MODULE 3

8 Entity Authentication

The last security service that we will discuss in detail is entity authentication. This is probably the security service that is provided by the most diverse range of mechanisms, including several that are not inherently cryptographic. Naturally our focus will be on the use of cryptography to provide entity authentication. Since many cryptographic entity authentication mechanisms rely on randomly generated numbers, we will choose this chapter to have a discussion about random number generation. We will also discuss the wider notion of providing freshness in cryptography.

At the end of this chapter you should be able to:

- Discuss a number of different mechanisms for randomly generating values that are suitable for use in cryptography.
- Compare different techniques for providing freshness.
- Recognise a number of different approaches to providing entity authentication.
- Appreciate the limitations of password-based approaches to providing entity authentication.
- Explain the principle behind dynamic password schemes.

8.1 Random number generation

The relationship between cryptography and randomness is extremely important. Many cryptographic primitives cannot function securely without randomness. Indeed, there are many examples of cryptosystems failing not because of problems with the underlying cryptographic primitives, but because of problems with their sources of randomness. It is thus vital that we understand what randomness is and how to produce it.

8.1.1 The need for randomness

Most cryptographic primitives take structured input and turn it into something that has no structure. For example:

- The ciphertext produced by a block or stream cipher should have no apparent structure. If this is not the case then useful information may be provided (leaked) to an attacker who observes the ciphertext. Indeed, there are many applications where ciphertext is used as a source of randomness. We have already seen this in Section 4.6.2 when we observed that ciphertext can be used to generate keystream for a 'stream cipher'.
- The output of a hash function should have no apparent structure. Although we did not state this explicitly as one of our hash function properties, we noted in Section 6.2.2 that hash functions are often used to generate cryptographic keys.

Just as importantly, many cryptographic primitives *require* sources of randomness in order to function. For example:

- Any cryptographic primitive that is based on symmetric keys requires a source of randomness in order to generate these keys. The security of symmetric cryptography relies on the fact that these keys cannot be predicted in any way.
- Many cryptographic primitives require the input of other types of randomly generated numbers such as salts (see Section 8.4.2) and initial variables (see Section 4.6.2). We have also seen in Section 5.3.4 that public-key cryptosystems are normally probabilistic, in the sense that they require fresh randomness each time that they are used.
- We will see in Chapter 9 that sources of randomness are very important for providing freshness in cryptographic protocols.

Given this intricate relationship, we could probably have had a general discussion about random number generation almost anywhere in our review of mechanisms for implementing security services. However, we choose to have it now because a significant number of the cryptographic mechanisms for providing entity authentication require randomly generated numbers as a means of providing freshness. We will discuss freshness mechanisms in Section 8.2. We will discuss randomness in the specific context of cryptographic key generation in Section 10.3.

8.1.2 What is randomness?

People have been trying for hundreds of years to define precisely what is meant by the word 'random'. In fact 'randomness', by its very nature, defies classification rules. Nonetheless, we all have an intuitive feel for what 'random' should mean.

These notions are all to do with ideas such as 'unpredictability' and 'uncertainty'. We would like randomly generated numbers in general to be hard to predict and appear to have no relationship with previous randomly generated numbers. By the same measure, we would like randomly generated bits to be apparently unpredictable sequences of bits.

Ironically, however, although randomness is hard to define, there are many ways that we would like randomness to behave that are easily identifiable. A random number generation process is often assessed by applying a series of statistical tests. Many of these tests are fairly intuitive and include checks such as, on average, over the generation of many random outputs:

- does 1 appear in the output of the process approximately as often as 0 does?
- does a 0 follow a 1 in the output of the process approximately as often as a 1 follows a 1?
- does the string 000 occur in the output of the process approximately as often as the string 111?

Human intuition often confuses randomness with evenly spaced distributions. For example, many people believe that the binary string 10101010 is more likely to have been produced by a truly uniform random source than the binary string 11111111. In fact, the probability that a uniform random source produces these two outputs is exactly the same, and is equal to the probability that it produces a string with no apparent pattern such as 11010001. By the same token, some bank customers get concerned when the bank issues them with the PIN 3333, when in fact this should be just as unlikely to occur as the PIN 7295. (Of course, a bank customer is perhaps more likely to change a PIN to something like 3333, so there is a genuine case for considering 3333 to be a less secure PIN in practice, but it is not a failure of the bank's random PIN generation process!) Statistical tests provide rigorous methods for assessing a random generation process that are much more reliable than human intuition.

8.1.3 Non-deterministic generators

There are two general approaches to generating randomness. First we will look at *non-deterministic* generation, which relies on unpredictable sources in the physical world. This is a compelling, but often expensive, approach to producing randomness. On the other hand, there are many situations where we are willing to compromise and use a 'cheaper' source of randomness. In this case, *deterministic* generation techniques, which we examine in Section 8.1.4, can be adopted.

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. Generally speaking, these are the best sources of 'true' randomness. 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. In some cases the randomly generated output is produced too slowly to be of much practical use.

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. But they are also of lower quality and easier for an attacker to access or compromise. When using software-based techniques it may be advisable to combine a number of different software-based non-deterministic generators.

NON-DETERMINISTIC GENERATORS IN PRACTICE

Non-deterministic generators work by measuring the physical phenomena and then converting the measurements into a string of bits. In some cases the initial binary string that is generated may need some further processing. For example, if the source was based on mouse clicks then periods of user inactivity may have to be discarded.

Regardless of the underlying technique used, there are 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 (indeed, this is the very point of using physical phenomena as a randomness source).

For these reasons, in most cryptographic applications deterministic sources of randomness tend to be preferred.

8.1.4 Deterministic generators

The idea of a *deterministic* random generator may sound like an oxymoron, since anything that can be determined cannot be truly random. The term *pseudorandom* (which we introduced in Section 4.2.1) is often used to describe both a deterministic generator and its output.

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. However, as we just hinted, the output of a deterministic generator is certainly not randomly generated. In fact, anyone who knows the information that is input to the deterministic generator can completely predict the output. Each time the algorithm is run using the same input, the same output will be produced. Such predictability is, in at least one sense, the opposite of randomness.

However, if we use a *secret* input into the deterministic generator then, with careful design of the generation process, we might be able to generate output that 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 Figure 8.1.

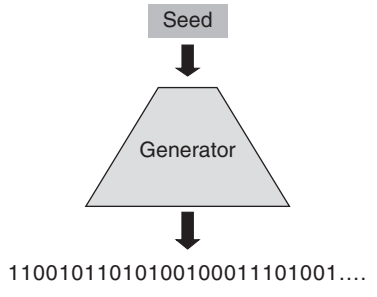

Figure 8.1. Basic model of a deterministic generator

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. Following our standard assumptions of Section 1.5.1, we normally assume that the details of the generator are publicly known, even if they are not.

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 (see Section 4.2.4), since the keystream generator for a stream cipher is a deterministic generator whose output is used to encrypt plaintext (see Section 4.2.1).
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.

Of course, deterministic generators are, in some sense, a bit of a cheat. They generate pseudorandom output but they require random input in the form of the seed to operate. So we still require a source of randomness for the seed. If necessary, we also require a means of securely distributing the seed.

However, the seed is relatively short. It is normally a symmetric key of a standard recommended length, such as 128 bits. We are still faced with the problem of generating this seed, but once we address this we can use it to produce long streams of pseudorandom output. The use of relatively expensive non-deterministic generators might be appropriate for short seed generation. Alternatively, a more secure deterministic generator could be used for this purpose such as one installed in secure hardware (see Section 10.3).

The case for using deterministic generators is similar to the case we made for using stream ciphers in Section 4.2.2. Deterministic generators thus provide an attractive means of converting relatively costly random seed generation into a more 'sustainable' source of pseudorandom output. As remarked earlier, however, deterministic generators are often points of weakness in real cryptosystems. It is important to identify potential points of weakness:

Cryptanalysis of the generator. Deterministic generators are cryptographic algorithms and, as such, 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. However, it is not uncommon for cryptographic applications to use well-respected encryption algorithms but employ 'home-made' deterministic generators to produce keys. This may arise because some system designers are wise enough not to attempt to build their own encryption algorithms, but fail to appreciate that designing secure deterministic generators is just as complex.

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. Thus most of the issues concerning key management discussed in Chapter 10 are also relevant to the management of seeds.

We end this brief discussion of generating randomness by summarising the different properties of non-deterministic and deterministic generators in Table 8.1.

8.2 Providing freshness

Before we discuss entity authentication mechanisms, there is one important set of mechanisms that we need to add to our cryptographic toolkit. These are not really cryptographic 'primitives' because, on their own, they do not achieve any security goals. *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.

Table 8.1: 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

Note that what entity authentication primarily requires is a notion of *liveness*, which is an indication that an entity is currently active. A freshness mechanism does not provide this by default, since just because a message is 'new' does not imply that the sender is 'alive'. For example, an attacker could intercept a 'fresh' message and then delay relaying it to the intended receiver until some point in the future. When the receiver eventually receives the message they may well be able to identify that it is fresh (not a replay) but they will not necessarily have any assurance that the sender is still 'alive'. However, all freshness mechanisms can be used to provide liveness if they are managed appropriately, particularly by controlling the window of time within which a notion of 'freshness' is deemed acceptable.

There are three common types of freshness mechanism, which we now review.

8.2.1 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. The granularity of time involved might vary considerably between applications. For some applications the date might suffice, however, more commonly an accurate time (perhaps to the nearest second) is more likely to be required.

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

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.

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. How much drift is acceptable before Bob rejects the time associated with a message? 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. The most obvious 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.

Communication delays. It is inevitable that there will be some degree of communication delay between Alice sending, and Bob receiving, a message. This tends to be negligible compared to clock drift and can also be managed using windows of acceptability.

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.

8.2.2 Sequence numbers

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

The idea is best illustrated by means of 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. We illustrate the case of N_{AB}.

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 $N_{AB} = T_{new}$.
2. Alice sends her message to Bob along with the latest sequence number value, which is T_{new}.
3. Alice increments the sequence number N_{AB} by one (in other words, she sets $N_{AB} = T_{new} + 1$) 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 T_{new} sent by Alice with the most recent value of the sequence number N_{AB} on his database. Suppose this is $N_{AB} = T_{old}$.
5. If $T_{new} > T_{old}$ then Bob accepts the latest message as fresh and he updates his stored value of N_{AB} from T_{old} to T_{new}.
6. If $T_{new} \leq T_{old}$ then Bob rejects the latest message from Alice as not being fresh.

This is just one example of the way in which sequence numbers can work. The basic principle is that messages are only accepted as fresh if the latest sequence number has not been used before. The simplest way of doing this is to make sure that, each time a new message is sent, the sequence number is increased.

Note that an alternative technique would be to associate each new message with a unique identification number, but not necessarily one that is bigger than the last one sent. In this case Bob would have to maintain a database that consisted of all previous identification numbers sent (not just the most recent one). He would then have to search this database every time that a new message was received in order to check that the identification number had not been used before. Clearly this would be inefficient in terms of time and storage space.

In the above example, note that Alice increments her sequence number N_{AB} by one each time that she sends a message, but Bob only checks whether $T_{new} > T_{old}$, not that $T_{new} = T_{old} + 1$, which is what we might expect. If $T_{new} > T_{old} + 1$ then this suggests that, between the last message Bob received and the current message, some messages from Alice to Bob have got lost. This might itself be a problem, so Bob will need to decide whether the fact that there are missing messages is important. However, the sequence number is primarily there to provide *freshness*. The fact that $T_{new} > T_{old}$ is enough to gain this assurance. It also allows Bob to resynchronise by updating his version of N_{AB} to the latest sequence number T_{new}.

It is worth briefly considering the extent to which sequence numbers address the four concerns that we raised with clock-based mechanisms:

Existence of clocks. The communicating parties no longer require clocks.

Synchronisation. In order to stay synchronised, communicating parties need to maintain a database of the latest sequence numbers. Our simple example included a mechanism for making sure that this database is kept up to date.

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 $T_{new} > T_{old}$, but *also* if $T_{new} = T_{old}$, since there is a chance that the previous message from Alice to Bob has not yet arrived. Note that this issue is not relevant 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.

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.

The obvious *cost* of using sequence numbers is the need to maintain databases of their latest values. Another possible problem arises if sequence numbers have a limited size and eventually cycle around again. Nonetheless, this type of mechanism is popular in applications where maintaining synchronised clocks

is unrealistic. The most compelling such example is probably mobile phone networks, where it is impractical to rely on millions of handsets throughout a network keeping an accurately synchronised notion of clock-based time (see Section 12.3.4).

8.2.3 Nonce-based mechanisms

One problem that is shared by both clock-based mechanisms and sequence numbers is the need for some integrated infrastructure. In the former this was a shared clocking mechanism, in the latter it was a synchronised database of sequence numbers. *Nonce-based* mechanisms do not have this need. Their only requirement is the ability to generate *nonces* (literally, 'numbers used only once'), which are randomly generated numbers for one-off use. Note that the term 'nonce' is sometimes used, more literally, to mean numbers that are *guaranteed* to be used only once. We will use it in a slightly more relaxed way to mean numbers that *with high probability* are used only once.

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, where by 'fresh' we mean that the received message must have been created *after* the nonce was generated.

To see why freshness is provided here, recall that the nonce was generated randomly for one-off use. As we know from Section 8.1, a good random number generator should not produce predictable output. Thus it should be impossible for an adversary to be able to anticipate a nonce in advance. If the same nonce reappears in a later message then it must be the case that this later message was created by someone *after* the generation of the nonce. In other words, the later message is fresh.

We re-emphasise this important point by considering the simplest possible example. 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 surely is no coincidence! What 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 certainly 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. What we can guarantee is 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 do not suffer from any of the problems that we identified for the previous freshness mechanisms, except for the familiar need to set a window of acceptance beyond which a nonce will no longer be regarded as fresh. After all, in our simple example we stated that Bob sent the nonce 'straight back'. How much delay between sending and receiving the nonce should Alice regard as being 'straight back'? Nonce-based mechanisms do, however, 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.

8.2.4 Comparison of freshness mechanisms

Choosing an appropriate freshness mechanisms is application dependent. The appropriate mechanism depends on which of the various problems can best be overcome in the environment in which they will be deployed. Table 8.2 contains a simplified summary of the main differences between the three types of freshness mechanism that we have discussed.

Note that there are other differences that might be influential in selecting a suitable freshness mechanism for an application. For example, sequence numbers and nonces are not, by definition, bound to a notion of clock-based time. Hence, if using these mechanisms in an application that requires a notion of 'timeliness' (for

Table 8.2: Summary of properties 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

example, for entity authentication) then they require a degree of management. For sequence numbers this management involves monitoring the time periods between received sequence numbers. For nonces, it involves monitoring the delay between sending and receiving the nonce.

8.3 Fundamentals of entity authentication

Recall from Section 1.3.1 that 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 identity then we cannot be certain whom we are trying to authenticate. 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'.

The word *entity* is itself problematic. We will avoid philosophical questions and not propose any formal definition, other than to comment that an 'entity' in the subsequent discussion could be a human user, a device or even some data. To appreciate the problems of defining a rigorous notion of an 'entity', consider the following question: when someone types their password into a computer then is the entity that is being authenticated the person, or their password? This essentially relates to the same 'human–computer gap' that we commented on when discussing digital signatures in Section 7.4.3.

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. For example, when someone presents their card and PIN at an ATM then they are engaging in unilateral entity authentication to the bank. The bank does not authenticate itself to the customer. Indeed this 'weakness' of ATM authentication has been regularly exploited by attackers who present fake ATMs to bank customers in order to harvest their card details and PINs. If the entity authentication process had been mutual then the customer would have been able to reject the bank. In fact, ATMs attempt to weakly authenticate themselves simply by 'looking like' genuine ATMs, but a determined attacker can easily make something that defeats this by also 'looking like' a real ATM.

8.3.1 A problem with entity authentication

It is important to recognise that 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.

To see this, 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 (fairly reasonably in this case) 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 (we can at least prevent the attacker controlling the amount that is withdrawn).

Fortunately, cryptography can provide a means of prolonging an entity authentication check in many situations. The solution is to combine entity authentication with the establishment of a cryptographic key. Through the entity authentication we gain assurance that the key was established with the claimed entity. Every time the key is correctly used in the future then it should be the case that the entity who was authenticated is involved in that session, since nobody

else should know the key. While this does not help us much in our scenario where the attacker is standing next to Alice at the ATM, it does help us defend against attackers who are attacking the ATM network and attempting to modify or send spoof messages over it. We will look at cryptographic protocols for implementing this process in Section 9.4.

8.3.2 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 (see Chapter 9). 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. For example, the process of establishing a symmetric key commonly involves mutual entity authentication in order to provide the two communicating entities with assurance that they have agreed a key with the intended partner. We discuss this scenario in more detail in Section 9.4.

8.3.3 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 for doing this. Note that:

- As we observed earlier, providing identity information is not normally enough to achieve entity authentication. Entity authentication also requires a notion of freshness, as discussed in Section 1.3.1.
- 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:
  1. 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, as we discussed in Section 6.2.2, cryptography can play a role in the secure storage of passwords.

2. Almost all of these approaches require a cryptographic protocol (the subject of Chapter 9) as part of their implementation.

We now review the main categories of identity information that are used when providing entity authentication.

SOMETHING THE CLAIMANT HAS

For human users, identity information can be based on something physically held by the user. This is a familiar technique for providing access control in the physical world, where the most common identity information of this type is a physical key. This technique can also be used for providing identity information in the electronic world. Examples of mechanisms of this type include:

Dumb tokens. By 'dumb' we mean 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. It is quite easy for anyone determined enough to either build, or purchase, 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 (see, for example, Section 12.4). Smart cards are also widely used for other applications, such as electronic ticketing, physical access control, identity cards (see Section 12.6.3), etc.

Smart tokens. Smart cards are special examples of a wider range of technologies that we will refer to as *smart tokens*. Some 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. We will discuss an application of this type in Section 8.5.

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

One of the highest profile, and most controversial, methods of providing identity information is to base it on physical characteristics of the claimant, which in this case is normally a human user. 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.

Identification based on biometrics is a compelling approach for human users because many biometric characteristics appear to be fairly effective at separating individuals. However, there are many implementation issues, both technical, practical and sociological. Hence biometric techniques need to be adopted with care.

We will not discuss biometrics any further here since they are of peripheral relevance to cryptography. We recognise biometrics primarily as a potentially useful source of identity information.

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.

Indeed, in many applications, identity information of this type often *is* a cryptographic key. However, strong cryptographic keys are usually far too long for a human user to remember and hence 'know'. There is 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. Of course, 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. We revisit this issue in Section 10.6.3.

We will now proceed to look more closely at ways in which cryptography can be employed to assist in the provision of identity information based on something that the claimant knows.

8.4 Passwords

Passwords are still one of the most popular techniques for providing identity information, although they have many obvious flaws. We will briefly look at some of these flaws as motivation for enhanced techniques. We will also reexamine the use of cryptography for password protection. Note that in this section we use the term 'password' very loosely, since much of our discussion equally applies to the likes of PINs and passphrases.

8.4.1 Problems with passwords

The attractive properties of passwords are that they are simple and familiar. These are the reasons that they are ubiquitously used as identity information. However, they have several flaws that severely limit the security of any application that employs them:

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 simply exhaustively tries all the 'likely' passwords and hopes to eventually find the correct one (see Section 1.6.5). Clever pneumonic techniques can slightly increase the size of a usable password space. Where users are requested to adopt complex passwords a usability–security tradeoff is soon reached, since it is more likely that a complex password will be transferred into 'something the claimant has' in the form of a written note, which in many cases defeats the purpose of using passwords in the first place. Moving from passwords to

passphrases can improve this situation by significantly increasing the password space, however, many of the other problems with passwords remain.

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. We have just noted that 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.

For the latter reason, passwords should be cryptographically protected at all times, as we will discuss in just a moment.

It is best to regard passwords as a rather fragile means of providing identity information. In particular, the problem of repeatability means that passwords on their own do not really provide entity authentication as we defined it, since there is no strong notion of freshness. In applications where strong entity authentication is required then passwords are best employed in conjunction with other entity authentication techniques, if at all. However, the advantages of passwords mean that they will probably always find use in applications where security is a relatively low priority.

8.4.2 Cryptographic password protection

Consider a large organisation that wishes to authenticate many users onto its internal system using passwords. One obvious way of implementing this is to use a system that compares offered passwords with those stored on a centralised password database. This presents the password database as a highly attractive target for attackers, 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. This is because, in order to authenticate a user, the system does not actually need to know a user's password. Rather, the device simply needs to know whether a supplied password is the correct one. The point is that while a user does need to enter the correct password, the system does not need to store a copy of this password in order to verify that it is correct.

In Section 6.2.2 we described an application of hash functions that implemented password database protection. The idea is to store hashes of the passwords, rather than the actual passwords in the password database. This allows them to be checked, while preventing anyone who gains access to the password database from recovering the passwords themselves. We observed that any function that is regarded as being one-way (which includes hash functions) could be used to provide this service.

As an example of a cryptographic primitive being used in a different way to create a one-way function, Figure 8.2 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 (see Section 8.1.3). The salt is used to uniquely modify the DES encryption algorithm (see Section 4.4) in 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.

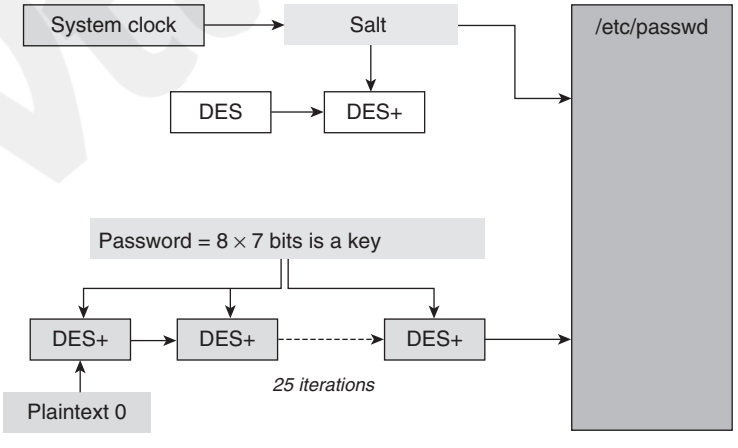

Figure 8.2. One-way function for UNIX password protection

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.

8.5 Dynamic password schemes

As just observed, 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.

Dynamic password schemes are important entity authentication mechanisms and are widely deployed in token-based technologies for accessing services such as internet or telephone banking.

8.5.1 Idea behind dynamic password schemes

At its heart, a dynamic password scheme uses a 'password function' rather than a password. If a claimant, which we will assume 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 that we will shortly discuss, 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 that we discussed in

Section 8.2 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.

8.5.2 Example dynamic password scheme

We now give an example of a dynamic password scheme.

DYNAMIC PASSWORD SCHEME DESCRIPTION

Before any authentication attempts are made, the user is given a token on which the password function has already been implemented in the form of a symmetric cryptographic algorithm A (this could be an encryption algorithm) with symmetric key K. While the algorithm A could be standard across the entire system, the key K is shared only by the server and the token held by the user. Note that a different user, with a different token, will share a different key with the server. Thus, as far as the server is concerned, correct use of key K will be associated with a specific user.

A further feature of this example scheme is that the user has some means of identifying themselves to the token, otherwise anyone who steals the token could pass themselves off as the user. In our example, this process will be implemented using a PIN. The token will only activate if the user enters the correct PIN.

Figure 8.3 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.

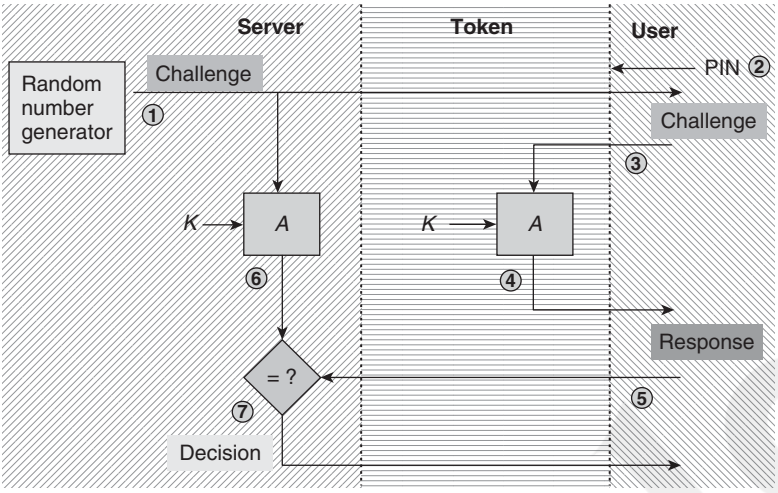

Figure 8.3. Example of a dynamic password scheme based on challenge–response

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 (recall our discussion in Section 8.2.3 regarding windows of acceptance for nonce-based freshness mechanisms). 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.

ANALYSIS OF DYNAMIC PASSWORD SCHEME

The dynamic password scheme in Figure 8.3 merits a closer look, just to make sure that we appreciate both what has been gained in comparison to conventional passwords and the limitations of this idea.

Firstly, we establish the basis for assurance that the user is who they claim to be (the security 'bottom line'). From the perspective of the server, the only entity apart from itself that can compute the correct response is the only other entity in possession of both the algorithm A and the key K. The only other entity to know K is the token. The only way of accessing the token is to type in the correct PIN. Knowledge of the PIN is therefore the basis for assurance of authentication. So long as only the correct user knows the correct PIN, this dynamic password scheme will successfully provide entity authentication of the user.

This does not sound like a big improvement on password-based entity authentication, since we are essentially using a conventional password scheme to authenticate the user to the token. However, there are 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 PIN into 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 (see Section 8.1) 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.

DYNAMIC PASSWORD SCHEMES IN PRACTICE

The relative ease of use and low cost of dynamic password schemes has seen their use increase significantly in recent years. They are now fairly popular entity authentication mechanisms for applications such as online banking (for example, EMV-CAP, discussed in Section 12.4.5). There is a great deal of variation in the ways in which these schemes operate. As well as varying in the underlying freshness mechanism, they also vary in the extent to which the user authenticates to the token. Techniques include:

- the user authenticates directly to the token (as in our example);
- the user presents some authentication data, such as a PIN, to the server; this could happen:
  - directly, for example the user presents the PIN to the server using a separate communication channel such as a telephone line;
  - indirectly, for example, the PIN is also an input into the cryptographic computation on the token, thus allowing it to be checked by the server when it conducts the verification step;

- there is no authentication between the user and the token (in which case we have one-factor authentication that relies on the correct user being in possession of the token).

8.6 Zero-knowledge mechanisms

We now briefly discuss an even stronger cryptographic primitive that can be used to support entity authentication. *Zero-knowledge mechanisms* bring security benefits but have practical costs. Nonetheless, it is worth at least discussing the idea behind them, just to indicate that it is feasible, even though they are not as commonly implemented in real systems as the previously discussed techniques.

8.6.1 Motivation for zero-knowledge

The entity authentication techniques that we have looked at thus far have two properties that we might deem undesirable.

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. As another example, the dynamic password scheme based on challenge–response requires the smart token and the server to share a key. 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 (see the remark about *key exposure* in Section 10.2.1).

It would seem unlikely that 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, but amazingly zero-knowledge mechanisms can do precisely this.

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.

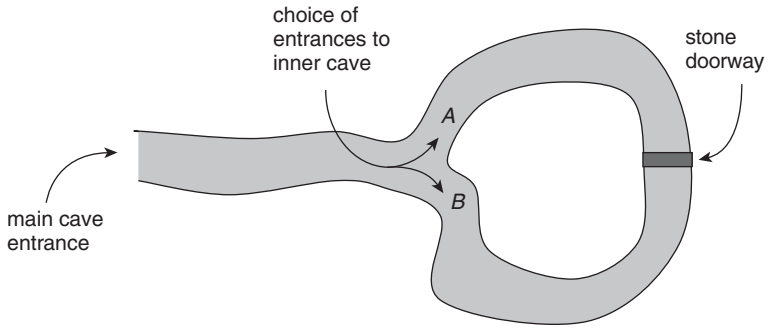

Figure 8.4. Popular analogy of a zero-knowledge mechanism

8.6.2 Zero-knowledge analogy

We will avoid discussion of the details of any zero-knowledge mechanism, but instead present 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. 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. Putting it another way, 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 $\frac{1}{2^n}$ 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.

8.6.3 Zero-knowledge in practice

An obvious question to ask about the zero-knowledge analogy is why we did not just stand at the cave split, send the guide down one tunnel, and then request that he came out of the other tunnel. This would demonstrate knowledge of the key phrase without revealing it. The reason is that this was indeed just an 'analogy' for the way in which cryptographic zero-knowledge mechanisms work.

Cryptographic zero-knowledge mechanisms require a number of 'runs' (or *rounds*) of a test of knowledge of a cryptographic secret, with each round passed bringing greater assurance to the verifier that the prover knows a cryptographic secret. However, each round also involves more computation and adds to the time taken to make the entity authentication decision. Zero-knowledge mechanisms normally use techniques similar to those of public-key cryptography which, as we discussed in Section 5.4.2, are typically more computationally expensive than those of symmetric cryptography. Thus zero-knowledge primitives are more

expensive mechanisms to use than the previous entity authentication mechanisms that we have discussed.

8.7 Summary

In this chapter we discussed mechanisms for providing entity authentication. Since strong entity authentication mechanisms require an assurance of freshness, we first reviewed freshness mechanisms. Since an important means of providing freshness mechanisms is challenge–response, which relies on random number generation, we began with a discussion of random number generation. Thus, some of this chapter dealt with issues of wider significance than entity authentication.

However, our treatment of entity authentication was also incomplete. One of the most important classes of cryptographic mechanism for providing entity authentication are authentication and key establishment (AKE) protocols. We have chosen to defer our discussion of AKE protocols until Section 9.4. The reason that we delay is because consideration of AKE protocols really requires a better understanding of what a cryptographic protocol is.

Lastly, we note that entity authentication is a service that is often provided in combination with other services. While there are applications where one entity is simply required to identify itself to another, in many applications the real reason for requiring entity authentication is to provide a platform on which other security services can be built. Hence entity authentication mechanisms are often components of more complex cryptographic protocols. Again, AKE protocols will provide an example of this.

8.8 Further reading

Generating good randomness is one of the most essential requirements for many cryptosystems. A good overview of appropriate techniques can be found in RFC 4086 [63]. Another source of reliable information is ISO/IEC 18031 [11]. NIST 800-22 [145] provides a suite of statistical tests of randomness, which can be used to measure the effectiveness of any random generator. Ferguson, Schneier and Kohno [75] has an interesting chapter on practical random generation. An informative and fun site about randomness and its applications beyond cryptography, which also provides output from a non-deterministic generator, is maintained by Mads Haahr [94]. CrypTool [52] has implementations of various statistical randomness tests.

Ferguson, Schneier and Kohno [75] also have an informative chapter on using clocks as freshness mechanisms. There are also discussions of different freshness mechanisms in Menezes, van Oorschot and Vanstone [123], and Dent and Mitchell [55]. The Network Time Protocol provides a means of synchronising

clocks in packet-switched networks such as the Internet and is specified in RFC 1305 [124].

Entity authentication is a security service that can be implemented in a variety of different ways, many of which involve cryptography being used alongside other technologies. A comprehensive overview of smart cards, smart tokens and their applications is provided by Mayes and Markantonakis [120]. A detailed investigation of different biometric techniques can be found in Jain, Flynn and Ross [101], while Gregory and Simon [93] is a more accessible introduction to biometrics.

An interesting set of experiments were conducted by Yan et al. [205] concerning memorability and security of passwords as entity authentication mechanisms. There is a good chapter on password security in Anderson [23]. FIPS 181, the Automated Password Generator [76], creates pronounceable passwords from DES, illustrating the use of cryptographic primitives as a source of pseudorandomness. A survey of alternatives to conventional passwords based on graphical techniques was conducted by Suo, Zhu and Owen [188]. Of more significant cryptographic interest are dynamic password schemes. RSA Laboratories are one of the main commercial providers of products implementing dynamic password schemes and they maintain several interesting simulations of their products, as well as providing a home for the One-Time Password Specifications (OTPS) [114]. Wikipedia provides a good portal page on dynamic password schemes [201] that includes comparisons of different approaches to generating dynamic passwords and mentions various other commercial vendors of products implementing dynamic password schemes.

The main ISO standard relating to entity authentication is ISO/IEC 9798 [19], which includes a part relating to zero-knowledge mechanisms. Examples of zero-knowledge mechanisms can be found in, for example, Stinson [185] and Vaudenay [194]. The original inspiration for the zero-knowledge protocol analogy that we described is Quisquater et al. [72].

8.9 Activities

1. Cryptography and randomness are connected in many different ways:
   (a) Provide some examples of why randomness is needed in cryptography.
   (b) Provide some examples of how cryptography can be used to provide randomness.
2. Suggest appropriate mechanisms for generating randomness for the following applications:
   (a) generating a cryptographic key on a laptop for use in an email security application;
   (b) generating a master key for a hardware security module;
   (c) generating keystream for a stream cipher on a mobile phone;

(d) generating a one-time pad key for a high-security application;
(e) generating a nonce on a server for use in a dynamic password scheme.

3. In Section 8.1.3 we provided some examples of software-based non-deterministic random number generation techniques. Find out which (combinations of) these techniques are currently recommended from:

   (a) a security perspective;
   (b) a practical perspective.

4. One technique for proving freshness is to use a clock-based mechanism.

   (a) What standard means are there of providing an internationally recognised notion of clock-based time?
   (b) Explain why it is important to protect the integrity of a timestamp.
   (c) Describe in detail how to provide integrity protection for a timestamp.

5. In practice we often have to be more pragmatic about implementing security controls than the theory suggests:

   (a) Under what circumstances might it make sense for an application that employs sequence numbers to accept a sequence number as 'fresh' even if the most recently received sequence number is not greater in value than the previously received sequence number?
   (b) Suggest a simple 'policy' for managing this type of situation.

6. A nonce, as we defined it in Section 8.2.3, is in most cases a pseudorandom number.

   (a) Explain why this means that we cannot guarantee that a particular nonce has not been used before.
   (b) What should we do if we require a *guarantee* that each nonce is used at most once?
   (c) The terms *nonce* and *salt* are used in different areas of cryptography (and not always consistently). Conduct some research on the use of these terms and suggest, in general, what the difference is between them.

7. For each of the following, explain what problems might arise if we base a freshness mechanism on the suggested component:

   (a) an inaccurate clock;
   (b) a sequence number that regularly cycles around;
   (c) a nonce that is randomly generated from a small space.

8. Freshness and liveness are closely related concepts.

   (a) Explain the difference between freshness and liveness by providing examples of applications that require these slightly different notions.
   (b) For each of the freshness mechanisms discussed in this chapter, explain how to use the freshness mechanism to provide a check of liveness.

9. What is the size of the password space if we permit passwords to consist only of:

   (a) eight alphabetic characters (not case sensitive);
   (b) eight alphabetic characters (case sensitive);
   (c) six alphanumeric characters (case sensitive);
   (d) eight alphanumeric characters (case sensitive);
   (e) ten alphanumeric characters (case sensitive);
   (f) eight alphanumeric characters and keypad symbols (case sensitive);

10. FIPS 181 describes a standard for an automated password generator.

   (a) What desirable password properties do passwords generated using FIPS 181 have?
   (b) How does FIPS 181 generate the required randomness?

11. Let E be a symmetric encryption algorithm (such as AES), K be a publicly known symmetric key, and P be a password. The following function F has been suggested as a one-way function suitable for storing passwords:

$$F(P) = E_K(P) \oplus P.$$

   (a) Explain in words how to compute $F(P)$ from P.
   (b) Since the key K is publicly known, explain why an attacker cannot reverse $F(P)$ to obtain P.
   (c) What advantages and disadvantages does this one-way function have over the UNIX password function described in Section 8.4.2?

12. An alternative function for storing passwords is *LAN Manager hash*.

   (a) Which applications use LAN Manager hash?
   (b) Explain how LAN Manager hash uses symmetric encryption to protect a password.
   (c) What criticisms have been made about the security of LAN Manager hash?

13. Passwords stored on a computer in encrypted form are a potential attack target. Explain how encrypted passwords can be attacked by using:

   (a) an exhaustive search;
   (b) a dictionary attack;
   (c) rainbow tables.

14. Biometric technologies provide a source of identity information that could be used as part of an entity authentication mechanism based on use of an electronic identity card.

   (a) What biometric technology might be suitable for such an application?
   (b) What issues (technical, practical and sociological) might arise through the use of your chosen biometric technology?

(c) Where might you also deploy cryptographic mechanisms in the overall implementation of such a scheme?

15. There are several commercial technologies for implementing dynamic passwords based on security tokens that employ a clock-based mechanism. Find a commercial product based on such a mechanism.

(a) What 'factors' does your chosen product rely on to provide authentication?
(b) In what ways is your chosen technology stronger than basic (static) passwords?
(c) Explain how the underlying mechanism in your chosen technology differs from the challenge–response mechanism that we looked at in Section 8.5.
(d) Find out how your chosen technology manages the issues that were raised in Section 8.2.1 concerning clock-based mechanisms.

16. Explain how to implement a dynamic password scheme based on the use of sequence numbers.

17. A telephone banking service uses a dynamic password scheme that employs a clock-based mechanism but does not use any authentication between the user and the token.

(a) What is the potential impact if the token is stolen?
(b) How might the bank address this risk through token management controls and authentication procedures?

18. Explain why a stream cipher would be a poor choice for the encryption mechanism used to compute responses to challenges in a dynamic password scheme based on challenge–response.

19. Some online banks implement the following dynamic password scheme:

- When a user wishes to log on they send a request for a 'one-time' password in order to access the banking service.
- The bank generates a one-time password and sends this by SMS to the user's mobile phone.
- The user reads the SMS and enters the one-time password in order to access the service.
- If the presented one-time password is correct, the user is given access to the service.

Compare this approach with the dynamic password schemes that we discussed in this chapter, from a:

(a) security perspective;
(b) efficiency perspective;
(c) business perspective (costs, processes, business relationships).

20. Our list of general categories of identification information in Section 8.3.3 is by no means exhaustive.

    (a) Another possible category of identification information is *somewhere the claimant is*. Provide an example of entity authentication based on location information.

    (b) Can you think of any other categories of identification information?

21. In Table 1.2 we provided a mapping of cryptographic primitives that can be used to help provide different cryptographic services (rather than largely providing these services on their own). Now that we have completed our review of cryptographic primitives, attempt to explain all the 'yes' entries in this table by providing examples of each primitive being used as part of the provision of the relevant service.

VtuCircle

9 Cryptographic Protocols

We have now completed our review of the most fundamental cryptographic primitives that make up the cryptographic toolkit. We have seen that each primitive provides very specific security services that can be applied to data. However, most security applications require different security services to be applied to a variety of items of data, often integrated in a complex way. What we must now do is examine how cryptographic primitives can be combined together to match the security requirements of real applications. This is done by designing cryptographic protocols.

We begin this chapter with a brief introduction to the idea of a cryptographic protocol. We then examine and analyse some very simple cryptographic protocols. Finally, we discuss the important class of cryptographic protocols that provide entity authentication and key establishment.

At the end of this chapter you should be able to:

- Explain the concept of a cryptographic protocol.
- Analyse a simple cryptographic protocol.
- Appreciate the difficulty of designing a secure cryptographic protocol.
- Justify the typical properties of an authentication and key establishment protocol.
- Appreciate the significance of the Diffie–Hellman protocol and variants of it.
- Compare the features of two authentication and key establishment protocols.

9.1 Protocol basics

We begin by providing two different, but related, motivations for the need for cryptographic protocols.

9.1.1 Operational motivation for protocols

It is rare to deploy a cryptographic primitive in isolation to provide a single security service for a single piece of data. There are several reasons for this. Many applications:

Have complex security requirements. For example, if we wish to transmit some sensitive information across an insecure network then it is likely that we will want confidentiality *and* data origin authentication guarantees (see Section 6.3.6 and Section 7.4.2).

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.

We thus require a process for specifying precisely how to apply cryptographic primitives during the exchange of data between entities in such a way that the necessary security goals are met.

9.1.2 Environmental motivation for protocols

An alternative motivation for cryptographic protocols comes from the environments in which they are likely to be deployed.

The idea of a *protocol* should be quite familiar. We engage in protocols in many aspects of daily life. For example, most cultures have established protocols that run

when two people who have not met before are introduced to one another (such as smile, shake hands, then exchange business cards). Diplomatic protocols are an example of a more formally documented class of protocols. These are procedures designed to achieve diplomatic goals, independent of the culture and language of the participating countries and diplomats. Indeed, it is for the very reason that countries, languages and diplomats are all *different* that diplomatic protocols are necessary. A potentially more familiar example of a protocol is the sequence of procedural and legal processes involved in sale or purchase of a property within a particular legal jurisdiction.

For similar reasons, protocols are important for electronic communications. Different computing devices run on different hardware platforms, using different software, and communicate in different languages. When considering environments such as the Internet, it at first seems incredible that this diversity of devices can communicate with one another at all. The secret is protocols, in this case communication protocols. The communication protocol TCP/IP enables any device connected to the Internet to talk to any other device connected to the Internet. TCP/IP provides a common process for breaking data into small packets, addressing them, routing them through a network, reassembling them, and finally checking that they have arrived correctly. They can then be interpreted and processed by the receiving device.

In a similar way, a cryptographic protocol provides a common process that allows security goals to be achieved between a number of devices, regardless of the nature of the devices involved.

9.1.3 Components of a cryptographic protocol

A *cryptographic protocol* is a specification of all the events that need to take place in order to achieve some required security goals. In particular, a cryptographic protocol needs to specify:

The protocol assumptions – any prerequisite assumptions concerning the environment in which the protocol will be run. While this in practice involves assumptions about the entire environment (including, for example, security of devices used in the protocol), we will generally restrict our attention to clarifying the cryptographic assumptions, such as the strength of cryptographic primitives used and the possession of cryptographic keys by the participating entities. *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*?

If a cryptographic protocol is followed correctly, in other words all protocol messages are well formed and occur in the correct order, and all actions complete successfully, then hopefully the security goals will be met. If at some stage a protocol message is not received correctly, or an action fails, then the protocol itself is said to *fail* and none of the security goals can be safely considered to have been met. It is wise to have pre-specified rules for deciding how to proceed following a protocol failure. In the simplest case this might involve a rerun of the protocol.

Even in simple applications, such as sending an integrity protected message from one entity to another, we still need to spell out the precise steps that need to be taken in order to achieve the required security goal. Cryptography is thus always used within a cryptographic protocol of some sort, albeit sometimes a rather simple one.

However, most cryptographic protocols are complicated to design and analyse. It is failures in the design and implementation of cryptographic protocols that lead to many security weaknesses in real cryptographic applications. In this chapter we will explain some of the subtleties involved in protocol design and demonstrate some of the basic skills required to 'make sense' of a cryptographic protocol.

9.2 From objectives to a protocol

The design of a cryptographic protocol is a process that commences with a real security problem that needs to be solved and ends with the specification of a cryptographic 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. While we will focus on security objectives, it is important to realise that there may be other objectives that are also important, 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.

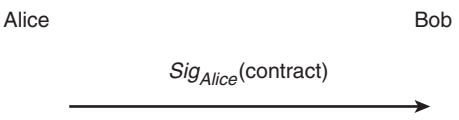

Figure 9.1. A simple cryptographic protocol providing non-repudiation

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 do not specify *which* algorithm is to be used. Nor do we specify whether the digital signature scheme is with appendix or message recovery (see Section 7.3.3). 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.

9.2.2 Challenges of the stages of protocol design

The very simple example that we have just discussed is so elementary that it hides the complexity that is normally involved with each design stage of a cryptographic protocol. While we will shortly examine a slightly more complex protocol, it is important to note that most applications have much more sophisticated security requirements. This introduces complexity throughout the design process.

DEFINING THE OBJECTIVES

It can be very difficult to determine in advance exactly what security requirements a particular application has. Failure to get this correct from the outset is likely to have serious consequences. Thus great care needs to be taken to conduct a sufficiently rigorous risk analysis exercise in advance, so that the security objectives that are defined are complete.

DETERMINING THE PROTOCOL GOALS

The translation of the security goals into cryptographic requirements is, in theory, the most straightforward of the design stages. However, just like any translation exercise, this needs to be done by someone sufficiently expert that the conversion process is accurately conducted.

SPECIFYING THE PROTOCOL

Designing a cryptographic protocol that meets the specified goals can be a very difficult task. This difficulty often comes as a surprise to system designers without cryptographic expertise, who may be tempted to design their own cryptographic protocols. Even if strong cryptographic primitives are used, an insecure protocol will not meet the intended security objectives.

This is true even for the most basic security goals. In Section 9.4 we will discuss cryptographic protocols that are designed to meet the relatively straightforward security goals of mutual entity authentication and key establishment. Hundreds of cryptographic protocols have been proposed to meet these security goals, but many contain design flaws.

The simple message here is that, just as for the design of cryptographic primitives, all three of the design stages (but most importantly the last one) are best left to experts. Indeed, even among such experts, the process of designing cryptographic protocols that can be *proven* to implement specified cryptographic goals remains a challenging one (see Section 3.2.5).

STANDARDS FOR CRYPTOGRAPHIC PROTOCOLS

In the light of the difficulties just discussed about designing cryptographic protocols, one sensible strategy would be to only use cryptographic protocols that have been adopted in relevant standards. For example:

- the PKCS standards include some cryptographic protocols for implementing public-key cryptography;
- ISO/IEC 11770 specifies a suite of cryptographic protocols for mutual entity authentication and key establishment;
- SSL/TLS specifies a protocol for setting up a secure communication channel (see Section 12.1).

The adoption of standardised cryptographic protocols is highly recommended, however, there are two potential issues:

Application complexity. Many applications have sufficiently complex security goals that there may not be an already approved cryptographic protocol that meets the precise application security goals. For major applications it may be necessary to design an entirely new dedicated standard. For example, the *Trusted Computing Group* have had to design and standardise their own set of cryptographic protocols for the implementation of trusted computing. Indeed, unusually, this process required the design of a new cryptographic primitive as well as cryptographic protocols.

Precision of fit. If a standardised protocol is considered for use then it must be the case that the application security goals are precisely those of the standard protocol. If a standard protocol needs to be even slightly changed then it may be the case that the protocol no longer meets its original security goals. This issue

also applies to cryptographic primitives themselves, since if we even slightly amend the key schedule of AES then the resulting algorithm is no longer AES.

9.2.3 Assumptions and actions

We now reconsider the simple cryptographic protocol shown in Figure 9.1. Recall the four components of a cryptographic protocol that we identified in Section 9.1.3, namely assumptions, flow, messages and actions. In fact, Figure 9.1 only describes the flow (one message from Alice to Bob) and the message (a contract digitally signed by Alice).

There are several problems that could arise with the protocol in Figure 9.1:

1. If Alice and Bob have not agreed on the digital signature scheme that they are going to use then Bob will not know which verification algorithm to use.
2. If Alice does not already possess a signature key then she will not be able to digitally sign the contract.
3. If Bob does not have access to a valid verification key that corresponds to Alice's signature key then he will not be able to verify the digital signature.
4. If Bob does not verify the digital signature received from Alice then he cannot have any assurance that Alice has provided him with correctly formed data that can later be used to settle a potential dispute.

ASSUMPTIONS

The simple protocol in Figure 9.1 only makes sense if we make the following *assumptions* regarding the environment in which the protocol is run. Before the protocol is run:

- Alice and Bob agree on the use of a strong digital signature scheme. *This addresses the first problem.*
- Alice has been issued with a signature key. *This addresses the second problem.*
- Bob has access to a verification key corresponding to Alice's signature key. *This addresses the third problem.*

Indeed, it may be appropriate to generalise these assumptions to one that states that before the protocol is run there exists a supporting public-key management system for overseeing the management of all required cryptographic keys (see Chapter 11).

ACTIONS

The description of the simple protocol in Figure 9.1 is only complete if we specify the following *action* that needs to take place as part of the protocol. After receiving the message from Alice:

- Bob verifies the digital signature received from Alice. *This addresses the fourth problem.*

This action should really be specified as part of the protocol itself. It is common practice to leave certain actions as implicit in the description of a cryptographic protocol. This, however, is slightly dangerous. For example, SSL/TLS is commonly adopted to secure the communication channel between a client and a web browser (see Section 12.1). During this protocol the web server provides a digitally signed public-key certificate (see Section 11.1.2) to the client in order to facilitate entity authentication of the web server. The implicit action of *verifying the public-key certificate received from the web server* is often ignored, thus exposing this protocol to a range of attacks.

9.2.4 The wider protocol design process

While the focus of this chapter is on the *design* of cryptographic protocols, it is important to recognise that the design is only one stage in a wider process. Just as we discussed for cryptographic primitives in Section 3.2.4, it is more likely that security problems arise from failures to implement a cryptographic protocol properly. This can manifest itself in numerous ways, including:

- weakness in the implementation of a specific cryptographic primitive used by the protocol;
- instantiation of a cryptographic primitive used in the protocol by a weak cryptographic algorithm;
- failure to implement the entire protocol correctly (for example, omitting an important action);
- weakness in the supporting key management processes.

Coupled with the difficulties in designing secure cryptographic protocols that we discussed in Section 9.2.2, it should be clear that the entire deployment process of a cryptographic protocol requires great care.

9.3 Analysing a simple protocol

In this section we will look at another simple cryptographic protocol, but one that has more security goals than the example in Section 9.2. We argued throughout Section 9.2 that cryptographic protocol design was best left to experts, thus the reason for studying this simple application is to provide insight into the complexities of cryptographic protocol design, rather than to develop proficiency in it. There are two other reasons for studying such an example in some depth:

1. We will see that there are many different ways, each with its own subtle advantages and disadvantages, of designing a cryptographic protocol that meets some specific security goals.
2. While designing proprietary cryptographic protocols is not generally recommended, it is useful to be able to analyse, at least at a high level, whether a given cryptographic protocol achieves its goals.

9.3.1 A simple application

We now describe a simple security scenario. This is probably too simple a scenario to have any real application, however, even this scenario has sufficient complexity to provide us with an example that we can analyse.

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 our main security objective, which we will refer to as a check of liveness (see Section 8.2).

In order to make the example slightly more interesting, we 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. We 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

We now translate these objectives into concrete 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*. When designing protocol goals to meet these objectives it can be helpful to consider what could go wrong, hence we will motivate each protocol goal by a potential failure of the protocol if this goal is not met.

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

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.

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.

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).

Note that it is the combination of the first two goals that provides the basic guarantee that Alice is alive. However, we will see that the third goal not only provides more precision, but in some circumstances is essential.

Table 9.1: Notation used during protocol descriptions

r_B	A nonce generated by Bob
$\|$	Concatenation
Bob	An *identifier* for Bob (perhaps his name)
$\text{MAC}_K(data)$	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

We will now examine seven candidate cryptographic protocols and discuss the extent to which they meet the three security goals. Most importantly, we will see that:

- there is more than one cryptographic protocol that meets these goals;
- some protocols only meet the goals if we make some additional assumptions;
- some protocols that appear at first to meet the goals, in fact, do not.

Table 9.1 indicates the notation used to describe the seven candidate protocols.

9.3.2 Protocol 1

Figure 9.2 shows the protocol flow and messages of our first candidate protocol. We describe this protocol in detail in order to clarify the notation.

PROTOCOL ASSUMPTIONS

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

Bob has access to a source of randomness. This is necessary because the protocol requires Bob to be able to generate a nonce. We naturally also assume that this generator is 'secure' in order to guarantee unpredictability of the output.

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.

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.

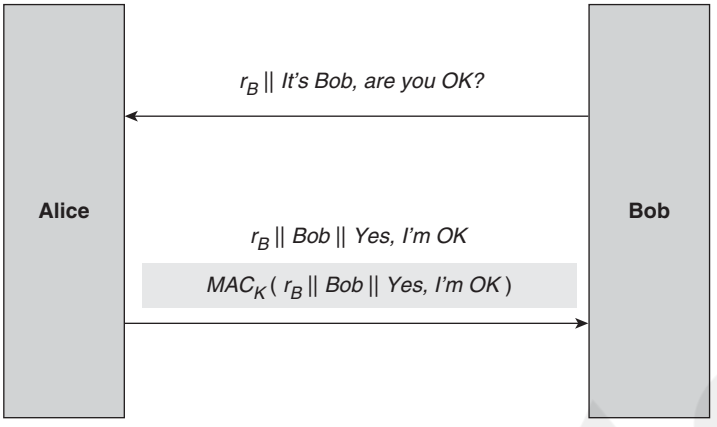

Figure 9.2. Protocol 1

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. We will discuss suitable protocols in Section 9.4. Technically, 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 r_B (this is an implicit action that is not described in Figure 9.2, as is the fact that he stores it for later checking purposes).
   (b) Bob concatenates r_B 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 r_B 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 (all of which are implicit actions that are not shown in Figure 9.2):
   (a) Bob checks that the received reply text consists of a valid r_B (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

We now check whether, if it successfully completes, Protocol 1 meets the required goals:

Data origin authentication of Alice's reply. Under our 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.

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 discussed in Section 8.2.3, 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. Firstly, and most importantly, the reply contains the nonce r_B, 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 r_B in the reply makes it almost certain that the reply corresponds to his request.
2. The reply contains the identifier *Bob*.

It will not be immediately obvious why both of these pieces of data are needed (the first might seem enough). However, in Protocol 3 we will discuss what might happen if the identifier *Bob* is removed from this protocol.

Thus we deduce that Protocol 1 does indeed meet the three security goals for our simple application. Note that all four of the components of a cryptographic protocol that we identified in Section 9.1.3 play a critical role in Protocol 1:

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. For example, we will see in Protocol 3 what happens if the identifier Bob is omitted from the reply text.

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.

Informal assurance that Alice is indeed alive comes from the fact that a valid MAC is produced on a message that includes a newly generated nonce. Only Alice could have generated the MAC and, because she includes the nonce, she must have done this after Bob made his request. However, such informal arguments have no place in cryptographic analysis because it is the details that are important. We will later examine several protocols that appear to satisfy a similar informal analysis, but which fail to meet the security goals.

REMARKS

We have seen that Protocol 1 meets the security goals and hence is a suitable protocol to use in our simple application. We will use Protocol 1 as a 'benchmark' protocol against which later protocols will be compared. We have described Protocol 1 in greater detail than we intend to treat later protocols. By doing so, we have hopefully clarified notation and how to interpret the figures indicating protocol messages and flow.

9.3.3 Protocol 2

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

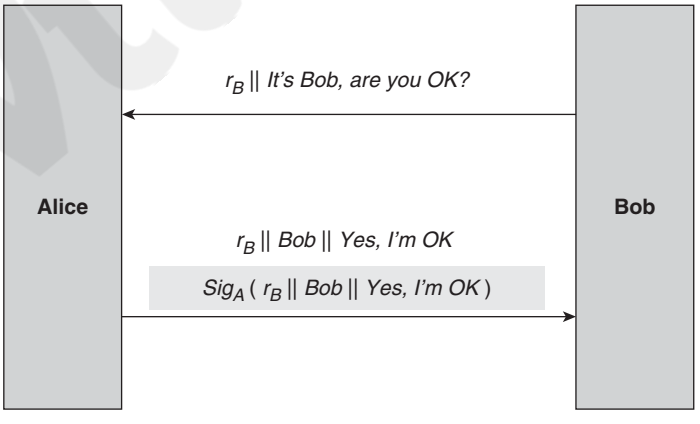

Figure 9.3. Protocol 2

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 our 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.

We 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.

The real differences between these two protocols are primarily in the key management issues that arise from the different assumptions. We will discuss these in much greater detail in Chapters 10 and 11. It suffices to note that this is such an application-dependent issue that many cryptographic protocols come in two different 'flavours', along the lines of Protocol 1 and Protocol 2.

A good example is the suite of authentication and key establishment protocols standardised in ISO 11770. Many of the protocols proposed in part 2 of ISO 11770, which is concerned with symmetric techniques, have public-key analogues in part 3 of the standard.

9.3.4 Protocol 3

From Figure 9.4 it should be clear that the protocol flow and messages of our third candidate protocol are almost identical to Protocol 1.

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 r_B in the reply appears, superficially, to provide this assurance since r_B 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', we call this a *reflection attack* against Protocol 3. The attack is depicted in Figure 9.5.

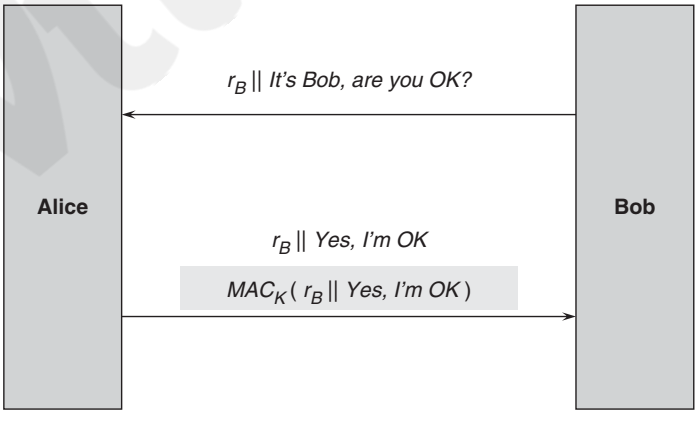

Figure 9.4. Protocol 3

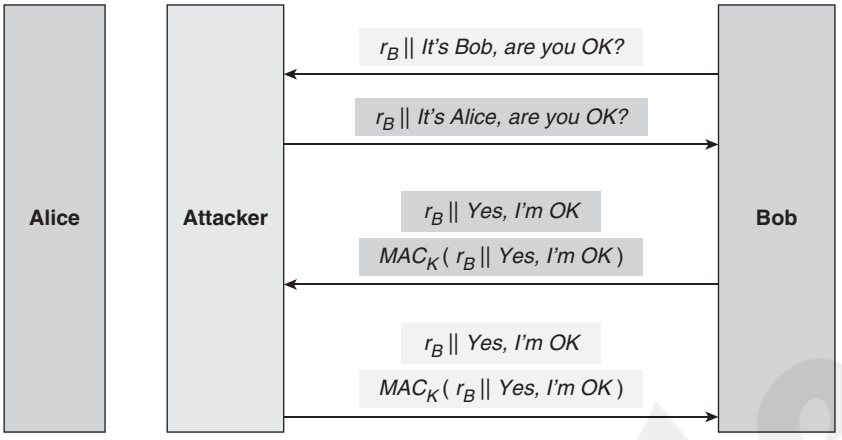

Figure 9.5. Reflection attack against Protocol 3

To see how the reflection attack works, we assume that an attacker is able to intercept and block all communication between Alice and Bob. We 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 tempting to suggest that Bob will regard the receipt of a message containing his nonce r_B as rather strange and will surely reject it. However, we must 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!

We can regard 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. This is why we describe the two runs as 'nested'.

Thus, if this reflection attack is feasible, then Protocol 3 does not meet the third security goal. It is tempting to respond to this by pointing out that if this reflection attack is *not* feasible then Protocol 3 is secure. However, by this stage in our cryptographic studies it should be clear that this is not the attitude of a wise cryptographic designer. If there are circumstances under which a cryptographic protocol might fail then we should really consider the protocol as insecure.

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

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. This is a cumbersome solution that makes Protocol 3 less efficient. Further, the additional actions that Bob needs to perform during the protocol are not 'obvious' and, although they could be clearly specified in the protocol description, some implementations might fail to include them.

Include an identifier. A far better solution would be to 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. There are many different identifiers that could be used, but one of the simplest is to include the name of the intended recipient in the reply, in other words add an identifier *Bob* into the reply text. If we do this then we convert Protocol 3 into Protocol 1.

REMARKS

Protocol 3 has raised an important issue. Even when considering such a simple set of protocol goals, we have come up against a subtle attack. It is generally regarded as good practice in the design of cryptographic protocols to include the identifiers of recipients in protocol messages to prevent reflection attacks of this type.

There are several other general classes of attack that can be launched against cryptographic protocols. These include *interleaving attacks*, which exploit several parallel protocol sessions and switch messages from one protocol run into another. Hopefully our earlier remark in Section 9.2.2 about the dangers of inexperienced designers inventing their own cryptographic protocols is now better justified.

9.3.5 Protocol 4

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

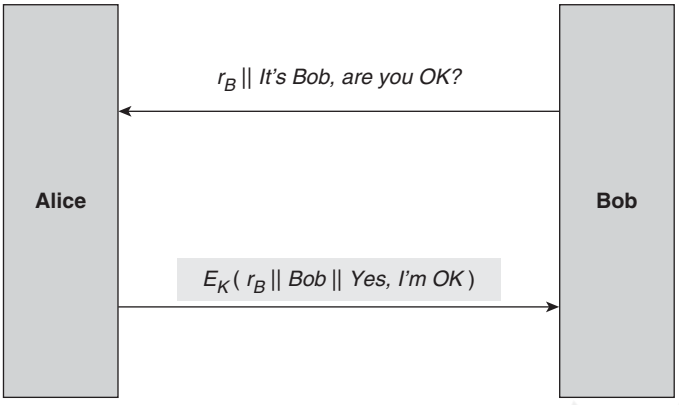

Figure 9.6. Protocol 4

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). Note that, just as for the previous protocols, this assumption does not specify precisely how this encryption algorithm should be substantiated. Thus it could, for example, be either a stream cipher or a block cipher. If it is a block cipher then it could be using any mode of operation. We will see shortly that this ambiguity might lead to problems.

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. We need to 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. We presented this argument in Section 6.3.1. A key management purist might also choose to point out there is inherent danger in using encryption to provide data origin authentication

because the same key K might later be used for encryption, thus abusing the principle of key separation (see Section 10.6.1).

The case for. In Section 6.3.1 we outlined a number of problems that may arise if encryption is used to provide data origin authentication. These mainly arose when the plaintext was long and unformatted. However, 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).

It is safest to argue that Protocol 4 does not meet the three security goals, since the 'case for' requires some caveats concerning the type of encryption mechanism used. For example, it is not likely to meet the goals if we implement E using a stream cipher.

There are cryptographic protocols in standards that do use encryption to deduce data origin authentication in the style of Protocol 4. Such standards normally include advice on what type of encryption mechanism it is 'safe' to use. Good advice in this case would be to use a block cipher in an authenticated-encryption mode of operation, as discussed in Section 6.3.6. We will see such an example in Section 9.4.3. However, encryption tends only to be used in this way if confidentiality of the message data is also required. In Protocol 4 this is not the case, so it would be much better to use Protocol 1.

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. We discussed mechanisms for doing this in Section 8.2.1.

PROTOCOL DESCRIPTION

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

- Instead of generating a nonce r_B, 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.

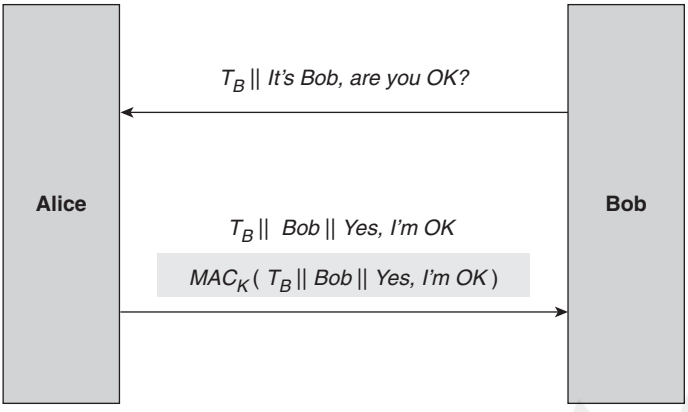

Figure 9.7. Protocol 5

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 T_B, which Bob generated at the start of the protocol. Thus, by the principles discussed in Section 8.2.1, 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.

REMARKS

Protocol 5 can be thought of as the 'clock-based' analogue of Protocol 1. Many cryptographic protocols come in two different 'flavours' such as Protocol 1 and Protocol 5, depending on the type of freshness mechanism preferred.

Note that there is 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.

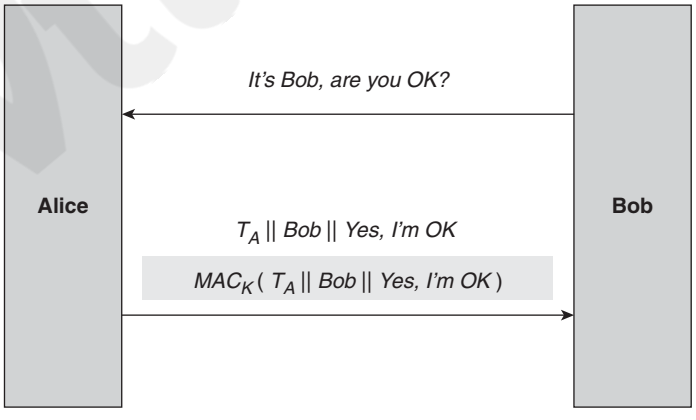

Figure 9.8. Protocol 6

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 T_A concatenated to his identifier *Bob* and a meaningful response to his query (in this case, *Yes, I'm OK*).
   (b) Bob verifies T_A 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. Thus, by the principles discussed in Section 8.2.1, 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. Nonetheless, it requires rather more complicated assumptions than Protocol 1, which seem unnecessary for this simple application.

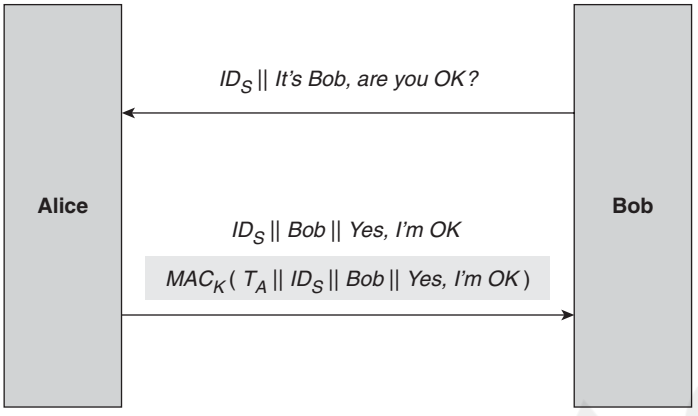

Figure 9.9. Protocol 7

9.3.8 Protocol 7

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

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 ID_S 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 ID_S 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.

However, there is a 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 (see Section 8.2.1).

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.

9.3.9 Simple protocol summary

That is enough protocol variants for now! Hopefully the important points have been highlighted as a result of this analysis:

There is no one correct way to design a cryptographic protocol. Of the seven variants that we studied, three provide all three security goals, despite being different protocols. The choice of the most suitable protocol design thus depends on what assumptions are most suitable for a given application environment.

Designing cryptographic protocols is hard. The deficiencies of several of these protocol variants are very subtle. Given that this application is artificially simple, the complexity of designing protocols for more intricate applications should be clear.

9.4 Authentication and key establishment protocols

The security goals of our simple protocol were rather basic, making it hard to justify the need for such a protocol in a real application. However, the dissection of the simple protocol variants has demonstrated the type of analytical skills required to examine more complex cryptographic protocols with more realistic collections of security goals.

We now reconsider *AKE protocols* (authentication and key establishment), which were introduced at the end of Chapter 8. There are literally hundreds of proposed AKE protocols, since an AKE protocol often has to be tailored to the precise needs of the application for which it is designed. However, 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. Recall from Section 8.3.1 that a problem with entity authentication is that it is achieved only for an instant in time. In practice, we often desire this achievement to be extended over a longer period of time (a *session*). One way of doing this is to bind the establishment of a symmetric key to the entity authentication process. In this way, later use of the key during a session continues to provide confidence that the communication is being conducted between the parties who were authenticated at the instant in time that the key was established. Thus we can maintain, at least for a while, the security context achieved during entity authentication. Of course, exactly *how long* this can be maintained is a subjective and application-dependent issue.

9.4.1 Typical AKE protocol goals

We now break down the general security objectives of an AKE protocol being run between Alice and Bob into more precise security goals. These will not be universal for all AKE protocols, hence we will refer to these as 'typical' security goals that are to be achieved on completion of an AKE protocol:

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

The motivation for the first five of these security goals should be self-evident. The last two goals are more subtle and not always required.

The goal of mutual key establishment is that Alice and Bob do end up with the same key. In many AKE protocols it suffices that key confirmation is *implicit*, with Alice and Bob assuming that they have established the same key because they believe that the protocol completed successfully. Mutual key confirmation goes one step further by requiring *evidence* that the same key has been established. This evidence is usually a cryptographic computation made using the established key.

Mutual key establishment does not impose any requirements on how the established key is generated. Hence, in many cases it may be acceptable that Alice (say) generates a symmetric key and transfers it to Bob during the AKE protocol, in which case Alice has full control of the choice of key. Unbiased key control is required in applications where Alice and Bob do not trust each other to generate a key. Alice may, for example, believe that Bob might, accidentally or deliberately, choose a key that was used on some prior occasion. Unbiased key control is normally achieved either by:

- Generating the key using a component of 'randomness' from each of Alice and Bob, so that the resulting key is not predictable by either of them. This is often termed *joint key control*.
- Using a trusted third party to generate the key.

We choose to distinguish between two families of AKE protocols. We will say that an AKE protocol is based on:

Key agreement, if the the key is established from information that is contributed by each of Alice and Bob; we will discuss an AKE protocol based on key agreement in Section 9.4.2;

Key distribution, if the key is generated by one entity (which could be a trusted third party) and then distributed to Alice and Bob; we will discuss an AKE protocol based on key distribution in Section 9.4.3.

9.4.2 Diffie–Hellman key agreement protocol

The *Diffie–Hellman key agreement protocol*, which we will henceforth refer to simply as the *Diffie–Hellman protocol*, is one of the most influential cryptographic protocols. Not only does it predate the public discovery of RSA, but it remains the basis for the vast majority of modern AKE protocols based on key agreement. We will explain the idea behind the Diffie–Hellman protocol and then examine an example of an AKE protocol that is based on it.

IDEA BEHIND THE DIFFIE–HELLMAN PROTOCOL

The Diffie–Hellman protocol requires the existence of:

- A public-key cryptosystem with a special property, which we discuss shortly. 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, which we discuss shortly. By a 'combination' function, we mean 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. Indeed, it is often used to establish a symmetric key, which can then be used to secure such a channel. It is important to remember that, unless otherwise stated, we assume that all the exchanged messages take place over an unprotected (public) channel that an attacker can observe and, potentially, modify. 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}. As we will discuss in a moment, 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.

The somewhat surprising aspect of the Diffie–Hellman protocol is that *without sharing any secret information*, Alice and Bob are able to jointly generate a secret value by communicating only over a public channel. This was a revolutionary idea when it was first proposed in 1976, and remains a slightly counterintuitive one. This property makes the Diffie–Hellman protocol extremely useful.

INSTANTIATION OF THE DIFFIE–HELLMAN PROTOCOL

In order to fully specify the Diffie–Hellman protocol, we need to find a suitable public-key cryptosystem and a suitable function F. Fortunately, we will not need

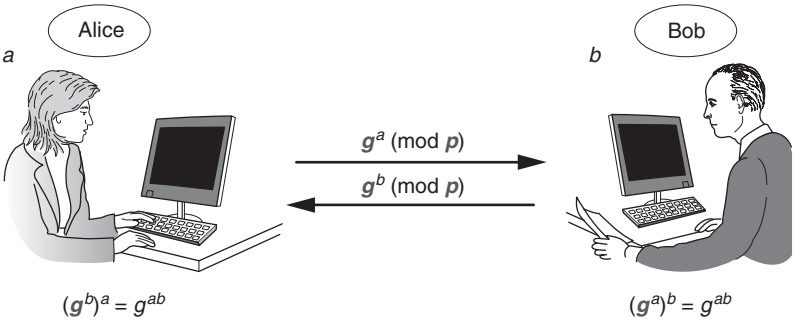

Figure 9.10. Diffie–Hellman protocol

any new public-key cryptosystems to describe the most common instantiation of the Diffie–Hellman protocol. This is because the ElGamal cryptosystem (see Section 5.3) has precisely the property that we need. Equally fortunately, the required function F is very simple.

Just as explained in Section 5.3.4 for ElGamal, we can 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. Note that all calculations are performed modulo p and thus we omit mod p in each computation for convenience. For a reminder of the basic rules of exponentiation, see Section 1.6.1.

1. 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.
2. Bob randomly generates a positive integer b and calculates g^b. Bob sends his public key g^b to Alice.
3. Alice uses g^b and her private key a to compute $(g^b)^a$.
4. Bob uses g^a and his private key b to compute $(g^a)^b$.
5. The special combination function property that we need is 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:

$$(g^a)^b = (g^b)^a = g^{ab}.$$

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. Recall from Section 5.3.3 that

from this information the attacker is unable to calculate either a or b because of the difficulty of the discrete logarithm problem.

2. The main purpose of the Diffie–Hellman protocol is to establish a common cryptographic key K_{AB}. There are two reasons why the shared value $Z_{AB} = g^{ab}$ 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 (see Section 10.3.2). 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. The protocol we have described is just one instantiation of the Diffie–Hellman protocol. In theory, 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. Hence, in contrast to Section 7.3.4, this time ElGamal is 'special'. Note that an important alternative manifestation of the Diffie–Hellman protocol is when an elliptic-curve-based variant of ElGamal is used (see Section 5.3.5), resulting in a protocol with shorter keys and reduced communication bandwidth.

ANALYSIS OF THE DIFFIE–HELLMAN PROTOCOL

We will now test the Diffie–Hellman protocol against the typical AKE protocol security goals that we identified in Section 9.4.1:

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 g^a and g^b) 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 $Z_{AB} = g^{ab}$ 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 g^a 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 $Z_{AB} = g^{ab}$ that he particularly 'likes'. This type of 'attack' is somewhat theoretical since, in practice, the values involved are so large that it would be very hard to conduct (Bob would probably have to try out too many choices of b). Hence it would seem reasonable to argue that joint (and hence unbiased) key control is achieved since any 'manipulation' that Bob can conduct is in most cases rather contrived.

Thus, from the above analysis, the Diffie–Hellman protocol achieves the goals relating to key establishment, but not the goals relating to authentication. We will now show that this is sufficiently problematic that this basic version of the Diffie–Hellman protocol is not normally implemented without further modification.

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. This is arguably the most well known attack against a cryptographic protocol and is one that the designers of any cryptographic protocol need to take measures to prevent.

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.

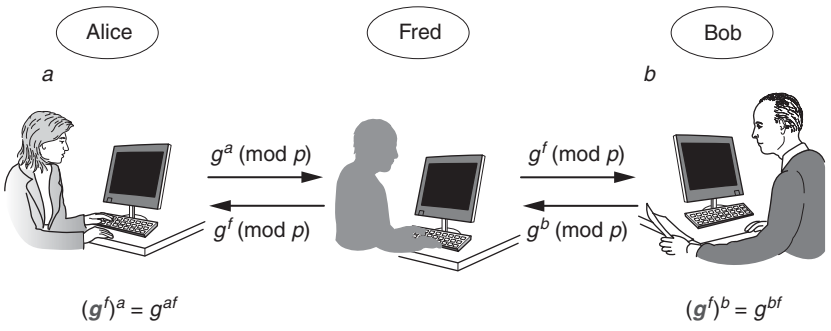

Figure 9.11. Man-in-the-middle attack against the Diffie–Hellman protocol

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 g^f 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 and 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.

Note that at the end of this man-in-the-middle attack, Fred cannot determine the shared value g^{ab} 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 K_{AF} from g^{af} and then encrypts a message to Bob using this key, Bob will not be able to decrypt it successfully because the key K_{BF} that he derives from his shared value g^{bf} will be different from K_{AF}.
- 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 K_{AF}, Fred (who is the only person who can derive both K_{AF} and K_{BF}) can decrypt the ciphertext using K_{AF} to learn the plaintext. He can then re-encrypt the plaintext using K_{BF} 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. To solve this problem, we need to strengthen the Diffie–Hellman protocol so that it meets the authentication goals of Section 9.4.1 as well as the key establishment goals.

AKE PROTOCOLS BASED ON DIFFIE–HELLMAN

Although the basic Diffie–Hellman protocol that we described in Section 9.4.2 does not provide authentication, there are many different ways in which it can be adapted to do so.

We now describe one 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 (see Section 11.1.2). 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 g^a. Alice sends g^a 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 g^b. Next, Bob signs a message that consists of Alice's name, g^a and g^b. Bob then sends g^b 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, g^a and g^b, which she then sends back to Bob. Finally, Alice uses g^b and her private key a to compute $(g^b)^a$.

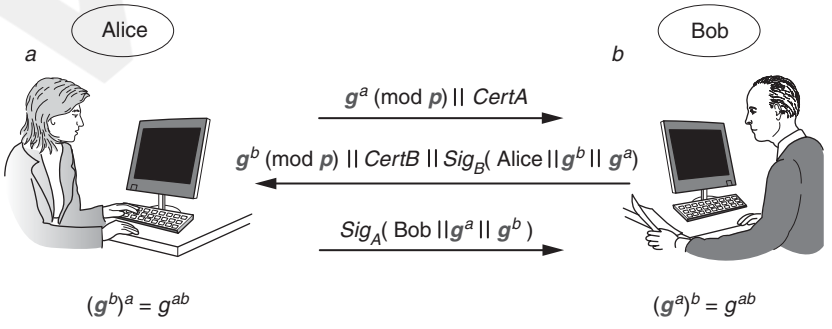

Figure 9.12. Station-to-station protocol

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 g^a and his private key b to compute $(g^a)^b$.

With the exception of the first two, the extent to which the goals of Section 9.4.1 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, g^a and g^b are thus also effectively randomly generated values. Hence we can consider g^a and g^b as being nonces (see Section 8.2.3). At the end of the second STS protocol message, Alice receives a digital signature from Bob on a message that includes her 'nonce' g^a. Similarly, at the end of the third STS protocol message, Bob receives a digital signature from Alice on a message that includes his 'nonce' g^b. Hence, by the principles that we discussed in Section 8.2.3, 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.

Thus, unlike the basic Diffie–Hellman protocol, the STS protocol meets the first five typical AKE protocol goals of Section 9.3.1.

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 K_{AT} with the TTP.
- Bob has already established a long-term shared symmetric key K_{BT} with the TTP.
- Alice and Bob are both capable of randomly generating nonces.

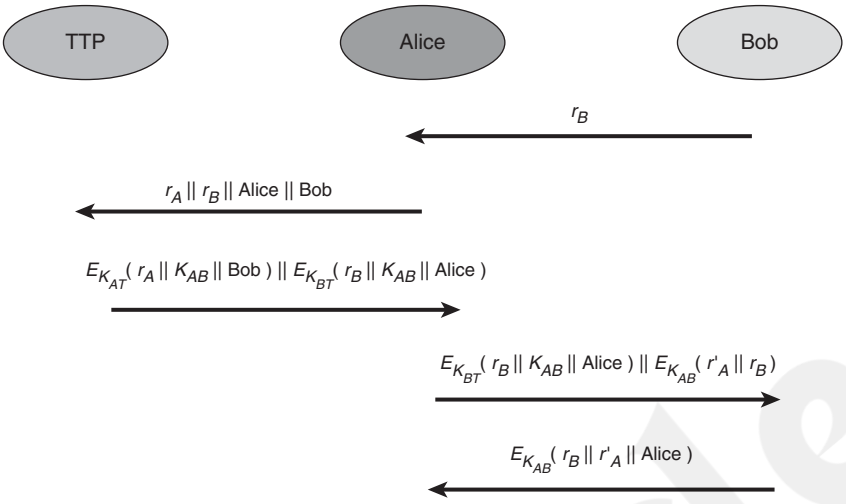

Figure 9.13. An AKE protocol from ISO 9798-2

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 r_B and sending it to Alice.
2. Alice randomly generates a nonce r_A 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 r_A and r_B.
3. The TTP generates a symmetric key K_{AB} and then encrypts it twice. The first ciphertext is intended for Alice and encrypted using K_{AT}. The plaintext consists of r_A, K_{AB} and Bob's name. The second ciphertext is intended for Bob and encrypted using K_{BT}. The plaintext consists of r_B, K_{AB} and Alice's name. The two ciphertexts are sent to Alice.
4. Alice decrypts the first ciphertext using K_{AT} and checks that it contains r_A and Bob's name. She extracts K_{AB}. She then generates a new nonce r'_A. Next, she generates a new ciphertext by encrypting r'_A and r_B using K_{AB}. 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 K_{BT} and checks that it contains r_B and Alice's name. He extracts K_{AB}. He then decrypts the second ciphertext using K_{AB} and checks to see if it contains r_B. He extracts r'_A. Finally, he encrypts r_B, r'_A and Alice's name using K_{AB} and sends this ciphertext to Alice.
6. Alice decrypts the ciphertext using K_{AB} and checks that the plaintext consists of r_B, r'_A and Alice's name. If it does then the protocol concludes successfully.

PROTOCOL ANALYSIS

We now analyse this protocol to see if it meets the typical goals of an AKE protocol specified in Section 9.3.1.

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

1. 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 r_B. Thus this ciphertext is fresh. But who encrypted it? Whoever encrypted it must have known the key K_{AB}. Bob received this key by successfully using K_{BT} to decrypt a ciphertext, which resulted in a correctly formatted plaintext message consisting of r_B, K_{AB} 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 K_{BT}. The format of this plaintext is essentially an 'assertion' by the TTP that the key K_{AB} has been freshly issued (because r_B is included) for use in communication between Bob (because it is encrypted using K_{BT}) 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 K_{AB}. Hence Bob can be sure that he has just been talking to Alice.
2. Alice's perspective is similar. At the end of the last protocol message, Alice receives an encrypted version of her nonce r'_A. This ciphertext, which was encrypted using K_{AB}, is thus fresh. In the third protocol message Alice receives an assertion from the TTP that K_{AB} has been freshly (because r_A is included) issued for use for communication between Alice (because it is encrypted using K_{AT}) 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 K_{AB}. Thus Alice can be sure that she has just been talking to Bob.

Note that our assertions that only Alice and Bob have access to K_{AB} of course assume that the TTP is not 'cheating', since the TTP also knows K_{AB}. However, the whole point of this protocol is that the TTP is *trusted* not to misbehave in this type of way.

Mutual data origin authentication: This is interesting, because 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, recall the 'case for' in our analysis of Protocol 4 in Section 9.3.5. Throughout our 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, such as those discussed in Section 6.3.6. 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 of Section 9.4.1 are provided. A similar AKE protocol is used by the widely deployed *Kerberos* protocol.

VtuCircle