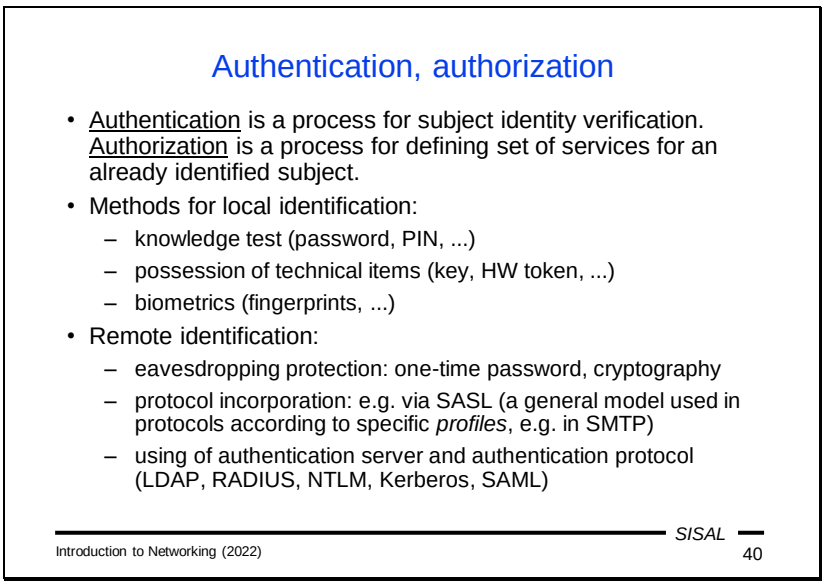


Before we talk about security, we should clarify two terms that are often confused.
- Authentication is a process used by subjects (usually users) to prove their identity, i.e. it is an answer to the question “who am I”.
- Authorization is a process by which an authority (usually a server) assigns permissions to a subject that has already succeeded in the authentication process, i.e. it is an answer to the question “what can I do”.

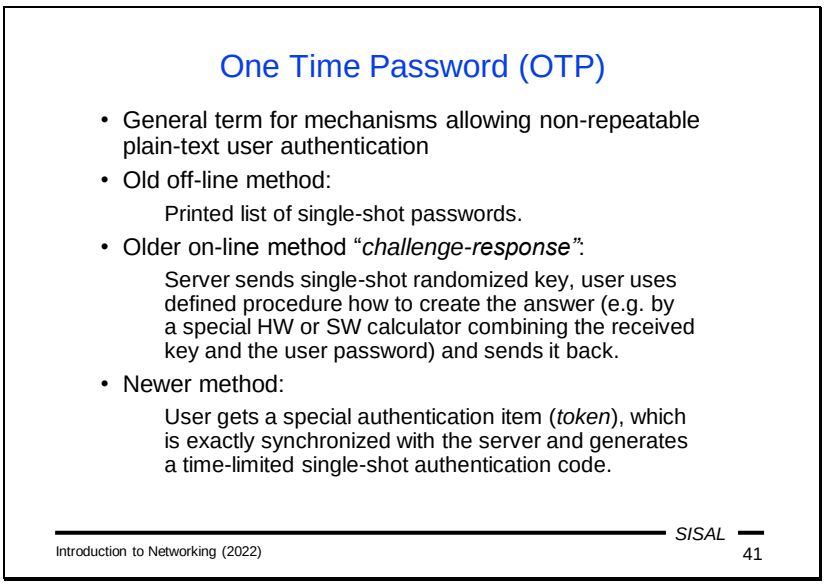
In general, there are three basic methods of proving identity:
- “What do I know”. The easiest way is to test a piece of knowledge, a password (or in a simpler version only a number, a PIN), an answer to one of a set of predefined questions etc. The big disadvantage of these methods is that the knowledge can be revealed. On the other hand, there are also some advantages, especially when authentication is used to access a less important resource: easy implementation, simple sharing of the knowledge and easy use of remote resources.
- “What do I have”. It is a bit safer to prove identity by holding a technical item. It can be either a physical object (a hardware token) or a piece of data (an electronic key). It is a bit more complicated to steal or copy the item.
- The safest proof of identity is biometrics – a fingerprint, a retinal scan, voice recognition etc. – however, if you need to prove your identity for a remote source, these methods have a difficulty in connecting the device that retrieves the biometric data to the server that verifies it.

A combination of these methods is also used.

If we want to authenticate for a remote service (a server), there are several other factors to keep in mind.
If we use the knowledge test, there is a risk of eavesdropping if we are using an insecure channel. The oldest way to protect against reusing revealed information is to

use modified information that is valid for one access only (a one-time password). Another way is to eliminate the problem of an insecure channel and use cryptography to create a secure channel.
Another problem is that most protocols have no means for secure authentication and they need to be extended to be able to transmit the necessary information. There is a framework called Simple Authentication and Security Layer that allows using many authentication methods and incorporates authentication into most of the important protocols. For each protocol, there is a special *profile* defining how to use it in the protocol.
It is also a common requirement to centralize the authentication for a set of resources, to create a special central authentication server (an *identity provider*) that uses a special protocol to communicate either with a client (to get authentication data from him/her and then to provide him/her an authentication result that can be used to prove identity to a server), or with a server (a *service provider*) who communicates with the client by itself.

Slide 41

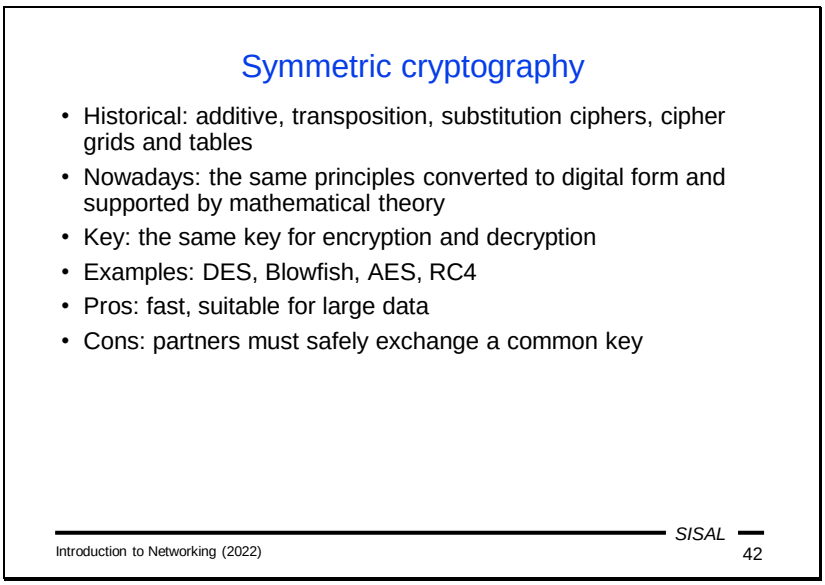


In the past, a very simple method for plain-text authentication was used. A server generated a list of passwords, a user printed this list and used the printed passwords one by one for subsequent authentication attempts.

Later, the so-called *challenge-response* method was developed. A server sends a randomly generated but unique sequence of characters (a challenge) and a client must send an exact response. The response could be generated either by a special hardware calculator, or by a piece of software. The challenge-response principle is widely used in various communication schemes. A conceptually similar system is also used nowadays to validate various web application operations by sending a mail with a URL that need to be visited to complete the operation.

A newer method uses special hardware items (tokens) that are precisely synchronized with a server and generate a special code used for identification. The validity of the code is limited to a few seconds and a single usage so that it cannot be stolen and misused.

Slide 42

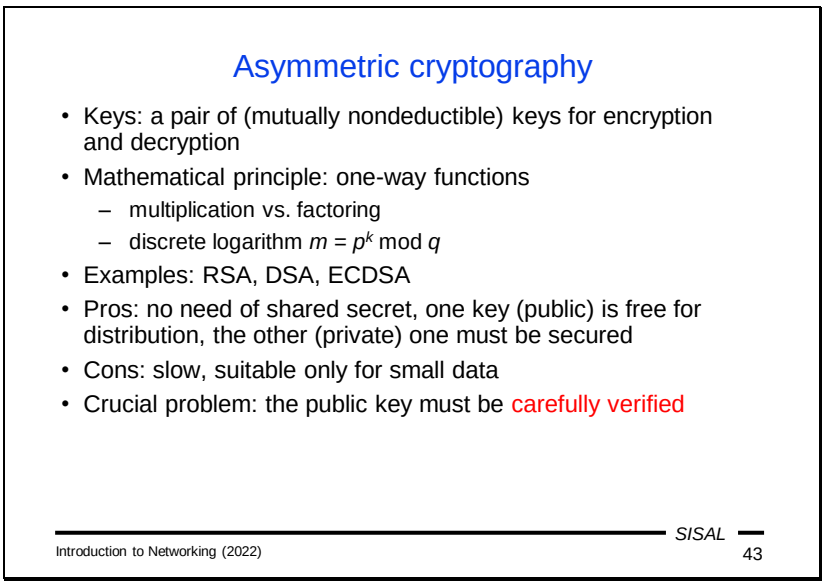


Cryptography plays an important role in contemporary communications. There are three types of cryptographic algorithms that are used in collaboration to achieve the goals of data encryption and origin verification.

The first group are **symmetric encryption** methods.

The history of these algorithms is really very long. Since ancient times people have needed to protect information transported to another place in the world. There are several methods of encoding by substituting or transposing characters. Another approach is to use a grid mask that shows only a part of the text written into a grid. This part of the text is written and by turning the grid mask repeatedly we can encode the entire text. A common feature is that our partner must have the same piece of information (the substitution or transposition key, the grid mask etc.) for decoding that we use for encoding. Of course, nowadays we need different methods of encryption based on complicated mathematical principles.

The basic advantage of current symmetric cryptography methods is that they are very fast and therefore suitable for encrypting large amounts of data. However, the catch is that we haven't solved our problem – the parties needed to securely transfer a piece of information (the message), now they can send it encrypted, but they still need to securely transfer another piece of information (the key). Of course, they can use another, safer method to transport the (small) key, but the fact is that we have only **reduced the size** of the problem.



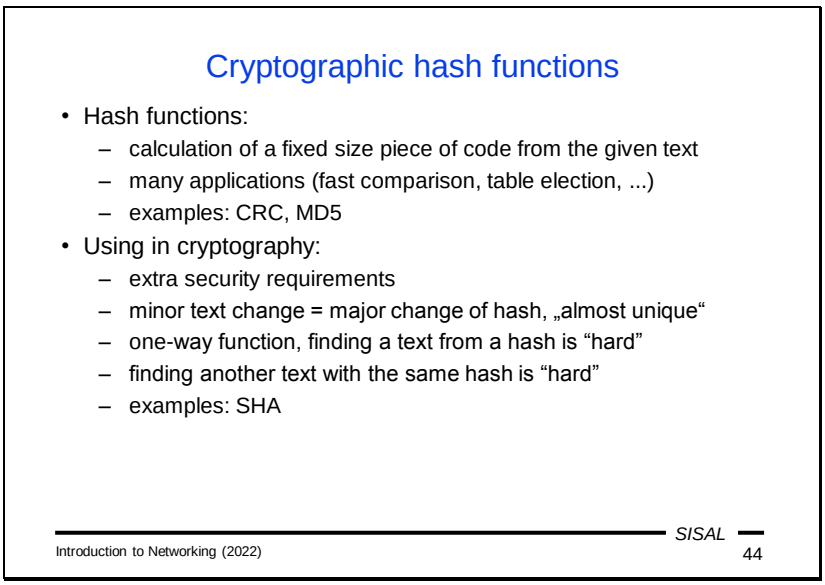
The solution of the problem how to transport securely encrypted data over an insecure channel comes with the **asymmetric cryptography**.

The basic idea is to use two different keys, one for encryption and one for decryption. Of course, it must not be possible to deduce one key from the other. Then you can simply publish one key from the pair while the other should be stored in a safe place.

Just to get an idea of how it works, consider ordinary multiplication – you can easily multiply two numbers but if you have a product, it can be difficult to find its factors (if they are large prime numbers). Such functions are called **one-way** functions. A more sophisticated mathematical one-way function is a discrete logarithm.

The advantages over symmetric cryptography are clear – there is no need to transfer a *shared secret* between the parties. However, a big complication is that these methods are extremely slow and thus only suitable for small amounts of data. But we already solved the problem of reducing the size of transported data using symmetric cryptography!

So we avoid the problem of necessity of sharing a common secret, but there is another problem: we need to **trust** the published key, i.e. to believe that it is correct and that the corresponding secure key really belongs to the person or service we expected.



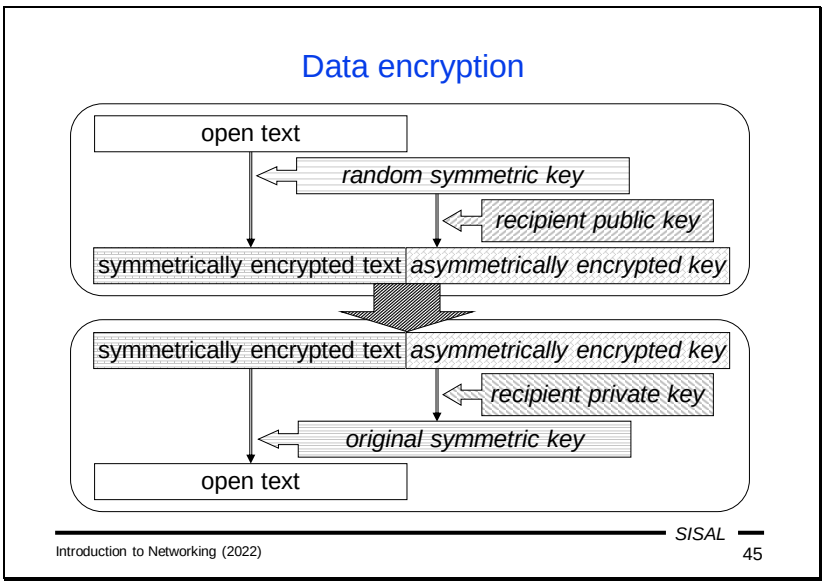
The last type of cryptographic algorithm we will talk about is a **hash function**.

In general, a *hash function* is a function that creates a code of short fixed length from any data. These functions are widely used in programming, e.g. for a fast search through a table, fast comparison of texts etc. The principle is that instead of working with an entire piece of data, we work with its hash only. Of course, the code cannot be unambiguous, so either we accept the risk (e.g. when we compare two versions of a source code file, the probability that they differ but their hash values are the same, is so low that we can ignore it), or we use the hash only to reduce the number of elements we have to deal with and check completely.

However, such weak algorithms are not acceptable for use in cryptography; there are additional conditions for a good cryptographic hash algorithm:

- It is important that even a minor change in the original data causes a fundamental change in the hash value, so the hash is “almost unique”.
- The function must be a one-way function; finding a text that corresponds to a given hash must be “difficult”, as must be finding another text that has the same hash value as a given text.

Slide 45



Now we can look at two basic operations used in cryptography, data encryption and origin verification.

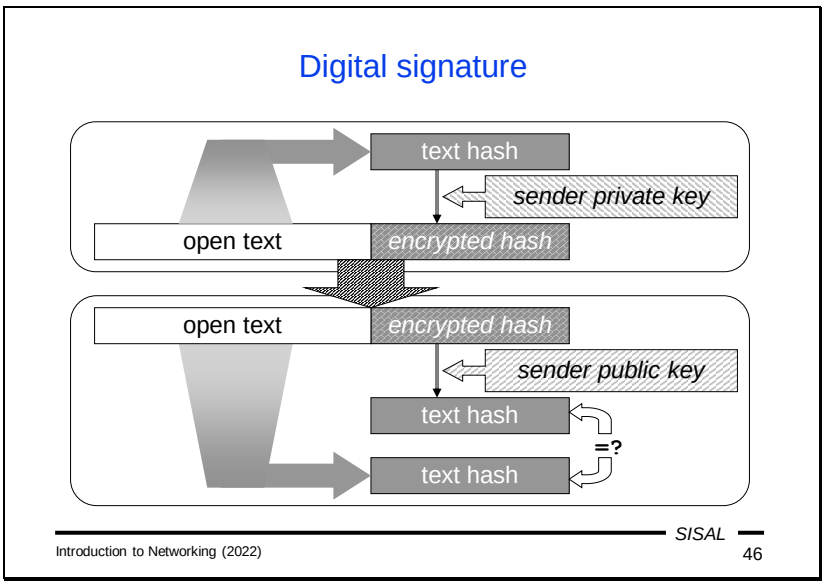
For efficient **data encryption**, we use a combination of symmetric and asymmetric algorithms.

Suppose that we have a text that should be sent encrypted over an insecure channel (e.g. e-mail). We generate a random key key_{sym} and symmetrically encrypt the text. The text can be large since we use symmetric cryptography which is fast enough. Now we need to pass key_{sym} to the recipient, so we need to encrypt it. But the key is short, so we can now use asymmetric cryptography (the speed is not an issue here) and encrypt key_{sym} with the **recipient's public key**. This encrypted key is then attached to the encrypted text.

After successful delivery, the encrypted key is taken and the **recipient's private key** is used to decrypt it. Now we have the original key_{sym} and the data can be symmetrically decrypted.

Note: Especially for e-mail, this approach has also another big advantage – if you have more than one recipient, you encrypt the data **just once**, then encrypt key_{sym} separately for each recipient using his/her public key and attach all the encrypted keys for all recipients to the encrypted data.

Slide 46



For **digital signatures**, we use a combination of asymmetric cryptography and a hash function.

The sender selects a hash function, takes the text (either plaintext or encrypted) and calculates its hash. Then it takes the **sender's private key** and encrypts the hash. The encrypted hash is then attached to the original text (along with the hash algorithm parameters that were used).

The recipient takes the text and uses the given hash function and parameters for a new hash value calculation. Then it takes the **sender's public key** and decrypts the hash originally calculated and encrypted by the sender. If both hashes are equal, we believe that the text was really sent by a person who had access to the sender's private key and that the message has not been modified.

Note that I didn't say "mail was sent by the right person"; we only know that the sender was able to work with the private key, not that it was really the right person…!

Be careful about the properties of these two cryptographic operations: Data encryption protects data from being read by anyone while a digital signature prevents anyone from modifying data sent by the original sender, or from sending completely different data without the recipient revealing it.

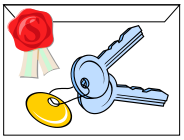
Diffie-Hellman algorithm

- Method of information exchange between two partners via an open channel to get a common secret (e.g. a symmetric key)
- Used in many protocols exploring the symmetric cryptography
- Procedure description:
  1. Alice picks a secret number a and public primes p and q.
  2. She computes $A = p^a \bmod q$ and sends p, q, and A to Bob.
  3. Bob picks a secret number b, computes $B = p^b \bmod q$ and sends B to Alice.
  4. Alice computes $s = B^a \bmod q$ and Bob the same $s = A^b \bmod q$.
- Principle:
  - $A^b = (p^a)^b = p^{ab} = p^{ba} = (p^b)^a = B^a$
  - Without knowing of secret numbers a and b and when choosing large prime numbers p and q, computing the s from A and B is considered to be “hard”.

There is another widely used mathematical principle that allows two partners to agree on a common secret using communication over an open plain-text channel, the so-called **Diffie-Hellman** algorithm. It is also based on one-way functions and the idea is that if both sides keep a secret number and only products of the discrete logarithm are sent via the open channel, then no one who does not know the secret factors, can calculate the final shared secret.

Slide 48

Public key authenticity



- Does the identity tag belong to the key?
  - people usually can make sure that they communicate with the proper partner prior to disclosing some secret
  - the key may be and should be verified from independent sources
  - applications must use some automated way
- Authenticity is verified by a third party and its signature is appended; this can be
  - someone who is known by me and I have his or her key verified ("*web of trust*")
  - well-known public certification authority; listed e.g. in browsers, however such list's credibility is not absolutely sure

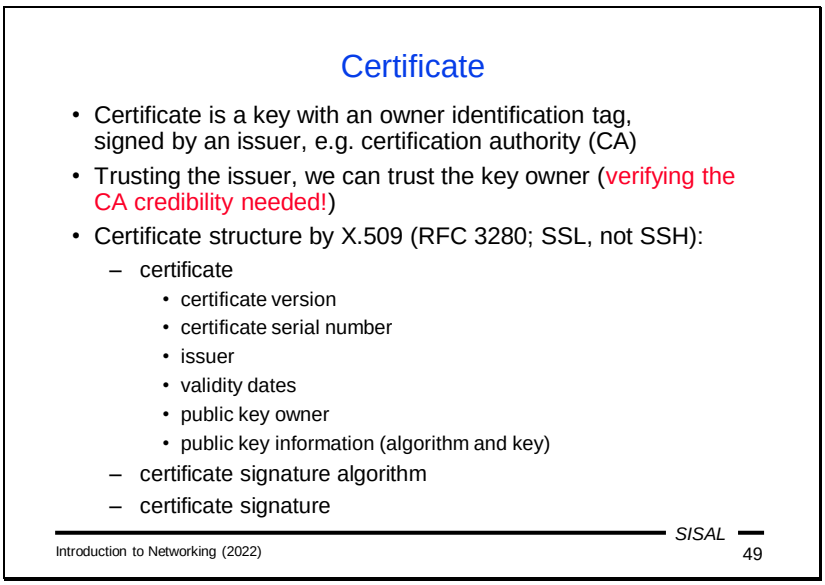
As mentioned earlier, asymmetric cryptography has one fundamental problem: how to prove or verify whether a (public) key belongs to a specific identity (person or service)? If we communicate in person with someone known to us, we can usually recognize whether the person is really our partner – by their greeting, speech style, referring to some common experience etc. If we are not absolutely sure, we can use another independent source to verify the identity. However, if we need to prove identity in software, we must involve some technical means.

In real life, a similar situation is using normal keys. If you have many keys, you will probably make tags to identify them and attach the tags to keyrings. And because you did this by yourself, you will probably believe that the tags match the keys. However, if you are dealing with a key received from someone else, you must believe that its tag is correct. If you want to be sure, you can e.g. arrange that the key with the tag is to be placed in an envelope and sealed by an authority you trust.

The same principle applies to software keys. You can attach **an identity tag** to a key and let someone **confirm** that this coupling is correct. There are various approaches to this type of verification.

- For example, PGP keys for e-mail signing use the method of a *web of trust*. A user creates a key and an identity tag and asks some other users to sign his key+tag. This signed key is then published and all users that trust any of the signers' keys, may decide to trust your key as well from now on. And when you sign other people's keys, the web of trust grows.
- The Public Key Infrastructure framework instead uses special organizations called Certification Authorities (CA) for signing. A CA signs your key+tag pair and if someone trusts this CA, your key is trusted, too.

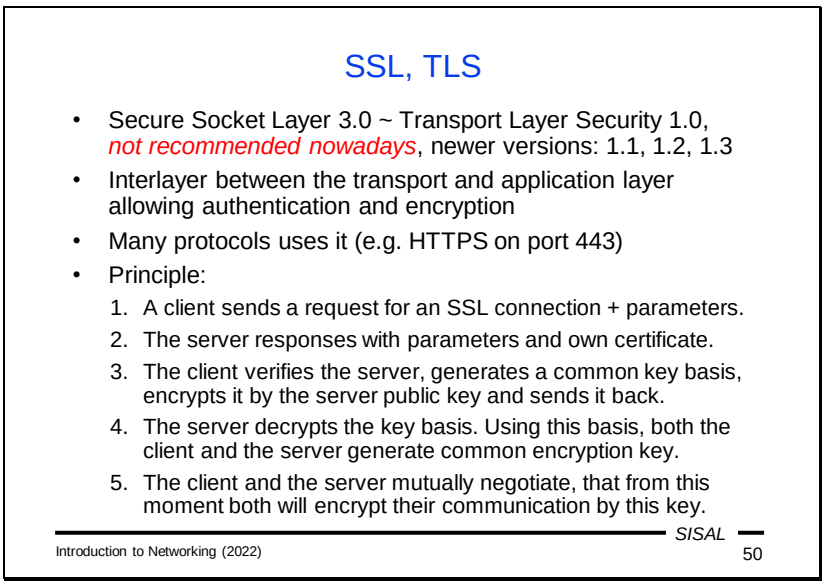
Slide 49



A **certificate** is a public key with a corresponding identity tag attached and signed by a Certification Authority. A user or client that needs to validate the authenticity of a key, downloads its certificate and checks the signature in it using the public key of the signing CA (CA_1). If this key is trusted by the application performing the validation (either it is stored locally in the application, or in the operating system), the original key is trusted as well. If the key of CA_1 is not trusted, the application must get a certificate of it (it can be attached to the original certificate, or the application must download it). This certificate is signed by another CA; let's call it CA_2. Now we must validate the signature of CA_2 using a similar principle. This is a so-called "*trust chain*" that ends either with a known CA, or with failure.

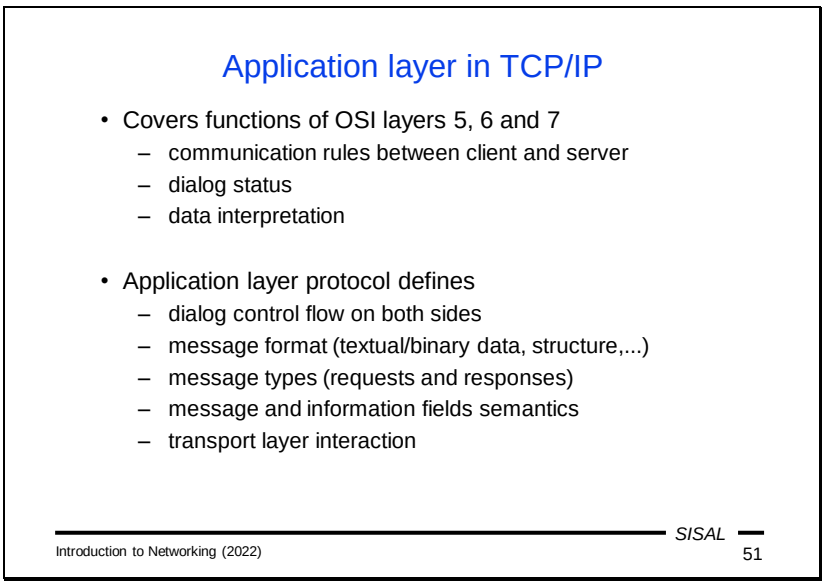
The list of "trusted" CAs is distributed together with the operating system or with software that will use it. For instance, if you install a web browser, a list of CAs trusted by the browser manufacturer is installed at the same time. There are some changes in the list from time to time and the browser automatically updates the list to keep it up-to-date. The problem is that from time to time a CA with a bad reputation (e.g. a CA that does not sufficiently verify key ownership) gets onto a list of trusted CAs. Once this is discovered, all the keys signed by this CA must be revoked.

Slide 50



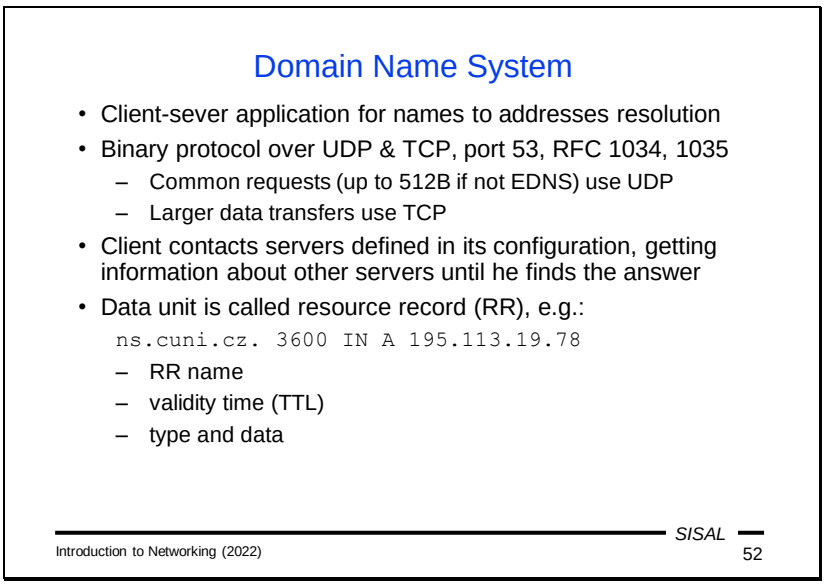
Now we have enough information to explain how the old TCP/IP protocols are currently used in a way that ensures their security. The general method is to insert a special intermediate layer in between the transport and application layers; this layer manages endpoint identification and data encryption. The layer was originally called Secure Socket Layer and the message “s” was appended to the names of the protocols that used it. For example, the URL scheme “https” means “HTTP over SSL”. The important fact is that it is **not a new protocol**; the scheme only says that the SSL interlayer is used. In version 3.0 SSL was renamed to Transport Layer Security 1.0 after slight modifications, but the terms SSL and TLS are still both used interchangeably (except when talking about their version numbers). TLS development still continues and it is now not recommended to use the oldest version 1.0, which was based on SSL.

Slide 51



After the preceding introductory chapters we can now start to talk about the most important protocols at the **application layer**. We already know that in TCP/IP, this layer covers the work of the top three OSI layers. It means that an application protocol in TCP/IP must specify:

- How the dialog can proceed. This means information such as who initiates the connection, who starts the dialog, what kind of operations can be requested and what responses to such requests can be etc.
- A message format. In general, the protocols are either
  - *textual* – i.e. you can look at the data sent by both sides in a text editor, e.g. integer values are sent in their text interpretation
  - *binary* – i.e. the data is structured as a stream of blocks, bytes or even bits, so to understand it you must use a piece of software that converts it into a textual form, e.g. integer values are sent as a sequence of bits representing the value
- Message types. A set of message types for different requests and a set of possible responses to them.
- Data semantics. Each data field (both text and binary) must be described so that there is solely one interpretation of it possible.
- Which transport protocol will be used in which situation and how. In the case of UDP, e.g. a maximum message size should be defined, in the TCP case, e.g. a grouping or splitting of messages into blocks can be described.



The Domain Name System (DNS) is a service for *resolving* domain names to addresses and vice versa. In fact, it provides even more information about stored domain names, but IP address queries are the most frequent.

The service is implemented by the protocol of the same name, which uses both the UDP and TCP transport protocols.

- Common queries are handled in UDP. Questions and answers are short and the overhead of establishing a TCP connection is unnecessary. In addition, UDP allows the client to respond more quickly to any slow or missing server responses. Originally, the limit for a UDP message was 512 bytes, but today the extended protocol (EDNS) is commonly used, which allows the client and server to agree on larger sizes.
- If a response exceeds the size limit, the server sends only the part that fits in the message and sets the TC (truncated) flag. The client can then repeat the query using TCP to obtain a complete answer. Some data exchanges (e.g. database updates between servers) take place directly in TCP.

A common client implementation is to sequentially query the servers it has in its configuration (think about why it has **addresses** of servers there, not their names), gradually increasing its timeout for receiving a response (usually starting at 0.5 sec) until a response is received. If the response does not contain the necessary information, it should include a link to other servers that should be queried to get it.

The protocol is binary; each message contains a header and then a number of so-called *resource records* (RR). Each record contains the name, time-to-live (TTL) in seconds, record type, and corresponding data.

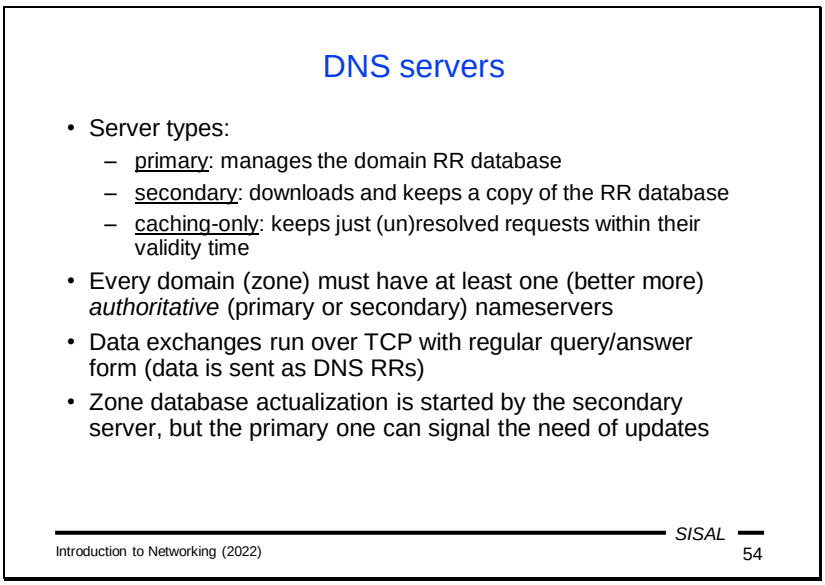
DNS resource records

Type	RR name semantics	Data semantics
SOA	domain name	general domain attributes
NS	domain name	domain nameserver name
A	host name	host IPv4 address
AAAA	host name	host IPv6 address
PTR	reverse name (e.g. IP address 1.2.3.4 is represented by `4.3.2.1.in-addr.arpa`, ::1 by `1.0...0.ip6.arpa`)	host domain name
CNAME	alias name	canonical (real) host name
MX	domain/host name	mailserver name and priority

Overview of the most important record types:

- SOA (Start Of Authority) is the initial record of each domain and contains information such as the data version number, administrator address, etc.
- NS is a type of record for name servers that maintain a database of records for a given domain (usually each domain has more than one NS RR).
- A is a record containing the IPv4 address for a given name.
- AAAA is a record for an IPv6 address. The name of the record is perhaps a bit strange and one would expect something like A6. Indeed, this type of record was introduced as a first draft, but did not work well, and was therefore replaced. The name AAAA indicates that the record contains 4x longer data than record A…
- PTR is a reverse record that is used to convert addresses to names. Since there must be a name and not an address on the left side of each RR, it is necessary to somehow convert the address into the name format. For IPv4, the conversion $IP_1.IP_2.IP_3.IP_4 \rightarrow IP_4.IP_3.IP_2.IP_1$.in-addr.arpa is used. The reason for the reverse byte order is the reverse hierarchy of names (right to left) and addresses (left to right). This allows, for example, placing the server responsible for the $IP_1.IP_2.IP_3.*$ network, and thus the $IP_3.IP_2.IP_1$.in-addr.arpa domain, directly in this network. For IPv6, the address is divided into a series of half bytes; their hexadecimal values are separated by periods, and the ip6.arpa domain is appended. Every computer connected to the Internet should have a reverse record, because some servers require clients that connect to them to have such an entry.
- CNAME (Canonical NAME) is an alias creation record. On the left side is the name of the alias, on the right side is the canonical (or real) name of the computer (beware, there is a common misunderstanding of the left and right sides). Both the record name and the description in the RFC clearly specify that the alias should not lead to another alias. The reason for this is the reduction of additional queries. Unfortunately, this restriction is inconvenient and this rule is commonly violated.

- MX (Mail eXchanger) is a record defining which server receives mail for a given domain or computer (usually more than one MX exists for a domain). This is a nice example of how to separate a domain record from the record of a particular service in that domain. It works very well with e-mail and it is a pity that a similar simple mechanism has not been introduced for the WWW. As a result, **host**-specific (A or AAAA) records are now commonly defined for a **domain** so that users can omit writing "www" in browsers. A solution using a special "web server" record would be more conceptual.

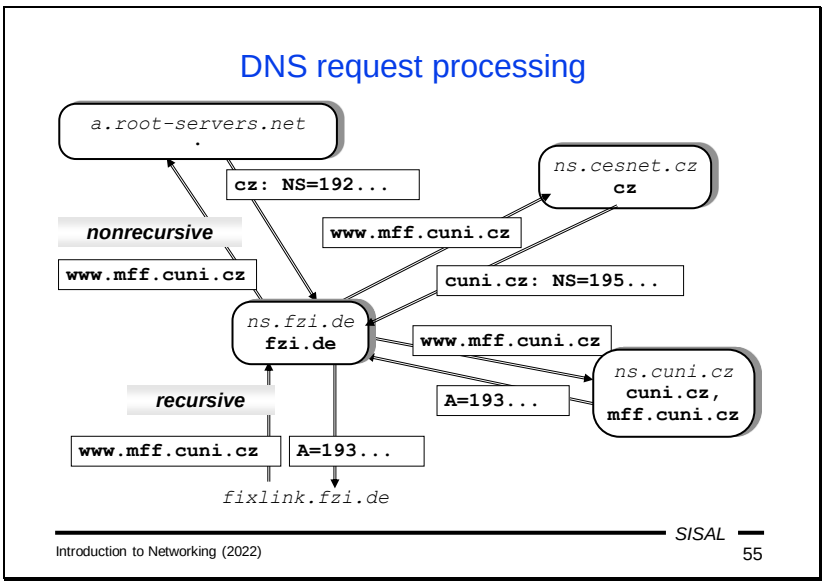


For handling queries belonging to a particular domain, we divide servers into three groups:

- The most important server is the so-called primary (or "master") server. It manages the domain RR database.
- Several secondary (or "slave") servers are usually set up as backups. They periodically download the current contents of the database from the primary server.
- All other servers, if they access any records for a foreign domain, usually cache them for some time, which is why we call them caching-only.

The first two types are collectively referred to as authoritative servers because their data comes from a trusted source. All authoritative servers return answers marked with the authority flag, and clients can choose any of them for making a query.

Database updates are initiated by secondary servers depending on a period defined in the SOA record. Outside of these periodic times, the primary server may inform the secondary servers that the data has changed to such an extent that it is appropriate to perform an out-of-order update. This mechanism reduces the volume of communication between servers, but a consequence is that even an authoritative server can return outdated data to clients for a short time after a data change.



Let's now look at an example of the query flow.

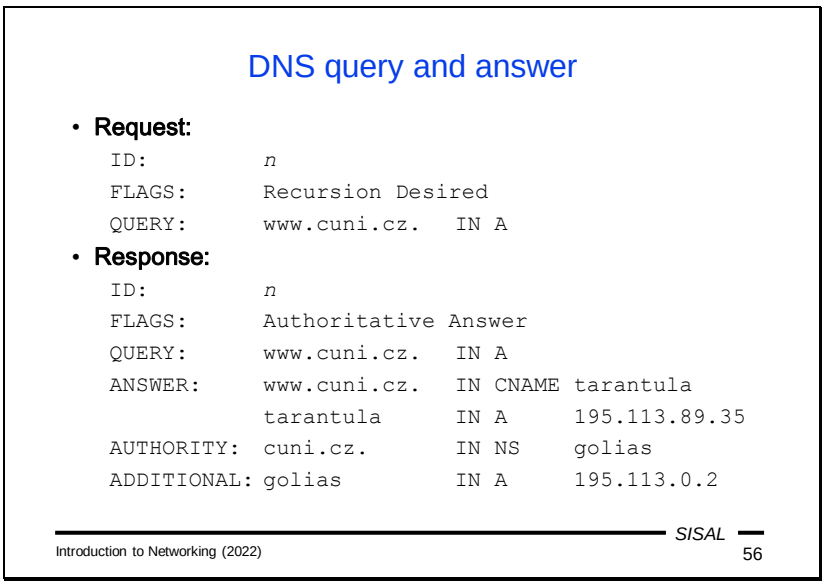
- Let's imagine that a user on a computer in a domain types the address www.mff.cuni.cz into the browser's address bar. In the usual implementation of a client on a workstation, a query is sent to a name server in the domain where the query originated. The query will be **recursive**, which means that the server takes responsibility for complete processing the query and sending the response. If the selected server does not currently have any relevant information in its cache, it needs to start looking for it.
- The server has no information about the domains mff.cuni.cz, cuni.cz or cz. It must therefore contact one of the so-called root name servers, whose addresses it has in its configuration. However, these servers do not handle queries recursively – they will only find the most relevant answer to the question in their database and send it. In our case, it will be an answer such as "send your query to the name server named… cz, whose address is…".
- The server caches this information and continues by querying the proposed server. In this way, all intermediate servers are cached until the query arrives at some authoritative server, which sends the final response.
- This response is also cached and finally sent to the client that originally requested it.

You may be interested in why we send the entire query to the root server, when it will most likely answer us only with a record for cz. There are several reasons for this:

- The basic problem is that DNS is designed so that a **dot** can be part of a **name**! It's as if you could use slashes or backslashes in file names in the file system. This decision brings certain advantages (I can set up a website www.group.abc.com despite the non-existent domain group.abc.com), but also a number of disadvantages. In short, a DNS client cannot decide which dots in a name are

really domain delimiters and which are not. Therefore, the query must be sent in its entirety – to someone who has this information.

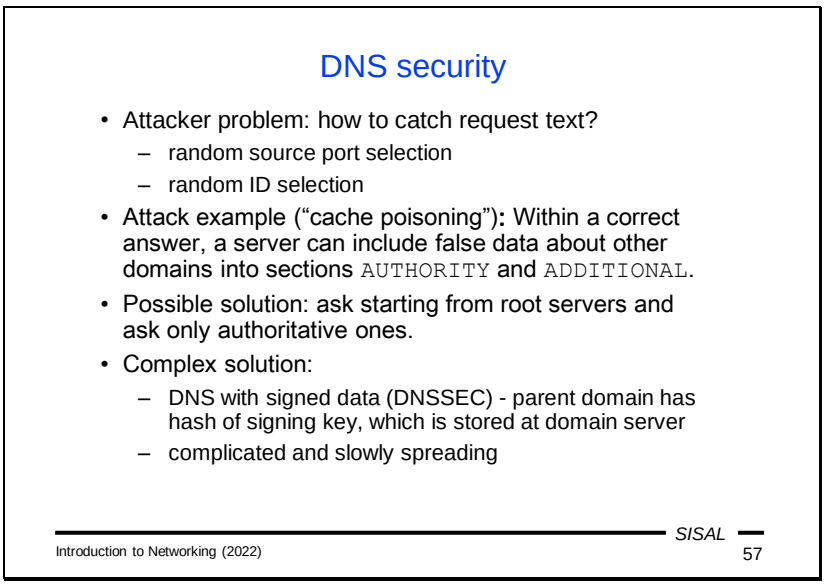
- In the example in the picture, we see that we saved one step by sending the entire query instead of a partial query "where is the server for mff.cuni.cz“ to the server for cuni.cz. The server for cuni.cz is a secondary server for mff.cuni.cz, so it could immediately send us the required final authoritative response.



Before discussing DNS security, let's look at what a request and response look like.

A DNS query consists of a header that contains, among other things, a 2-byte random number (identifier) of the query and flags (e.g. a recursive response request), and a so-called QUERY section with a single RR containing the queried name and type (this record is exceptional in that it does not have a data part).

The answer again contains the query identifier and flags (e.g. the authority of the answer) in the header and the repeated query in the QUERY section. This is followed by an ANSWER section containing RRs with answers, an AUTHORITY section containing a list of name servers that can give an authoritative answer or at least authoritative information about some of the parent domains contained in the query, and an ADDITIONAL section containing additional information (addresses of name servers from the AUTHORITY section, addresses of the MX servers listed in the ANSWER section, etc.).



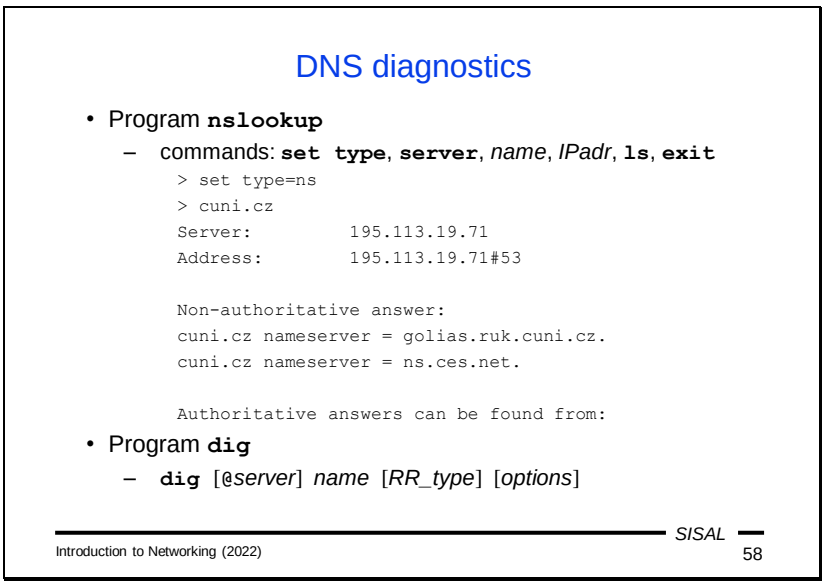
DNS security is based on the fact that it is not easy for a potential attacker to get to the exact content of the query in order to spoof the answer. If it can attack and eavesdrop on the local network, an attack is possible. However, if the query leaves the local network, it is routed between the network routers to the destination server, and it is difficult for an attacker to get to the query there. The second option for an attacker is to expect the question, guess its content and send a false answer. This procedure is complicated by the random selection of a source port of the UDP query and a random ID – this gives billions of possibilities.

It is therefore necessary for an attacker to choose other ways. As an example of a protocol weakness, we can mention the so-called *cache poisoning* attack. An attacker legally operates a server for a domain and forces the client to send him a DNS query (for example, by a spamming advertisement). The query contains a correct answer (making the attack more insidious), but the AUTHORITY section contains faked information that e.g. the server for the cz domain is also this attacker's server. And if the client is naive, it caches this false information and from then on the attacker gains complete control over queries directed to the cz domain from the given client (or worse – from the whole network).

The solution for the client, of course, is to proceed with queries from the root servers and cache only the information to which authoritative responses lead.

Weak DNS security has led to the creation of another extension (DNSSEC), the essence of which is the signing of all records in the domain with a key that is stored in the parent domain. An attacker who does not know the key cannot forge records, and cannot forge the key without access to the server of the parent domain (e.g. to forge data about the domain cuni.cz, he would have to attack the server for the

domain cz). The problem with DNSSEC is that to ensure sufficient security, the protocol became extremely complicated, as is the management of signed domains, and as a result it has spread very slowly.



If we need to check the behavior of the *resolver* (the operating system component responsible for using the DNS service), we have the **nslookup** program available. After starting, the program displays a prompt and we can enter commands for specifying the server we are querying and query parameters, as well as the names we want to resolve. The program can also be called from the command line and the query can be specified using parameters.

On UNIX computers we usually have a program with a similar function called **dig** or **drill**.

Summary 3

- Describe three basic groups of cryptographic algorithms and the differences between them.
- Describe the principles of encryption and digital signatures.
- Describe the purpose and principle of the Diffie-Hellman algorithm.
- How is the authenticity of a public key verified?
- What is the difference between HTTP and HTTPS?

- When is UDP used in DNS and when is TCP used?
- Describe the purpose and behavior of the secondary name server of a domain.
- Describe the procedure for resolving a DNS query.
- Describe the security risks of DNS.