Entity Authentication

Entity Authentication is one of the security service. Many cryptographic entity authentication mechanisms rely on randomly generated numbers.

3.1 Random Number Generation:

Many cryptographic primitives cannot function securely without randomness

3.1.1 The need for randomness:

- Most cryptographic primitives take structured input and turn it into something that has no structure.
- Many cryptographic primitives *require* sources of randomness in order to function.

3.1.2 What is randomness?

Randomness is about ideas such as 'unpredictability' and 'uncertainty'. A random number generation process is often assessed by applying a series of statistical tests.

3.1.3 Non-deterministic generators

There are two general approaches to generating randomness.

A *non-deterministic generator* is based on the randomness produced by physical phenomena and therefore provides a source of 'true randomness' in the sense that the source is very hard to control and replicate. Non-deterministic generators can be based on hardware or software.

- **HARDWARE-BASED NON-DETERMINISTIC GENERATORS**

  Hardware-based non-deterministic generators rely on the randomness of physical phenomena. Generators of this type require specialist hardware. Examples include:

➢ measurement of the time intervals involved in radioactive decay of a nuclear atom;

➢ semiconductor thermal (Johnson) noise, which is generated by the thermal motion of electrons;

➢ instability measurements of free running oscillators;

➢ white noise emitted by electrical appliances;

➢ quantum measurements of single photons reflected into a mirror.

Hardware-based generators provide a continuous supply of randomly generated output for as long as the power required to run the generator lasts, or until the process ceases

to produce output. However, because specialist hardware is required, these types of generator are relatively expensive.

- **SOFTWARE-BASED NON-DETERMINISTIC GENERATORS**

Software-based non-deterministic generators rely on the randomness of physical phenomena detectable by the hardware contained in a computing device.

Examples include:

- ➢ capture of keystroke timing;
- ➢ outputs from a system clock;
- ➢ hard-drive seek times;
- ➢ capturing times between interrupts (such as mouse clicks);
- ➢ mouse movements;
- ➢ computations based on network statistics.

These sources of randomness are cheaper, faster and easier to implement than hardware-based techniques.

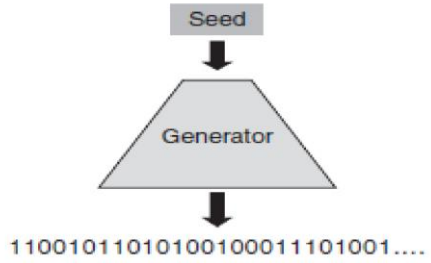
Disadvantage: Two problems with non-deterministic generators:

1. They tend to be expensive to implement.
2. It is, essentially, impossible to produce two identical strings of true randomness in two different places.

3.1.4 Deterministic generators

BASIC MODEL OF A DETERMINISTIC GENERATOR

- A *deterministic generator* is an algorithm that outputs a pseudorandom bit string, in other words a bit string that has no apparent structure.
- The output of a deterministic generator is certainly not randomly generated. If anyone who knows the information that is input to the deterministic generator can completely predict the output.
- If input into the deterministic generator is secret then, with careful design of the generation process, generated output will have no apparent structure.
- It will thus *appear* to have been randomly generated to anyone who does not know the secret input. This is precisely the idea behind a deterministic generator.
- The basic model of a deterministic generator is shown in the below Figure



Basic model of a deterministic generator

The two components of this model are:

A seed. The secret information that is input into the deterministic generator is often referred to as a *seed*. This is essentially a cryptographic key. The seed is the only piece of information that is definitely not known to an attacker. Thus, to preserve the unpredictability of the pseudorandom output sequence it is important both to protect this seed and to change it frequently.

The generator. This is the cryptographic algorithm that produces the pseudorandom output from the seed.

DETERMINISTIC GENERATORS IN PRACTICE

A deterministic generator overcomes the two problems that we identified for non-deterministic generators:

1. They are cheap to implement and fast to run. It is no coincidence that deterministic generators share these advantages with stream ciphers, since the keystream generator for a stream cipher is a deterministic generator whose output is used to encrypt

2. Two identical pseudorandom outputs can be produced in two different locations. All that is needed is the same deterministic generator and the same seed.

The seed is relatively short. It is normally a symmetric key of a standard recommended length, such as 128 bits.

Properties of non-deterministic and deterministic generators:

Non-deterministic generators	Deterministic generators
Close to truly randomly generated output	Pseudorandom output
Randomness from physical source	Randomness from a (short) random seed
Random source hard to replicate	Random source easy to replicate
Security depends on protection of source	Security depends on protection of seed
Relatively expensive	Relatively cheap

Cryptanalysis of the generator. Deterministic generators are cryptographic algorithms and are always vulnerable to potential weaknesses in their design. Use of a well-respected deterministic generator is probably the best way of reducing associated risks.

Seed management. If the same seed is used twice with the same input data then the same pseudorandom output will be generated. Thus seeds need to be regularly updated and managed. The management of seeds brings with it most of the same challenges as managing keys, and presents a likely target for an attacker of a deterministic generator.

3.2 Providing freshness

- *Freshness mechanisms* are techniques that can be used to provide assurance that a given message is 'new', in the sense that it is not a *replay* of a message sent at a previous time.
- The main threat that such mechanisms are deployed against is the capture of a message by an adversary, who then later replays it at some advantageous time.
- Freshness mechanisms are particularly important in the provision of security services that are time-relevant, of which one of the most important is entity authentication.

There are three common types of freshness mechanism.

3.2.1 Clock-based mechanisms

- A *clock-based* freshness mechanism is a process that relies on the generation of some data that identifies the time that the data was created. This is sometimes referred to as a *timestamp*.
- This requires the existence of a clock upon which both the creator of the data and anyone checking the data can rely.

For example, suppose that Alice and Bob both have such a clock and that Alice includes the time on the clock when she sends a message to Bob. When Bob receives the message, he checks the time on his clock and if it 'matches' Alice's timestamp then he accepts the message as fresh.

Clock-based freshness mechanisms seem a natural solution, however, they come with four potential implementation problems:

1. **Existence of clocks**. Alice and Bob must have clocks. For many devices, such as personal computers and mobile phones, this is quite reasonable. It may not be so reasonable for other devices, such as certain types of smart token.

2. **Synchronisation**.

➢ Alice's and Bob's clocks need to be reading the same time, or sufficiently close to the same time.

➢ The clocks on two devices are unlikely to be perfectly synchronised since clocks typically suffer from *clock drift*.

➢ Even if they drift by a fraction of one second each day, this drift steadily accumulates.

Solution:

- One solution might be to only use a highly reliable clock, for example one based on a widely accepted time source such as universal time.
- Another solution might be to regularly run a resynchronisation protocol.
- Next solution is to define a window of time within which a timestamp will be accepted. Deciding on the size of this *window of acceptability* is application dependent and represents a tradeoff parameter between usability and security.

3. **Communication delays**. It is inevitable that there will be some degree of communication delay between Alice sending, and Bob receiving, a message.
4. **Integrity of clock-based data**. Bob will normally require some kind of assurance that the timestamp received from Alice is correct. This can be provided by conventional cryptographic means, for example using a MAC or a digital signature. However, such an assurance can only be provided when Bob has access to the cryptographic key required to verify the timestamp.

3.2.2 Sequence numbers

In applications where clock-based mechanisms are not appropriate, an alternative mechanism is to use ***logical***time. Logical time maintains the order in which messages or sessions occur and is normally instantiated by a ***counter* or *sequence number***.

An example: Suppose Alice and Bob regularly communicate with one another and wish to ensure that messages that they exchange are fresh. Alice can do this by maintaining two sequence numbers for communicating with Bob, which are counters denoted by $N_{AB}$and N_{BA}. Alice uses sequence number N_{AB}as a counter for messages that she sends to Bob, and

sequence number N_{BA}as a counter for messages that she receives from Bob. Both sequence numbers work in the same way.

When Alice sends a message to Bob:

1. Alice looks up her database to find the latest value of the sequence number N_{AB}. Suppose that at this moment in time N_{AB}= *Tnew* .

2. Alice sends her message to Bob along with the latest sequence number value, which is *Tnew* .

3. Alice increments the sequence number $N_{AB}$by one (in other words, she sets N_{AB}= *Tnew* + 1) and stores the updated value on her database. This updated value will be the sequence number that she uses next time that she sends a message to Bob.

When Bob receives the message from Alice:

4. Bob compares the sequence number *Tnew* sent by Alice with the most recent value of the sequence number N_{AB}on his database. Suppose this is N_{AB}= *Told* .

5. If *Tnew* > *Told* then Bob accepts the latest message as fresh and he updates his stored value of $N_{AB}$from *Told* to *Tnew* .

6. If *Tnew* ≤ *Told* then Bob rejects the latest message from Alice as not being fresh.

Sequence numbers address the four concerns that are raised with clock-based mechanisms:

1. **Existence of clocks**. The communicating parties no longer require clocks.
2. **Synchronisation**. In order to stay synchronised, communicating parties need to maintain a database of the latest sequence numbers.
3. **Communication delays**. These only apply if messages are sent so frequently that there is a chance that two messages arrive at the destination in the reverse order to which they were sent. If this is a possibility then there remains a need to maintain the equivalent of a window of acceptability, except that this will be measured in terms of acceptable sequence number differences, rather than time.

For example, Bob might choose to accept the message as fresh not just if *Tnew* > *Told* , but *also* if *Tnew* = *Told* , since there is a chance that the previousmessage from Alice to Bob has not yet arrived. This issue is notrelevant if either:

- delays of this type are not likely (or are impossible);
- Bob is more concerned about the possibility of replays than the implications of rejecting genuine messages.

4. **Integrity of sequence numbers**. Just as for clock-based time, an attacker who can freely manipulate sequence numbers can cause various problems in any protocol that relies on them. Thus sequence numbers should have some level of cryptographic integrity protection when they are sent.

3.2.3 Nonce-based mechanisms

- One problem that is shared by both clock-based mechanisms and sequence numbers is the need for some integrated infrastructure. This problem solved using a mechanism called *Nonce-based* mechanisms.
- Their only requirement is the ability to generate *nonces* (literally, 'numbers used only once'), which are randomly generated numbers for one-off use.
- The general principle is that Alice generates a nonce at some stage in a communication session (protocol).
- If Alice receives a subsequent message that contains this nonce then Alice has assurance that the new message is fresh, whereby 'fresh' we mean that the received message must have been created *after* the nonce was generated.

Suppose that Alice generates a nonce and then sends it in the clear to Bob. Suppose then that Bob sends it straight back. Consider the following three claims about this simple scenario:

Alice cannot deduce anything from such a simple scenario:

- This is not true, although it is true that she cannot deduce very much. She has just received a message consisting of a nonce from someone. It could be from anyone.
- However, it consists of a nonce that she has just generated. This means is that it is virtually certain that whoever sent the nonce back to her (and it might not have been Bob) must have seen the nonce that Alice sent to Bob.
- In other words, this message that Alice has just received was almost certainly sent by someone *after* Alice sent the nonce to Bob.
- In other words, the message that Alice has just received is not authenticated, but it is fresh.

There is a chance that the nonce could have been generated before:

- This is true, there is a 'chance', but if we assume that the nonce has been generated using a secure mechanism and that the nonce is allowed to be sufficiently large then it is a very small chance.

- This is the same issue that arises for any cryptographic primitive. If Alice and Bob share a symmetric key that was randomly generated then there is a 'chance' that an adversary could generate the same key and be able to decrypt ciphertexts that they exchange.
- We can guarantee that by generating the nonce using a secure mechanism, the chance of the nonce having been used before is so small that we might as well forget about it.

Since a nonce was used, Bob is sure that the message from Alice is fresh.

- This is not true, he certainly cannot. As far as Bob is concerned, this nonce is just a number. It could be a copy of a message that was sent a few days before.
- Since Bob was not looking over Alice's shoulder when she generated the nonce, he gains no freshness assurance by seeing it.
- If Bob has freshness requirements of his own then he should also generate a nonce and request that Alice include it in a later message to him.

Nonce-based mechanisms, come with two costs:

1. Any entity that requires freshness needs to have access to a suitable generator, which is not the case for every application.

2. Freshness requires a minimum of two message exchanges, since it is only obtained when one entity receives a message back from another entity to whom they earlier sent a nonce. In contrast, clock-based mechanisms and sequence numbers can be used to provide freshness directly in one message exchange.

3.2.4 Comparison of freshness mechanisms

	Clock-based	Sequence numbers	Nonce-based
Synchronisation needed?	Yes	Yes	No
Communication delays	Window needed	Window needed	Window needed
Integrity required?	Yes	Yes	No
Minimum passes needed	1	1	2
Special requirements	Clock	Sequence database	Random generator

3.3 Fundamentals of entity authentication

- Entity authentication is the assurance that a given entity is involved and currently active in a communication session. This means that entity authentication really involves assurance of both:

Identity. the identity of the entity who is making a claim to be authenticated;

Freshness. that the claimed entity is 'alive' and involved in the current session.

- If we fail to assure ourselves of freshness then we could be exposed to *replay attacks*, where an attacker captures information used during an entity authentication session and replays it a later date in order to falsely pass themselves off as the entity whose information they 'stole'.
- If entity authentication is only used to provide assurance of the identity of one entity to another (and not vice versa) then we refer to it as *unilateral* entity authentication.
- If both communicating entities provide each other with assurance of their identity then we call this *mutual* entity authentication.

3.3.1 A problem with entity authentication

Entity authentication is a security service that is only provided for an 'instant in time'. It establishes the identity of a communicating entity at a specific moment, but just seconds later that entity could be replaced by another entity, and we would be none the wiser.

Consider the following very simple attack scenario:

- Alice walks up to an ATM, inserts her payment card and is asked for her PIN. Alice enters her PIN. This is an example of entity authentication since the card/PIN combination is precisely the information that her bank is using to 'identify' Alice.
- As soon as the PIN is entered, Alice is pushed aside by an attacker who takes over the communication session and proceeds to withdraw some cash.
- The communication session has thus been 'hijacked'. Note that there was no failure of the entity authentication mechanism in this example.
- The only 'failure' is that it is assumed that the communication that takes place just a few seconds after the entity authentication check is still with the entity who successfully presented their identity information to the bank via the ATM.
- This instantaneous aspect of entity authentication might suggest that for important applications we are going to have to conduct almost continuous entity authentication in order to have assurance of the identity of an entity over a longer period of time.
- In the case of the ATM, we would thus have to request Alice to enter her PIN every time she selects an option on the ATM. This will really annoy Alice and does not even protect against the above attack, since the attacker can still push Alice aside at the end of the transaction and steal her money.

The solution is to **combine entity authentication** with the establishment of a **cryptographic key.**

3.3.2 Applications of entity authentication

Entity authentication tends to be employed in two types of situation:

Access control. Entity authentication is often used to directly control access to either physical or virtual resources. An entity, sometimes in this case a human user, must provide assurance of their identity in real time in order to have access. The user can then be provided with access to the resources immediately following the instant in time that they are authenticated.

As part of a more complex cryptographic process. Entity authentication is also a common component of more complex cryptographic processes, typically instantiated by a cryptographic protocol. In this case, entity authentication is normally established at the start of a connection. An entity must provide assurance of their identity in real time in order for the extended protocol to complete satisfactorily.

3.3.3 General categories of identification information

One of the prerequisites for achieving entity authentication is that there is some means of providing information about the identity of a *claimant* (the entity that we are attempting to identify). There are several different general techniques :

- Providing identity information is not normally enough to achieve entity authentication. Entity authentication also requires a notion of freshness.
- Different techniques for providing identity information can be, and often are, combined in real security systems.
- Cryptography has a dual role in helping to provide entity authentication:
    i. Some of these approaches involve identity information that may have little to do with cryptography (such as possession of a token or a password). Cryptography can still be used to support these approaches. For example: cryptography can play a role in the secure storage of passwords.
    ii. Almost all of these approaches require a cryptographic protocol as part of their implementation.

Something The Claimant Has:

For human users, identity information can be based on something physically held by the user. This technique can also be used for providing identity information in the electronic world. Examples of mechanisms of this type include:

Dumb tokens:

Dumb means a physical device with limited memory that can be used to store identity information. Dumb tokens normally require a reader that extracts the identity information from the token and then indicates whether the information authenticates the claimant or not.

One example of a dumb token is a plastic card with a magnetic stripe. The security of the card is based entirely on the difficulty of extracting the identity information from the magnetic stripe, a reader that can extract or copy this information. Hence this type of dumb token is quite insecure.

- ➢ In order to enhance security, it is common to combine the use of a dumb token with another method of providing identification, such as one based on something the user knows.

For example, in the banking community plastic cards with magnetic stripes are usually combined with a PIN, which is a piece of identity information that is required for entity authentication but that is not stored on the magnetic stripe.

Smart cards:

A smart card is a plastic card that contains a chip, which gives the card a limited amount of memory and processing power. The advantage of this over a dumb token is that the smart card can store secret data more securely and can also conduct cryptographic computations. However, like dumb tokens, the interface with a smart card is normally through an external reader.

Smart cards are widely supported by the banking industry, where most payment cards now include a chip as well as the conventional magnetic stripe Smart cards are also widely used for other applications, such as electronic ticketing, physical access control, identity cards etc.

Smart tokens:

Smart tokens have their own user interface. This can be used, for example, to enter data such as a challenge number, for which the smart token can calculate a cryptographic response.

All types of smart token (including smart cards) require an interface to a computer system of some sort. This interface could be a human being or a processor connected to a reader. As

with dumb tokens, smart tokens are often implemented alongside another identification method, typically based on something that the user knows.

SOMETHING THE CLAIMANT IS

The field of *biometrics* is devoted to developing techniques for user identification that are based on physical characteristics of the human body.

A biometric mechanism typically converts a physical characteristic into a digital code that is stored on a database. When the user is physically presented for identification, the physical characteristic is measured by a reader, digitally encoded, and then compared with the template code on the database. Biometric measurements are often classified as either being:

Static, because they measure unchanging features such as fingerprints, hand geometry, face structure, retina and iris patterns.

Dynamic, because they measure features that (slightly) change each time that they are measured, such as voice, writing and keyboard response times.

SOMETHING THE CLAIMANT KNOWS

Basing identity information, at least partially, on something that is known to the claimant is a very familiar technique. Common examples of this type of identity information include PINs, passwords and passphrases. This is the technique most immediately relevant to cryptography since identity information of this type, as soon as it is stored anywhere on a device, shares many of the security issues of a cryptographic key.

In many applications, identity information of this type often *is* a cryptographic key. Strong cryptographic keys are usually far too long for a human user to remember and hence 'know'. This can be some good news and some potentially bad news concerning the use of cryptographic keys as identity information:

1. Most information systems consist of networks of devices and computers. These machines are much better at 'remembering' cryptographic keys than humans!

   Thus, if the claimant is a machine then it is possible that a cryptographic key can be something that is 'known'.
2. Where humans are required to 'know' a cryptographic key, they normally activate the key by presenting identity information that is easier to remember such as a PIN, password or passphrase. This reduces the effective security of that cryptographic key from that of the key itself to that of the shorter information used to activate it.

3.4 Passwords

3.4.1 Problems with passwords

Length.: Since passwords are designed to be memorised by humans, there is a natural limit to the length that they can be. This means that the *password space* (all possible passwords) is limited in size, thus restricting the amount of work required for an exhaustive search of all passwords.

Complexity.

- The full password space is rarely used in applications because humans find randomly generated passwords hard to remember. As a result we often work from highly restricted password spaces, which greatly reduces the security. This makes dictionary attacks possible, where an attacker exhaustively tries all the 'likely' passwords and hopes to eventually find the correct one.
- Clever pneumonic techniques can slightly increase the size of a usable password space.
- Moving from passwords to passphrases can improve this situation by significantly increasing the password space.

Repeatability:

- For the lifetime of a password, each time that it is used it is exactly the same. This means that if an attacker can obtain the password then there is an (often significant) period of time within which the password can be used to fraudulently claim the identity of the original owner.
- One measure that can be taken to restrict this threat is to regularly force password change. However, this again raises a usability issue since regular password change is confusing for humans and can lead to insecure password storage practices.

Vulnerability.:

The consequences of 'stealing' a password can be serious. However, passwords are relatively easy for an attacker to obtain:

- ➢ they are most vulnerable at point of entry, when they can be viewed by an attacker watching the password holder (a technique often referred to as *shoulder surfing*);

- ➢ they can be extracted by attackers during social engineering activities, where a password holder is fooled into revealing a password to an attacker who makes claims.

  for example: to be a system administrator (an attack that is sometimes known as *phishing*);
- ➢ they can be obtained by an attacker observing network traffic or by an attacker who compromises a password database.

3.4.2 Cryptographic password protection

- Consider a large organisation that wishes to authenticate many users onto its internal system using passwords.

One way of implementing this is to use a system that compares offered passwords with those stored on a centralised password database. Since this database potentially contains a complete list of account names and passwords. Even if this database is managed carefully, the administrators of the system potentially have access to this list, which may not be desirable.

- One area where cryptography can be used to help to implement an identification system based on passwords is in securing the password database.

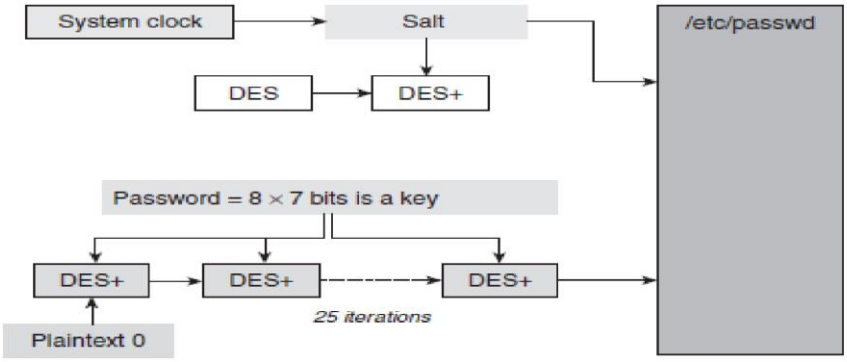
As an example of a cryptographic primitive being used in a different way to create a one-way function, below Figure illustrates the basic idea behind the function that was used in many early UNIX operating systems for password database protection.

- In the password database in the UNIX system, often identified by */etc/passwd*, every user has an entry that consists of two pieces of information:

Salt. This is a 12-bit number randomly generated using the system clock. The salt is used to uniquely modify the DES encryption algorithm nin a subtle way. We denote the result of this unique modification by DES+.

Password image. This is the result that is output after doing the following:

1. Convert the 8 ASCII character password into a 56-bit DES key. This is straightforward, since each ASCII character consists of 7 bits.



One-way function for UNIX password protection

2. Encrypt the plaintext consisting of all zeros (64 zero bits) using the uniquely modified DES+ with the 56-bit key derived from the password.
3. Repeat the last encryption step 25 times (in other words, we encrypt the all zero string 25 times). This last step is designed to slow the operation down in such a way that it is not inconvenient for a user, but much more difficult for an attacker conducting a dictionary attack.

When a user enters their password, the system looks up the salt, generates the modified DES+ encryption algorithm, forms the encryption key from the password, and then conducts the multiple encryption to produce the password image. The password image is then checked against the version stored in */etc/passwd*. If they match then the password is accepted.

3.5 Dynamic password schemes

Two of the main problems with passwords are vulnerability (they are quite easy to steal) and repeatability (once stolen they can be reused).

A ***dynamic password scheme***, also often refered to as a *one-time password scheme*, preserves the concept of a password but greatly improve its security by:

1. limiting the exposure of the password, thus reducing vulnerability;

2. using the password to generate dynamic data that changes on each authentication attempt, thus preventing repeatability.

3.5.1 Idea behind dynamic password schemes

A dynamic password scheme uses a 'password function' rather than a password. If a claimant, is a human user, wants to authenticate to a device, such as an authentication server, then the user inputs some data into the function to compute a value that is sent to the device. There are thus two components that we need to specify:

The password function: Since this function is a cryptographic algorithm, it is usually implemented on a smart token. In the example, we will assume that this is a smart token with an input interface that resembles a small calculator.

The input: We want the user and the device to agree on an input to the password function, the result of which will be used to authenticate the user. Since the input must be fresh, any of the freshness mechanisms could be used. All of these techniques are deployed in different commercial devices, namely:

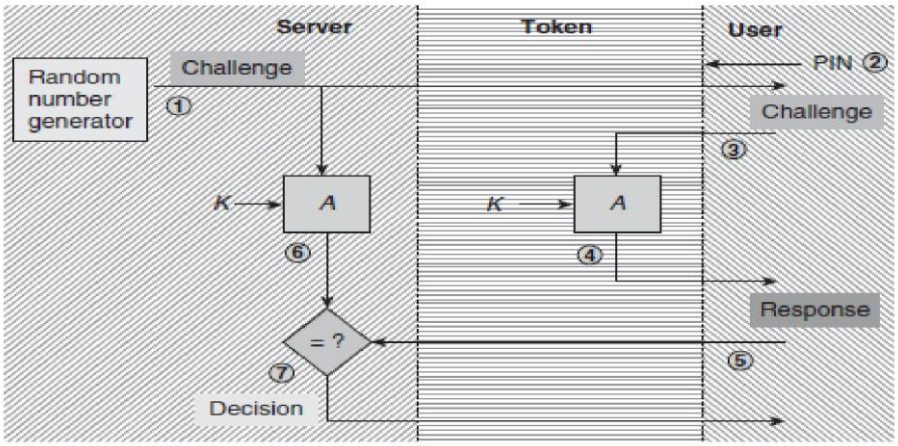
- ***Clock-based***. The user and the device have synchronised clocks and thus the current time can be used to generate an input that both the user and the device will 'understand'.
- ***Sequence numbers*.** The user and the device both maintain synchronised sequence numbers.
- ***Nonce-based***. The device randomly generates a number, known as a *challenge*, and sends it to the user, who computes a cryptographic *response*. Such mechanisms are often referred to as *challenge–response* mechanisms.

3.5.2 Example dynamic password scheme

Figure shows an authentication attempt using this dynamic password scheme:

1. The server randomly generates a challenge and sends it to the user. It is possible that the user first sends a message to the server requesting that the server send them a challenge.
2. The user authenticates themselves to the token using the PIN.
3. If the PIN is correct then the token is activated. The user then uses the token interface by means of a keypad to enter the challenge into the token.
4. The token uses the password function to compute a response to the challenge. If algorithm *A* is an encryption algorithm then the challenge can be regarded as a plaintext and the response is the ciphertext that results from applying encryption algorithm *A* using key *K*. The token displays the result to the user on its screen.
5. The user sends this response back to the server. This step might involve the user reading the response off the screen of the token and then typing it into a computer that is being used to access the authentication server.
6. The server checks that the challenge is still valid. If it is still valid, the server inputs the challenge into the password function and computes the response, based on the same algorithm *A* and key *K*.

7. The server compares the response that it computed itself with the response that was sent by the user. If these are identical then the server authenticates the user, otherwise the server rejects the user.



Example of a dynamic password scheme based on challenge–response

ANALYSIS OF DYNAMIC PASSWORD SCHEME:

There can be several significant improvements:

Local use of PIN:

With regard to security at the user end, the main difference is that the user uses the PIN to authenticate themselves to a small portable device that they have control over. The chances of the PIN being viewed by an attacker while it is being entered are lower than for applications where a user has to enter a PINinto a device not under their control, such as an ATM. Also, the PIN is only transferred from the user's fingertips to the token and does not then get transferred to any remote server.

Two factors. Without access to the token, the PIN is useless. Thus another improvement is that we have moved from *one-factor* authentication (something the claimant knows, namely the password) to *two-factor* authentication (something the claimant knows, namely the PIN, *and* something the claimant has, namely the token).

Dynamic responses. The biggest security improvement is that every time an authentication attempt is made, a different challenge is issued and therefore a different response is needed. Of course, because the challenge is randomly generated there is a *very small* chance that the same challenge is issued on two separate occasions. But assuming that a good source of randomness is used then this chance is so low that we can dismiss it. Hence anyone who

succeeds in observing a challenge and its corresponding response cannot use this to masquerade as the user at a later date.

3.6 Zero-knowledge mechanisms

3.6.1 Motivation for zero-knowledge:

- **Requirement for mutual trust**. Firstly, they are all based on some degree of trust between the entities involved.

For example, passwords often require the user to agree with the server on use of a password, even if the server only stores a hashed version of the password. However, there are situations where entity authentication might be required between two entities who are potential adversaries and do not trust one another enough to share *any* information.

- **Leaking of information**. Secondly, they all give away some potentially useful information on each occasion that they are used. Conventional passwords are catastrophic in this regard since the password is fully exposed when it is entered, and in some cases may even remain exposed when transmitted across a network. Our example dynamic password scheme is much better, but does reveal valid challenge–response pairs each time that it is run.

➢ Entity authentication could be provided in such a way that no shared trust is necessary and *no knowledge at all* is given away during an authentication attempt.

➢ The requirement for a zero-knowledge mechanism is that one entity (the *prover*) must be able to provide assurance of their identity to another entity (the *verifier*) in such a way that it is impossible for the verifier to later impersonate the prover, even after the verifier has observed and verified many different successful authentication attempts.

3.6.2 Zero-knowledge analogy

Consider a popular analogy in which we will play the role of verifier. The setting is a cave shaped in a loop with a split entrance, as depicted in Figure 8.4.

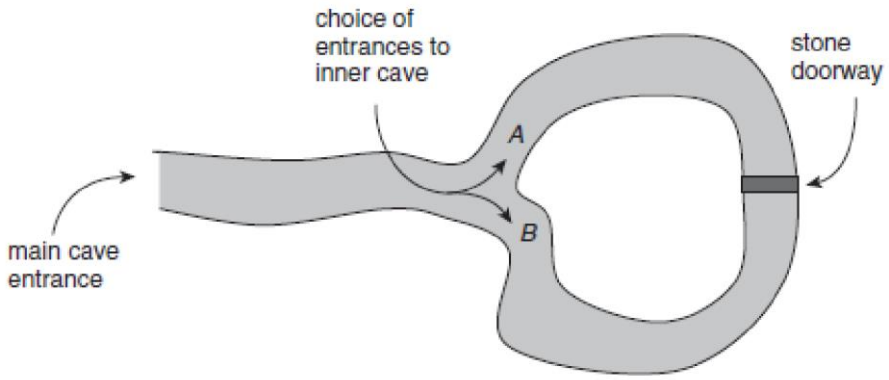


Figure 8.4. Popular analogy of a zero-knowledge mechanism

The back of the cave is blocked by a stone doorway that can only be opened by using a secret key phrase. We wish to hire a guide to make a circular tour of the entire cave system but need to make sure in advance that our guide knows the key phrase, otherwise we will not be able to pass through the doorway. The guide, who will be our prover (the entity authentication claimant), is not willing to tell us the key phrase, otherwise there is a risk that we might go on our own tour without hiring him. Thus we need to devise a test of the guide's knowledge before we agree to hire him.

The guide has a further concern. For all he knows, we are from a rival guiding company and are trying to learn the key phrase. He wants to make sure that no matter how rigorous a test is run, we will not learn *anything* that could help us to try to work out what the key phrase is. i. e he wants to make sure that the test is a zero-knowledge mechanism that verifies his claim to know the key phrase.

So here is what we do:

1. We wait at the main cave entrance and send the guide down the cave to the place where it splits into two tunnels, labelled *A* and *B*. We cannot see this tunnel split from the main entrance, so we send a trusted observer down with him, just to make sure he does not cheat during the test.
2. The guide randomly picks a tunnel entrance and proceeds to the stone door.
3. We toss a coin. If it is heads then we shout down the cave that we want the guide to come out through tunnel *A*. If it is tails then we shout down the cave that we want the guide to come out through tunnel *B*.
4. The observer watches to see which tunnel the guide emerges from.

Suppose we call heads (tunnel *A*). If the guide comes out of tunnel *B* (the wrong entrance) then we decide not to hire him since he does not appear to know the key phrase. However, if he comes out of tunnel *A* (the correct entrance) then one of two things have happened:

- The guide got lucky and chose tunnel *A* in the first place. In that case he just turned back, whether he knows the key phrase or not. In this case we learn nothing.
- The guide chose tunnel *B*. When we called out that he needed to come out of tunnel *A*, he used the key phrase to open the door and crossed into tunnel *A*. In this case the guide has demonstrated knowledge of the key phrase.

So if the guide emerges from tunnel *A* then there is a 50% chance that he has just demonstrated knowledge of the key phrase. The problem is that there is also a chance that he got lucky.

So we run the test again. If he passes a second test then the chances that he got lucky twice are now down to 25%, since he needs to get lucky in both independent tests. Then we run the test again, and again. If we run *n* such independent tests and the guide passes them all, then the probability that the guide does not know the key phrase is:

$$\frac{1}{2} \times \frac{1}{2} \times \cdots \times \frac{1}{2} = \left(\frac{1}{2}\right)^n = \frac{1}{2^n}.$$

Thus we need to insist on *n* tests being run, where $1/2^n$ is sufficiently small that we will be willing to accept that the guide almost certainly has the secret knowledge.

Meanwhile, the guide will have done a great deal of walking around the cave system and using the key phrase, without telling us any information about the key phrase. So the guide will also be satisfied with the outcome of this process.

Cryptographic Protocols

9.1 Protocol basics

9.1.1 Operational motivation for protocols

Many applications:

- **Have complex security requirements**. For example, if we wish to transmit some sensitive information across an insecure network then there should be confidentiality *and* data origin authentication guarantees .
- **Involve different data items with different security requirements**. Most applications involve different pieces of data, each of which may have different security requirements.

  For example, an application processing an online transaction may require the purchase details (product, cost) to be authenticated, but not encrypted, so that this

information is widely available. However, the payment details (card number, expiry date) are likely to be required to be kept confidential. It is also possible that different requirements of this type arise for efficiency reasons, since all cryptographic computations (particularly public-key computations) have an associated efficiency cost. It can thus be desirable to apply cryptographic primitives only to those data items that strictly require a particular type of protection.

- **Involve information flowing between more than one entity**. It is rare for a cryptographic application to involve just one entity, such as when a user encrypts a file for storage on their local machine. Most applications involve at least two entities exchanging data.

  For example, a card payment scheme may involve a client, a merchant, the client's bank and the merchant's bank (and possibly other entities).

- **Consist of a sequence of logical (conditional) events**. Real applications normally involve multiple operations that need to be conducted in a specific order, each of which may have its own security requirements.

  For example, it does not make any sense to provide confidentiality protection for the deduction of a sum from a client's account and issue some money from a cash machine until entity authentication of the client has been conducted.

9.1.2 Components of a cryptographic protocol

A cryptographic protocol needs to specify:

The protocol assumptions – any prerequisite assumptions concerning the environment in which the protocol will be run. This involves assumptions about the entire environment (including, for example, security of devices used in the protocol). *What needs to have happened before the protocol is run*?

The protocol flow – the sequence of communications that need to take place between the entities involved in the protocol. Each message is often referred to as being a *step* or *pass* of the protocol. *Who sends a message to whom, and in which order*?

The protocol messages – the content of each message that is exchanged between two entities in the protocol. *What information is exchanged at each step*?

The protocol actions – the details of any actions (operations) that an entity needs to perform after receiving, or before sending, a protocol message. *What needs to be done between steps*?

9.2 From objectives to a protocol

9.2.1 Stages of protocol design

There are three main stages to the process of designing a cryptographic protocol:

- **Defining the objectives**. This is the problem statement, which identifies what the problem is that the protocol is intended to solve. Particularly performance-related objectives.
- **Determining the protocol goals**. This stage translates the objectives into a set of clear cryptographic requirements. The protocol goals are typically statements of the form *at the end of the protocol, entity X will be assured of security service Y*. We will see some examples shortly.

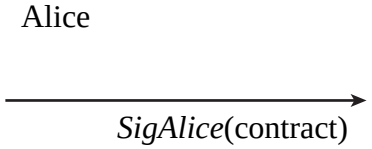


Figure 9.1. A simple cryptographic protocol providing non-repudiation

- **Specifying the protocol**. This takes the protocol goals as input and involves determining some cryptographic primitives, message flow and actions that achieve these goals.

A very simple example of these stages would be the following:

Defining the objectives. Merchant Bob wants to make sure that a contract that he will receive from Alice cannot later be denied.

Determining the protocol goals. At the end of the protocol Bob requires non- repudiation of the contract received from Alice.

Specifying the protocol. A protocol to achieve this simple goal is given in Figure 9.1. In this protocol there is only one message, which is sent from Alice to Bob. This message consists of the contract, digitally signed by Alice. The notation Sig_{Alice}represents a generic digital signature algorithm. We assume that if a digital signature scheme with appendix is used then part of Sig_{Alice}(contract) is a plaintext version of the contract.

9.3 Analysing a simple protocol

9.3.1 A simple application

THE OBJECTIVES

In this scenario we suppose that Alice and Bob have access to a common network. Periodically, at any time of his choosing, Bob wants to check that Alice is still ‘alive’ and

connected to the network. This is main security objective, which will be referred to as a check of liveness.

Lets assume that Alice and Bob are just two entities in a network consisting of many such entities, all of whom regularly check the liveness of one another, perhaps every few seconds. Thus set a secondary security objective that whenever Bob receives any confirmation of liveness from Alice, he should be able to determine precisely which liveness query she is responding to.

THE PROTOCOL GOALS

Whenever Bob wants to check that Alice is alive he will need to send a *request* to Alice, which she will need to respond to with a *reply*.

At the end of any run of a suitable cryptographic protocol, the following three goals should have been met:

1. **Data origin authentication of Alice's reply**. If this is not provided then Alice may not be alive since the reply message might have been created by an attacker.
2. **Freshness of Alice's reply.** If this is not provided then, even if there is data origin authentication of the reply, this could be a replay of a previous reply.

   In other words, an attacker could observe a reply that Alice makes when she *is* alive and then send a copy of it to Bob at some stage after Alice has expired. This would be a genuine reply created by Alice. But she would not be alive and hence the protocol will have failed to meet its objectives.
3. **Assurance that Alice's reply corresponds to Bob's request**. If this is not provided then it is possible that Bob receives a reply that corresponds to a different request (either one of his own, or of another entity in the network).

Table 9.1: Notation used during protocol descriptions

r_B	A nonce generated by Bob
$\|$	Concatenation
Bob	An *identifier* for Bob (perhaps his name)
$MAC_K(data)$	A MAC computed on *data* using key *K*
$E_K(data)$	Symmetric encryption of *data* using key *K*
$Sig_A(data)$	A digital signature on *data* computed by Alice
T_A	A timestamp generated by Alice
T_B	A timestamp generated by Bob
ID_S	A session identifier

CANDIDATE PROTOCOLS

Figure 9.2 shows the protocol flow and messages of our first candidate protocol.

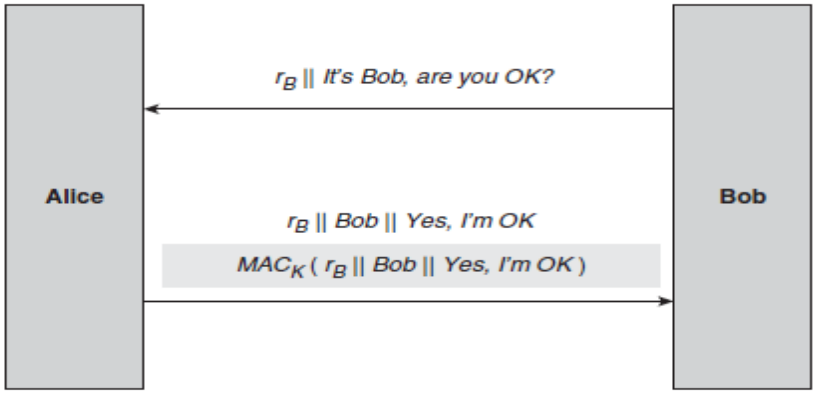


Figure 9.2. Protocol 1

PROTOCOL ASSUMPTIONS

There are three assumptions that we make before running this protocol:

1. **Bob has access to a source of randomness**. This is necessary because the protocol requires Bob to be able to generate a nonce and also assume that this generator is 'secure' in order to guarantee unpredictability of the output.
2. **Alice and Bob already share a symmetric key *K* that is known only to them**. This is necessary because the protocol requires Alice to be able to generate a MAC that Bob can verify.
3. **Alice and Bob agree on the use of a strong MAC algorithm**. This is necessary because if the MAC algorithm is flawed then data origin authentication is not necessarily provided by it.

If Alice and Bob do not already share a symmetric key then they will need to first run a different protocol in order to establish a common symmetric key *K* . If Alice and Bob have not already agreed on the use of a strong MAC algorithm to compute the MAC then Alice could indicate the choice of MAC algorithm that she is using in her *reply*.

PROTOCOL DESCRIPTION

Protocol 1 consists of the following steps:

1. Bob conducts the following steps to form the request:
   a) Bob generates a nonce *rB* (this is an implicit action that is not described in Figure 9.2.
   b) Bob concatenates *rB* to the text *It's Bob, are you OK?*. This combined data string is the request.
   c) Bob sends the request to Alice.
2. Assuming that she is alive and able to respond, Alice conducts the following steps to form the reply:
   a) Alice concatenates the nonce *rB* to identifier *Bob* and the text *Yes, I'm OK*. We will refer to this combined data string as the *reply text*.
   b) Alice computes a MAC on the reply text using key *K* (this is an implicit action). The reply text is then concatenated to the MAC to form the reply.
   c) Alice sends the reply to Bob.
3. On receipt of the reply, Bob makes the following checks.:
   a) Bob checks that the received reply text consists of a valid *rB* (which he can recognise because he generated it and has stored it on a local database) concatenated to his identifier *Bob* and a meaningful response to his query (in this case, *Yes, I'm OK*).
   b) Bob computes a MAC on the received reply text with key *K* (which he shares with Alice) and checks to see if it matches the received MAC.

c) If both of these checks are satisfactory then Bob accepts the reply and ends the protocol. We say that the protocol successfully *completes* if this is the case.

PROTOCOL ANALYSIS

Protocol 1 meets the required goals:

a) **Data origin authentication of Alice's reply**. Under second assumption, the only entity other than Bob who can compute the correct MAC on the reply text is Alice. Thus, given that the received MAC is correct, the received MAC must have been computed by Alice. Thus Bob indeed has assurance that the reply (and by implication the reply text) was generated by Alice.

b) **Freshness of Alice's reply**. The reply text includes the nonce r_B, which Bob generated at the start of the protocol. Thus, by the principles, the reply is fresh.

c) **Assurance that Alice's reply corresponds to Bob's request**.

There are two pieces of evidence in the reply that provide this:

- Firstly, and most importantly, the reply contains the nonce *rB*, which Bob generated for this run of the protocol. By our first protocol assumption, this nonce is very unlikely to ever be used for another protocol run, thus the appearance of *rB* in the reply makes it almost certain that the reply corresponds to his request.
- The reply contains the identifier *Bob*.

Four of the components of a cryptographic protocol:

- **The protocol assumptions**. If the protocol assumptions do not hold then, even when the protocol successfully completes, the security goals are not met.

  For example, if a third entity Charlie also knows the MAC key *K* then Bob cannot be sure that the reply comes from Alice, since it could have come from Charlie.
- **The protocol flow**. Clearly the two messages in this protocol must occur in the specified order, since the reply cannot be formed until the request is received.
- **The protocol messages**. The protocol goals are not necessarily met if the content of the two messages is changed in any way.
- **The protocol actions**. The protocol goals are not met if any of the actions are not undertaken.

  For example, if Bob fails to check that the MAC on the reply text matches the received MAC then he has no guarantee of the origin of the reply.

9.3.3 Protocol 2

Figure 9.3 shows the protocol flow and messages of our second candidate protocol.

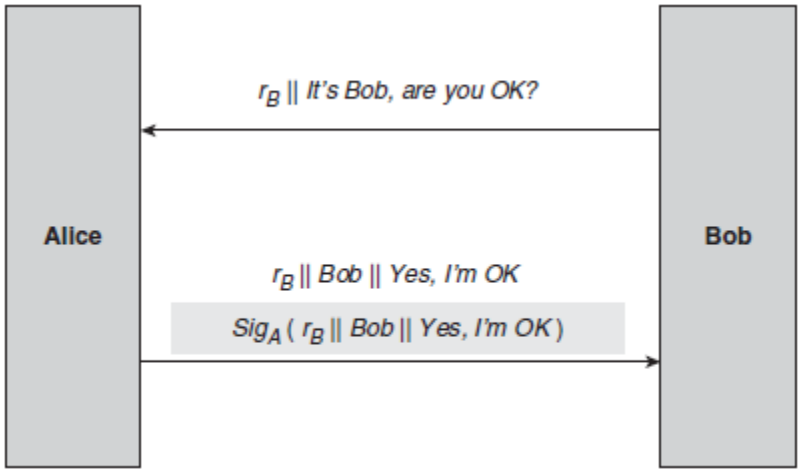


Figure 9.3. Protocol 2

PROTOCOL ASSUMPTIONS

As can be seen from Figure 9.3, Protocol 2 is very similar to Protocol 1. In fact, it is in the protocol assumptions that the main differences lie:

Bob has access to a source of randomness. As for Protocol 1.

Alice has been issued with a signature key and Bob has access to a verification key corresponding to Alice's signature key. This is the digital signature scheme equivalent of the second assumption for Protocol 1.

Alice and Bob agree on the use of a strong digital signature scheme.

PROTOCOL DESCRIPTION

The description of Protocol 2 is exactly as for Protocol 1, except that:

Instead of computing a MAC on the reply text, Alice digitally signs the reply text using her signature key.

Instead of computing and comparing the received MAC on the reply text, Bob verifies Alice's digital signature on the reply text using her verification key.

PROTOCOL ANALYSIS

The analysis of Protocol 2 is exactly as for Protocol 1, except for:

Data origin authentication of Alice's reply. Under second assumption, the only entity who can compute the correct digital signature on the reply text is Alice. Thus, given that her digital signature is verified, the received digital signature must have been computed by Alice.

Thus Bob indeed has assurance that the reply (and by implication the reply text) was generated by Alice.

Therefore deduce that Protocol 2 also meets the three security goals.

REMARKS

Protocol 2 can be thought of as a public-key analogue of Protocol 1. So which one is better?

- It could be argued that, especially in resource-constrained environments, Protocol 1 has an advantage in that it is more computationally efficient, since computing MACs generally involves less computation than signing and verifying digital signatures.
- However, it could also be argued that Protocol 2 has the advantage that it could be run between an Alice and Bob who have not pre-shared a key, so long as Bob has access to Alice's verification key.

9.3.4 Protocol 3

PROTOCOL ASSUMPTIONS

These are identical to Protocol 1.

PROTOCOL DESCRIPTION

This is identical to Protocol 1, except that in Protocol 3 the identifier *Bob* is omitted from the reply text.

PROTOCOL ANALYSIS

This is identical to Protocol 1, except for:

Assurance that Alice's reply corresponds to Bob's request.

As argued for Protocol 1, the inclusion of the nonce rB in the reply appears, superficially, to provide this assurance since rB is in some sense a unique identifier of Bob's request. However, there is an attack that can be launched against Protocol 3 in certain environments which shows that this is not always true. Since the attacker plays the role of a 'mirror', and this is a reflection *attack* against Protocol 3. The attack is depicted in Figure 9.5.

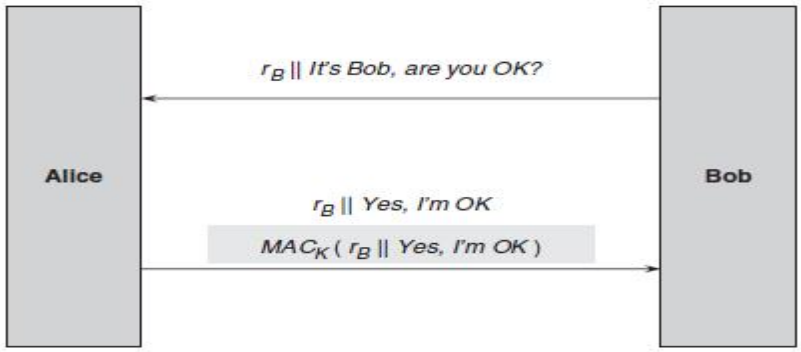


Figure 9.4. Protocol 3

The reflection attack working:

Lets assume that an attacker is able to intercept and block all communication between Alice and Bob and also assume that Bob normally recognises that incoming traffic may be from Alice through the use of the channel, rather than an explicit identifier. This is perhaps an unreasonable assumption, but we are trying to keep it simple. Thus, even if Alice is no longer alive, the attacker can pretend to be Alice by sending messages on this channel. The reflection attack works as follows:

1. Bob initiates a run of Protocol 3 by issuing a request message.
2. The attacker intercepts the request message and sends it straight back to Bob, except that the text It's Bob is replaced by the text It's Alice.
3. At this point it is to suggest that Bob will regard the receipt of a message containing his nonce rB as rather strange and will surely reject it. However, resist the temptation to anthropomorphise the analysis of a cryptographic protocol and recall that in most applications of this type of protocol both Alice and Bob will be computing devices following programmed instructions. In this case Bob will simply see a request message that appears to come from Alice and, since he is alive, will compute a corresponding reply message. He then sends this reply to Alice.
4. The attacker intercepts this reply message and sends it back to Bob.
5. Bob, who is expecting a reply from Alice, checks that it contains the expected fields and that the MAC is correct. Of course it is, because he computed it himself!

The reflection attack described in Figure 9.5 as two nested runs of Protocol 3:

• The first run is initiated by Bob, who asks if Alice is alive. He thinks that he is running it with Alice, but instead he is running it with the attacker.

• The second run is initiated by the attacker, who asks if Bob is alive. Bob thinks that this request is from Alice, but it is from the attacker. Note that this run of Protocol 3 begins after the first run of the protocol has begun, but completes before the first run ends.

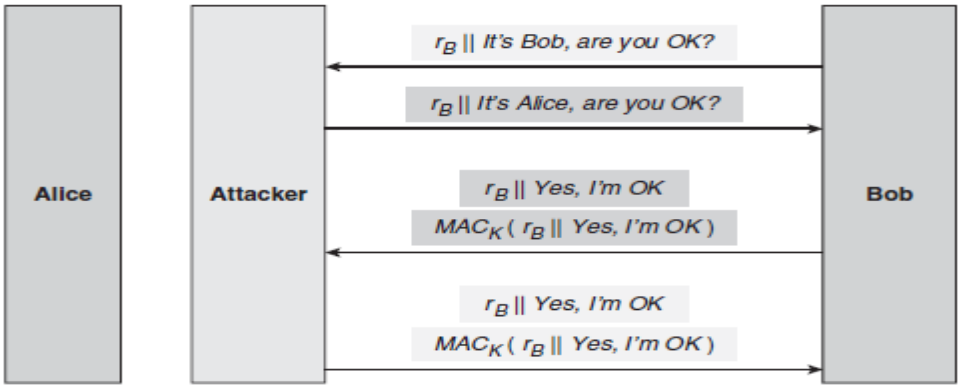


Figure 9.5. Reflection attack against Protocol 3

A better response would be to repair Protocol 3. There are two options:

1. **Include an action to check for this attack:**

This would involve Bob keeping a note of all Protocol 3 sessions that he currently has open. He should then check whether any request messages that he receives match any of his own open requests.

2. **Include an identifier.**

Include some sort of identifier in the reply that prevents the reflection attack from working. There is no point in doing so in the request since it is unprotected and an attacker could change it without detection.

9.3.5 Protocol 4

Figure 9.6 shows the protocol flow and messages of our fourth candidate protocol.

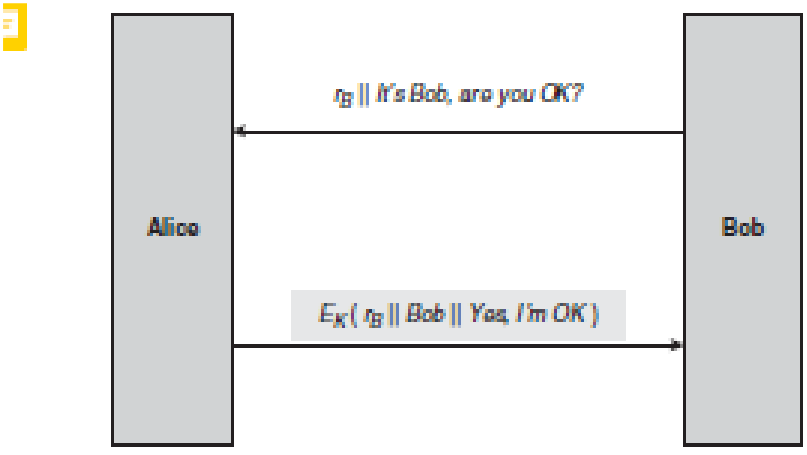


Figure 9.6. Protocol 4

PROTOCOL ASSUMPTIONS

These are identical to Protocol 1, expect that we assume that Alice and Bob have agreed on the use of a strong symmetric encryption algorithm E (rather than a MAC).

PROTOCOL DESCRIPTION

The description of Protocol 4 is exactly as for Protocol 1, except that:

- Instead of computing a MAC on the reply text, Alice uses E to encrypt the reply text using key K.
- Alice does not send the reply text to Bob.
- Instead of computing and comparing the received MAC on the reply text, Bob simply decrypts the received encrypted reply text.

PROTOCOL ANALYSIS

The analysis of Protocol 4 is exactly as for Protocol 1, except for the issue of data origin authentication of Alice's reply. Consider whether encryption can be used in this context to provide data origin authentication.

There are two arguments:

The case against:

This is perhaps the purist's viewpoint. Protocol 4 does not provide data origin authentication because encryption does not, in general, provide data origin authentication.

The case for:

Problems that may arise if encryption is used to provide data origin authentication. These mainly arose when the plaintext was long and unformatted. In this case the reply text is short and has a specific format. Thus, if a block cipher such as AES is used then it is possible that the reply text is less than one block long, hence no 'block manipulation' will be possible. Even if the reply text is two blocks long and ECB mode is used to encrypt these two blocks, the format of the reply text is specific and any manipulation is likely to be noticed by Bob (assuming of course that he checks for it).

9.3.6 Protocol 5

Protocol 5, depicted in Figure 9.7, is very similar to Protocol 1, except that the nonce generated by Bob is replaced by a timestamp generated by Bob.

PROTOCOL ASSUMPTIONS

These are the same as the assumptions for Protocol 1, except that the need for Bob to have a source of randomness is replaced by:

Bob can generate and verify integrity-protected timestamps:

This requires Bob to have a system clock. Requiring T_B to be integrity-protected means that it cannot be manipulated by an attacker without subsequent detection of this by Bob.

PROTOCOL DESCRIPTION : The description of Protocol 5 is exactly as for Protocol 1, except that:

- Instead of generating a nonce rB, Bob generates an integrity-protected timestamp T_B. This is then included in both the request (by Bob) and the reply (by Alice).
- As part of his checks on the reply, Bob checks that the reply text includes T_B.

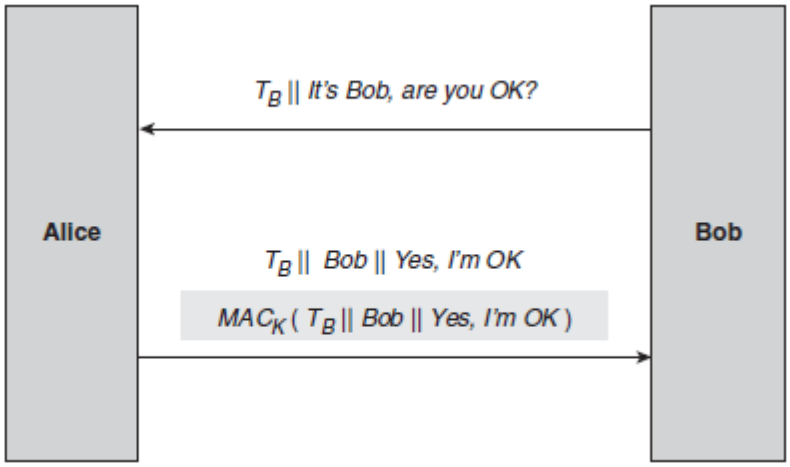


Figure 9.7. Protocol 5

PROTOCOL ANALYSIS

The analysis of Protocol 5 is similar to Protocol 1.

Data origin authentication of Alice's reply. As for Protocol 1.

Freshness of Alice's reply. The reply text includes the timestamp TB, which Bob generated at the start of the protocol. The reply is fresh.

Assurance that Alice's reply corresponds to Bob's request. There are two pieces of evidence in the reply that provide this:

1. The reply contains the timestamp T_B, which Bob generated for this run of the protocol. Assuming that the timestamp is of sufficient granularity that it is not possible for Bob to have issued the same timestamp for different protocol runs (or that it includes a unique session identifier), the presence of T_B indicates that the reply matches the request.
2. The reply contains the identifier Bob, preventing reflection attacks. Thus Protocol 5 meets the three security goals.

Thus Protocol 5 meets the three security goals.

REMARKS

Protocol 5 can be thought of as the 'clock-based' analogue of Protocol 1.

No need for Alice to share a synchronised clock with Bob for Protocol 5 to work. This is because only Bob requires freshness, hence it suffices that Alice includes Bob's timestamp without Alice necessarily being able to 'make sense' of, let alone verify, it.

One consequence of this is that it is important that T_B is integrity-protected. To see this, suppose that T_B just consists of the time on Bob's clock, represented as an unprotected timestamp (perhaps just a text stating the time). In this case the following attack is possible:

1. At 15.00, the attacker sends Alice a request that appears to come from Bob but has T_B set to the time 17.00, which is a time in the future that the attacker anticipates that Bob will contact Alice.
2. Alice forms a valid reply based on T_B being 17.00 and sends it to Bob.
3. The attacker intercepts and blocks the reply from reaching Bob, then stores it.
4. The attacker hits Alice over the head with a blunt instrument. (Less violent versions of this attack are possible!)
5. At 17.00, Bob sends a genuine request to Alice (recently deceased).
6. The attacker intercepts the request and sends back the previously intercepted reply from Alice.
7. Bob accepts the reply as genuine (which it is) and assumes that Alice is OK (which she most definitely is not). This attack is only possible because, in this example, we allowed the attacker to 'manipulate' T_B. By assuming that T_B is a timestamp that cannot be manipulated in such a way, this attack is impossible.

9.3.7 Protocol 6

Protocol 6 is shown in Figure 9.8.

PROTOCOL ASSUMPTIONS These are the same as the assumptions for Protocol 1, except that the need for Bob to have a random generator is replaced by:

Alice can generate timestamps that Bob can verify. As part of this assumption we further require that Alice and Bob have synchronised clocks

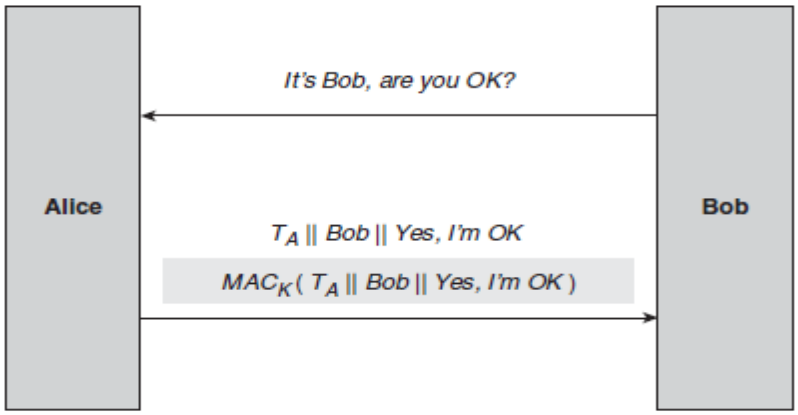


Figure 9.8. Protocol 6

PROTOCOL DESCRIPTION

The description of Protocol 6 is slightly different from Protocol 1, so we will explain it in more detail than the last few protocols.

1. Bob conducts the following steps to form the request:
    a) Bob forms a simplified request message that just consists of the text It's Bob, are you OK?.
    b) Bob sends the request to Alice.
2. Assuming that she is alive and able to respond, Alice conducts the following steps to form the reply:
    (a) Alice generates a timestamp T_A and concatenates it to identifier Bob and the text Yes, I'm OK, to form the reply text.
    (b) Alice computes a MAC on the reply text using key K. The reply text is then concatenated to the MAC to form the reply.
    (c) Alice sends the reply to Bob.
3. On receipt of the reply, Bob makes the following checks:
    a) Bob checks that the received reply text consists of a timestamp TA concatenated to his identifier Bob and a meaningful response to his query (in this case, Yes, I'm OK).
    b) Bob verifies TA and uses his clock to check that it consists of a fresh time.
    c) Bob computes a MAC on the received reply text with key K and checks to see if it matches the received MAC.
    d) If both these checks are satisfactory then Bob accepts the reply and ends the protocol.

PROTOCOL ANALYSIS

The analysis of Protocol 6 is similar to Protocol 1.

Data origin authentication of Alice's reply. As for Protocol 1.

Freshness of Alice's reply. The reply text includes the timestamp T_A. The reply is fresh.

Assurance that Alice's reply corresponds to Bob's request. Unfortunately this is not provided, since the request does not contain any information that can be used to uniquely identify it. Thus Protocol 6 does not meet all three security goals.

REMARKS

Protocol 6 has only failed on a technicality. It could easily be 'repaired' by including a unique session identifier in the request message, which could then be included in the reply text.

9.3.8 Protocol 7

Seventh protocol variant is closely related to Protocol 6, and is depicted in Figure 9.9.

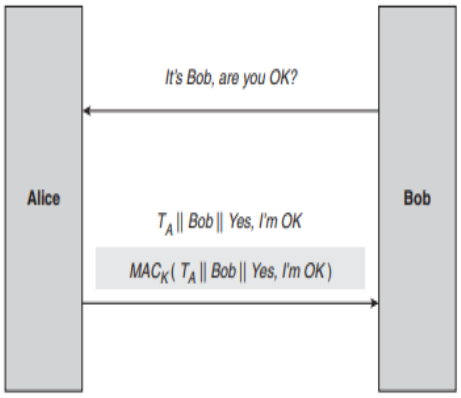


Figure 9.8. Protocol 6

PROTOCOL ASSUMPTIONS

These are the same as the assumptions for Protocol 6.

PROTOCOL DESCRIPTION

The description of Protocol 7 is almost the same as Protocol 6. The only differences are:

- Bob includes a unique session identifier IDS in the request, which Alice includes in the reply text. This identifier is not necessarily randomly generated (unlike the nonces that were used in some of the previous variants).
- The reply text that is sent in the clear by Alice differs from the reply text on which Alice computes the MAC. The difference is that T_A is included in the latter, but not the former.

PROTOCOL ANALYSIS

The analysis of Protocol 7 is similar to Protocol 6. The inclusion of the session identifier IDS is intended to remove the concerns about linking the reply to the request. The omission of T_A from the reply text that is sent in the clear at first just looks like a saving in bandwidth, since:

- Alice and Bob have synchronised clocks, by our assumptions,
- it is not strictly necessary that the data on which the MAC is computed matches the reply text, so long as Bob receives all the critical data that he needs to check the MAC.

Problem:

Bob does not know T_A. Even if they have perfectly synchronised clocks, the time that Alice issues T_A will not be the same time that Bob receives the message due to communication delays. Thus Bob does not know all the reply text on which the MAC is computed, and hence cannot verify the MAC to obtain data origin authentication. The only option is for Bob to check all the possible timestamps T_A within a reasonable window and hope that he finds one that matches. While this is inefficient, it is worth noting that this technique is sometimes used in real applications to cope with time delays and clock drift.

REMARKS

Protocol 7 is easily fixed by including T_A in both versions of the reply text, as is done in Protocol 6. Nonetheless, this protocol flaw demonstrates how sensitive cryptographic protocols are to even the slightest 'error' in their formulation.

9.4 Authentication and key establishment protocols

AKE protocols (authentication and key establishment):

The two main security objectives of an AKE protocol are always:

Mutual entity authentication:

Occasionally just unilateral entity authentication.

Establishment of a common symmetric key:

Regardless of whether symmetric or public-key techniques are used to do this. It should not come as a surprise that these two objectives are required together in one protocol.

Need to authenticate key holders:

Key establishment makes little sense without entity authentication. It is hard to imagine any applications where we would want to establish a common symmetric key between two parties without at least one party being sure of the other's identity. Indeed, in many applications mutual entity authentication is required. The only argument for not featuring entity

authentication in a key establishment protocol is for applications where the authentication has already been conducted prior to running the key establishment protocol.

Prolonging authentication:

The result of entity authentication can be prolonged by simultaneously establishing a symmetric key. A problem with entity authentication is that it is achieved only for an instant in time.

9.4.1 Typical AKE protocol goals

Mutual entity authentication:

Alice and Bob are able to verify each other's identity to make sure that they know with whom they are establishing a key.

Mutual data origin authentication: Alice and Bob are able to be sure that information being exchanged originates with the other party and not an attacker.

Mutual key establishment: Alice and Bob establish a common symmetric key.

Key confidentiality: The established key should at no time be accessible to any party other than Alice and Bob.

Key freshness: Alice and Bob should be happy that (with high probability) the established key is not one that has been used before.

Mutual key confirmation: Alice and Bob should have some evidence that they both end up with the same key.

Unbiased key control: Alice and Bob should be satisfied that neither party can unduly influence the generation of the established key.

9.4.2 Diffie–Hellman key agreement protocol

IDEA BEHIND THE DIFFIE–HELLMAN PROTOCOL

The Diffie–Hellman protocol requires the existence of:

- A public-key cryptosystem with a special property, We denote the public and private keys of Alice and Bob in this cryptosystem by (P_A, S_A) and (P_B, S_B), respectively. These may be temporary key pairs that have been generated specifically for this protocol run, or could be long-term key pairs that are used for more than one protocol run.
- A combination function F with a special property. By a 'combination' function, means a mathematical process that takes two numbers x

and y as input, and outputs a third number which we denote F(x, y). Addition is an example of a combination function, with F(x, y) = x + y.

The Diffie–Hellman protocol is designed for environments where secure channels do not yet exist, it is often used to establish a symmetric key, which can then be used to secure such a channel.

The basic idea behind the Diffie–Hellman protocol is that:

1. Alice sends her public key P_A to Bob.
2. Bob sends his public key P_B to Alice.
3. Alice computes $F(S_A, P_B)$. Note that only Alice can conduct this computation, since it involves her private key S_A.
4. Bob computes $F(S_B, P_A)$. Note that only Bob can conduct this computation, since it involves his private key S_B. The special property for the public-key cryptosystem and the combination function F is that $F(S_A, P_B) = F(S_B, P_A)$.

At the end of the protocol Alice and Bob will thus share this value, which we denote Z_{AB}, this shared value Z_{AB} can then easily be converted into a key of the required length. Since the private keys of Alice and Bob are both required to compute Z_{AB}, it should only be computable by Alice and Bob, and not anyone else (an attacker) who observed the protocol messages. Note that this is true despite the fact that the attacker will have seen P_A and P_B.

INSTANTIATION OF THE DIFFIE–HELLMAN PROTOCOL

For ElGamal, choose two public system wide parameters:

• a large prime p, typically 1024 bits in length;

• a special number g (a primitive element).

The Diffie–Hellman protocol is shown in Figure 9.10 and proceeds as follows.

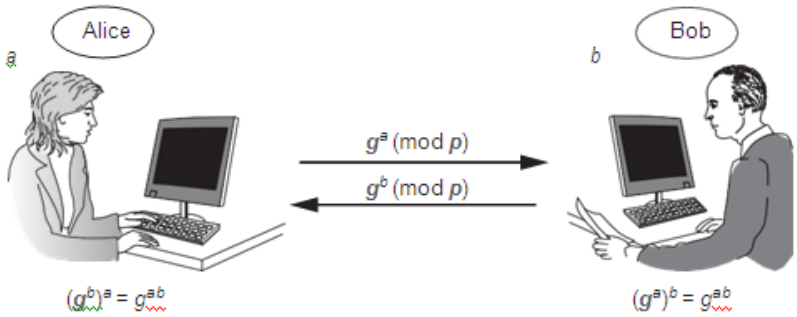


Figure 9.10. Diffie-Hellman protocol

Note that all calculations are performed modulo p and thus we omit mod p in each computation for convenience.

a) Alice randomly generates a positive integer a and calculates g^a. This is, effectively, a temporary ElGamal key pair. Alice sends her public key g^a to Bob.

b) Bob randomly generates a positive integer b and calculates g^b. Bob sends his public key g^b to Alice.

c) Alice uses g^b and her private key a to compute $(g^b)^a$.

d) Bob uses g^a and his private key b to compute$(g^a)^b$

e) The special combination function property that is needed that raising a number to the power a and then raising the result to the power b is the same as raising the number to the power b and then raising the result to the power a, which means that: $(g^a)^b = (g^b)^a$.

So Alice and Bob have ended up with the same value at the end of this protocol.

There are several important issues to note:

1. It is widely believed that the shared value $Z_{AB} = g^{ab}$ cannot be computed by anyone who does not know either a or b. An attacker who is monitoring the communication channel only sees g^a and g^b.
2. The main purpose of the Diffie–Hellman protocol is to establish a common cryptographic key *KAB*. There are two reasons why the shared value $ZAB = gab$ is unlikely to itself form the key in a real application:
    - Z_{AB} is not likely to be the correct length for a cryptographic key. If we conduct the Diffie–Hellman protocol with p having 1024 bits, then the shared value will also be a value of 1024 bits, which is much longer than a typical symmetric key.
    - Having gone through the effort of conducting a run of the Diffie–Hellman protocol to compute Z_{AB}, Alice and Bob may want to use it to establish several

different keys. Hence they may not want to use Z_{AB} as a key, but rather as a seed from which to derive several different keys. The rationale behind this is that Z_{AB} is relatively expensive to generate, both in terms of computation and communication, whereas derived keys K_{AB} are relatively cheap to generate from Z_{AB}.

3. Any public-key cryptosystem that has the right special property and for which a suitable combination function F can be found, could be used to produce a version of the Diffie–Hellman protocol. In this case:
    - very informally, the special property of ElGamal is that public keys of different users can be numbers over the same modulus p, which means that they can be combined in different ways;
    - the combination function F , which is $F(x, g^y) = (g^y)^x$, has the special property that it does not matter in which order the two exponentiations are conducted, since:

$F(x, g^y) = (g^y)^x = (g^x)^y = F(y, g^x)$.

It is not possible to use keys pairs from any public-key cryptosystem to instantiate the Diffie–Hellman protocol. In particular, RSA key pairs cannot be used because in RSA each user has their own modulus n, making RSA key pairs difficult to combine in the above manner.

ANALYSIS OF THE DIFFIE–HELLMAN PROTOCOL

Mutual entity authentication: There is nothing in the Diffie–Hellman protocol that gives either party any assurance of who they are communicating with. The values *a* and *b* (and hence *ga* and *gb*) have been generated for this protocol run and cannot be linked with either Alice or Bob. Neither is there any assurance that these values are fresh.

Mutual data origin authentication: This is not provided, by the same argument as above.

Mutual key establishment: Alice and Bob do establish a common symmetric key at the end of the Diffie–Hellman protocol, so this goal is achieved.

Key confidentiality. The shared value $Z_{AB}=gab$ is not computable by anyone other than Alice or Bob. Neither is any key K_{AB} derived from Z_{AB}. Thus this goal is achieved.

Key freshness. Assuming that Alice and Bob choose fresh private keys *a* and *b* then Z_{AB} should also be fresh. Indeed, it suffices that just one of Alice and Bob choose a fresh private key.

Mutual key confirmation. This is not provided, since neither party obtains any explicit evidence that the other has constructed the same shared value Z_{AB}.

Unbiased key control. Both Alice and Bob certainly contribute to the generation of Z_{AB}. Technically, if Alice sends *ga* to Bob before Bob generates *b*, then Bob could 'play around' with a few candidate choices for *b* until he finds a *b* that results in a $Z_{AB}=gab$ that he particularly 'likes'.

MAN-IN-THE-MIDDLE ATTACK ON THE DIFFIE–HELLMAN PROTOCOL

The man-in-the-middle attack is applicable to any situation where an attacker (Fred, in Figure 9.11) can intercept and alter messages sent on the communication channel between Alice and Bob.

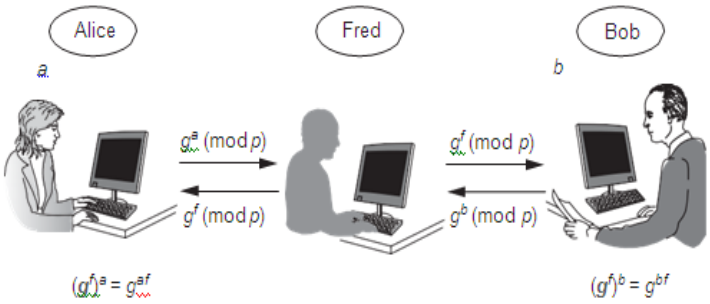


Figure 9.11. Man-in-the-middle attack against the Diffie-Hellman protocol

The man-in-the middle attack works as follows (where all calculations are modulo p):

1. Alice begins a normal run of the Diffie–Hellman protocol depicted in Figure 9.10. She randomly generates a positive integer a and calculates g^a. Alice sends g^a to Bob.
2. Fred intercepts this message before it reaches Bob, generates his own positive integer f , and calculates g^f . Fred then claims to be Alice and sends g^f to Bob instead of g^a.
3. Bob continues the Diffie–Hellman protocol as if nothing untoward has happened. Bob randomly generates a positive integer b and calculates g^b. Bob sends g^b to Alice.
4. Fred intercepts this message before it reaches Alice. Fred then claims to be Bob and sends g^f to Alice instead of g^b.
5. Alice now believes that the Diffie–Hellman protocol has successfully completed. She uses gf and her private integer a to compute $g^{af}= (g^f)^a$.
6. Bob also believes that it has successfully completed. He uses g^f& b to compute $g^{bf}= (g^f)^b$.
7. Fred computes $g^{af}=(g^a)^f$ and $g^{bf}= (g^b)^f$. He now has two different shared values, g^{af} , which he shares with Alice, and g^{bf} , which he shares with Bob.

At the end of this man-in-the-middle attack, all three entities hold different beliefs:

- Alice believes that she has established a shared value with Bob. But she is wrong, because she has established a shared value with Fred.
- Bob believes that he has established a shared value with Alice. But he is wrong, because he has established a shared value with Fred.
- Fred correctly believes that he has established two different shared values, one with Alice and the other with Bob.

At the end of this man-in-the-middle attack, Fred cannot determine the shared value *gab* that Alice and Bob would have established had he not interfered, since both *a* and *b* remain secret to him, protected by the difficulty of the discrete logarithm problem. Nonetheless, Fred is now in a powerful position:

- If Fred's objective was simply to disrupt the key establishment process between Alice and Bob then he has already succeeded. If Alice derives a key *KAF* from *gaf* and then encrypts a message to Bob using this key, Bob will not be able to decrypt it successfully because the key *KBF* that he derives from his shared value *gbf* will be different from *KAF* .
- Much more serious is the situation that arises if Fred remains on the communication channel. In this case, if Alice encrypts a plaintext to Bob using key *KAF* , Fred (who is the only person who can derive both *KAF* and *KBF*) can decrypt the ciphertext using *KAF* to learn the plaintext. He can then re-encrypt the plaintext using *KBF* and send this to Bob. In this way, Fred can 'monitor' the *encrypted* communication between Alice and Bob without them being aware that this is even happening.
- This man-in-the middle attack was only able to succeed because neither Alice nor Bob could determine from whom they were receiving messages during the Diffie–Hellman protocol run.

AKE PROTOCOLS BASED ON DIFFIE–HELLMAN

Way of building in authentication: The *station-to-station* (STS) protocol makes an additional assumption that Alice and Bob have each established a long-term signature/verification key pair and have had their verification keys certified. The STS protocol is shown in Figure 9.12 and proceeds as follows (where all calculations are modulo *p*):

1. Alice randomly generates a positive integer *a* and calculates *ga*. Alice sends *ga* to Bob, along with the certificate *CertA* for her verification key.
2. Bob verifies *CertA*. If he is satisfied with the result then Bob randomly generates a positive integer *b* and calculates *gb*. Next, Bob signs a message that consists of Alice's

name, *ga* and *gb*. Bob then sends *gb* to Alice, along with the certificate *CertB* for his verification key and the signed message.

3. Alice verifies *CertB*. If she is satisfied with the result then she uses Bob's verification key to verify the signed message. If she is satisfied with this, she signs a message that consists of Bob's name, *ga* and *gb*, which she then sends back to Bob. Finally, Alice uses *gb* and her private key *a* to compute (*gb*)*a*.
4. Bob uses Alice's verification key to verify the signed message that he has just received. If he is satisfied with the result then Bob uses *ga* and his private key *b* to compute (*ga*)*b*.

With the exception of the first two, the goals are met for the STS protocol are just as for the basic Diffie–Hellman protocol (in other words, they are all met except for key confirmation). It remains to check whether the first two authentication goals are now met:

Mutual entity authentication. Since *a* and *b* are randomly chosen private keys, *ga* and *gb* are thus also effectively randomly generated values. Hence we can consider *ga* and *gb* as being nonces. At the end of the second STS protocol message, Alice receives a digital signature from Bob on a message that includes her 'nonce' *ga*. Similarly, at the end of the third STS protocol message, Bob receives a digital signature from Alice on a message that includes his 'nonce' *gb*. Hence, by the principles , mutual entity authentication is provided, since both Alice and Bob each perform a cryptographic computation using a key only known to them on a nonce generated by the other party.

Mutual data origin authentication. This is provided, since the important data that is exchanged in the main messages is digitally signed.

9.4.3 An AKE protocol based on key distribution

The STS protocol is an AKE protocol based on key agreement and the use of public-key cryptography. We will now look at an AKE protocol based on key distribution and the use of symmetric cryptography. This protocol is a simplified version of one from ISO 9798-2. This protocol involves the use of a trusted third party (denoted TTP).

PROTOCOL DESCRIPTION

The idea behind this AKE protocol is that Alice and Bob both trust the TTP. When Alice and Bob wish to establish a shared key K_{AB}, they will ask the TTP to generate one for them, which will then be securely distributed to them. The protocol involves the following assumptions:

- Alice has already established a long-term shared symmetric key *KAT* with the TTP.
- Bob has already established a long-term shared symmetric key *KBT* with the TTP.

- Alice and Bob are both capable of randomly generating nonces.

There is a further assumption made on the type of encryption mechanism used, but we will discuss that when we consider data origin authentication.

The protocol is shown in Figure 9.13 and proceeds as follows:

1. Bob starts the protocol by randomly generating a nonce *rB* and sending it to Alice.
2. Alice randomly generates a nonce *rA* and then sends a request for a symmetric key to the TTP. This request includes both Alice's and Bob's names, as well as the two nonces *rA* and *rB*.
3. The TTP generates a symmetric key *KAB* and then encrypts it twice. The first ciphertext is intended for Alice and encrypted using *KAT* . The plaintext consists of *rA*, *KAB* and Bob's name. The second ciphertext is intended for Bob and encrypted using *KBT* . The plaintext consists of *rB*, *KAB* and Alice's name. The two ciphertexts are sent to Alice.
4. Alice decrypts the first ciphertext using *KAT* and checks that it contains *rA* and Bob's name. She extracts *KAB*. She then generates a new nonce *rA*j . Next, she generates a new ciphertext by encrypting *rA*j and *rB* using *KAB*. Finally, she forwards the second ciphertext that she received from the TTP, and the new ciphertext that she has just created, to Bob.
5. Bob decrypts the first ciphertext that he receives (which is the second ciphertext that Alice received from the TTP) using *KBT* and checks that it contains *rB* and Alice's name. He extracts *KAB*. He then decrypts the second ciphertext using *KAB* and checks to see if it contains *rB*. He extracts *rA*j . Finally, he encrypts *rB*, *rA*j and Alice's name using *KAB* and sends this ciphertext to Alice. Alice decrypts the ciphertext using *KAB* and checks that the plaintext consists of *rB*, *rA*j and Alice's name. If it does then the protocol concludes successfully.

PROTOCOL ANALYSIS

Mutual entity authentication: We will divide this into two separate cases.

First we look at Bob's perspective. At the end of the fourth protocol message, the second ciphertext that Bob receives is an encrypted version of his nonce *rB*. Thus this ciphertext is fresh. But who encrypted it? Whoever encrypted it must have known the key *KAB*. Bob received this key by successfully using *KBT* to decrypt a ciphertext, which resulted in a correctly formatted plaintext message consisting of *rB, KAB* and Alice's name. Thus Bob can be sure that this ciphertext originated with the TTP, since the TTP is the only entity other than Bob who knows *KBT* . The format of this plaintext is essentially an 'assertion' by the

TTP that the key *KAB* has been freshly issued (because *rB* is included) for use in communication between Bob (because it is encrypted using *KBT*) and Alice (because her name is included). Thus the entity that encrypted the second ciphertext in the fourth protocol message must have been Alice because the TTP has asserted that only Alice and Bob have access to *KAB*. Hence Bob can be sure that he has just been talking to Alice.

1. Alice's perspective is similar. At the end of the last protocol message, Alice receives an encrypted version of her nonce *rA*j . This ciphertext, which was encrypted using *KAB*, is thus fresh. In the third protocol message Alice receives an assertion from the TTP that *KAB* has been freshly (because *rA* is included) issued for use for communication between Alice (because it is encrypted using *KAT*) and Bob (because his name is included). Thus the entity that encrypted the last protocol message must have been Bob, again because the TTP has asserted that only Alice and Bob have access to *KAB*. Thus Alice can be sure that she has just been talking to Bob.

Mutual data origin authentication: We use symmetric encryption throughout this protocol and do not apparently employ any mechanism to explicitly provide data origin authentication, such as a MAC. While symmetric encryption does not normally provide data origin authentication. Throughout current protocol, the plaintexts are strictly formatted and fairly short, hence it might be reasonable to claim that encryption alone is providing data origin authentication, so long as a strong block cipher such as AES is used. However, the standard ISO 9798-2 goes further, by specifying that the 'encryption' used in this protocol must be such that data origin authentication is also essentially provided. One method would be to use an authenticated-encryption primitive. This goal is thus also met.

Mutual key establishment: At the end of the protocol Alice and Bob have established K_{AB}, so this goal is met.

Key confidentiality: The key K_{AB}can only be accessed by an entity who has knowledge of either K_{AT}, K_{BT}or K_{AB}. This means either the TTP (who is trusted), Alice or Bob. So this goal is met.

Key freshness: This goal is met so long as the TTP generates a fresh key K_{AB}. Again, we are *trusting* that the TTP will do this.

Mutual key confirmation: Both Alice and Bob demonstrate that they know $K_{AB}$by using it to encrypt plaintexts (Alice in the fourth protocol message; Bob in the last protocol message). Thus both confirm knowledge of the shared key.

Unbiased key control: This is provided because $K_{AB}$is generated by the TTP.

Thus we conclude that all the goals are provided. A similar AKE protocol is used by the widely deployed *Kerberos* protocol.