Handout 5 (Protocols)

Protocols are the computer science equivalent to fractals and the Mandelbrot set in mathematics. With the latter you have a simple formula which you just iterate and then you test whether a point is inside or outside a region, and voila something magically happened.[1] Protocols are similar: they are simple exchanges of messages, but in the end something "magical" can happen—for example a secret channel has been established or two entities have authenticated themselves to each other. The problem with magic is of course it is poorly understood and even experts often got, and get, it wrong with protocols.

To have an idea what kind of protocols we are interested, let us look at a few examples. One example are (wireless) key fobs which operate the central locking system and the ignition in a car.

The point of these key fobs is that everything is done over the "air"—there is no physical connection between the key, doors and engine. So we must achieve security by exchanging certain messages between the key fob on one side and doors and engine on the other. Clearly what we like to achieve is that I can get into my car and start it, but that thieves are kept out. The problem is that everybody can "overhear" or skim the exchange of messages between the key fob and car. In this scenario the simplest attack you need to defend against is a person-in-the-middle attack. Imagine you park your car in front of a supermarket. One thief follows you with a strong transmitter. A second thief "listens" to the signal from the car and wirelessly transmits it to the "colleague" who followed you and who silently enquires about the answer from the key fob. The answer is then send back to the thief at the car, which then dutifully opens and possibly starts. No need to steal your key anymore.

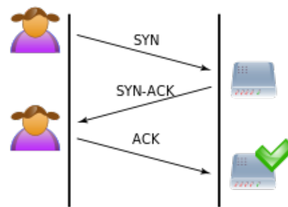
But there are many more such protocols we like to consider. Other examples are wifi—you might sit at a Starbucks and talk wirelessly to the free access point there and from there talk with your bank, for example. Also even if your have to touch your Oyster card at the reader each time you enter and exit the Tube, it actually operates wirelessly and with appropriate equipment over some quite large distance. But there are many many more examples (Bitcoins, mobile phones,...). The common characteristics of the protocols we are interested in here is that an adversary or attacker is assumed to be in complete control over the network or channel over which you exchanging messages. An attacker can install a packet sniffer on a network, inject packets, modify packets,

[1] http://en.wikipedia.org/wiki/Fractal, http://en.wikipedia.org/wiki/Mandelbrot_set

replay old messages, or fake pretty much everything. In this hostile environment, the purpose of protocols (that is exchange of messages) is to achieve some security goal, for example only allow the owner of the car in but everybody else should be kept out.

The protocols we are interested here are generic descriptions of how to exchange messages in order to achieve a goal, be it establishing a mutual secure connection or being able to authenticate to a system. Unlike the distant past where for example we had to meet a person in order to authenticate him or her (via a passport for example), the problem we are facing on the Internet is that we cannot easily be sure who we are "talking" to. The obvious reason is that only some electrons arrive at our computer; we do not see the person, or computer, behind the incoming electrons (messages).

To start, let us look at one of the simplest protocols that are part of the TCP protocol (which underlies the Internet). This protocol does not do anything security relevant, it just establishes a "hello" from a client to a server which the server answers with "I heard you" and the client answers in turn with something like "thanks". This protocol is often called a *three-way handshake*. Graphically it can be illustrated as follows



On the left-hand side is a client, say Alice, on the right-hand side is a server, say. Time is running from top to bottom. Alice initial SYN message needs some time to travel to the server. The server answers with SYN-ACK, which will require some time to arrive at Alice. Her answer ACK will again take some time to arrive at the server. After the messages are exchanged Alice and the server simply have established a channel to communicate over. Alice does not know whether she is really talking to the server (somebody else on the network might have intercepted her message and replied in place of the server). Similarly, the server has no idea who it is talking to. That this can be established depends on what is exchanged next and is the point of the protocols we want to study in more detail.

Before we start in earnest, we need to fix a more convenient notation for protocols. Drawing pictures like the one above would be awkward in the long-run. The notation already abstracts away from a few details we are not interested in: for example the time the messages need to travel between endpoints. What we are interested in is in which order the messages are sent. For the SYN-ACK protocol we will therefore use the notation

$$
\begin{array}{rl}
A \rightarrow S: & SYN \\
S \rightarrow A: & SYN_ACK \\
A \rightarrow S: & ACK
\end{array}
\tag{1}
$$

The left-hand side specifies who is the sender and who is the receiver of the message. On the right of the colon is the message that is send. The order from top to down specifies in which order the messages are sent. We also have the convention that messages like above SYN are send in clear-text over the network. If we want that a message is encrypted, then we use the notation

$$\{msg\}_{K_{AB}}$$

for messages. The curly braces indicate a kind of envelope which can only be opened if you know the key K_{AB} with which the message has been encrypted. We always assume that an attacker, say Eve, cannot get the content of the message, unless she is also in the possession of the key. We explicitly exclude in our study that the encryption can be broken.[2] It is also possible that an encrypted message contains several parts. In this case we would write something like

$$\{msg_1, msg_2\}_{K_{AB}}$$

But again Eve would not be able to know this unless she also has the key. We also allow the possibility that a message is encrypted twice under different keys. In this case we write

$$\{\{msg\}_{K_{AB}}\}_{K_{BC}}$$

The idea is that even if attacker Eve has the key K_{BC} she could decrypt the outer envelop, but still do not get to the message, because it is still encrypted with the key K_{AB}. Note, however, while an attacker cannot obtain the content of the message without the key, encrypted messages can be observed and be recorded and then replayed at another time, or send to another person!

Another very important point is that the notation for protocols such as shown in (1) is a schema how the protocol should proceed. It could be instantiated by an actual protocol run between Alice, say, and the server Calcium at King's. In this case the specific instance would look like

$$
\begin{array}{rl}
\text{Alice} \rightarrow \text{Calcium}: & SYN \\
\text{Calcium} \rightarrow \text{Alice}: & SYN_ACK \\
\text{Alice} \rightarrow \text{Calcium}: & ACK
\end{array}
$$

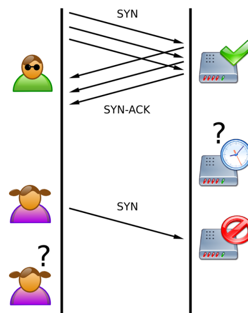
But a server like Calcium of course needs to serve many clients. So there could be the same protocol also running with Bob, say

[2]...which of course is what a good protocol designer needs to ensure and more often than not protocols are broken. For example Oyster cards contain a very weak encryption mechanism which has been attacked.

$$\begin{array}{rl} \text{Bob} \rightarrow \text{Calcium}: & SYN \\ \text{Calcium} \rightarrow \text{Bob}: & SYN_ACK \\ \text{Bob} \rightarrow \text{Calcium}: & ACK \end{array}$$

And these two instances of the protocol could be running in parallel or be at different stages. So the protocol schema shown in (1) can be thought of how two programs need to run on the side of A and S in order to successfully complete the protocol. But it is really just a blue print how the communication is supposed to proceed.

This is actually already a way how such protocols can fail. Although very simple the SYN_ACK protocol can cause headaches for system administrators where an attacker starts the protocol, but does not complete it. This looks graphically like



The attacker sends lots of SYN requests which the server dutifully answers, but needs to keep track of such protocol exchanges. So every time a little bit of memory resource will be eaten away on the server side until all resources are exhausted and when Alice tries to contact the server then the server is overwhelmed and does not respond anymore. This kind of attack are called *SYN floods*.[3]

After reading four pages, you might be wondering where the magic is. For this let us take a closer look at authentication protocols.

Authentication Protocols

The simplest authentication protocol between principals A and B, say is

$$A \rightarrow B : K_{AB}$$

It can be sought of as A sends a common secret to B like a password. The idea is that if only A and B know the key K_{AB} then this should be sufficient for B to infer it is talking to A. But this is of course too naive, if the message can be observed by everybody else on the network. Eve could just record this message A just send, and next time send the same message to B and B would believe it

[3] http://en.wikipedia.org/wiki/SYN_flood

talked to A. But actually it talked to Eve which now clears out As back account if B had been a bank.

A more sophisticated protocol which tries to avoid the replay attack is as follows

$$\begin{aligned} A \rightarrow B &: \; HELLO \\ B \rightarrow A &: \; N \\ A \rightarrow B &: \; \{N\}_{K_{AB}} \end{aligned}$$

With this protocol the idea is that A first sends a message to B saying "I want to talk to you". B sends then a challenge in form of a random number N. In protocols such random numbers are often called *nonce*. What is the purpose of this nonce? Well, if an attacker records A answer, it will not make sense to replay this message, because next time this protocol is run the nonce B sends will be different. So if we run this protocol, what can B infer: it has send out an (unpredictable) nonce to A and received this challenge back, but encoded under the key K_{AB}. If B assumes only A and B know the key K_{AB} and the nonce is unpredictable, then B is able to infer it must be talking to A. Of course the implicit assumption on this inference are that nobody else knows about the key K_{AB} and nobody else can decrypt the message. B of course can decrypt the answer from A and check whether the answer corresponds to the challenge (nonce) B has send earlier.

But what about A? Can A make any assumptions about who it talks to? It dutifully answered the challenge and hopes its bank, say, will be the only one to understand her answer. But is this the case? No! Lets consider an attacker Eve who has control over the network. She could have intercepted the message $HELLO$ and just replied herself to A using a random number… for example one which she observed in a previous run of this protocol. Remember that if a message is send without curly braces it is sent in clear text. Then A would encrypt the nonce with the key K_{AB} and send it back to Eve. She just throws the answer away. A would hope that she talked to B because she followed the protocol, but unfortunately she cannot be sure who she is talking to.

The solution is to follow a *mutual challenge-response* protocol. There A already starts off with a challenge (nonce) on her own.

$$\begin{aligned} A \rightarrow B &: \; N_A \\ B \rightarrow A &: \; \{N_A, N_B\}_{K_{AB}} \\ A \rightarrow B &: \; N_B \end{aligned}$$

As seen, B receives this nonce, N_A, adds his own nonce, N_B and encrypts it with the key K_{AB}. A receives this message, is able to decrypt it since we assume she has the key K_{AB}, and sends back the nonce of B. Let us analyse which assumptions A and B can make after the protocol has run. B received a challenge and answered correctly to A (in the encrypted message). An attacker would just not be able to answer this challenge correctly because the attacker is assumed to not be in the possession of the key K_{AB}; so could not have formed this message. It could also not have just replayed an old message, because A would send out

each time a fresh nonce. So with this protocol you can ensure also for A that it talks to B. I leave you to argue that B can be sure to talk to A. Of course these arguments will depend on the assumptions that only A and B know the key K_{AB} and that nobody can break the encryption unless they have this key and that the nonces are fresh each time the protocol is run.

There might be something mysterious about the nonces, the random numbers, that are sent around. They need to be unpredictable and in this way fulfil an important role in protocols. Suppose

1. I generate a nonce and send it to you encrypted with a key we share
2. you increase it by one, encrypt it under a key I know and send it back to me

In our notation this would correspond to the protocol

$$\begin{array}{l} I \to Y: \ \{N\}_{K_{IY}} \\ Y \to I: \ \{N+1\}_{K_{IY}} \end{array}$$

What can I infer from this simple exchange:

- you must have received my message (it could not just be deflected by somebody on the network, because the response required some calculation; doing the calculation and sending the answer requires the key K_{IY})
- you could only have generated your answer after I send you my initial message (since my N is always new, it could not have been a message that was generated before I myself knew what N is)
- if only you and me know the key K_{IY}, the message must have come from you

Even if this does not seem much information I can glean from such an exchange, it is in fact the basic building blocks for establishing some secret or achieving some security goal (like authentication).

While the mutual challenge-response protocol solves already the authentication problem, there are some problems. One is of course that it requires a pre-shared secret key. That is something that needs to be established beforehand. Not all situations allow such an assumption. For example if I am a whistle blower (say Snowden) and want to talk to a journalist (say Greenwald) then I might not have a secret pre-shared key.

Another problem is that such mutual challenge-response systems often work in the same system in the "challenge mode" but also in the "response mode". For example if two servers want to talk to each other—they would need the protocol in response mode, but also if they want to talk to other servers in challenge mode. Similarly if you in an military aircraft you have to challenge everybody you see, in case there is a friend amongst the targets you like to shoot, but you also have to respond to any of your own anti-aircraft guns on the ground lest they shoot you. In these situations you have to be careful to not decode, or answer, your own challenge. Recall the protocol is

$$A \to B:\ N_A$$
$$B \to A:\ \{N_A, N_B\}_{K_{AB}}$$
$$A \to B:\ N_B$$

but it does not specify who is A and who is B. If, as supposed, the protocol works in response and in challenge mode, then A will be A in one instance, but B in the other. I hope this makes sense. Let us look at the details and lets assume our adversary is E who just deflects our messages back to us.

	challenge mode:		response mode:	
1)	$A \to E$:	N_A		
2)			$E \to A$:	N_A
3)			$A \to E$:	$\{N_A, N'_A\}_{K_{AB}}$
4)	$E \to A$:	$\{N_A, N'_A\}_{K_{AB}}$		
5)	$A \to E$:	N'_A		

In the first step we challenge E with a nonce we created. Since we also run the protocol in "response mode", E can now feed us the same challenge in step 2. We do not know where it came from (it's over the air), but if we are in an aircraft we should better quickly answer it, otherwise we risk to be shot. So we add our own challenge N'_A and encrypt it under the secret key K_{AB} (step 3). Now E does not need to know this key in order to form the correct answer for the first protocol. It will just replays this message back to us in the challenge mode (step 4). I happily accept this message—after all it is encrypted under the secret key K_{AB} and it contains the correct challenge from me, namely N_A. So I accept that E is a friend and send even back the challenge N'_A. The problem is that E now starts firing at me and I have no clue what is going on. I might suspect, erroneously, that an idiot must have leaked the secret key. Because I followed in both cases the protocol to the letter, but somehow E, unknowingly to me with my help, managed to disguise as a friend. As a pilot, I would be a bit peeved at that moment and would have preferred the designer of this challenge-response protocol had been a tad smarter. For one thing they violated the best practice in protocol design of using the same key, K_{AB}, for two different purposes—challenging and responding. They better had used two different keys. This would have averted this attack and would have saved me a lot of trouble.

Trusted Third Parties

One limitation the protocols we discussed so far is that they pre-suppose a secret shared key. As already mentioned, this is a convenience we cannot always assume. How to establish a secret key then? Well, if both parties, say A and B, mutually trust a third party, say S, then they can use the following protocol:

$$A \to S:\ A, B$$
$$S \to A:\ \{K_{AB}\}_{K_{AS}} \text{ and } \{\{K_{AB}\}_{K_{BS}}\}_{K_{AS}}$$
$$A \to B:\ \{K_{AB}\}_{K_{BS}}$$
$$A \to B:\ \{m\}_{K_{AB}}$$

The assumption in this protocol is that A and S share a secret key, and also B and S (S being the trusted third party). The goal is that A can send B a message m under a shared secret key K_{AB}, which at the beginning of the protocol does not exist yet. How does this protocol work? In the first step A contacts S and says that it wants to talk to B. In turn S invents a new key K_{AB} and sends two messages back to A: one message is $\{K_{AB}\}_{K_{AS}}$ which is encrypted with the key A and S share, and also the message $\{\{K_{AB}\}_{K_{BS}}\}_{K_{AS}}$. which is encrypted with K_{AB} but also a second time with K_{BS}. The point of the second message is that it is a message intended for B. So a receives both messages and can decrypt them—in the first case it obtains the key K_{AB} which S suggested to use. In the second case it obtains a message it can forward to B. B receives this message and since it knows the key it shares with S obtains the key K_{AB}. Now A and B can start to exchange messages with the shared secret key K_{AB}. What is the advantage of S sending A two messages instead of contacting B instead? Well, for one there can now be a time-delay between the second and third step in the protocol. At some point in the past A and S need to have come together to share a key, similarly B and S. After that B does not need to be "online" anymore until A actually starts sending messages to B. A and S can completely on their own negotiate a new key.

The major limitation of this protocol however is that I need to trust a third party. And in this case completely, because S can of course also read easily all messages A sends to B. The problem is that I cannot really think of any institution who could serve as such a trusted third party. One would hope the government would be such a trusted party, but in the Snowden-era we know that this is wishful thinking in the West, and if I lived in Iran or North Korea, for example, I would not even start to hope for this.

The cryptographic "magic" of public-private keys seems to offer an elegant solution for this, but as we shall see in the next section, this requires some very clever protocol design.

Averting Person-in-the-Middle Attacks

The idea of public-private key encryption is that one can make public the key P^{pub} which people can use to encrypt messages for me. and I can use my key P^{priv} to be the only one that can decrypt them.

Keyfobs - protocol

Further Reading

http://www.cs.ru.nl/~rverdult/Gone_in_360_Seconds_Hijacking_with_Hitag2-USENIX_2012.pdf