Handout 5 (Protocols)

Protocols are the computer science equivalent to fractals and the Mandelbrot set in mathematics. With the latter two you have a simple formula, which you just iterate and then you test whether a point is inside or outside a region…it does not look exciting, but 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. This can happen even in face of strong adversaries who have complete control over the network involved in the message exchange. 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 in, 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, as was the case with the old solid metal keys. With the key fobs we must achieve security by exchanging certain messages between the key fob on one side and the doors and engine on the other. Clearly what we like to accomplish 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. For this imagine you park your car in front of a supermarket. One thief follows you with a strong transmitter. A second thief "listens" to the signals from the car and wirelessly transmits them to the "colleague" who followed you. This thief silently enquires what the key fob answers. This answer is then send back to the thief at the car. If done properly, the car will dutifully open and possibly start. No need to steal your keys anymore. That this is an attack one needs to reckon with is demonstrated by the fact that dodgy websites[2] sell the necessary equipment for top Ruble. This webpage is notable for the very helpful picture of a person-in-the-middle attack (see Figure 1).

But there are many more such protocols we like to study. Another example is Wifi—you might sit at a Starbucks and talk wirelessly to the free access point there and from there talk to your bank (see The Guardian article cited at

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

[2] `http://autokeydevices.com/product/wave/` … funnily this webpage says "not intended for illegal use", but I have a hard time finding any legal purpose for such a device.

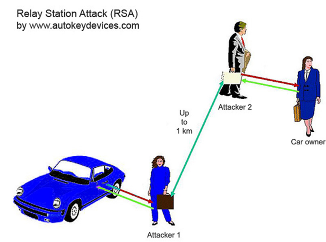


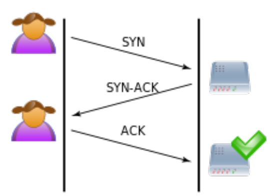
Figure 1: From a dodgy webpage about modern car theft. Note the stylish attackers.

the very end of this handout). Moreover, even if you have to touch in and out your Oyster card at the reader each time you enter or exit the Tube, it actually operates wirelessly and with appropriate equipment over some quite large distance (several meters). But there are many, many more examples for protocols (Bitcoins, Tor, mobile phones,...).

The common characteristics of the protocols we are interested in is that an adversary or attacker is assumed to be in complete control over the network or channel over which we exchanging messages. An attacker can install a packet sniffer on a network, inject packets, intercept packets, modify packets, replay old messages, or fake pretty much everything else. In this hostile environment, the purpose of a protocol (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. 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. Whether they can authenticate themselves 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 we will adopt 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

$$\begin{array}{l} A \rightarrow S: \; SYN \\ S \rightarrow A: \; SYN\_ACK \\ A \rightarrow S: \; ACK \end{array} \tag{1}$$

The left-hand side of each clause 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 SYN above, are send in cleartext over the network. If we want that a message is encrypted, then we use the notation

$$\{msg\}_K$$

for messages. The curly braces indicate a kind of envelope which can only be opened if you know the key K with which the message has been encrypted. We always assume that an attacker, say Eve, cannot get to 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.[3] It is also possible that an encrypted message contains several parts. In this case we would write something like

$$\{msg_1, msg_2\}_K$$

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

$$\{\{msg\}_{K_1}\}_{K_2}$$

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

This protocol is called lockstep protocol. The idea is that even if attacker Eve has the key K_2 she could decrypt the outer envelop, but still does not get to the message, because it is still encrypted with the key K_1. 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 our 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

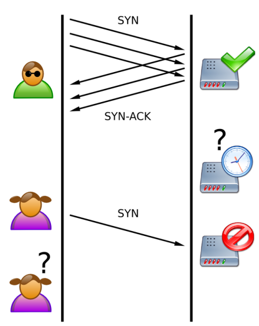
$$\begin{array}{rl} \text{Alice} \rightarrow \text{Calcium}: & SYN \\ \text{Calcium} \rightarrow \text{Alice}: & SYN\_ACK \\ \text{Alice} \rightarrow \text{Calcium}: & ACK \end{array}$$

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

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

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 blueprint for 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 then does not complete it. This looks graphically like



The attacker sends lots of SYN requests which the server dutifully answers. But in doing so the server needs to keep track of such protocol exchanges. As a result every time the protocol is initiated a little bit of memory will be eaten away on the server side until all memory is exhausted. When poor Alice then tries to contact the server, it is overwhelmed and does not respond anymore. This kind of attack is called *SYN floods*.[4]

After reading four pages, you might be wondering where the magic is with protocols. 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 thought of as A sends a common secret to B, for example 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 in the context where the message can be observed by everybody else on the network. Eve, for example, could just record this message A just sent, and next time sends the same message to B. B has no other choice than believing it talks to A. But actually it talks to Eve, who now clears out A's bank account assuming B had been a bank.

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

$$\begin{array}{l} A \rightarrow B : \ HELLO \\ B \rightarrow A : \ N \\ A \rightarrow B : \ \{N\}_{K_{AB}} \end{array}$$

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's answer, it will not make sense to replay this message, because next time this protocol is run, the nonce B sends out will be different. So if we run this protocol, what can B infer? It has sent 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 is 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 sent earlier.

But what about A? Can A make any inferences about whom it talks to? It dutifully answered the challenge and hopes his or her bank, say, will be the only one to understand her answer. But is this the case? No! Let us consider again 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

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

that if a message is sent without curly braces it is sent in clear text. A would encrypt the nonce with the key K_{AB} and send it back to Eve. She just throws away the answer. A would hope that she talked to B because she followed the protocol, but unfortunately she cannot be sure who she is talking to—it might be Eve.

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

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

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} too, and sends back the nonce of B. Let us analyse which inferences A and B can make after the protocol has run. B received a challenge and answered correctly to A (inside the encrypted message). An attacker would 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 is not able to generate this message. It could also not have been the case that it is an old message replayed, 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.

The purpose of the nonces, the random numbers that are sent around, might be a bit opaque. Because they are unpredictable they 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

$$
\begin{aligned}
I \to Y &: \; \{N\}_{K_{IY}} \\
Y \to I &: \; \{N+1\}_{K_{IY}}
\end{aligned}
$$

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 have sent 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 we can glean from such an exchange, it is in fact the basic building block in protocols for establishing some secret or for achieving some security goal (like authentication). This is what I meant by magic: we send around "just" some random numbers, but actually can use them to make some meaningful inferences.

While the mutual challenge-response protocol solves the authentication problem, there are some limitations. 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 whistleblower (say Snowden) and want to talk to a journalist (say Greenwald) then I might not have a secret pre-shared key.

Another limitation 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 are 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

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

but it does not specify who is A and who is B. If 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 let us 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 a fighter aircraft we 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—namely challenging and responding. They better had used two different keys. This would have averted this attack and would have saved me a lot of inconvenience.

Trusted Third Parties

One limitation the protocols we discussed so far have 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:

$$
\begin{array}{ll}
A \to S: & A, B \\
S \to A: & \{K_{AB}\}_{K_{AS}} \text{ and } \{\{K_{AB}\}_{K_{BS}}\}_{K_{AS}} \\
A \to B: & \{K_{AB}\}_{K_{BS}} \\
A \to B: & \{m\}_{K_{AB}}
\end{array}
$$

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_{AS} 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, there can 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 and does not solve the authentication problem completely.

Averting Person-in-the-Middle Attacks

The idea of public-private key encryption is that one can publish the key K^{pub} which people can use to encrypt messages for me and I can use my private key K^{priv} to be the only one that can decrypt them. While this sounds all good, it relies on the ability that people can associate me with my public key. That is not as trivial as it sounds. For example, if I would be the government, say Theresa Mayhem, and try to find out who are the trouble makers in the country, I would publish an innocent looking webpage and say I am The Guardian newspaper (or alternatively The Sun for all the juicy stories), publish a public key on it, and then just wait for incoming messages.

This problem is supposed to be solved by using certificates. The purpose of certification organisations is that they verify that a public key, say K^{pub}_{Bob}, really belongs to Bob. This is also the mechanism underlying the HTTPS protocol. The problem is that this system is essentially completely broken…but this is a story for another time. Suffice to say for now that one of the main certification organisations, VeriSign, has limited its liability to $100 in case it issues a false certificate. This is really a joke and really the wrong incentive for the certification organisations to clean up their mess.

The problem we want to study closer here is that protocols based on public-private key encryption are susceptible to simple person-in-the-middle attacks. Consider the following protocol where A and B attempt to exchange secret messages using public-private keys.

- A sends public key to B
- B sends public key to A
- A sends a message encrypted with B's public key,
  B decrypts it with its private key
- B sends a message encrypted with A's public key,
  A decrypts it with its private key

In our formal notation for protocols, this would look as follows:

$$
\begin{aligned}
A \to B &: \ K_A^{pub} \\
B \to A &: \ K_B^{pub} \\
A \to B &: \ \{A, m\}_{K_B^{pub}} \\
B \to A &: \ \{B, m'\}_{K_A^{pub}}
\end{aligned}
$$

Since we assume an attacker, say E, has complete control over the network, E can intercept the first two messages and substitutes her own public key. The protocol run would therefore be

$$
\begin{aligned}
&1. \quad A \to E : \ K_A^{pub} \\
&2. \quad E \to B : \ K_E^{pub} \\
&3. \quad B \to E : \ K_B^{pub} \\
&4. \quad E \to A : \ K_E^{pub} \\
&5. \quad A \to E : \ \{A, m\}_{K_E^{pub}} \\
&6. \quad E \to B : \ \{E, m\}_{K_B^{pub}} \\
&7. \quad B \to E : \ \{B, m'\}_{K_E^{pub}} \\
&8. \quad E \to A : \ \{E, m'\}_{K_A^{pub}}
\end{aligned}
$$

where in steps 6 and 8, E can modify the messages by including the E in the message. Both messages are received encrypted with E's public key; therefore it can decrypt them and repackage them with new content. A and B have no idea that they talking to an attacker. To them all messages look legit. Because E can modify messages, it seems very difficult to defend against this attack.

But there is a clever trick…dare I say some magic which makes this attack very difficult to perform on people who know each other—but not necessarily have a shared key. Modify the protocol above so that A and B send their messages in two halves, like

$$
\begin{aligned}
&1. \quad A \to B : \ K_A^{pub} \\
&2. \quad B \to A : \ K_B^{pub} \\
&3. \quad \qquad\qquad \{A, m\}_{K_B^{pub}} \mapsto H_1, H_2 \\
&\quad \qquad\qquad\quad \{B, m'\}_{K_A^{pub}} \mapsto M_1, M_2 \\
&4. \quad A \to B : \ H_1 \\
&5. \quad B \to A : \ \{H_1, M_1\}_{K_A^{pub}} \\
&6. \quad A \to B : \ \{H_2, M_1\}_{K_B^{pub}} \\
&7. \quad B \to A : \ M_2
\end{aligned}
$$

The idea is that in step 3, A encrypts the message (with B's public key) and then splits the encrypted message into two halves. Say the encrypted message is

$$\underbrace{\texttt{0X1peUVTGJK0XI7G+H70mMjAM8piY0sI}}_{\{A,m\}_{K_B^{pub}}}$$

then A splits it up into two halves

$$\underbrace{\texttt{0X1peUVTGJK0XI7G}}_{H_1} \qquad \underbrace{\texttt{+H70mMjAM8piY0sI}}_{H_2}$$

Similarly B splits its message into two halves M_1 and M_2. However, A initially only sends the first half H_1 to B. Which B answers with the message consisting of the received H_1 and its own first half M_1 encrypted with A's public key. The message in step 5. A receives this message, decrypts it and **only** when the H_1 matches with its first half it send out earlier, A will send out the second half; see step 6. For this, A adds the received M_1 and encrypts both parts with B's public key. Finally B checks whether the received M_1 matches with its first half, and if yes sends A its second half M_2. Now A and B are in the possession of H_1 and H_2, respectively M_1 and M_2, and can decrypt the corresponding messages.

Now the big question is, why on earth does this splitting of messages in half and additional message exchange help with defending against person-in-the-middle attacks? Well, let's try to be an attacker. As before we intercept the messages where public keys are exchanged and inject our own.

$$\begin{array}{ll} 1. & A \to E: \; K_A^{pub} \\ 2. & E \to B: \; K_E^{pub} \\ 3. & B \to E: \; K_B^{pub} \\ 4. & E \to A: \; K_E^{pub} \end{array}$$

Now A and B build the message halves:

$$\{A, m\}_{K_E^{pub}} \;\mapsto\; H_1, H_2 \qquad \{B, m'\}_{K_E^{pub}} \;\mapsto\; M_1, M_2$$

and A sends E its first half of the message.

$$5. \quad A \to E: \; H_1$$

Neither E nor B can do much with this message. Remember it is only half of some "garbled" text that cannot be decrypted. E could try to forward the message to B and see what its reply is.

$$\begin{array}{ll} 6. & E \to B: \; H_1 \\ 7. & B \to E: \; \{H_1, M_1\}_{K_E^{pub}} \end{array}$$

Although E can decrypt the message with its private key, but it only gets the halves H_1 and M_1 which are of no use yet. In order to get more information it can send the message to A with A's public key.

$$8. \quad E \to A : \{H_1, M_1\}_{K_A^{pub}}$$

A would receive this message, decrypt it and find out it matches with its expectation. It therefore sends out the message

$$9. \quad A \to E : \{H_2, M_1\}_{K_E^{pub}}$$

Now E is in the possession of H_1 and H_2, which it can join together in order to obtain $\{A, m\}_{K_E^{pub}}$ which it can decrypt. It seems like from now on all is lost, but let's see: in order to stay undetected it must send a message to B. It now has two options: one is to use the newly obtained knowledge and modify A's message to be

$$\{E, m\}_{K_B^{pub}} \mapsto H_1', H_2'$$

But notice since E changed the message, it will now receive two different halves. Let us call them H_1' and H_2'. If E now sends B the H_2', B will be in the possession of H_1 and H_2'. But after joining both halves it will not be able to decrypt the resulting message—the two halves simply do not fit. It can send out the original H_2 as follows:

$$10. \quad E \to B : \{H_2, M_1\}_{K_B^{pub}}$$

In this case B can make sense out of the message and as a result sends E back its second half M_2.

$$11. \quad B \to E : M_2$$

E might be ecstatic by now, because it has now also received M_1 and M_2 which it can join to get $\{B, m'\}_{K_E^{pub}}$. It can decrypt this message but still is not finished completely, because it has to send A a message. It could try to build the message $\{E, m'\}_{K_A^{pub}}$, but like above A would not be able to make sense out of the two halves (which again do not fit together). So one option is to send M_2.

With this the protocol has ended. E was able to decrypt all messages, but what messages did A and B receive and from whom? Was E able to modify the messages? If yes, were A and B able to find out that something strange is going on and probably not talk on this channel anymore? I leave you to think about it.[5]

I hope you have thought about all these questions. Maybe you noticed that there is a way to defeat the lockstep protocol. If an attacker could only forward the (unmodified) messages, then all would be great. Because then it could be

[5] Consider the case where A sends the message „How is your grandmother?" to B, and B send the message „How is the weather in London today" to A. Another possibility: what if A and B include a voice message in there messages.

used to establish secret keys using the Hellman-Diffie technique (see further reading). That E was able to decrypt all messages is of no importance for the Hellman-Diffie technique.

Unfortunately, E can create completely fake messages. Let us look at this possibility: E intercepts again the keys from A and B, and substitutes its own keys.

$$\begin{array}{ll} 1. & A \to E: \ K_A^{pub} \\ 2. & E \to B: \ K_E^{pub} \\ 3. & B \to E: \ K_B^{pub} \\ 4. & E \to A: \ K_E^{pub} \end{array}$$

Now A and B build again their message halves:

$$\{A, m\}_{K_E^{pub}} \mapsto H_1, H_2 \qquad \{B, m'\}_{K_E^{pub}} \mapsto M_1, M_2$$

A sends its first half H_1.

$$5. \quad A \to E: \ H_1$$

At this stage of the protocol, also E creates two messages and halves them, say

$$\{E, m_E\}_{K_E^{pub}} \mapsto C_1, C_2 \qquad \{E, m'_E\}_{K_E^{pub}} \mapsto D_1, D_2$$

But notice that E has to make up these messages out of thin air. No information from A and B is usable yet—remember the half H_1 on its own cannot be decrypted. E can then send C_1 to B, which dutifully responds

$$\begin{array}{ll} 6. & E \to B: \ C_1 \\ 7. & B \to E: \ \{C_1, M_1\}_{K_E^{pub}} \end{array}$$

Next E has to send a message to A—it can use the made up D_1 and the H_1 received earlier.

$$8. \quad E \to A: \ \{H_1, D_1\}_{K_A^{pub}}$$

A can verify it received H_1 and thus sends out

$$9. \quad A \to E: \ \{H_2, D_1\}_{K_E^{pub}}$$

With this E is in the possesion of both halves from A. In order to get the reply from B, E can send the message

$$10. \quad E \to B: \ \{C_2, M_1\}_{K_E^{pub}}$$

and B can verify that it received M_1. So it answer with

$$11. \quad B \to E: \; M_2$$

Finally E can complete the protocol with sending D_2 to A:

$$12. \quad E \to A: \; D_2$$

A and B receive expected messages and were able to verify their first halves. That means they do not suspect anything dodgy going on: E has sucessfully managed a man-in-the middle attack. In case A and B are computers, there is not much that can prevent this attack. In case they are humans, there are a few things they can do. For example A and B can craft their messages such that they include a specific question only A and B are likely to be able to answer, or include a voice message which identifies A and B by their voice. The point is E should not be able to create legit looking messages. Humans can do this if they have some minimal knowledge of the protocol partner (for example know their voice from TV); but computers cannot. The conclusion is that there is no protocol that can establish a trusted connection without any preshared information. The solution that has evolved over the years is to use certificates which have been created by an authority we (or better the browser) already trust.

Key Fob Protocol

Recall from the beginning that a person-in-the middle attack can easily be mounted at the key fob and car protocol unless we are careful. If you look at actual key fob protocols, they use a variant of the protocol described above. Suppose C is the car and T is the key fob (transponder). The HiTag2 protocol used in cars of VW & friends is as follows:

1. C generates a random number N
2. C calculates $\{N\}_K \mapsto F, G$
3. $C \to T$: N, F
4. T calculates $\{N\}_K \mapsto F', G'$
5. T checks that $F = F'$
6. $T \to$ C: N, G'
7. C checks that $G = G'$

The assumption is that the key K is only known to the car and the transponder. The claim is that C and T can authenticate to each other. Again, I leave it to you to find out, if this protocol is immune from person-in-the-middle attacks. (Hint: Does it establish a trusted connection from "zero"?)

Further Reading

- A nice video explaining the Hellman-Diffie key exchange technique is here

  https://www.youtube.com/watch?v=YEBfamv-_do

  The main point of this technique is that no sensitive information is sent over the network—both parties create the key together, but on their computer, not over the network. While the technique is cryptographic magic, it can be attacked when messages can be manipulated during transit. Remember that the lockstep protocol can only be attacked by either passively forwarding the messages (without being able to modify them) or by creating complete fake messages.

- A blogpost that describes the first few milliseconds of an HTTPS connection is at

  http://www.moserware.com/2009/06/first-few-milliseconds-of-https.html

  It disentangles every message sent between a client and a server.

- If you want to know more about how cars can be hijacked, the paper

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

  is quite amusing to read. Obviously an even more amusing paper would "Dismantling Megamos Crypto: Wirelessly Lockpicking a Vehicle Immobilizer" by the same authors, but because of the court injunction by VW, we are denied this entertainment. UPDATE: This paper is now in the public domain.

- Man-in-the-middle-attacks from the "wild" are described with real data in the blog post

  http://www.renesys.com/2013/11/mitm-internet-hijacking

  The conclusion in this post is that man-in-the-middle-attacks can be launched from any place on Earth—it is not required that you sit in the "middle" of the communication of two people. You just have to route their traffic through a node you own.

- An article in The Guardian from 2013 reveals how GCHQ and the NSA at a G20 Summit in 2009 sniffed emails from Internet cafes, monitored phone calls from delegates and attempted to listen on phone calls which were made by Russians and which were transmitted via satellite links:

http://www.theguardian.com/uk/2013/jun/16/gchq-intercepted-communications-g20-summits

…all in the name of having a better position for negotiations. Hmmm…

- A paper guessing how the NSA can decrypt so much of the encrypted Internet traffic:

  https://weakdh.org/imperfect-forward-secrecy.pdf