CHAPTER

13

Introduction to Cryptographic Protocols

Cryptographic protocols consist of an exchange of messages between participants. We've already seen a simple cryptographic protocol in Chapter 11.

Creating secure protocols is challenging. The main problem is that as a designer or implementer, you are not in control. Up to now we have been designing a system, and have had control over the behavior of various parts. Once you start communicating with other parties, you have no control over their behavior. The other party has a different set of interests than you do, and he could deviate from the rules to try to get an advantage. When working on protocols, you must assume that you are dealing with the enemy.

13.1 Roles

Protocols are typically described as being executed by Alice and Bob, or between a customer and a merchant. Names like ''Alice,'' ''Bob,'' ''customer,'' and ''merchant'' are not really meant to identify a particular individual or organization. They identify a role within the protocol. If Mr. Smith wants to communicate with Mr. Jones, he might run a key agreement protocol. Mr. Smith could take the role of Alice, and Mr. Jones the role of Bob. The next day the roles might be reversed. It is important to keep in mind that a single entity can take on any of the roles.[1] This is especially important to remember when you analyze the protocol for security. We've already seen the

[1]In protocols with three or more participants, it is even possible for a single person to take on more than one role at the same time.

man-in-the-middle attack on the DH protocol. In that attack, Eve takes on the roles of both Alice and of Bob. (Of course, Eve is just another role, too.)

13.2 Trust

Trust is the ultimate basis for all dealings that we have with other people. If you don't trust anybody with anything at all, why bother interacting with them? For example, buying a candy bar requires a basic level of trust. The customer has to trust the merchant to provide the candy and give proper change. The merchant has to trust the customer to pay. Both have recourse if the other party misbehaves. Shoplifters are prosecuted. Cheating merchants risk bad publicity, lawsuits, and getting punched in the nose.

There are several sources of trust:

Ethics Ethics has a large influence in our society. Although very few, if any, people behave ethically all the time, most people behave ethically most of the time. Attackers are few. Most people pay for their purchases, even when it would be laughably easy to steal them.

Reputation Having a ''good name'' is very important in our society. People and companies want to protect their reputation. Often the threat of bad publicity gives them an incentive to behave properly.

Law In civilized societies, there is a legal infrastructure that supports lawsuits and prosecution of people who misbehave. This gives people an incentive to behave properly.

Physical Threat Another incentive to behave properly is the fear of harm if you cheat and are caught. This is one of the sources of trust for drug deals and other illegal trades.

MAD A Cold War term: Mutually Assured Destruction. In milder forms, it is the threat to do harm to both yourself and the other party. If you cheat your friend, she might break off the friendship, doing you both harm. Sometimes you see two companies in a MAD situation, especially when they file patent infringement lawsuits against each other.

All of these sources are mechanisms whereby a party has an incentive not to cheat. The other party knows this incentive, and therefore feels he can trust his opponent to some extent. This is why these incentives all fail when you deal with completely irrational people: you can't trust them to act in their own best interest, which undermines all these mechanisms.

It is hard to develop trust over the Internet. Suppose Alice lives abroad and connects to the ACME website. ACME has almost no reason to trust Alice; of the mechanisms of trust we mentioned, only ethics remains. Legal recourse against private individuals abroad is almost impossible, and prohibitively expensive in most cases. You can't effectively harm their reputation or threaten them, even with MAD.

There is nevertheless a basis of trust between Alice and ACME, because ACME has a reputation to protect. This is important to remember when you design a protocol for e-commerce. If there are any failure modes (and there always are), the failure should be to ACME's advantage, because ACME has an incentive to settle the matter properly by manual intervention.[2] If the failure is to Alice's advantage, the issue is less likely to be settled properly. Furthermore, ACME will be vulnerable to attackers who try to induce the failure mode and then profit by it.

Trust is not a black-and-white issue. It is not that you either trust someone or you don't trust him. You trust different people to different degrees. You might trust an acquaintance with $100 but not with your lottery ticket that just won a $5,000,000 prize. We trust the bank to keep our money safe, but we get receipts and copies of canceled checks because we don't fully trust their administration. The question ''Do you trust him?'' is incomplete. It should be ''Do you trust him with *X*?''

13.2.1 Risk

Trust is fundamental to business, but it is usually expressed as risk rather than trust. Risk can be seen as the converse of trust. Risks are evaluated, compared, and traded in many forms.

When working on cryptographic protocols, it is easier to talk in terms of trust than in terms of risks. But a lack of trust is simply a risk, and that can sometimes be handled by standard risk-management techniques such as insurance. We talk about trust when we design protocols. Always keep in mind that business people think and talk in terms of risks. You'll have to convert between the two perspectives if you want to be able to talk to them.

13.3 Incentive

The incentive structure is another fundamental component of any analysis of a protocol. What are the goals of the different participants? What would they like to achieve? Even in real life, analyzing the incentive structure gives insightful conclusions.

[2] Almost all telephone, mail, and electronic commerce to individuals follows this rule by having the customer pay for the order before it is shipped.

Several times every week we get press reports that announce things like, ''New research has shown that....'' Our first reaction is always to ask: who paid for the research? Research whose results are advantageous to the party who paid for it is always suspect. Several factors are at play here. First, the researchers know what their customer wants to hear, and know they can get repeat contracts if they produce ''good'' results. This introduces a bias. Second, the sponsor of the research is not going to publish any negative reports. Publishing only the positive reports introduces another bias. Tobacco companies published ''scientific'' reports that nicotine was not addictive. Microsoft pays for research that ''proves'' that open source software is bad in some way. Don't ever trust research that supports the company that paid for it.

The authors are personally quite familiar with these pressures. During our years as consultants, we performed many security evaluations for paying customers. We were often harsh—the average product we evaluated was quite bad—and our evaluations often had significantly negative components. That didn't always make us popular with our customers. One of them even called Bruce and said: ''Stop your work and send me your bill. I've found someone who is cheaper and who writes better reports.'' Guess which meaning of ''better'' was intended here?

We see exactly the same problem in other areas. As we wrote the first edition of this book, the press was filled with stories about the accounting and banking industries. Analysts and accountants were writing reports favorable for their clients rather than unbiased evaluations. We blame the incentive structure that gave these people a reason to bias their reports. Looking at the incentives is quite instructive, and something we've done for years. With a bit of practice, it is surprisingly easy, and it yields valuable insights.

If you pay your management in stock options, you give them the following incentive structure: increase the share price over the next three years and make a fortune; decrease the share price, and get a golden handshake. It is a ''Heads I win a lot, tails I win a little'' incentive, so guess what some managers do? They go for a high-risk, short-term strategy. If they get the opportunity to double the amount they gamble they will always take it, because they will only collect the winnings and never pay the loss. If they can inflate the share price for a few years with bookkeeping tricks, they will, because they can cash out before they are found out. Some of the gambles fail, but others pay the bills.

A similar thing happened with the savings and loan industry in the United States in the 1980s. The federal government liberalized the rules, allowing S&Ls to invest their money more freely. At the same time, the government guaranteed the deposits. Now look at the incentive structure. If the investments pay off, the S&L makes a profit, and no doubt management gets a nice bonus. If the investments lose money, the federal government pays off the depositors. Not surprisingly, a bunch of S&Ls lost a lot of money on high-risk investments—and the federal government picked up the bill.

Fixing the incentive structure is often relatively easy. For example, instead of the company itself paying for the audit, the stock exchange can arrange and pay for the audit of the books. Give the auditors a significant bonus for every error they find and you'll get a much more accurate report.

Examples of undesirable incentive structures abound. Divorce lawyers have an incentive to make the divorce very acrimonious, as they are paid for every hour spent fighting over the estate. It is a safe bet that they will advise you to settle as soon as the legal fees exceed the value of the estate.

In American society, lawsuits are common. If an accident happens, every participant has a great incentive to hide, deny, or otherwise avoid the blame. Strict liability laws and huge damage awards might seem good for society at first, but it greatly hinders our ability to figure out why the accident happened, and how we can avoid it in future. Liability laws that are supposed to protect consumers make it all but impossible (as an example) for a company like Firestone to admit there is a problem with their product so we can all learn how to build better tires.

Cryptographic protocols interact in two ways with incentive structures. First, they rely on incentive structures. Some electronic payment protocols do not stop the merchant from cheating the customer, but provide the customer with proof of the cheating. This works because it creates a cryptographic forensic trail. The merchant now has an incentive not to have people out there with proof they were cheated. The proof could be used either in a court case or just to damage the reputation of the merchant.

Second, cryptographic protocols change the incentive structure. They make certain things impossible, removing them from the incentive structure. They can also open up new possibilities and new incentives. Once you have online banking, you create an incentive for a thief to break into your computer and steal your money by that method.

At first, incentives look as if they are mostly materialistic, but that is only part of it. Many people have nonmaterialistic motives. In personal relationships, the most fundamental incentives have little to do with money. Keep an open mind, and try to understand what drives people. Then create your protocols accordingly.

13.4 Trust in Cryptographic Protocols

The function of cryptographic protocols is to minimize the amount of trust required. This is important enough to repeat. The function of cryptographic protocols is to minimize the amount of trust required. This means minimizing both the number of people who need to trust each other and the amount of trust they need to have.

One powerful tool for designing cryptographic protocols is the paranoia model. When Alice takes part in a protocol, she assumes that all other participants are conspiring together to cheat her. This is really the ultimate conspiracy theory. Of course, each of the other participants is making the same assumption. This is the default model in which all cryptographic protocols are designed.

Any deviations from this default model must be explicitly documented. It is surprising how often this step is overlooked. We sometimes see protocols used in situations where the required trust is not present. For example, most secure websites use the SSL protocol. The SSL protocol requires trusted certificates. But a certificate is easy to get. The result is that the user is communicating securely with a website, but he doesn't know which website he is communicating with. Numerous phishing scams against PayPal users have exploited this vulnerability, for example.

It is very tempting not to document the trust that is required for a particular protocol, as it is often ''obvious.'' That might be true to the designer of the protocol, but like any module in the system, the protocol should have a clearly specified interface, for all the usual reasons.

From a business point of view, the documented trust requirements also list the risks. Each point of required trust implies a risk that has to be dealt with.

13.5 Messages and Steps

A typical protocol description consists of a number of messages that are sent between the participants of the protocol and a description of the computations that each participant has to do.

Almost all protocol descriptions are done at a very high level. Most of the details are not described. This allows you to focus on the core functionality of the protocol, but it creates a great danger. Without careful specifications of all the actions that each participant should take, it is extremely difficult to create a safe implementation of the protocol.

Sometimes you see protocols specified with all the minor details and checks. Such specifications are often so complicated that nobody fully understands them. This might help an implementer, but anything that is too complicated cannot be secure.

The solution is, as always, modularization. With cryptographic protocols, as with communication protocols, we can split the required functionality into several protocol layers. Each layer works on top of the previous layer. All the layers are important, but most of the layers are the same for all protocols. Only the topmost layer is highly variable, and that is the one you always find documented.

13.5.1 The Transport Layer

Network specialists must forgive us for reusing one of their terms here. For cryptographers, the transport layer is the underlying communication system that allows parties to communicate. This consists of sending strings of bytes from one participant to another. How this is done is irrelevant for our purposes. What we as cryptographers care about is that we can send a string of bytes from one participant to the other. You can use UDP packets, a TCP data stream, e-mail, or any other method. In many cases, the transport layer needs some additional encoding. For example, if a program executes multiple protocols simultaneously, the transport layer must deliver the message to the right protocol execution. This might require an extra destination field of some sort. When using TCP, the length of the message needs to be included to provide message-oriented services over the stream-oriented TCP protocol.

To be quite clear, we expect that transport layer to transmit arbitrary strings of bytes. Any byte value can occur in the message. The length of the string is variable. The string received should, of course, be identical to the string that was sent; deleting trailing zero bytes, or any other modification, is not allowed.

Some transport layers include things like magic constants to provide an early detection of errors or to check the synchronization of the TCP stream. If the magic constant is not correct on a received message, the rest of the message should be discarded.

There is one important special case. Sometimes we run a cryptographic protocol over a cryptographically secured channel like the one we designed in Chapter 7. In cases like that, the transport layer also provides confidentiality, authentication, and replay protection. That makes the protocol much easier to design, because there are far fewer types of attack to worry about.

13.5.2 Protocol and Message Identity

The next layer up provides protocol and message identifiers. When you receive a message, you want to know which protocol it belongs to and which message within that protocol it is.

The protocol identifier typically contains two parts. The first part is the version information, which provides room for future upgrades. The second part identifies which particular cryptographic protocol the message belongs to. In an electronic payment system, there might be protocols for withdrawal, payment, deposit, refund, etc. The protocol identifier avoids confusion among messages of different protocols.

The message identifier indicates which of the messages of the protocol in question this is. If there are four messages in a protocol, you don't want there to be any confusion about which message is which.

Why do we include so much identifying information? Can't an attacker forge all of this? Of course he can. This layer doesn't provide any protection against active forgery; rather, it detects accidental errors. It is important to have good detection of accidental errors. Suppose you are responsible for maintaining a system, and you suddenly get a large number of error messages. Differentiating between active attacks and accidental errors such as configuration and version problems is a valuable service.

Protocol and message identifiers also make the message more self-contained, which makes much of the maintenance and debugging easier. Cars and airplanes are designed to be easy to maintain. Software is even more complex—all the more reason why it should be designed for ease of maintenance.

Probably the most important reason to include message identifying information has to do with the Horton Principle. When we use authentication (or a digital signature) in a protocol, we typically authenticate several messages and data fields. By including the message identification information, we avoid the risk that a message will be interpreted in the wrong context.

13.5.3 Message Encoding and Parsing

The next layer is the encoding layer. Each data element of the message has to be converted to a sequence of bytes. This is a standard programming problem and we won't go into too much detail about that here.

One very important point is the parsing. The receiver must be able to parse the message, which looks like a sequence of bytes, back into its constituent fields. This parsing must not depend on contextual information.

A fixed-length field that is the same in all versions of the protocol is easy to parse. You know exactly how long it is. The problems begin when the size or meaning of a field depends on some context information, such as earlier messages in the protocol. This is an invitation to trouble.

Many messages in cryptographic protocols end up being signed or otherwise authenticated. The authentication function authenticates a string of bytes, and usually it is simplest to authenticate the message at the level of the transport layer. If the interpretation of a message depends on some contextual information, the signature or authentication is ambiguous. We've broken several protocols based on this type of failure.

A good way to encode fields is to use Tag-Length-Value or TLV encoding. Each field is encoded as three data elements. The tag identifies the field in question, the length is the length of the value encoding, and the value is the actual data to be encoded. The best-known TLV encoding is ASN.1 [64], but it is incredibly complex and we shy away from it. A subset of ASN.1 could be very useful.

Another alternative is XML. Forget the XML hype; we're only using XML as a data encoding system. As long as you use a fixed Document Template Definition (DTD), the parsing is not context-dependent, and you won't have any problems.

13.5.4 Protocol Execution States

In many implementations, it is possible for a single computer to take part in several protocol executions at the same time. To keep track of all the protocols requires some form of protocol execution state. The state contains all the information necessary to complete the protocol.

Implementing protocols requires some kind of event-driven programming, as the execution has to wait for external messages to arrive before it can proceed. This can be implemented in various ways, such as using one thread or process per protocol execution, or using some kind of event dispatch system.

Given an infrastructure for event-driven programming, implementing a protocol is relatively straightforward. The protocol state contains a state machine that indicates the type of message expected next. As a general rule, no other type of message is acceptable. If the expected type of message arrives, it is parsed and processed according to the rules.

13.5.5 Errors

Protocols always contain a multitude of checks. These include verifying the protocol type and message type, checking that it is the expected type of message for the protocol execution state, parsing the message, and performing the cryptographic verifications specified. If any of these checks fail, we have encountered an error.

Errors need very careful handling, as they are a potential avenue of attack. The safest procedure is not to send any reply to an error and immediately delete the protocol state. This minimizes the amount of information the attacker can get about the protocol. Unfortunately, it makes for an unfriendly system, as there is no indication of the error.

To make systems usable, you often need to add error messages of some sort. If you can get away with it, don't send an error message to the other parties in the protocol. Log an error message on a secure log so the system administrator can diagnose the problem. If you *must* send an error message, make it as uninformative as possible. A simple ''There was an error'' message is often sufficient.

One dangerous interaction is between errors and timing attacks. Eve can send a bogus message to Alice and wait for her error reply. The time it takes

Alice to detect the error and send the reply often provides detailed information about what was wrong and exactly where it went wrong.

Here is a good illustration of the dangers of these interactions. Years ago, Niels worked with a commercially available smart card system. One of the features was a PIN code that was needed to enable the card. The four-digit PIN code was sent to the card, and the card responded with a message indicating whether the card was now enabled or not. Had this been implemented well, it would have taken 10,000 tries to exhaust all the possible PIN codes. The smart card allowed five failed PIN attempts before it locked up, after which it would require special unlocking by other means. The idea was that an attacker who didn't know the PIN code could make five attempts to guess the four-digit PIN code, which gave her a 1 in 2000 probability of guessing the PIN code before the card locked up.

The design was good, and similar designs are widely used today. A 1 in 2000 chance is good enough for many applications. But unfortunately, the programmer of that particular smart card system made a problematic design decision. To verify the four-digit PIN code, the program first checked the first digit, then the second, etc. The card reported the PIN code failure as soon as it detected that one of the digits was wrong. The weakness was that the time it took the smart card to send the ''wrong PIN'' error depended on how many of the digits of the PIN were correct. A smart attacker could measure this time and learn a lot of information. In particular, the attacker could find out at which position the first wrong digit was. Armed with that knowledge, it would take the attacker only 40 attempts to exhaustively search the PIN space. (After 10 attempts the first digit would have to be right, after another 10 attempts the second, etc.) After five tries, her chances of finding the correct PIN code rose to 1 in 143. That is much better for the attacker than the 1 in 2000 chance she should have had. If she got 20 tries, her chances rose to 60%, which is a lot more than the 0.2% she should have had.

Even worse, there are certain situations where having 20 or 40 tries is not infeasible. Smart cards that lock up after a number of failed PIN tries always reset the counter once the correct PIN has been used, so the user gets another five tries to type the correct PIN the next time. Suppose your roommate has a smart card like the one described above. If you can get at your roommate's smart card, you can run one or two tries before putting the smart card back. Wait for him to use the card for real somewhere, using the correct PIN and resetting the failed-PIN attempt counter in the smart card. Now you can do one or two more tries. Soon you'll have the whole PIN code because it takes at most 40 tries to find it.

Error handling is too complex to give you a simple set of rules. This is something we as a community do not know enough about yet. At the moment, the best advice we can give is to be very careful and reveal as little information as possible.

13.5.6 Replay and Retries

A replay attack occurs when the attacker records a message and then later resends that same message. Message replays have to be protected against. They can be a bit tricky to detect, as the message looks exactly like a proper one. After all, it *is* a proper one.

Closely related to the replay attack is the retry. Suppose Alice is performing a protocol with Bob, and she doesn't get a response. There could be many reasons for this, but one common one is that Bob didn't receive Alice's last message and is still waiting for it. This happens in real life all the time, and we solve this by sending another letter or e-mail, or repeating our last remark. In automated systems this is called a retry. Alice retries her last message to Bob and again waits for a reply.

So Bob can receive replays of messages sent by the attacker and retries sent by Alice. Somehow, Bob has to deal properly with them and ensure correct behavior without introducing a security weakness.

Sending retries is relatively simple. Each participant has a protocol execution state of some form. All you need to do is keep a timer and send the last message again if you do not receive an answer within a reasonable time. The exact time limit depends on the underlying communication infrastructure. If you use UDP packets (a protocol that uses IP packets directly), there is a reasonable probability that the message will get lost, and you want a short retry time, on the order of a few seconds. If you send your messages over TCP, then TCP retries any data that was not received properly using its own timeouts. There is little reason to do a retry at the cryptographic protocol level, and most systems that use TCP do not do this. Nevertheless, for the rest of this discussion we are going to assume that retries are being used, as the general techniques of handling received retries also work even if you never send them.

When you receive a message, you have to figure out what to do with it. We assume that each message is recognizable, so that you know which message in the protocol it is supposed to be. If it is the message you expect, there is nothing out of the ordinary and you just follow the protocol rules. Suppose it is a message from the ''future'' of the protocol; i.e., one that you only expect at a later point in time. This is easy; ignore it. Don't change your state, don't send a reply, just drop it and do nothing. It is probably part of an attack. Even in weird protocols where it could be part of a sequence of errors induced by lost messages, ignoring a message has the same effect as the message being lost in transit. As the protocol is supposed to recover from lost messages, ignoring a message is always a safe solution.

That leaves the case of ''old'' messages: messages you already processed in the protocol you are running. There are three situations in which this could occur. In the first one, the message you receive has the same message identification as the previous one you responded to, and it is identical in

content to the message you responded to, too. In this case, the message is probably a retry, so you send exactly the same reply you sent the first time. Note that the reply should be the same. Don't recompute the reply with a different random value, and don't just assume that the message you get is identical to the first one you replied to. You have to check.

The second case is when you receive a message that has the same message identification as the message you last responded to, but the message contents are different. For example, suppose in the DH protocol Bob receives the first message from Alice, and then later receives another message that claims to be the first message in the protocol, but which contains different data while still passing the relevant integrity checks. This situation is indicative of an attack. No retry would ever create this situation, as the resent message is never different from the first try. Either the message you just received is bogus, or the earlier one you responded to is bogus. The safe choice is to treat this as a protocol error, with all the consequences we discussed. (Ignoring the message you just received is safe, but it means that fewer forms of active attacks are detected as such. This has a detrimental effect on the detection and response parts of the security system.)

The third case is when you receive a message that is even older than the previous message you responded to. There is not much you can do with this. If you still have a copy of the original message you received at that phase in the protocol, you can check if it is identical to that one. If it is, ignore it. If it is different, you have detected an attack and should treat it as a protocol error. Many implementations do not store all the messages that were received in a protocol execution, which makes it impossible to know whether the message you receive now is or is not identical to the one originally processed. The safe option is to ignore these messages. You'd be surprised how often this actually happens. Sometimes messages get delayed for a long time. Suppose Alice sends a message that is delayed. After a few seconds, she sends a retry that does arrive, and both Alice and Bob continue with the protocol. Half a minute later, Bob receives the original message. This is a situation in which Bob receives a copy of—in protocol terms—a very old message.

Things get more complicated if you have a protocol in which there are more than two participants. These exist, but are beyond the scope of this book. If you ever work on a multiparty protocol, think carefully about replay and retries.

One final comment: it is impossible to know whether the last message of a protocol arrived or not. If Alice sends the last message to Bob, then she will never get a confirmation that it arrived. If the communication link is broken and Bob never receives the last message, then Bob will retry the previous message but that will not reach Alice either. This is indistinguishable to Alice from the normal end of the protocol. You could add an acknowledgment from

Bob to Alice to the end of the protocol, but then this acknowledgment becomes the new last message and the same problem arises. Cryptographic protocols have to be designed in a way that this ambiguity does not lead to insecure behavior.

13.6 Exercises

Exercise 13.1 Describe a protocol you engage in on a regular basis. This might be ordering a drink at a local coffee shop or boarding an airplane. Who are the explicit actors directly involved in this protocol? Are there other actors involved peripherally in this protocol, such as during the setup phase? For simplicity, list at most 5 actors. Create a matrix, where each row is labeled by an actor and each column is labeled by an actor. For each cell, describe how the actor in the row trusts the actor in the column.

Exercise 13.2 Consider the security of your personal computer. List the attackers who might break into your computer, their incentives, and the associated costs and risks to the attacker.

Exercise 13.3 Repeat exercise 13.2, except for a bank instead of your personal computer.

Exercise 13.4 Repeat exercise 13.2, except for a computer at the Pentagon instead of your personal computer.

Exercise 13.5 Repeat exercise 13.2, except for a computer belonging to a criminal organization instead of your personal computer.