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.

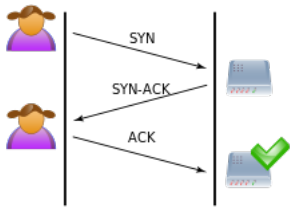
But there are many more such protocols we like to consider. Other examples are wifi—you might sit at a Starbucks and talk wirelessly to the free access point there and from there talk with your bank, for example. Also even if your have to touch your Oyster card at the reader each time you enter and exit 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

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

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

$$\begin{array}{l} \text{Alice} \to \text{Calcium}: \ SYN \\ \text{Calcium} \to \text{Alice}: \ SYN\_ACK \\ \text{Alice} \to \text{Calcium}: \ ACK \end{array}$$

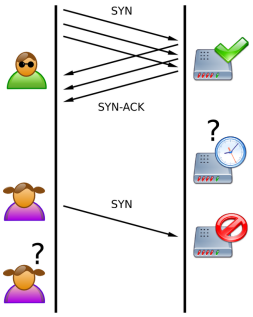
But a server like Calcium of course needs to serve many clients. So there could be the same protocol also running with Bob, say

[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.

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

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

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

Keyfobs - protocol

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

Further Reading

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