CHAPTER

14

Key Negotiation

Finally, we are ready to tackle the key negotiation protocol. The purpose of this protocol is to derive a shared key that can then be used for the secure channel we defined in Chapter 7.

Complete protocols get quite complicated, and it can be confusing to present the final protocol all at once. Instead, we will present a sequence of protocols, each of which adds a bit more functionality. Keep in mind that the intermediate protocols are not fully functional, and will have various weaknesses.

There are different methods for designing key negotiation protocol, some with supporting proofs of security and some without. We designed our protocol from the ground up—not only because it leads to a cleaner explanation, but also because it allows us to highlight nuances and challenges at each stage of the protocol's design.

14.1 The Setting

There are two parties in the protocol: Alice and Bob. Alice and Bob want to communicate securely. They will first conduct the key negotiation protocol to set up a secret session key k, and then use k for a secure channel to exchange the actual data.

For a secure key negotiation, Alice and Bob must be able to identify each other. This basic authentication capability is the subject of the third part of this book. For now, we will just assume that Alice and Bob can authenticate messages to each other. This basic authentication can be done using RSA

signatures (if Alice and Bob know each other's keys or are using a PKI), or using a shared secret key and a MAC function.

But wait! Why do a key negotiation if you already have a shared secret key? There are many reasons why you might want to do this. First of all, the key negotiation can decouple the session key from the existing (long-term) shared key. If the session key is compromised (e.g., because of a flawed secure channel implementation), the shared secret still remains safe. And if the shared secret key is compromised *after* the key negotiation protocol has been run, the attacker who learns the shared secret key still does not learn the session key negotiated by the protocol. So yesterday's data is still protected if you lose your key today. These are important properties: they make the entire system more robust.

There are also situations in which the shared secret key is a relatively weak one, like a password. Users don't like to memorize 30-letter passwords, and tend to choose much simpler ones. A standard attack is the *dictionary attack*, where a computer searches through a large number of simple passwords. Although we do not consider them here, some key negotiation protocols can turn a weak password into a strong key.

14.2 A First Try

There are standard protocols you might use to do key negotiation. A well-known one based on the DH protocol is the Station-to-Station protocol [34]. Here we will walk you through the design of a different protocol for illustrative purposes. We'll start with the simplest design we can think of, shown in Figure 14.1. This is just the DH protocol in a subgroup with some added authentication. Alice and Bob perform the DH protocol using the first two messages. (We've left out some of the necessary checks, for simplicity.) Alice then computes an authentication on the session key k and sends it to Bob, who checks the authentication. Similarly, Bob sends an authentication of k to Alice.

We don't know the exact form of the authentication at the moment. Remember, we said we assume that Alice and Bob can authenticate messages to each other. So Bob is able to check $\text{AUTH}_A(k)$ and Alice is able to check $\text{AUTH}_B(k)$. Whether this is done using digital signatures or using a MAC function is not our concern here. This protocol merely turns an authentication capability into a session key.

There are some problems with this protocol:

- The protocol is based on the assumption that (p, q, g) are known to both Alice and Bob. Choosing constants for these values is a bad idea.
- It uses four messages, whereas it is possible to achieve the goal using only three.

Alice		**Bob**
Known: (p, q, g)		Known: (p, q, g)
$x \in_{\mathcal{R}} \{1, \ldots, q-1\}$		
	$\xrightarrow{X := g^x}$	
		$y \in_{\mathcal{R}} \{1, \ldots, q-1\}$
	$\xleftarrow{Y := g^y}$	
$k \leftarrow Y^x$		$k \leftarrow X^y$
	$\xrightarrow{\text{AUTH}_\text{A}(k)}$	
		check $\text{AUTH}_\text{A}(k)$
	$\xleftarrow{\text{AUTH}_\text{B}(k)}$	
check $\text{AUTH}_\text{B}(k)$		

Figure 14.1: A first attempt at key negotiation.

- The session key is used as an input to the authentication function. This is not a problem if the authentication function is strong, but suppose the authentication function leaks a few bits about the session key. That would be bad. It certainly would require a new analysis of the entire protocol. A good rule of thumb is to use a secret only for a single thing. Here k will be used as a session key, so we don't want to use it as an argument to the authentication function.
- The two authentication messages are too similar. If, for example, the authentication function is a simple MAC using a secret key known to both Alice and Bob, then Bob could just send the authentication value he received from Alice, and he would not need the secret key to complete the protocol. Thus Alice would not be convinced by the last authentication message.
- Implementations have to be careful not to use k until the authentication messages have been exchanged. This is not a major issue and is a rather simple requirement, but you wouldn't believe what sometimes happens when people try to optimize a program.

We will fix all of these problems over the course of this chapter.

14.3 Protocols Live Forever

We've emphasized the importance of designing systems to withstand the future. This is even more important for protocols. If you limit the size of database fields to 2000 bytes, it might be a problem for some users, but you

can remove the limit in the next version. Not so for protocols. Protocols are run between different participants, and every new version needs to be interoperable with the old version. Modifying a protocol and still keeping it compatible with older versions is rather complicated. Before you know it, you have to implement several versions of the protocol, with a system to decide which version to use.

The protocol version switch becomes a point of attack, of course. If an older protocol is less secure, an attacker has an incentive to force you to use that older protocol. You'd be surprised at how many systems we've seen that suffer from what's known as a version-rollback attack.

It is of course impossible to know all the future requirements, so it might be necessary to define a second version of a protocol at some point. However, the cost of having several protocol versions is high, especially in overall complexity.

Successful protocols live almost forever (we don't care about unsuccessful ones). It is extremely difficult to completely remove a protocol from the world. So it is even more important to design protocols to be future-proof. This is why we can't specify a fixed set of DH parameters for our key negotiation protocol. Even if we chose them to be very large, there is always a danger that future cryptanalytical improvements might force us to change them.

14.4 An Authentication Convention

Before we go on, we will introduce an authentication convention. Protocols often have many different data elements, and it can be hard to figure out exactly which data elements need to be authenticated. Some protocols break because they neglect to authenticate certain data fields. We use a simple convention to solve these problems.

In our protocols, every time a party sends an authentication, the authentication data consists of all the data exchanged so far: all the previous messages, and all the data fields that precede the authentication in the authenticator's message. The authenticator will also cover (be computed over) the identities of the communicants. In the protocol shown in Figure 14.1, Alice's authenticator would not be on k, but on Alice's identifier, Bob's identifier, X, and Y. Bob's authenticator would cover Alice's identifier, Bob's identifier, X, Y, and $\textsc{Auth}_A$.

This convention removes many avenues of attack. It also costs very little. Cryptographic protocols don't exchange that much data, and authentication computations almost always start by hashing the input string. Hash functions are so fast that the extra cost is insignificant.

This convention also allows us to shorten the notation. Instead of writing something like $\textsc{Auth}_A(X, Y)$ we simply write $\textsc{Auth}_A$. As the data to

be authenticated is specified by the convention, we no longer need to write it down explicitly. All further protocols in this book will use this convention.

Just as a reminder: authentication functions only authenticate a string of bytes. Each string of bytes to be authenticated must start with a unique identifier that identifies the exact point in the protocol where this authenticator is used. Also, the encoding of the previous messages and the data fields into this string of bytes must be such that the messages and fields can be recovered from the string without further context information. We've already talked about this in detail, but it is an important point that is easily overlooked.

14.5 A Second Attempt

How do we fix the problems of the previous protocol? We don't want to use a constant DH parameter set, so we'll let Alice choose it and send it to Bob. We'll also collapse the four messages into two, as shown in Figure 14.2. Alice starts by choosing DH parameters and her DH contribution, and sends it all to Bob with an authentication. Bob has to check that the DH parameters are properly chosen and that X is valid. (See Chapter 11 for details of these checks.) The rest of the protocol is similar to the previous version. Alice receives Y and $\textsc{Auth}_\text{B}$, checks them, and computes the DH result.

Alice		**Bob**
Choose suitable (p, q, g) $x \in_{\mathcal{R}} \{1, \ldots, q-1\}$		
	$\xrightarrow{(p,q,g),\ X := g^x,\ \textsc{Auth}_\text{A}}$	
		Check (p, q, g), X, $\textsc{Auth}_\text{A}$ $y \in_{\mathcal{R}} \{1, \ldots, q-1\}$
	$\xleftarrow{Y := g^y,\ \textsc{Auth}_\text{B}}$	
Check Y, $\textsc{Auth}_\text{B}$ $k \leftarrow Y^x$		$k \leftarrow X^y$

Figure 14.2: A second attempt at key negotiation.

We no longer have fixed DH parameters. We use only two messages, we don't use the authentication key directly in any way, and our authentication convention ensures that the strings being authenticated are not similar.

But now we have some new problems:

- What do we do if Bob wants a larger DH prime than Alice? Perhaps Bob has stricter security policies and thinks the DH prime chosen by Alice isn't secure enough. Bob will have to abort the protocol. Maybe he could send an error message along the lines of ''Require DH prime to be at least k bits long,'' but that gets messy and complicated. Alice would have to restart the protocol with new parameters.
- There is a problem with the authentication. Bob isn't sure he is talking to Alice at all. Anybody can record the first message that Alice sends and then later send it to Bob. Bob thinks the message comes from Alice (after all, the authentication checked), and finishes the protocol, thinking he shares a key k with Alice. The attacker doesn't learn k, as he doesn't know x, and without k the attacker cannot break into the rest of the system that uses k. But Bob's logs will show a completed authenticated protocol with Alice, and that is a problem by itself, as it provides erroneous information to investigating administrators.

Bob's problem is called a lack of ''liveness.'' He isn't sure that Alice is ''alive,'' and that he's not talking to a replaying ghost. The traditional way to solve this is to make sure that Alice's authenticator covers a random element chosen by Bob.

14.6 A Third Attempt

We will fix these problems with a few more changes. Instead of Alice choosing the DH parameters, she will simply send her minimal requirements to Bob, and Bob will choose the parameters. This does increase the number of messages to three. (It turns out that most interesting cryptographic protocols require at least three messages. We don't know why, they just do.) Bob only sends a single message: the second one. This message will contain his authenticator, so Alice should send a randomly chosen element in the first message. We use a random nonce for this.

This leads to the protocol shown in Figure 14.3. Alice starts by choosing s, the minimal size of the prime p she wants to use. She also chooses a random 256-bit string as nonce N_a and sends them both to Bob. Bob chooses a suitable DH parameter set and his random exponent, and sends the parameters, his DH contribution, and his authenticator to Alice. Alice completes the DH protocol as usual with the added authenticator.

There is one more problem to be solved. The final result k is a variable-sized number. Other parts of the system might find this difficult to work with. Furthermore, k is computed using algebraic relations, and leaving algebraic

structure in a cryptographic system always scares us. There are a few places where you absolutely need such structure, but we avoid it wherever possible. The danger of algebraic structure is that an attacker might find some way of exploiting it. Mathematics can be an extremely powerful tool. Over the past few decades, we have seen many new proposals for public-key systems, almost all of which have been broken—mostly due to the algebraic structure they contained. Always remove any algebraic structure that you can.

Alice		**Bob**
$s \leftarrow \min p$ size $N_a \in_{\mathcal{R}} 0, \ldots, 2^{256} - 1$		
	$\xrightarrow{s,\, N_a}$	
		Choose (p, q, g) $x \in_{\mathcal{R}} \{ 1, \ldots, q-1 \}$
	$\xleftarrow{(p,q,g),\, X := g^x,\ \text{AUTH}_\text{B}}$	
Check (p, q, g), X, AUTH_B $y \in_{\mathcal{R}} \{ 1, \ldots, q-1 \}$		
	$\xrightarrow{Y := g^y,\ \text{AUTH}_\text{A}}$	
		Check Y, AUTH_A
$k \leftarrow X^y$		$k \leftarrow Y^x$

Figure 14.3: A third attempt at key negotiation.

The obvious solution is to hash the final key. This reduces it to a fixed size, and also destroys any remaining algebraic structure.

14.7 The Final Protocol

The final protocol is shown in short form in Figure 14.4. This is the form that is easiest to read and understand. However, we've left a lot of verification steps out of the protocol to make it easy to read and to focus on the key properties. We simply write ''Check (p, q, g),'' which stands for several verifications. To show you all the required cryptographic checks, the long form of the protocol is given in Figure 14.5.

Bob needs to choose a suitable size for p. This depends on the minimum size required by Alice and his own required minimum size. Of course, Bob should ensure that the value of s is reasonable. We don't want Bob to be required to start generating 100,000-bit primes just because he received an

unauthenticated message with a large value for s in it. Similarly, Alice should not have to start checking very large primes just because Bob sent them. Therefore, both Alice and Bob limit the size of p. Using a fixed maximum limits flexibility; if cryptanalytical progress suddenly forces you to use larger primes, then a fixed maximum is going to be a real problem. A configurable maximum brings with it all the problems of a configuration parameter that almost nobody understands. We've chosen to use a dynamic maximum. Both Alice and Bob refuse to use a prime that is more than twice as long as the prime they would prefer to use. A dynamic maximum provides a nice upgrade path and avoids excessively large primes. You can argue about whether the choice of the factor two is best. Maybe you should use three; it doesn't matter much.

Alice		**Bob**
$s \leftarrow \min p$ size $N_a \in_{\mathcal{R}} 0, \ldots, 2^{256} - 1$		
	$\xrightarrow{s,\ N_a}$	
		Choose (p, q, g) $x \in_{\mathcal{R}} \{1, \ldots, q-1\}$
	$\xleftarrow{(p,q,g),\ X := g^x,\ \text{AUTH}_\text{B}}$	
Check (p, q, g), X, AUTH_B $y \in_{\mathcal{R}} \{1, \ldots, q-1\}$		
	$\xrightarrow{Y := g^y,\ \text{AUTH}_\text{A}}$	
		Check Y, AUTH_A
$k \leftarrow \text{SHA}_d\text{-}256(X^y)$		$k \leftarrow \text{SHA}_d\text{-}256(Y^x)$

Figure 14.4: The final protocol in short form.

The rest of the protocol is just an expansion of the earlier short form. If Bob and Alice are smart, they'll both use caches of suitable DH parameters. This saves Bob from having to generate new DH parameters every time, and it saves Alice having to check them every time. Applications can even use a fixed set of DH parameters, or encode them as defaults in a configuration file, in which case you don't have to send them explicitly. A single DH parameter set identifier would be enough. But be careful when optimizing. Optimizations can end up modifying the protocol enough to break it. There are no simple rules we can give you to check if an optimization breaks a protocol or not. Protocol design is still more an art than a science, and there are no hard rules to live by.

Alice		**Bob**
$s_a \leftarrow \min p$ size		
$N_a \in_{\mathcal{R}} 0, \ldots, 2^{256} - 1$		
	$\xrightarrow{s_a,\, N_a}$	
		$s_b \leftarrow \min p$ size
		$s \leftarrow \max(s_a, s_b)$
		$s \stackrel{?}{\leq} 2 \cdot s_b$
		Choose (p, q, g) with $\log_2 p \geq s - 1$
		$x \in_{\mathcal{R}} \{ 1, \ldots, q - 1 \}$
	$\xleftarrow{(p, q, g),\, X := g^x,\ \textsc{Auth}_{\mathrm{B}}}$	
Check $\textsc{Auth}_{\mathrm{B}}$		
$s_a - 1 \stackrel{?}{\leq} \log_2 p \stackrel{?}{\leq} 2 \cdot s_a$		
$255 \stackrel{?}{\leq} \log_2 q \stackrel{?}{\leq} 256$		
Check p, q both prime		
$q \stackrel{?}{\mid} (p - 1) \wedge g \stackrel{?}{\neq} 1 \wedge g^q \stackrel{?}{=} 1$		
$X \stackrel{?}{\neq} 1 \wedge X^q \stackrel{?}{=} 1$		
$y \in_{\mathcal{R}} \{ 1, \ldots, q - 1 \}$		
	$\xrightarrow{Y := g^y,\ \textsc{Auth}_{\mathrm{A}}}$	
		Check $\textsc{Auth}_{\mathrm{A}}$
		$Y \stackrel{?}{\neq} 1 \wedge Y^q \stackrel{?}{=} 1$
$k \leftarrow \mathrm{SHA}_d\text{-}256(X^y)$		$k \leftarrow \mathrm{SHA}_d\text{-}256(Y^x)$

Figure 14.5: The final protocol in long form.

14.8 Different Views of the Protocol

There are a number of instructive ways to look at a protocol like this. There are a few properties that the protocol should have, and we can look at why the protocol provides them all.

14.8.1 Alice's View

Let's look at the protocol from Alice's point of view. She receives a single message from Bob. She's sure this message is from Bob because it is authenticated, and the authentication includes her random nonce N_a. There is no way anyone could forge this message or replay an old message.

Alice checks that the DH parameters are properly chosen, showing that the DH protocol has all its expected properties. So when she keeps y secret and sends out Y, she knows that only persons who know an x such that $g^x = X$ can compute the resulting key k. This is the basic DH protocol property. Bob authenticated X, and Alice trusts Bob to only do this when he is following the protocol. Thus, Bob knows the appropriate x, and is keeping it secret. Therefore, Alice is sure that only Bob knows the final key k that she derives.

So Alice is convinced she is really talking to Bob, and that the key she derives can be known only to her and Bob.

14.8.2 Bob's View

Now let's look at Bob's side. The first message he receives gives him almost no useful information; it basically states that someone out there has chosen a value s_a and some random bits N_a.

The third message (the second one Bob receives) is different. This is a message that definitely came from Alice, because Alice authenticated it, and we assumed at the outset that Bob can verify an authentication by Alice. The authentication includes X, a random value chosen by Bob, so the third message is not a replay but has been authenticated by Alice specifically for this protocol run. Also, Alice's authentication covers the first message that Bob received, so now he knows that the first message was proper, too.

Bob knows the DH parameters are safe; after all, he chose them. So just like Alice, he knows that only someone who knows a y such that $g^y = Y$ can compute the final key k. But Alice authenticated the Y she sent, and Bob trusts Alice, so she is the only person who knows the corresponding y. This convinces Bob that Alice is the only other person who can compute k.

14.8.3 Attacker's View

Finally, we look at the protocol from the viewpoint of an attacker. If we just listen in on the communications, we see all the messages that Alice and Bob exchange. But the key k is computed using the DH protocol, so as long as the DH parameters are safe, a passive attack like this is not going to reveal anything about k. In other words: we'll have to try an active attack.

One instructive exercise is to look at each data element and try to change it. Here we are quickly stopped by the two authentications. Alice's final authentication covers all the data that was exchanged between Alice and Bob. That means we can't change any data elements, other than to try a replay attack of a prerecorded protocol run. But the nonce and the random X value stop any replay attempts.

That doesn't mean we can't try to play around. We could, for example, change s_a to a larger value. As long as this larger value is acceptable to Bob, most of the protocol would complete normally. There are just three problems. First of all, increasing s_a isn't an attack because it only makes the DH prime larger, and therefore the DH parameters stronger. The second and third problems are the two authentications, which will both fail.

There are some other things that might look like attacks at first. For example, suppose Alice sends Bob s_a and N_a. Bob sends s_a and N_a to Charlie. Charlie replies to Bob with (p, q, g), X, and AUTH_C. Bob now turns this around and forwards (p, q, g) and X to Alice, along with a new authenticator AUTH_B that he computes. Alice replies to Bob with Y and AUTH_A. Bob then sends Y and a new authenticator AUTH_B that he computes to Charlie. What's the result of all this? Alice thinks she's sharing a key k with Bob when in fact she's sharing it with Charlie. And Charlie thinks he's sharing a key with Bob when he's in fact sharing it with Alice. Is this an attack? Not really. Notice that Bob could just do the normal key negotiation with both Alice and Charlie, and then forward all the messages on the secure channel (decrypting each message he receives from Alice and re-encrypting it to Charlie, and vice-versa). This has the same effect; Alice thinks she is communicating with Bob, and Charlie thinks he is communicating with Bob, but they are sending messages to each other instead. And in this scenario Bob knows more (and can do more) than if he ran the ''attack.'' It is true that Alice might send a message to Charlie that makes Charlie believe that Bob agreed to something, but that can only be to Bob's detriment. And an attack that harms the attacker is not one we worry about.

In the real world, you will find many protocols where there are unauthenticated data elements. Most designers wouldn't bother authenticating s_a in our protocol, because changing it would not lead to an attack. (Both Alice and Bob independently verify that the size of p is large enough for them.) Allowing attackers to play around is always a bad idea. We don't want to give them any more tools than necessary. And we can certainly imagine a situation where not authenticating s_a could be dangerous. For example, assume that Bob prefers to use DH parameters from a list built into the program, and only generates new parameters when necessary. As long as Alice and Bob choose to use DH prime sizes that are still in the list, Bob never generates a new parameter set. But this also means that Bob's parameter generation code and Alice's parameter verification code are never used and therefore unlikely to be properly tested. A bug in the parameter generation and testing code could remain hidden until an attacker increases s_a. Yes, this is an unlikely scenario, but there are thousands of unlikely scenarios that are all bad for security. And thousands of low-probability risks add up to a high-probability risk. This is why we are so paranoid about stopping any type of attack anytime we can. It gives us defense in depth.

14.8.4 Key Compromise

So what happens if some other part of the system is compromised? Let's have a look.

If Alice merely loses her authentication key without it becoming known to an attacker, she simply loses the ability to run this protocol. She can still use session keys that were already established. This is very much how you'd expect the protocol to behave. The same holds for Bob if he loses his key.

If Alice loses the session key, without it becoming known to an attacker, she will have to run the key negotiation protocol again with Bob to establish a new session key.

Things get worse if an attacker manages to learn a key. If Alice's authentication key is compromised, the attacker can impersonate Alice from that moment on until the time that Bob is informed and stops accepting Alice's authentications. This is an unavoidable consequence. If you lose your car keys, anyone who finds them can use the car. That is one of the main functions of keys: they allow access to certain functions. This protocol does have the desirable property that past communications between Alice and Bob still remain secret. Even knowing Alice's authentication key doesn't let the attacker find the session key k for a protocol that has already finished, even if the attacker recorded all the messages. This is called *forward secrecy*.[1] The same properties hold with regard to Bob's authentication key.

Finally, we consider the situation where the session key is compromised. The key k is the hash of g^{xy}, where both x and y are randomly chosen. This provides no information about any other key. It certainly provides no information about Alice's or Bob's authentication keys. The value of k in one protocol run is completely independent of the k in another protocol run (at least, it is if we assume that Alice and Bob use a good PRNG).

Our protocol offers the best possible protection against key compromises.

14.9 Computational Complexity of the Protocol

Let's have a look at the computational complexity of our solution. We'll assume that the DH parameter selection and verification are all cached, so we don't count them in the workload of a single protocol run. That leaves the following computations, which Alice and Bob must each perform:

- Three exponentiations in the DH subgroup.
- One authentication generation.

[1] You sometimes see the term *perfect forward secrecy*, or PFS, but we don't use words like ''perfect'' because it never is.

- One authentication verification.
- Various relatively efficient operations, such as random number generation, comparisons, and hash functions.

If symmetric-key authentication is used, the run time of the protocol is dominated by the DH exponentiations. Let's look at how much work that is. Bob and Alice each have to do three modular exponentiations with a 256-bit exponent. This requires about 1150 modular multiplications.[2] To get an idea of how much work this really is, we'll compare this to the computational cost of an RSA signature where the RSA modulus and the DH prime are the same size. For an s-bit modulus, the signature algorithm requires $3s/2$ multiplications if you do not use the CRT (Chinese Remainder Theorem). Using the CRT representation saves a factor of four, so the cost of an RSA signature on s-bit numbers is similar to the cost of doing $3s/8$ multiplications. This leads us to an interesting conclusion: RSA signatures are relatively slower than DH computations when the moduli are large, and relatively faster when the moduli are small. The break-even point is around 3000 bits. This is because DH always uses 256-bit exponents, and for RSA the exponent grows with the modulus size.

We conclude that for the public-key sizes we use, the DH computations cost roughly the same as an RSA signature computation. The DH operations are still the dominant factors in the computations for the protocol, but the cost is quite reasonable.

If RSA signatures are used for the authentication, the computational load more or less doubles. (We can ignore RSA verifications as they are very fast.) This still isn't excessive. CPU speeds are rapidly increasing, and in most practical implementations you'll see that communications delays and overhead take up more time than the computations.

14.9.1 Optimization Tricks

There are a few optimizations that can be applied to the DH operations. Using addition chain heuristics, each exponentiation can be done using fewer multiplications. Furthermore, Alice computes both X^q and X^y. You can use addition sequence heuristics to compute these two results simultaneously and save about 250 multiplications. See Bos [18] for a detailed discussion.

There are also various tricks that make it faster to generate a random y and compute g^y, but these tricks require so much extra system complexity that we'd rather not use them.

[2]This is for the simple binary exponentiation algorithm. A better-optimized algorithm reduces this to less than 1000 multiplications.

14.10 Protocol Complexity

This protocol is also an excellent example of why protocol design is so hideously difficult. Even a simple protocol like this quickly expands to a full page, and we didn't even include all the rules for DH parameter generation or the checks for the authentication scheme that are unknown at our abstraction level. Yet it is already difficult to keep track of everything that goes on. More complicated protocols get much larger. One particular smart card payment system that Niels worked on had a dozen or so protocols specified in 50 pages of symbols and protocol specifications, and that was using a proprietary, highly compact notation! There were 50 more densely written pages needed to cover the security-critical implementation issues.

Full documentation of a set of cryptographic protocols can run into hundreds of pages. Protocols quickly get too complicated to keep in your head, and that is dangerous. Once you don't understand it all, it is almost inevitable that a weakness slips in. The above-mentioned project was probably too complex to be fully understood, even by the designers.

A few years later Niels worked with another, commercially available smart card system. This was a well-known and established system that was widely used for many different smart card applications. One day Marius Schilder, a colleague, showed up with a question—or rather, with a large hole in the system. It turns out that two of the protocols had a destructive interference with each other. One protocol computed a session key from a long-term card key, a bit like the key negotiation protocol of this chapter. A second protocol computed an authentication value from the long-term card key. With a bit of tweaking, you could use the second protocol to let the smart card compute the session key, and then send half of the bits to you. With half of the key bits known, breaking the rest of the system was trivial. Oops! This bug was fixed in the next version, but it is a good illustration of the problems of large protocol specifications.

Real-world systems always have very large protocol specifications. Communicating is very complex, and adding cryptographic functions and distrust makes things even harder. Our advice: be very careful with protocol complexity.

One of the fundamental problems in this area is that there are no good modularization notations for protocols, so everything ends up being mixed together. We've already seen that here in this chapter: the DH parameter size negotiation, DH key exchange, and authentication are all merged together. This is not just a combination of loose parts; the specification and implementation mash them all together. It is rather like a really bad and complex

computer program without any modularization. We all know what that leads to, but we've developed modularization techniques to deal with program complexity. Unfortunately, we lack modularization techniques for protocols, and developing such modularization techniques may not be an easy task.

14.11 A Gentle Warning

We've tried to make the design of the protocol look as easy as possible. Please don't be fooled by this. Protocol design is fiendishly difficult, and requires a lot of experience. Even with lots of experience, it is very easy to get wrong. Though we've tried very hard to get everything right in this book, there is always a possibility that the key negotiation protocol we designed here is wrong. It is important to have professional paranoia and treat all protocols with skepticism.

14.12 Key Negotiation from a Password

So far, we've assumed there is an authentication system to base the key negotiation on. In many situations, all you have is a password. You could just use a MAC keyed with the password to run this protocol, but there is a problem: given a transcript from this protocol (acquired by eavesdropping on the communications), you can test for any particular password. Just compute the authentication value and see whether it is correct.

The problem with passwords is that people don't choose them from a very large set. There are programs that search through all likely passwords. Ideally we'd like a key negotiation protocol where an eavesdropper cannot perform an offline dictionary attack.

Such protocols exist; probably the best-known example is SRP [129]. They provide a significant security improvement. We do not describe password-based key negotiation protocols here. If you are interested in using a password-based key negotiation protocol, you should also be aware of the fact that there are multiple patents in this area.

14.13 Exercises

Exercise 14.1 In Section 14.5, we stated that a property of the protocol could result in providing erroneous information to investigating administrators. Give a concrete scenario where this could be a problem.

Exercise 14.2 Suppose Alice and Bob implement the final protocol in Section 14.7. Could an attacker exploit a property of this protocol to mount a denial-of-service attack against Alice? Against Bob?

Exercise 14.3 Find a new product or system that uses (or should use) a key negotiation protocol. This might be the same product or system you analyzed for Exercise 1.8. Conduct a security review of that product or system as described in Section 1.12, this time focusing on the security and privacy issues surrounding the key negotiation protocol.