we slide them by 3, all the 'A's become ‘D's, all the 'B's become 'E's, and so on. This is a very simple cipher; actually

attributed to Julius Caesar, but if the color names are encrypted with this method, the repeating pattern of their encoded letters are easily identified (the ASCII code for R followed by the ASCII code of E followed by the ASCII code of D are easy to recognize as an entity). So the sequence can be identified, and if the words are also known in this case just counting the number of letters is enough to identify the original words.

This simple example allows us to understand that cryptanalysis is not an analysis that is independent of the content of the message: a completely randomized message would be much more difficult to identify and analyze than a message that has a recognizable structure.

Some types of cryptographic attacks and algorithms rely on knowing part of the text. In some cases, knowing part of the text makes it possible to trace the key and then access the rest of the text. This is very significant when it comes to network communications, because in network communications the protocols at different levels usually have a certain amount of information that is in any case recognizable, e.g. in an IP packet we know that the header contains the sender address and the recipient address.

An encryption algorithm hides also redundant information, so applying a compression algorithm to an encrypted message does not bring much advantage, because it would reduce compression rate. If we want to use compression and encryption, we must first use compression and then encryption.

The One Time Pad Algorithm

The One Time Pad (OTP) is an algorithm, so to speak, "perfect". It is a very simple encryption algorithm, very simple to implement and has a characteristic that makes it particularly interesting: it is proven that without knowing the key it is impossible to go back to the original text. Now we look at it and then we discuss why, as we have such a good algorithm, we have to find others.

The EXCLUSIVE OR $\oplus$ (XOR) is a logical function where the result is True when the two operands are different, it operates on logical values but translated to binary digits, usually True=1 and False=0, its *truth table* is:

$0 \oplus 0 = 0$

$0 \oplus 1 = 1$

$1 \oplus 0 = 1$

$1 \oplus 1 = 0$

From this table it is clear that if $X \oplus Y \rightarrow Z$, then $Z \oplus Y \rightarrow X$ and $Z \oplus X \rightarrow Y$. The EXCLUSIVE OR is a

very simple function and also easy to implement as an electronic circuit.

The OTP mechanism is used to transfer a message from two parties. In the scientific literature it is common practice to identify the actors with personal names, they are usually called Alice (A) and Bob (B). So we will describe how Alice can securely transmit an *n*-bytes message (*PlainText*) to Bob.

The One Time Pad algorithm requires a setup phase:

- A random sequence of exactly *n* bytes is generated, this is the *One Time Pad* (OTP) or simply the *Key*;
- The OTP is securely shared between the two parts in some way (given in advance, sent in an already secured channel, etc.).

When both parties possess the key, they can use it to transmit messages. In case Alice wants to send a message to Bob:

- Alice calculates a *bit by bit* EXCLUSIVE OR of the *PlainText* with the *Key*, producing a *CypherText*:
  Encryption: PlainText ⊕ Key → CypherText
- The resulting *CypherText* is transmitted by Alice to Bob over a possibly insecure channel
- Bob calculates the *bit by bit* EXCLUSIVE OR of the *CypherText* and the *Key*, so recovering the original *PlainText*:
  Decryption: CypherText ⊕ Key → PlainText

Example

Let us suppose both Alice and Bob have already agreed upon a random OTP (*Key*).

Alice wants to send "HELLO" to Bob (in ASCII Code, spaces are here introduced for ease of reading):

	H	**E**	**L**	**L**	**O**
PlainText:	01001000	01000101	01001100	01001100	01001111
			⊕		
Key:	10101001	01110010	10110010	10000110	11000110
			=		
CypherText:	11100001	00110111	11111110	11001010	10001001

Bob receives the *CypherText* and applies the *Key*:

CypherText:	11100001	00110111	11111110	11001010	10001001
			⊕		
Key:	10101001	01110010	10110010	10000110	11000110
			=		
PlainText:	01001000	01000101	01001100	01001100	01001111
	H	**E**	**L**	**L**	**O**

The *PlainText* has been recovered.

As the *Key* is a random sequence, there is no way to say if a certain bit of the sequence is '0' or '1', as they are equally probable. Without the *Key*, given a *CypherText* bit, we have no chance of trying to guess what the original bit might be. This is true for single bits, for the whole message, for any fragment of our message.

In case a part of the *Key* is known, it is possible to recover the corresponding part in the *PlainText*, but without the rest of the *Key* there is no lack of information on the other bits of the *PlainText*. Therefore, the cryptanalysis mentioned before is not effective. In this case, to discover the remaining part of the original message, the remaining part of the *Key* is needed. All the information conveyed by the *Key* is needed in order to recover all the information of the text.

One Time Pad Problems

The strength of the OTP mechanism relies on the length of the *Key* being exactly equal to the length of the text and on using it just one time. The main drawback is that the transmission of the OTP has the same problems as the message.

In some cases this problem is easily solved, for example a diplomatic briefcase with a ribbon containing a very long key could be provided to an embassy and subsequently the messages that will be exchanged with that embassy can be encrypted by consuming the bits of the key gradually. These are very peculiar uses, but quite effective. It is clear that if we consider network communication, the real uses of the One Time Pad are very limited. There is a need for something more flexible, something that allows the exchange of a key that can be used for a very long time.

Let us consider each of the problems.

Offline transmission in advance

This is a general problem with keys: keys have to be agreed in some way by the parties. This need to agree on keys is the most important problem of using cryptography. The problem with cryptography is not so much algorithms and cryptanalysis, as well as key management: that is being sure that all and only the right subjects have access to the keys, then exchange them in advance, and be sure that the keys exchanged are the correct ones.

One time use

The One Time Pad is used only once, as the name suggests, this is an important aspect and limitation. If we tried to use the same key on two messages, we would weaken the security of the method.

Generating random numbers in a deterministic system is very difficult

General purposes computers use *pseudorandom* number generation where each number is computed from the previous one with some computation, this produces a sequence of numbers that repeat after a very long period. These are not true random numbers as a computer is a deterministic machine. In cryptography, however, this feature is deleterious. If it is possible to derive the next random number from a previous one, the protections offered by cryptographic algorithms or protocols become much less secure. Writing code that implements cryptographic algorithms is not trivial: there is a great deal of detail to watch out for, all the flaws that have been identified over the years by code developers that only the code developers in this specific area have learned to manage. This means that it is always a bad idea to develop and/or implement in-house cryptographic algorithms instead of using well-tested, widely used, extensively validated libraries.

Algorithms other than OTP

The other algorithms use fixed length keys. If we use an n bit key, we have 2^n different keys, if the value of n is big enough and you just apply a brute force attack (that is trying all the possible keys), it would require so many years to find the right one so that "probably" the content of the message is no more of interest. For example, with one of the fastest supercomputers available, checking a 256 bit key would require 3.67×10^{55} years. Clearly, using keys of sufficient length has some additional computational cost even for those who legitimately encrypt and decrypt, but the impact is limited.

There are Symmetric and Asymmetric algorithms.

Symmetric Algorithms

The most typical and old family of cryptographic algorithms, the ones we typically think of when we think about encrypting messages, are the ***symmetric algorithms***, called after the symmetry of the encryption and decryption operations. These algorithms use a single key that must be known to both the sender and the recipient; it is used to encrypt a message and then to decrypt it.

A simple symmetric schema could be the same as the OTP, as OTP is a symmetric algorithm, with the big difference that the *Key* has a fixed length (e.g. 256 bits) and it is repeated in some way, possibly after some complex change each time:

Encryption: *encryption_function*(PlainText, Key) → CypherText

Decryption: *decryption_function*(CypherText, Key) → PlainText

More in general, even if we use the same *Key* the function used to decrypt is not necessarily the same used to encrypt. For example, if we consider Caesar's cipher, it is clear that the function that deciphers the message slides the alphabet to the left and not to the right, but decryption uses the same key: number 3.

Some important symmetric key algorithms

DES (Data Encryption Standard)

This historical algorithm, now outdated, uses a 56-bit key that in the late 1970s (when the algorithm was developed) was absolutely adequate for many uses, because a brute force attack on a 56-bit key to get the message with the computing resources available at the time and in the contexts in which that algorithm was used, was impractical. 40 years later the DES algorithm would not last even a few minutes of a home computer. This is a very important aspect: the length of the key is chosen based on the *actual* available computing capabilities. A key that is considered robust enough now may no longer be that strong in 20 years. In general we tend to abound with the length of the key in order to ensure that there is some margin, both because computing capabilities increase, but also because cryptanalysis as a discipline is making progress and may be able to find weaknesses in the algorithm. This is quite an interesting aspect. Contrary to what one might think, normally a cryptographic algorithm does not fail suddenly, i.e. the day a flaw is discovered in an algorithm it instantly turns from secure to completely insecure and anyone can find the plaintext in a moment. What happens is that researchers analyze the algorithm and maybe found small weaknesses that can ease attacking it in certain cases. Then someone possibly founds some algorithm optimization to solve the problem. As time passes the algorithm is gradually weakened due to unsolvable problems, until it is no longer adequate for certain uses or at all. There is a progressive erosion of the defenses of cryptographic algorithms, and this progressive weakening is a more difficult problem to manage, because obviously we

cannot know what progress there will be in the future from the point of view of cryptanalysis.

AES (Advanced Encryption Standard, also known by its original name Rijndael)

AES was chosen based not only on robustness, but also on the efficiency of implementation in typical architectures. No secret algorithm was sought because a secret algorithm is not considered more robust than a non-secret one: on the contrary, the algorithm was intended to be analyzed by as many valuable cryptanalysts as possible to ensure that it was intrinsically robust. An algorithm that is safe but too slow, too expensive in terms of computing resources, would be not usable. This constraint was set to be as the most difficult for competitors to overcome, compared to that of robustness.

Asymmetric (or Public Key) Algorithms

Asymmetric algorithms do not use just one key, but two: one key is used to encrypt and the other to decrypt. Therefore, in the exchange of messages, sender and receiver do not use the same key. These keys are generated and managed in pairs: what a key (any of the pair) encrypts can be decrypted only by the other key of the pair.

The reason to use asymmetric algorithms derive from the problems symmetric cryptography has:

1. Confidential communications between 2 entities requires that a unique key be shared between them only, if the entities are n then $n\cdot(n-1)/2$ different keys are required to allow for confidential communication between each pair of the parties
2. Both sender and recipient know their shared key, so *non-repudiation* (repudiation is denying of being the author of a certain message) is not possible

The first problem is not about the number of keys that each subjects has to manage, the problem is *exchanging* them. Each subject must agree on a key with each of the other n-1 subjects, which is actually a complex operation; this makes symmetric cryptography impractical for many uses.

The second problem, the *non-repudiation*, lies in the fact that we often want to make sure that a certain message was actually sent by a certain subject. It is clear that if a subject receives a message from some other subject and the two subjects are the only ones that know the key used to encrypt it, each subject is sure that the other subject is the author of the message. Vice versa, since they only know the key, each subject knows that only the other subject can read the message. The problem appears when there is a need to prove this to a third party, for some kind of litigation, as this mechanism does not work: each party is able to create a message, encrypt it, and then say that the other party is the author. The *non-repudiation* problem is not solvable by using symmetric cryptography.

The asymmetric schema is similar to the symmetric one, but the keys used are the two that are

generated in pair:

Encryption: *encryption_function*(*PlainText*, ***Key1***) → *CypherText*

Decryption: *decryption_function*(*CypherText*, ***Key2***) → *PlainText*

The names ***Key1*** and ***Key2*** have been used instead of *EncryptionKey* and *DecryptionKey* because the two keys are mathematically interchangeable. The subject that generates the keys generally keeps one private/secret (called the *private key*) and makes the other public (called the *public key*), hence the name of the algorithm class that is also called the *public key algorithm*. More on the term *public* will be discussed later.

The function used to encrypt is not necessarily the same as the one used to decrypt. The important thing is that together they represent the cryptographic algorithm, one part relies on one key and the other part relies on the other key.

The two keys have such mathematical properties that discovering one key from the other is quite complex and requires an enormous amount of time. In particular, even knowing the public key, the *PlainText* and the *CypherText*, reconstructing the private key would require an unreasonable amount of time.

Asymmetric Algorithm Uses

Suppose (details will be given later) that the *public key* is publicly available and its owner is the sole to know the corresponding private key. Being the public key available to anyone, anyone can use it to encrypt a message, but only the owner of the corresponding private key will be able to decrypt the message, this provides **confidentiality** of the message transmission.

So we have moved from a situation where one subject needed *n* different keys to confidentially communicate with *n* other subjects to a situation where just one key (the receiver's public key) is needed for everyone to send a confidential message to a subject.

In order to allow *n* subjects to communicate to each other, each subject must have a pair of keys, so the total number of keys is $2 \cdot n$. The interesting point is not so much that the keys have decreased in number; but that being public keys these can be communicated in a much more simple way. If someone else gets to know those public keys, it is not a problem at all. Therefore, these keys can be really published, made available to everyone and everyone can use them without weakening the protection of the messages.

If the subject is the only owner of a private (secret) key, whose public counterpart is publicly available, anyone can verify (decrypt) messages encrypted with the private key of the subject. Then if it is possible to use the public key of a subject to decrypt a message, it means that only the subject that has the corresponding private key could have encrypted the message, this provides

for the **non-repudiation** of the message.

A problem is still to be solved: ensuring that a public key is really associated to the legitimate subject. Here a *super partes* certification method must be provided, this is provided by a *Public Key Infrastucture* (PKI), discussed later.

As a side note, asymmetric algorithms can also be used as a *challenge-response algorithm*. In order for subject A to *authenticate* (prove that a subject is genuinely who claims to be) subject B, A sends (in the clear) a random number to B; then B encrypts it with its private key and sends it back to A. If A is able to decrypt the message with the public key of B, A is sure that the other party is B, as it is the only one that can encrypt something the public key of B can decrypt.

Advantages and Disadvantages

The total number of keys is significantly decreased, which decreases the complexity of guaranteeing the correct management of all the keys (and only the owner knows the secret key). Since public keys are made publicly available, there is no need for specific agreements between different subjects. This use of public keys is, for example, very important in e-commerce, because e-commerce cannot require that there is an a priori agreement of the seller with customers.

To obtain this link between the two keys and, at the same time, guarantee that one cannot be computed from the other, and that what is encrypted with one is decrypted with the other, make asymmetric algorithms much more complex in terms of mathematical problems than symmetric algorithms. These mathematical problems essentially require slower algorithms than symmetric algorithms and the use of longer keys, at least with the currently used algorithms. Therefore, if on one hand there is a greater flexibility, on the other they are more onerous in terms of computing resources.

Prominent algorithm: RSA *(from the initials of Ronald **R**ivest, Adi **S**hamir, and Leonard **A**dleman)*
This algorithm is based on the complexity of prime factorization: it is easy to multiply two primes and obtain the result, but it is complex/intractable to find the two primes from the result. Prime factorization is a difficult problem. We also know that it is not easy to find very high prime numbers, as there is no rule that allows finding out prime numbers other than to try to factor them and discover if they are prime. This means that working on very large numbers, finding out which are the two prime numbers from which a given number is derived is a problem that requires significant computing resources. On the other hand, legitimate operations can be done with limited computing resources.

Prominent algorithm: Elliptic Cryptography *(ECC - Elliptic Curve Cryptography)*

The “difficult problem” in this case is the computation of the discrete logarithm (in a finite field). A

"finite field" means that integers values are in a limited set (very rough definition).

The ECC problem is more complex than prime factorization, so you get the same security with shorter keys. ECC algorithms are faster than RSA.

Comparison of Symmetric and Asymmetric Algorithms

Asymmetric algorithms exploit mathematical problems other than those of symmetric cryptography. This means that the lengths of the keys are not comparable to those of symmetric cryptography. The problem with symmetric encryption is just that the key is long enough to prevent trying all possible numbers that could match a key, so with a 128-bit key there are 2^{128} (that is $3.4 \cdot 10^{38}$) possible keys. Today's symmetrical algorithms typically use 128-, 256- and sometimes
512-bit keys.

If we consider RSA, for example, which exploits prime numbers, it is clear that not all keys are possible, because they are related to the presence of prime numbers. Not all numbers are prime, so not all numbers are possible keys. Very simplifying, to have the same algorithm robustness (therefore the same difficulty from the computational point of view) of a 128-bit symmetric algorithm, the RSA algorithm needs a key of the order of 1024 bits. Today's asymmetrical algorithms typically use 1024, 2048 and 4096 bit keys.

Hash Functions

Hash functions calculate a short fixed length summary of a text of arbitrary length; this summary is called a *digest*. A hash function must be simple and fast to calculate.

Security properties

Hash functions are "one- way functions" in the sense that it is very complex and slow from the digest to recover a *possible* text that generates it, but this text is hardly useful. This is a consequence of the fact that the original text digest is (and should be) larger than the digest, so virtually infinite sequences of bytes can result in a specific digest. Even if a text is found to have the given digest, it is extremely improbable it is the original text or even makes sense. Building two meaningful messages with the same digest or building a meaningful messages out of a given digest is extremely difficult, any modification of a message, even a single bit, will give a completely different digest (about 50% of its bits are different).

Integrity check

Given a message M, its digest D is computed by means of a hash function H: D = H(M)

A secure transmission of digest D enables to ensure that message M is correct: the receiver of message M calculates the digest on it and compares it with the one (D) received; if they are the identical, the message has not been modified. It is essential that the digest is transmitted in a

secure way, otherwise there is no protection at all.

This mechanism is also used as a verification of a downloaded software instead of other less powerful (but faster) algorithms known as CRCs (Cyclic Redundancy Checks), common in network protocols.

Some Important Hash Algorithms

The following are two of the most known hash algorithms.

MD5 (Message Digest #5, 1991)

MD5 requires a low computational power (compared to the majority of the other hash algorithms) and produces a 128-bit hash. Over time it has been gradually weakened to be completely unreliable for security purposes (at present it is possible to find two distinct messages with the same MD5 hash – which is called a *collision* – in seconds), anyway it is still useful in non-cryptographic applications, for example to discover transmission errors instead of the more common (and weak, but faster) CRC-32.

SHA-3 (Secure Hash Algorithm #3, 2015)

SHA are actually a family of secure hash algorithms, SHA-3 is the last version and the result of a competition among cryptographers. The algorithm can be tuned balancing security with speed; it produces hash values of 224, 256, 334 and 512 bits. It is 2-3 times slower than MD5 but no weakness have been found so far.

Signatures (Digital Signatures)

Digital signatures, called electronic signature in the legislation of some countries, take a message, calculate a hash, and encrypt it (just the hash) with the secret key of an asymmetric algorithm. This hash signed with the secret key is called a *signature*. A digital signature then does not encrypt the original message, just allow to guarantee that the corresponding text has not been modified, because otherwise the digest would change. It also guarantees that the origin can be verified (*non repudiation*), as it was actually encrypted with the private key of an asymmetric algorithm. This means that anyone can verify that the text is intact and has actually been generated by a certain subject.

Similar to digital signatures are **Message Authentication Codes** (MACs). They also aim to ensure the integrity and authenticity of a message. The digest is computed hashing the text to be protected and a *symmetric algorithm* key (there is no assurance of non-repudiation, so it is suitable only when this characteristic is not needed). One of the most known algorithm is HMAC: Keyed-hash MAC.

Public Key Infrastructures

The confidentiality, integrity and authenticity requirements of asymmetric encryption and digital signature operations require establishing a relationship of trust between the parties involved. As often the parties had no contact before, a trusted (by them) third party must provide the needed guarantees, in other words this third party must assure the authenticity of the owner of a public key.

A *Certificate Authority* (CA) is a part of a hierarchical trust model that guarantees for the user's identity. It must be recognized as "trusted" by all entities involved in the communication.

In order to assure about the authenticity of a subject, a CA signs a *digital certificate* that links the subject to its public key (it is conceptually similar to an Identity document of a person). The signature applied by the CA is a digital signature, so it is computed by using CA's private key. Of course, the digital certificate must be validated by using CA's public key.

A *Public Key Infrastructure* (PKI) is a hierarchical structure of CAs, where every CA has its public key signed by the previous level CA (which uses its private key for this purpose), the process goes up to the *Root Certificate Authority* that self-sign a certificate holding its public key by using its private key.

A hierarchical model allows building hierarchies of any depth. For example, within a Country we could have the municipal, …, regional certification authorities, up to a national root CA.

The root CA certificate must be made public in many locations and ways in order to make it impossible to substitute it with a forged one. As an example, the Operating Systems themselves are shipped with a set of root and trusted intermediate CAs. Figure 1 shows an excerpt of the Root Certificates list from a MS Windows 10 operating system.

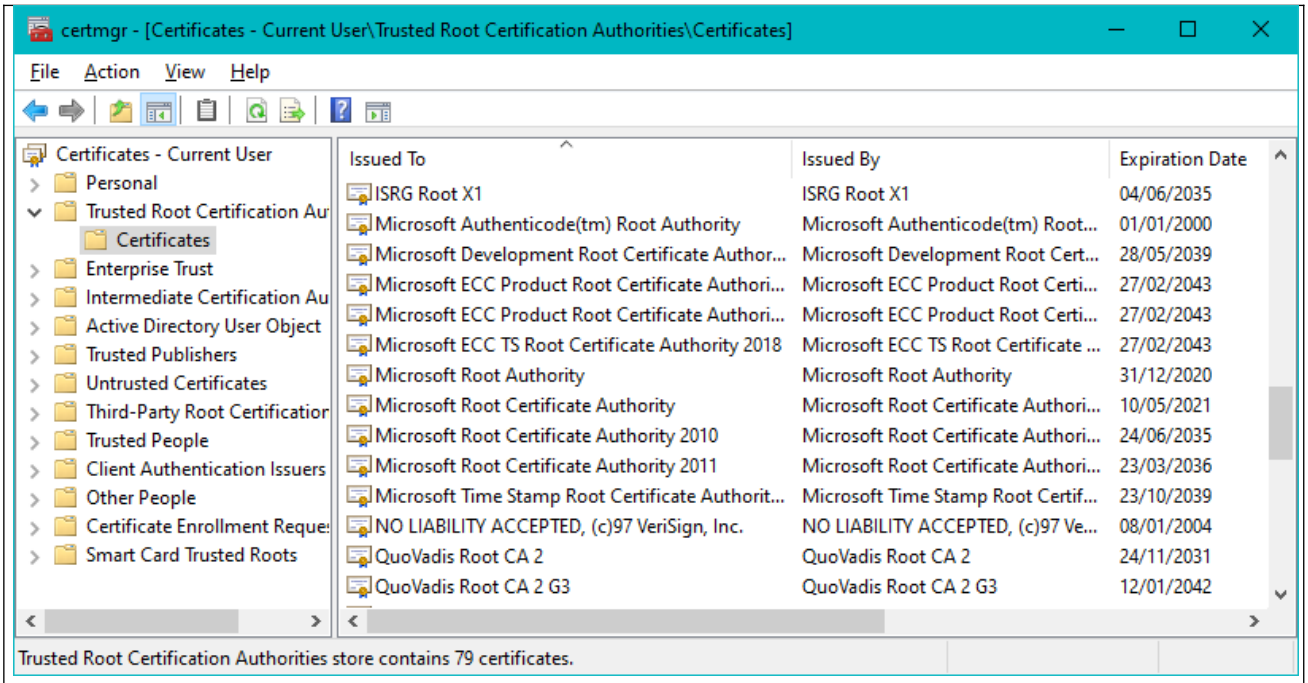

Figure 1. Windows 10 Certificate Manager

Components of a PKI

The main actor is the **Certification Authority (CA)**, it manages the cryptographic keys and certificates of the PKI. It is the component that holds the real keys and that then performs the cryptographic operations. Obviously, in this sense it is also the most critical element of the certification authority.

The **Registration Authority (RA)** manages certificate requests and provides issued certificates. It deals with the distribution of cryptographic tokens and software (it is the front-end of the CA). It is like a counter, for example, to which a citizen can go to request a digital certificate and from which it can be obtained. It can be a physical counter or a website, the type is associated to different levels of security. In addition an RA generally can also assist the user in generating the key pair and managing these certificates securely, for example by providing ad hoc software or hardware tools that support the applicant in generating signatures, encrypting documents, verifying signatures, and decrypting information.

A **Certificate Server** is directory where user certificates are published and in some way made searchable. The directory follows the X.500 standard and is queried by using the LDAP protocol.

The Digital Certificate

The digital certificate structure and format is described by the ITU-T X.509 standard (it is a standard of the International Telecommunication Union numbered X.509).

An X.509 digital certificate must contain at least the following fields:

DN (distinguish name) user: Gaius Julius Caesar

CN (common name): Julius Caesar

O (organization): Government

OU (organization unity): Senate

C (country): Roman Empire

This is the classic pattern of a certificate that meets the X.500 standard, of which X.509 is a member of the family. A certificate can have other attributes (e.g. email). There may be an IP address or a URL that corresponds to an X.500 Directory path (e.g. LDAP – Lightweight Directory Access Protocol):

IT / Roman Empire / Government / Julius Caesar / Gaius Julius Caesar /

An X.509 digital certificate must also contain at least the following fields:

DN Issuer: name of the Authority in X.500 format (the certification authority also has its own distinguished name, which appears in the certificate. Obviously the format with which it is represented is the same as the distinguished name of the subject to which the certificate applies.

Serial Number: serial number of the certificate held by the Authority (each certificate is uniquely identified by the serial number assigned to it by the authority. On the other hand, the pair (*serial number* & *issuer distinguish name*) uniquely identifies a certificate throughout the plethora of certificates that can be issued worldwide as a unique pair).

NotBefore: activation date

NotAfter: expiration date

These are two information strictly related to each other. A certificate therefore has a date of beginning of life and an expiration date, beyond which it is no longer valid. The reason derives from the desire to protect the keys from excessive use.

Public Key: the public key of the certified subject (the certificate contains personal data and the public key)

Public Key Algorithm: the algorithm the public key is suitable for

Signature Algorithm: the algorithm used by the Authority to sign the Certificate

Signature: the signature affixed to the Certificate by the Certificate Authority (therefore the

element that guarantees the authenticity of the certificate itself)

KeyUsage: the purposes the key can be used for

Generation and Revocation of Certificates

The RFC 2314 standard (from PKCS # 10) defines the format of the *certificateRequest* message and contains the public key and data of the applicant (both given to an RA). The request message requires the so-called *proof of possession* of the private key: the certificateRequest message must be signed with the private key of the applicant. The authority, by verifying this signature substantially verifies that the applicant actually possesses this key (otherwise the applicant would not have been able to produce this signature).

On the date corresponding to the "notAfter" attribute, the Certificate is considered "expired". The expiration is a mechanism that also serves to ensure that the same asymmetric keys associated with the digital certificate cannot be used for an excessive period of time, which could expose them to a series of cryptoanalytic attacks.

The expiration can refer to:

- **the certificate** (the association subject-key)
- **the associated keys**, in relation to the use for which they are intended (a key can be used for certain goals only and not for others)

An expired certificate cannot be used by its owner, since it has expired its use is in practice prohibited. In reality, nothing prevents the holder of the certificate from using it for other purposes but all those who communicate with him will somehow detect that this certificate has expired and will not accept it. The certificate is of course available to verify and decrypt messages issued before expiration. In some cases, it is possible to “renew” a certificate, which is usually not automatic.

Sometimes a certificate can be made invalid prior to expiration through *revocation* (this happens before the “notAfter” attribute value. The reason could be, for example, the key associated with the certificate has been compromised, the subject is no longer associated to the company or changed unit, the CA that issued the certificate (or one of its higher level CAs) has been compromised or other reasons. A revoked certificate cannot be used anymore, except when it is just kept “on hold”. This is a particular form of revocation in which the certificate is temporarily inserted in the CRL because of specific reasons (such as “possible key compromise”, “request of revocation by a person other than the owner”, “temporary non-use of the certificate” e.g. for vacation)”, etc. and its validity could be restored later. In case of “reactivation” from suspension, a new entry in the CRL is inserted with a with a special reason code: *removeFromCRL*, as no entry can ever be deleted from the CRL.

To inform the PKI users that a certificate has lost validity, its serial number is listed in a *Certificate Revocation List* (CRL). This list is issued periodically by the issuer Certification Authority and propagated (actively or passively) to all the users, so that all the subjects who have such a certificate can no longer use it. The CRL complies with ITU-T X.509 standard - rfc3280 - rfc4325 which means that its format is standardized. It is signed periodically by the CA and includes a field called the "nextUpdate" which contains the date and time in which the new Certificate Revocation List will be available. If this date belongs to the past it means that our CRL is "expired", if the date belongs to the future we know when we will have to update our list of revoked certificates. Each revoked certificate includes the reason of revocation. The CRL is provided as a service via the LDAP protocol.

As CRLs can become quite large, especially if the certification authority has hundreds of thousands or even millions of users, it can be partitioned and the CA just provides pieces of the Certificate Revocation List, for example referring to a certain period of time. Therefore, when it is necessary to verify a certificate, the citizen downloads the only piece of CRL of interest. Instead of downloading a CRL or a part of it, a service implementing the *Online Certificate Status Protocol* (OCSP - RFC 2560) can be used to find if a certain certificate number is in the CRL. The responses are signed by the CA in order to be trusted

The Chain of Trust

To verify a certificate, a subject must verify all the chain of the intermediate CAs starting from a trusted one, possibly the root CA. First the certificate to check is verified against the issuer CA, then the CA certificate is verified against its parent CA, and so on up to an intermediate trusted (by the user) CA or the root CA itself.

KeyUsage

Actually, keys can be used for different purposes and a certificate specify the allowed uses, among them we have:

- **nonRepudiation**: the key can be used for producing a signature with legal value which can then be opposed to third parties as a handwritten signature
- **dataEncipherment:** the key can be used to encrypt data
- **keyAgreement:** the key can be used to securely exchange keys

TimeStamping

A PKI can also be used to time stamping a document according to the Time Stamp Protocol (RFC 3161) to prove its existence in a point in the past. A CA will sign the document and includes a temporal mark. The signature also ensure that the document has not been altered after affixing the timestamp. A Time Stamp can extend the validity of an electronic signature beyond the lifetime of the digital certificate associated with the keys used, for example by requesting the affixing of a new time stamp on the contract every year that passes. The affixing will be done

every year using cryptographic techniques with a strength adequate for the period in which the signature itself is applied.

Standard Mechanisms and Formats

The following are the two standard mechanism and formats commonly used.

PKCS #7 - Public key cryptography standard #7

PKCS #7 describes the format of messages/files signed, encrypted (with symmetric and asymmetric encryption) and verifiable by MAC (digested data)

S/MIME - Secure Multipurpose Mail Extension

Secure Multipurpose Mail Extension (RFC 3851 - SMIMEv3) is an extension of the MIME standard (RFC 2045, 2046 and 2047 as an extension of RFC 822) and describes the encryption of mail messages. PKCS #7 entities are substantially translated into new MIME entities and therefore describe new containers in which we can find portions of signed or encrypted e-mail messages or signed or encrypted attachments.

Exchange and Storage Formats

The **PKCS #12 (Personal Information Exchange Syntax Standard #12)** defines a format for the exchange and secure storage (on file) of cryptographic keys and digital certificates (trusted certificate chain). Storage is safe because it is based on encryption of both the keys and the certificates, and of the entire chain of certificates a subject chooses to trust. It is the format used by browsers, e-mail clients and operating systems when a user exports a signature and encryption key and a certificate, for example for a backup copy or to copy it to another device.

A PKCS #12 encrypts the data with a symmetrical algorithm whose key is derived from a password provided by the user. It guarantees the integrity of the content through HMAC, or through a digital signature.

The benefits of using PKCS #12 are:

- there are **no implementation costs** as it is a format available in many libraries for many devices (from computers to cellphones)
- they are **already supported by most operating systems** and software, this means that it is immediately available
- it **uses "strong" symmetric algorithms** (the encryption that is applied is a strong, very resistant encryption, so they are keys that resist for example brute force attacks
- it is **portable on different devices**, it is possible to move a file to other devices or copy it to all the devices we use to have the keys always available

There are also some disadvantages:

- **it can be deleted, copied, transmitted**, this is an important problem because it can be stolen, erased, copied to take it away or just made unavailable
- **the private key must be used directly by the signing/decryption software**, this means that the software uses decrypts this file and accesses the key directly, so there is a moment when the key is in the clear on the system, unprotected, although maybe it is just the volatile memory which is the least easily accessible area; however it is available in clear text on the system in some way and this is a very strong limitation of this format
- **it possible to perform a brute force attacks**, for example, in the event that someone steals the PKCS #12 file, they can try all possible passwords in a separate computer system

Cryptographic Tokens

Cryptographic tokens are hardware devices that natively implement encryption algorithms. They do not allow direct access to cryptographic material from outside, they are the only enabled to access and use the key. Moreover, these commands can generally be executed only after the user, and the system on behalf of the user, authenticate to the token, for example by entering a PIN. Tokens can have a small Operating System through which the user requests key generation, encryption and data signature operations. The secure destruction of the token often means not only the cancellation, but also the inability to access the memory areas where the key was written, so to avoid any form of attack that we could not conceive today. The operating system and token hardware can be certified according to specific security standards.

The Challenge-Response Mechanism

A challenge/response mechanism allows two parties to authenticate without exposing useful information on the communication channel. The idea is that one party A “challenges” the other party B to operate on a random number (called *nonce* because it is a number used just one time) with its own key, for example with a One-Way function H. Then B returns H(nonce) to A. If B returns what A expects, A is sure about the identity of B. In the simplest case, B encrypts the nonce with a secret key known to A. A, which has the same key, in turn applies the function to the nonce using the same key, and verifies that the number given to it by B matches the one it calculated.

This is an authentication based on a password that has not been sent over the channel, so it is not possible to trace the key. What makes the mechanism effective is the fact that the random number is different for each session. Since the information read on the channel is different for each session, it is not useful for subsequent authentication. An advantage is that no synchronization is needed (differently from the One-Time-Password mechanism, where it is important to know the starting point of the Pad to use). As it is uneasy to be used by humans, it is

typically used in machine-to-machine authentication. A challenge-response mechanism is found in the network authentication mechanisms of ADSL routers to the provider authentication server.

Questions:

In separate file.

Attachments:

Powerpoint in a separate file.

References:

Every topic has its own page on Wikipedia, sometimes already too thorough for the purposes of this course; books are too specialized and are useful only after a very deep training on the specific subject.