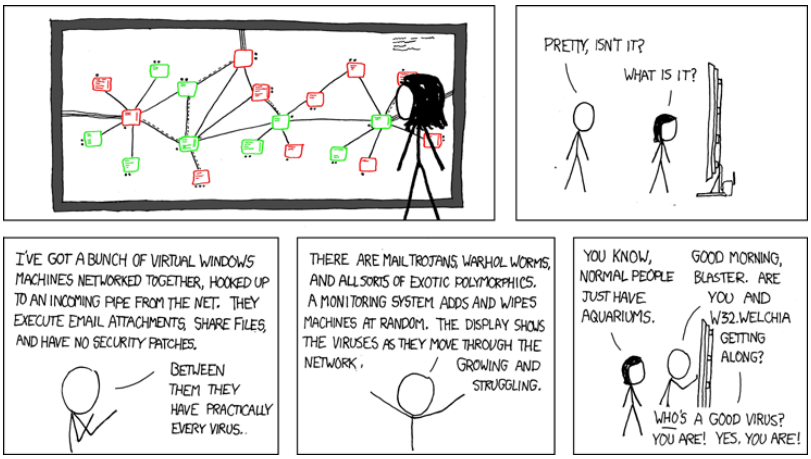
UNIVERSITÀ DEGLI STUDI DI TRIESTE

Marco Sgobino

Notes from the course of

Computer Networks II and Introduction to Cybersecurity



Academic Year 2021-2022

Contents

Part I

Security in Networking Protocols

Chapter 1

Transmission Control Protocol internals

1.1 TCP Basis

1.1.1 Recap

TCP is a protocol that has the following remarkable properties,

1. it is *connection-oriented*: before transmitting data, a connection must be established – this is different to what IP protocol does in network layer, since in IP protocol there lies no concept of connection, and messages are exchanged under the paradigm of *packet switching*[1];
2. it is *reliable*: it assures all segments are correctly delivered through use of *ACK* mechanism, and **at most once** – again, this is different to IP layer, since IP packets are just sent one after the other, unreliably, and there is no guarantee that they will be delivered, let alone in their correct ordering;
3. it offers a *sliding window* mechanism for congestion control and stream control. This assures read and send buffers are well-optimized in both sender and receiver – this feature is much important since it allows fine-tuning the connection and avoiding waste of resources;

[1] In telecommunications, packet switching is a method of grouping data into packets that are transmitted over a digital network. Packets are made of a header and a payload. Data in the header is used by networking hardware to direct the packet to its destination, where the payload is extracted and used by an operating system, application software, or higher layer protocols. Packet switching is the primary basis for data communications in computer networks worldwide.

4. it is *byte-oriented*: the byte stream is fragmented into multiple segments, and composed again after getting to destination – in this manner, huge payloads can be reliably delivered in multiple segments, each one having the correct order to the application. Application has the necessary information to easily reconstruct the correct ordering of the acquired information.

Ultimately, TCP allows creating connections between application processes, offering multiple services to the upper layers. Connections are possible thanks to the four above characteristics of the TCP layer.

SOCKETS

The TCP makes use of *sockets* abstraction as a way of serving data to the above application layer. The logical structure is the following one: there are two entities, the client and the server. The client first authenticates to the server; the server then opens a connection and the client executes the subsequent send-receive loop. Both server and client need to create a *socket s* (a UNIX-like file abstraction), that can be used to send and receive data. Client connect *s* to `IP-srv`, `port-srv`, while the server operates a similar procedure on its own sockets. Each socket is then bound to a specific IP address and a specific *port number*, according to the just-created connection specifics.

The communication takes place on *s* by means of an application protocol, with applications exchanging data each other. The connection is said to be *reliable*: losses and packet loss are carefully managed by the TCP protocol, and segments[2] are collected in the very same order as they are sent.

A very simplified pseudo-code at client's side for TCP is as follows,

```
 int s; s := socket(...);
connect(s, IP-srv, port-srv,...);
...
send (s, msg1, ...);
...
msg2 := receive(s,...);
...
```

where the notation ... denotes possibly code in between two instructions. In the above code, a client wants to send data to a remove server having a specific IP

[2] *Segments* refer to the particular kind of package that TCP protocol exchange. Usually, in literature one could find the term package referring to TCP segments; while this is not fully correct, it may be acceptable when there is no ambiguity.

address and a port number. The client first creates a socket, then connects it to the server IP address and to the corresponding port number. After connection has been established, the client can subsequently invoke `send()` to command TCP layer to deliver desired data to the other end. To the very same socket, data can be received as well: by invoking `receive()` the client can successfully retrieve eventual data sent by the server. Socket acts as an *abstraction* for the application layer, for which a socket can be simply thought as a **data sink** or **data source**.

At server side, the logical structure is quite different. A server creates a socket `s1`, chooses a port number to bind to that socket (usually a standard one), then it declares *willingness to accept connections* on `s1`, and finally it awaits for connection requests on `s1`. Upon receiving a connection request, the server handles it by creating *another* socket `s2` and managing the connection on this newly generated socket, this time with a different port number.

Basically, it creates two different sockets: the first is kept alive to accept new connections, and the second one is created on-demand, and it is required to actually manage the connection. A *different* socket is required for each connection. A server usually remains on *sleep* until a connection is requested, awaiting a request on the main socket.

```
s1 := socket(...);
bind(s1, portsrvm ...);
listen(s1,...);
s2 := accept(s1,...); // another socket
...
msg1 := receive(s2,...);
...
send(s2,msg2,...);
...
```

Details of the above code largely depend on the platform on which the server is operating.

1.1.2 TCP Implementation

TCP layer is built *on top* of the IP layer. Since there are many differences between TCP and IP, set aside their abstraction layer, TCP should be properly built to manage IP differences and quirks.

IP protocol operates between *nodes*: it is *connectionless*, **unreliable**, and is *message–oriented*. The **Maximum Transmission Unit** size of an IP packet is $MTU =$

$64KB$: this implicitly means a TCP segment can never be greater than the MTU, because a TCP segment is carried *inside* an IP packet.

In TCP, communication happens to be **bidirectional**, with a pattern that depends on the application protocol. The send–receive patterns heavily depends on the application itself - browser-related send–receive sequences are very different from, let's say, an e-mail client send–receive sequence. There is no guarantee that two different application protocols will exchange a similar amount and kind of TCP messages, let alone the very same ones.

As already mentioned, TCP layers communicate between themselves in terms of *segments*. A segment is a single *message between TCP layers*, and contains a *TCP header* and – eventually – data, that is said to be *payload*. The difference between information in header and information in payload is that the first one is strictly required by the TCP layer, while the second one is information required by the application layer. The latter may be absent in some segments.

Payload in fact can either be 0 bytes long or carry some information (wanted by the application layer). *A TCP segment must be small enough to fit in a single IP packet*, hence IP header and TCP segment size should be no greater than $64KB$, the MTU of an IP packet.

Since a packet is composed by a IP header, whose payload is a TCP segment, and the TCP segment is in turn composed by a TCP header, followed by eventual application data, the shape of a packet plus its segment can be summarized as follows,

```
| IP header | TCP header |  Payload ***  |
```

Usually, IP header size is usually 20 bytes, as well as TCP header that is 20 bytes. The IP datagram can be greater up to $64KB$, with the first 40 to 50 bytes reserved to headers. Thus,

```
| IP header | TCP header |  Payload ***  |
   20 byte     20 byte        greater
```

Segments can carry portions of the original data. For instance, in a video stream many, many segments should be sent to client in order to carry enough information and let application layer reconstruct the video correctly. For this reason, in application layer, one huge application message could correspond to *many segments* in TCP layer, in **both** directions. At TCP level multiple segments are usually required in order to send a single 'big' application-level message.

Basically, this means that while for the application layer a single `send()` may suffice to send an entire file or all the information required, from the TCP layer's

point of view many and many segments may be required, typically many more than the number of `send()` commands invoked by the application (at least, an number of them equal to the `send()` invocation should be sent).

The connection state

TCP takes care of creating *connections* between processes. There would be no actual TCP data exchange without an established connection. Connections do possess their own *state*, which fundamentally and univocally describes a connection. It is enough to represent each connection with their `<id>` and `<state>`, with the `<id>` field that contains essentially 4 informations,

1. the `local IP` address;
2. the `local port` number;
3. the `remote IP` address;
4. the `remote port` number.

Four and only four informations are required: the IP address and the port number of each connection's endpoint.

Each connection is then associated to a `state`, that serves as a description of the actual state in which the connection lies. The state of the connection describes precisely the condition in which the connection lies (is it running? is it closed?).

Information regarding a TCP connection must reside in the packet and in the segment header, so each received segment can be sorted out and managed properly. IP addresses are extracted from the IP header, while port numbers are extracted from TCP header, thus it suffices to look at the IP packet plus the TCP header.

Information in header is not sufficient: each endpoint must keep track of the connection with a *state variable*. The connection <state> includes information on the **Maximum Segment Size** (MSS), which is the maximum size of the *data part* of a segment that the other part is willing to receive. The MSS is negotiated upon connection opening – this value is, in practice, identical in both direction and **not arbitrary**. In most cases and for historical reasons, there are only 2 possible values that depend whether the connection:

- *lies on different networks* (connection through internet), MSS is 536 bytes (MTU=576), that is the maximum segment size that can fit in the smallest possible packet - historically this was the most reliable option;

- *lies on the same network* (connection through ethernet), MSS is 1460 bytes (MTU=1500), which corresponds to ethernet MTU minus the IP header and TCP header - this is a good choice in order to fit to a single ethernet frame.

The core idea is that each segment must be *sufficiently small* to fit in one packet along the full path, *in order to prevent fragmentation*. In fact, by transmitting larger segments one could end up with fragmented segments, with no real advantage and many disadvantages.

IP AND TCP HEADER STRUCTURE

IPv4 header format

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Version				IHL				DSCP						ECN		Total Length															
4	32	Identification																Flags			Fragment Offset											
8	64	Time To Live								Protocol								Header Checksum															
12	96	Source IP Address																															
16	128	Destination IP Address																															
20	160	Options (if IHL > 5)																															
⋮	⋮																																
56	448																																

Figure 1.1: Header of an IP packet. Source: Wikipedia.

Figure 1.1 shows the structure of an IP packet. Essentially,

Version, 4 bit The first header field in an IP packet is the four-bit version field. For IPv4, this is always equal to 4;

Internet Header Length (IHL), 4 bit The IPv4 header is variable in size due to the optional 14th field (options). The IHL field contains the size of the IPv4 header, it has 4 bit that specify the number of 32-bit words in the header. The minimum value for this field is 5, which indicates a length of $5 \times 32b = 160b = 20B$. As a 4-bit field, the maximum value is 15, this means that the maximum size of the IPv4 header is 15×32 bit = 480 bit = 60 bytes .;

Differentiated Services Code Point (DSCP), 6 bit Originally defined as the type of service (ToS), this field specifies differentiated services (DiffServ) per `RFC 2474`. Real-time data streaming makes use of the DSCP field. An example is Voice over IP (VoIP), which is used for interactive voice services;

Explicit Congestion Notification (ECN), 2 bit This field is defined in RFC 3168 and allows end-to-end notification of network congestion without drop-

ping packets. ECN is an optional feature available when both endpoints support it and effective when also supported by the underlying network;

Total Length, 16 bit This 16-bit field defines the entire packet size in bytes, including header and data. The minimum size is 20 bytes (header without data) and the maximum is 65535B. All hosts are required to be able to reassemble datagrams of size up to 576B, but most modern hosts handle much larger packets. Links may impose further restrictions on the packet size, in which case datagrams must be fragmented. Fragmentation in IPv4 is performed in either the sending host or in routers. Reassembly is performed at the receiving host;

Identification, 16 bit This field is an identification field and is primarily used for uniquely identifying the group of fragments of a single IP datagram. Some experimental work has suggested using the ID field for other purposes, such as for adding packet-tracing information to help trace datagrams with spoofed source addresses, but RFC 6864 now prohibit any such use;

Flags, 3 bit A three-bit field follows and is used to control or identify fragments. They are (in order, from most significant to least significant):

- bit 0: Reserved; must be zero;
- bit 1: Don't Fragment (DF);
- bit 2: More Fragments (MF).

If the DF flag is set, and fragmentation is required to route the packet, then the packet is dropped. This can be used when sending packets to a host that does not have resources to perform reassembly of fragments. It can also be used for path MTU discovery, either automatically by the host IP software, or manually using diagnostic tools such as ping or traceroute. For unfragmented packets, the MF flag is cleared. For fragmented packets, all fragments except the last have the MF flag set. The last fragment has a non-zero Fragment Offset field, differentiating it from an unfragmented packet;

Fragment offset, 13 bit This field specifies the offset of a particular fragment relative to the beginning of the original unfragmented IP datagram in units of eight-byte blocks. The first fragment has an offset of zero. The 13 bit field allows a maximum offset of $(2131) \times 8 = 65528B$, which, with the header length included $(65528 + 20 = 65548B)$, supports fragmentation of packets exceeding the maximum IP length of 65535B;

Time to live (TTL), 8 bit An eight-bit time to live field limits a datagram's lifetime to prevent network failure in the event of a routing loop. It is specified in seconds, but time intervals less than 1 second are rounded up to 1. In practice, the field is used as a hop count—when the datagram arrives at a router, the router decrements the TTL field by one. When the TTL field hits zero, the router discards the packet and typically sends an ICMP time exceeded message to the sender. The program traceroute sends messages with adjusted TTL values and uses these ICMP time exceeded messages to identify the routers traversed by packets from the source to the destination;

Protocol, 8 bit This field defines the protocol used in the data portion of the IP datagram. IANA maintains a list of IP protocol numbers as directed by RFC 790;

Header checksum, 16 bit The 16-bit IPv4 header checksum field is used for error-checking of the header. When a packet arrives at a router, the router calculates the checksum of the header and compares it to the checksum field. If the values do not match, the router discards the packet. Errors in the data field must be handled by the encapsulated protocol. Both UDP and TCP have separate checksums that apply to their data. When a packet arrives at a router, the router decreases the TTL field in the header. Consequently, the router must calculate a new header checksum;

Source address, 32 bit This field is the IPv4 address of the sender of the packet. Note that this address may be changed in transit by a network address translation device;

Destination address, 32 bit This field is the IPv4 address of the receiver of the packet. As with the source address, this may be changed in transit by a network address translation device;

Options, up to 288 bit Options are largely unused, and may be considered harmful by some router. This is the portion of the IP header that is variable in size.

On the TCP header, instead, Figure 1.2 shows the header structure,

Source port, 16 bit Identifies the sending port;

Destination port, 16 bit Identifies the receiving port;

Sequence number, 32 bit Will be covered later, its role is to keep track of the sequence of segments;

TCP segment header

Offsets	Octet	0								1								2								3							
Octet	Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
0	0	Source port																Destination port																
4	32	Sequence number																															
8	64	Acknowledgment number (if ACK set)																															
12	96	Data offset				Reserved 0 0 0				NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size														
16	128	Checksum																								Urgent pointer (if URG set)							
20	160	Options (if *data offset* > 5. Padded at the end with "0" bits if necessary.)																															
⋮	⋮																																
60	480																																

Figure 1.2: Header of a TCP segment.

Acknowledgment number, 32 bit Same as the above, but keeps track of the *successfully delivered* packages;

Data offset, 4 bit Specifies the size of the TCP header in 32-bit words. The minimum size header is 5 words and the maximum is 15 words thus giving the minimum size of 20 bytes and maximum of 60 bytes, allowing for up to 40 bytes of options in the header. This field gets its name from the fact that it is also the offset from the start of the TCP segment to the actual data;

Reserved, 3 bit For future use and should be set to zero;

Flags, 9 bit Contains 9 flags, with some important ones:

URG Indicates that the Urgent pointer field is significant;

ACK Indicates that the Acknowledgement field is significant. All packets after the initial SYN packet sent by the client should have this flag set;

PSH Push function. Asks to push the buffered data to the receiving application;

RST Resets the connection;

SYN *Synchronize sequence numbers*. Only the first packet sent from each end should have this flag set. Some other flags and fields change meaning based on this flag, and some are only valid when it is set, and others when it is clear;

FIN Last packet from sender.

Window size, 16 bit The size of the *receiving window*. It is the number of window size units that the sender of this very segment is willing to receive. Useful in flow control;

Checksum, 16 bit Checksum for error-checking;

Urgent pointer, 16 bit If the URG flag is set, then this 16-bit field is an offset from the sequence number indicating the last urgent data byte;

Options, variable 0–320 bit, in units of 32 bit Various options.

1.1.3 TCP Architecture

TCP bears an unpredictable and unrepeatable nature

TCP carries many different implementations, depending mostly on OS of choice. Several variants of its components have been written, with many of them largely optional, or deprecated. As already mentioned, execution flow at application level works independently and unpredictably with respect to the TCP-level flow: when an application sends something, multiple TCP packages are exchanged; how many and when they are sent is *not predictable*.

TCP works by first copying data that should be sent in a buffer, and then send them later.

The sequence of bytes is first copied to a buffer (sliding window) and the `send()` function is, for example, invoked – the exact time in which a segment is sent is **unpredictable**, **unrepeatable** and depends on many things, over which the application has little to no control. This means that application cannot *deterministically predict* the exact time in which the transmission will occur – instead, the TCP layer will deliver the payload at a specific time determined by the protocol[3].

On TCP, various *events* can induce a transmission of data:

- application invokes `send()` – in this case, the data in send's call argument will be inserted into the proper transmission buffer, and it will be transmitted somewhere in the (possibly immediate) future;
- application invokes `receive()` – this way, we will see in detail that the TCP protocol will send a *response* that *acknowledges* the receipt of a segment, for the other party's sake;

[3] However, also on TCP layer there is a level of unpredictability. Many more effects can concur: for instance, the Operating System may not be ready to send a segment, or may be busy with other resources. These situation will cause delay in sending information, even from the TCP layer's point of view.

- TCP layer *receives a segment* – the same as above;
- a *timeout* occurs – TCP layer will inform the other party of this;

Each of the above will trigger a transmission either immediately or *within a maximum predefined time* (in the case of a timeout, for instance). When a TCP layer "is touched" from above or below, **it reacts by transmitting a segment**. Transmission should occur *even if there is no useful data to transmit* (e.g. if the transmission buffer is empty) - in that case, a segment will only carry the header, which has information useful to the protocol itself. Basically, regardless of the presence of data in the transmission buffer the TCP protocol will exchange vital protocol information. Protocol information is, in fact, necessary to keep the connection alive and well-performing.

Upon transmission, TCP may deliver a varying number of bytes, ranging from empty up to a number of bytes *larger than Maximum Segment Size* (536 in Internet network, 1460 for same Ethernet network). Since huge payloads cannot fit into a single segment with a predefined MSS, the TCP protocol *fragments* them before delivery, resulting in *multiple segments delivered in sequence*. TCP protocol will try its best to send as many segments as possible, for efficiency's sake: it is not uncommon that 2 or 3 segments are initially delivered before receiving the other party's response.

CPU Cycle	0.3 ns
Main Memory Access (DRAM)	120ns
SSD	50–150 μs
HHD	10 ms
Internet, San Francisco to New York	40ms

Table 1.1: Some interesting metrics. Notice the order of magnitudes differences between CPU cycles and network delays. This table highlights the fact that bytes cannot be injected through the internetwork at full-speed: suppose one has a 100MB transmission buffer that is delivered across just 1ms - injected throughput would be $100 \cdot 10^6 \cdot 8b/(10^{-3}s)$, which yields an astonishing $8 \cdot 10^{11}b/s = 800Gb/s$, unsustainable for most of internetworks.

The TCP protocol starts with *slowly sending segments, increasing the exchange speed as the time goes on*. Acceleration mainly depends on the timing of *received* segments, by looking at the metrics of confirmation packets from receiver. As the sender acknowledges that the receiver is able to keep up with the increasing transmission speed, it raises up the delivery speed. Basically, TCP layer adapts its speed to 2 determining factors,

- the receiver's speed at processing packages;
- the internetwork's capability of deliver packages.

As we will see in following chapters, the first factor is handled by the **Flow Control** algorithm, while the second one is kept under control by the **Congestion Control** system.

HOW AN APPLICATION ASKS FOR DATA TO BE TRANSMITTED

The system call `send()` processes the data transmission through the network. The `send` function passes a memory buffer contained in application space (address, length), and copies bytes from transmission memory buffer in application space to transmission memory buffer in TCP layer (called **TX-buffer**).

```
public void write(byte[] b)
  throws IOException
```

Send is first invoked by application level, whose execution flow is completely independent from the TCP layer's one. Buffer at application layer is then copied to the TCP transmission buffer, to be sent immediately or later. New invokations of `send()` will copy data in TCP buffer **after** the data that is already present.

Invoking `send()` **does not guarantee** any immediate transmission: all it does is pushing application data into TX-buffer to be sent in future. TCP will then *independently* establish the proper moment and way to transmit data present in transmission buffer.

Suppose now one has to send N bytes with K consecutive `send()` invocations. How many segments will be exchanged? How many transmission events?

First and foremost, the number of segments will not depend on K, since no matter how many times `send()` is invoked, the end result will be to simply insert those N bytes in the TX-buffer, because the sole effect of `send()` is to insert the passed number of bytes into the TX-buffer, triggering a transmission. As a first approximation, the number of transmitted segments will roughly be

$$\#\text{transmitted} = \frac{N}{MSS} + 1,$$

with the last segment +1 smaller than the previous ones. Things, however, can go wrong and be much more complex due to packet loss and retransmissions.

Recall that for all these reasons, the exact number is neither predictable nor repeatable.

RETRIEVING DATA FROM TCP

Data retrieval occurs with another system call, `receive()`. System call `receive()` is quite similar to the `send()` call: both receiver's TCP layer and Application layer execution flows act independently to each other, and receiving buffer is not guaranteed to contain any data.

When data reaches the receiver, the data is copied into the receiving buffer (**RX-buffer**). The receiving buffer stores all received bytes and is flushed only when the application invokes `receive()`. The function `receive()` copies (at least a portion of) the receiver buffer into the application buffer by means of passing a buffer with known *address* and *length*. Function `receive()` copies bytes without exceeding the size of the buffer (*length*) and it **returns** how many bytes have been copied into the provided buffer. As argument, `receive()` also takes the number of bytes to fetch from the TCP receiving buffer. Basically, `receive()` needs to accept a buffer to which data should be copied, its length (in C length information is **not** intrinsic to arrays) and the expected number of bytes to be retrieved.

There are three possible outcomes for the `receive()` call:

- the *receive buffer is empty*: the application is suspended and the process is put to sleep until some data is available;
- *more than `length` bytes are available*: a `length` number of bytes is fetched and delivered to the application as soon as possible. More `receive()` calls are needed to retrieve all data and empty the RX-buffer;
- *less than `length` bytes are available*: all available bytes are copied as soon as possible, since they are less than the maximum deliverable value. The single `receive()` call will not yield all requested bytes, and more calls should be used.

Basically, each time a number of bytes is requested, TCP will provide *up to* that number of bytes. The application is held back only in the case when there is no data available.

Now with an example: let a buffer have length equal to 800 bytes. Suppose `receive()` is invoked in the following scenarios,

1. in the first scenario, there are 600 bytes in the RX-buffer. Upon `receive()` call, all 600 bytes are delivered to the application. The RX-buffer is now empty;

2. in the second scenario, there are 1000 bytes in the RX-buffer. Upon `receive()` invocation, only 800 of the 1000 bytes are saved into the buffer - 200 bytes will remain into the RX-buffer;

3. in the third scenario, there is *no data* in the TX-buffer. Application is suspended until there is data to retrieve. Suppose 500 bytes are collected in the RX-buffer - they will be immediately saved into the buffer provided by the application, and the application will retrieve them. RX-buffer is now emptied.

1.1.4 SEQUENCE NUMBERS

IP protocol is *unreliable* (packets can be lost, duplicated, or delivered in different order from which they were sent). To overcome this huge shortcoming, each data byte is implicitly identified by a 32 bit **sequence number**. A sequence number is a label for each transmitted and received byte, so that both correct ordering and amount of data can be properly determined. The association is *implicit* – the sender applies a sequence number to a segment, and the receiver uses that information to reconstruct the original order of segments. This way, segments are reconstructed even if they show up haphazardly, and the data can be assembled as it originally was.

There are many variables involved in sequence numbers. `snd.User` is the variable carrying the value of the next byte the **application** will send. The variable `snd.Next` carries the value of the next byte that the **TCP layer** will transmit – its value is contained in the TCP header (initial byte of the sequence, of course). Basically, `snd.User` refers to the next byte that will be filled by the invocation of `send()`, while `snd.Next` is the next byte yet to transmit by the TCP layer. The sequence number of application level must be computed from other information.

The variable `snd.Next` is carried in the TCP segment header, in *sequence number* space (16 bit).

In short,

- `snd.Next` is the boundary between transmitted data and yet-to-transmit data;

- `snd.User` is the boundary between in TX-buffer data (data sent by application) and not-yet-associated bytes, the free space in the buffer.

Upon sending a segment, the *sequence number* will be inferred from the `snd.Next` variable, and will be the *next byte the other endpoint expects*.

From the receiver's point of view, the variable `rcv.Next` is the boundary between content of RX-buffer and not-yet-received data (right boundary of the data currently in buffer). The variable `rcv.Next` is similar to the variable `rcv.Next` in the sense that it points at the next byte yet to receive. Only packets having the *expected sequence number* are collected and brought into the buffer.

There is a special case where some segment carries new parts of information and sequence numbers that are not collected, with a portion of them already collected: in that case the incoming segment will still be collected, with duplicated bytes thrown away. There may be two reasons for packet duplications: either IP layer duplicates them, or the TCP sender retransmits them because it thought they were lost. In order to be able to retransmit packets, TCP stores all sent data in buffer until acknowledgement has been received, since they could be retransmitted in the immediate future.

- `rcv.Next` points at the next byte that is not yet being received;
- `rcv.User` points to the next byte to be received by the application. After `receive()` invokation by the application, all bytes that have been successfully delivered to the application can now be safely deleted.

Each TCP layer has **both** TX-buffer and RX-buffer. Sequence numbers in the two directions are independent each other, and they will increase as much as there is new data incoming from a direction.

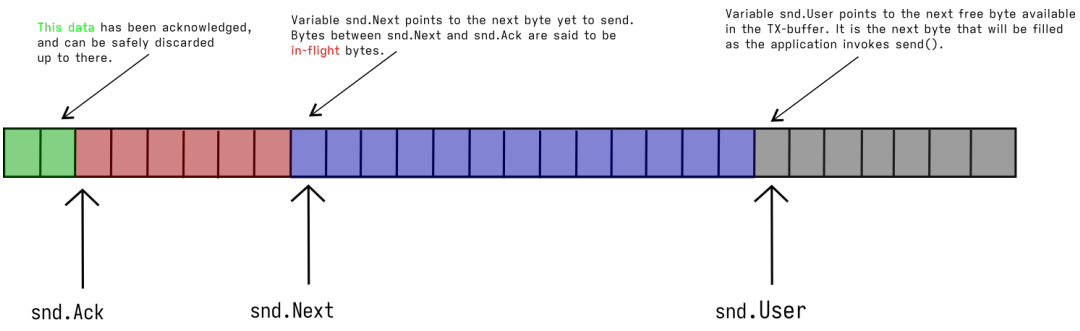


Figure 1.3: TX-Buffer and its flags `snd.Ack`, `snd.Next` and `snd.User`.

Each and every segment is accepted only if it contains *new* data: that is, when the sequence number is the expected one *or* when the segment contains novel sequence numbers.

Each segment also carries information on the *length* of its payload. This allows the receiver to efficiently reconstruct the segment, in order to compute the ending sequence number, and not only the starting sequence number.

An important remark is that **sequence numbers are 32-bit integers**. Therefore, the sequence numbers can only go from 0 to $2^{32} - 1$, so there may be *wrap-arounds* in an overflow-like fashion. TCP should internally handle these comparisons in a proper manner.

1.1.5 HANDLING DUPLICATES AND LOSS

IP is an unreliable protocol. This means that *packets can be loss*. Necessary mechanisms are two,

- the **retransmission**, or when a segment is regarded to be sent again;
- the **acknowledgement**, which is a kind of *notification of receipt*, that is sent by the receiver in order to make the other party sure that it has received and collected the data sent to it.

Acknowledgements are a mechanism that assures receipt of a message by means of *notifications*. Every header of a segment contains the sequence number of `snd.Next`. Every segment also *carries an information regarding the bytes that are received*: the **acknowledgement number**, which tracks the state of the RX-buffer by following the pointer `rcv.Next`. This way, having both sequence number and acknowledgement number, one can successfully track the state of a TCP connection. Upon sending a TCP segment, both sequence number and

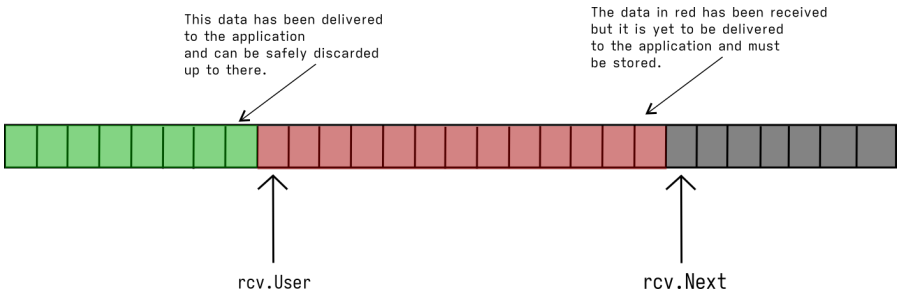


Figure 1.4: RX-buffer and its flags. Notice that RX-buffer only uses two flags instead of three (no ACK flag needed). Indeed, there is no need to keep track of sent ACKs: they will be sent as soon as segments are acquired.

acknowledgement number are delivered, so that the receiver can reconstruct the state of the sender RX-buffer.

Basically, sequence number tracks the bytes a party *has sent*, while acknowledgement number tracks the bytes a party *has successfully received*. Both informations must be exchanged at every delivered segment.

A fifth variable is thus needed: `snd.Ack`, which *points to the byte in TX-buffer before which any transmitted byte had been acknowledged*. This variable is only increased upon receiving data (for instance, upon receiving a sequence with a greater ACK number from the sender). Since it has been safely collected, all data before `snd.Ack` pointer can be forgotten.

Acknowledgements mechanism is crucial to a TCP connection, in order to provide **reliability** to a connection. Therefore, transmission of acknowledgements is always necessary even if TX-buffer is empty. In that case, no payload will be transmitted, and only protocol information is sent (increasing acknowledgement number).

An important note is that `snd.Ack` is only updated when the received sequence number `segment.ack` is *greater* than the value in `snd.Ack` - that is, when new data are correctly received.

For this reason one can say that the sequence number tracks `snd.Next`, while the acknowledgement number tracks `rcv.Next`. However, the sequence number will be at the value of the *first* sent byte, that is the first byte the receiver is expecting.

For instance, suppose the receiver has successfully received 130 bytes from a sender: its `rcv.Next` points to the 130th index in the RX-buffer, corresponding to the 131th byte. In order to inform the sender, all the new incoming segments from the receiver will carry 130 as sequence number (it points to the 131st element). Upon receiving such packets, the sender will successfully interpret the sequence number, and set the `snd.Ack` variable to 130: all the data prior to index 130 can be safely discarded.

Data bytes between pointers `snd.Ack` and `snd.Next` is said to be **in flight** data. These bytes have been transmitted but not yet acknowledged, and are potentially eligible for retransmission.

The reason why acknowledgements are important is pretty straightforward: without acknowledgements, there would be **no way for the sender to tell whether sent data has been received or not**. This means that even though the receiver has no data, acknowledgements are still required and should be properly sent.

In the end, acknowledgements are a crucial component of TCP, without them it wouldn't be possible to provide either a reliable or a predictable connection, with no chance of keeping track of actually and successfully delivered information.

1.1.6 Delayed Acknowledgement

Delayed Acknowledgement is a notorious TCP algorithm. Its core idea is not to answer immediately with an acknowledgement, but to await some arbitrary time interval T that depends on the operating system. The algorithm is pretty straightforward:

> Upon receiving a segment, if delayed-ack timer T has previously been set, *transmit immediately*. Else, set the delayed-ack timer T to a specific value.

When receiving a packet, simply set a timer, and send the response segment after timer expires. However, if you receive yet another segment before the timer expires, then send immediately the response.

Delayed-ack timer value depends on the operating system:

- RFC suggests $T = 500ms$;
- Windows has $T = 200ms$;
- Linux in general has $T = 40ms$;
- RHEL sets $T = 4ms$.

The reasoning is as follows: if there is not much data to transmit, it will be likely that the timer T will expire. Expiring timer means sending a packet with the proper acknowledgement number - a strategy that guarantees that the connection stays open and remains fresh. If the other part is sending a lot of data, the contrary will be more likely to occur, breaking the awaiting. In order to help the other part, acknowledges will be delivered as soon as a *second segment* is received.

The core idea is to both help the other end and minimize the number of segments to be sent: this is done by preventing to respond with a segment whose sole purpose is to inform of acknowledged data *at each segment received*, while still answering with ACK information even though there is little to no data received. In fact, a delay time T assured to save sending some segments, a feature that historically was of a crucial importance.

1.1.7 RETRANSMISSIONS

The connection state includes three more actors, related to *retransmission*:

- a **retransmission timer**, which counts the time interval until which the retransmission is held back;
- a variable that describes the *current duration of the retransmission timer interval*, the **RTO (Retransmission Time-Out)**;
- a **retransmission counter**, that counts how many retransmissions for a segment have been performed.

The **retransmission algorithm** is as follows:

> Upon transmitting segment S, *if timer is not already set*, then the counter is cleared and set to 0, with the timer set to the RTO value. At the beginning, then, a new counter is spawned and the timer is set to a value which is the RTO value.
>
> When an ACK is received, `snd.Ack` is updated to the maximum value between `snd.Ack` and `segment.Ack`. If `snd.Ack == snd.Next`, the timer is switched off: the sent data has been received by the other party correctly and as expected.
>
> There may be occasions in which the timer expires, though. When timer expires, the *counter is incremented* and if the counter has not yet reached a `MAX_COUNT` value, the segment is retransmitted. However, this time the timer value is set to RTO but `RTO = 2 * RTO`, which is the double of the original value. *Only in-flight data should be retransmitted* (and all of it after timer expires). Basically, data for which we are sure that it has been received should not be retransmitted in case the timer expires, and the sender should only focus on retransmitting not-yet-acknowledged data. At each retransmission, the RTO is doubled, and the retransmission counter is increased.
>
> When counter reached `MAX_COUNT` value, an excessive number of retransmissions has been attempted, and connection is closed.
>
> Upon receiving an ACK for **all** the in-flight data, switch the timer off (that is, when `snd.Ack == snd.Next`).
>
> A particular condition is when an ACK arrives *for only a portion* of the data that has been sent. In that case, the timer resets (the receiver has responded) to the RTO value, *as if those still not-ACKed*

had been sent now, and the sender awaits ACK for missing segments. This basically occurs when a partial ACK comes, and prevents unnecessary retransmissions.

Exact timings of retransmissions are hard to predict. A sender is completely blind if it does not receive any ACK: should it send again in-flight segments, or should it wait a little bit more? In practice, how long should the timer be set? Basically, a rule of thumb is that it should be "slightly larger" than Round–Trip Time. In order to do so, since RTT largely varies depending on connections, time, and many other conditions, RTO is **dinamically updated** by algorithms such as *Jacobson algorithm*.

Regarding the maximum number of retransmissions, value of `MAX_COUNT` largely depends on the operating system. For instance, Windows closes connection after 5 failed attempts, Linux after 15. It is simple to realise that there may be a lot of unnecessary retransmissions: the timer can, for instance, run out too soon for the acknowledgement to reach the sender. Unnecessary retransmissions are a waste of resources, and definitely a fundamental problem. The reason could be one of those:

- segments are *lost*, retransmissions are necessary;
- there could be segments *whose ACKs are lost*;
- RTO was set to a *too small value*.

Thus, RTO should be set to an appropriate value, since a too short value leads to high overhead and possibly many unnecessary retransmissions, while a too long value results in high latency and possibly a slow or non-responsive connection. RTO value is basically a *trade-off* between exhibiting few retransmissions and having a fast and responsive connection. RTO should be set **dinamically**: it should be slightly greater than the *RTT* (Round-Trip-Time), an idea from Jacobson algorithm. This is of a crucial importance for TCP to work. Initial RTO value is heuristic, and varies from one OS to another. Linux and Windows start from same value, macOS adopts a different value, and so on.

1.1.8 MULTIPLE DEFAULT GATEWAYS

More default gateways could be added to a single node. Reasons to add more than one default gateway all boil down to *failure avoidance*. To know whether a gateway has stopped working, a heuristic TCP algorithm tries to detect a gateway failure:

If the number of retransmissions is greater than `MAX_COUNT` divided by 2, the *connection* changes its default gateway (if possible). Moreover, if the number of connections that changed default gateway is greater than the number of open connections divided by 4, the *IP layer* changes the default gateway as well. This last feature speeds up reconfiguration of early connections that still have to make some retransmission attemps.

Basically, each connection can autonomously choose its own gateway, but the IP layer is able to **force** any – new or already present – connection to use a different default gateway. A connection that exceeds the total number of retransmissions will change default gateway, and if 4 or more connections' default gateway switches are detected the IP layer will switch to a different default gateway as well, forcing all newly created TCP connections to switch to the new gateway.

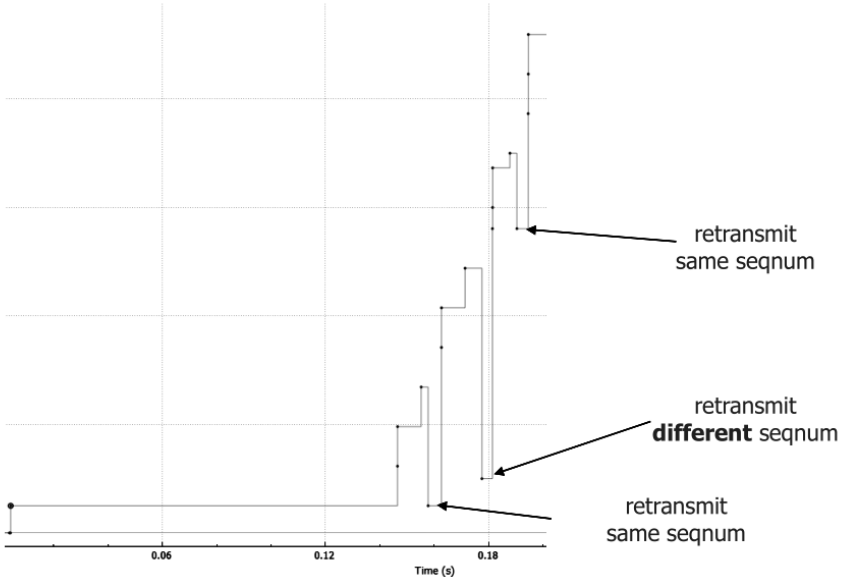


Figure 1.5: Time plot of sequence numbers. Dots correspond to sent segments. Some retransmissions occur.

1.1.9 Selective Acknowledgements

Suppose that 4 segments are sent, and the second one has been lost: in this case, the receiver got all segments except the second one, but it has no way to inform the sender to retransmit only the second segment. Sending ACK for

only the first segment would result in unnecessary retransmission of segments 3 and 4, which were instead correctly received.

For this reason, the receive buffer could end up having some **gaps**, missing bytes that are supposed to be received. The solutions are **Selective Acknowledgements** (SACK), a special kind of acknowledgements that carry both *left and right sequence number edges* of each out-of-order **block** in RX-buffer, this way correctly informing the other party of a *specific* missing portion of the data, avoiding unnecessary retransmissions. Selective Acknowledgements can selectively mark which bytes should be transferred again, and only those.

SACK protocol must be *supported by both members* of a connection in order to be useful. The protocol is very convenient, since many segments can be lost when a sender tries to deliver dozens of segments at once. This way, TCP can achieve *efficient handling* of the connection, minimizing the number of unnecessary retransmissions.

In SACK-supporting segments, a `TCP Option` field exist, in which both `left edge` and `right edge` are inserted, and they allow for correct data reconstruction after it has been lost, only for that portion delimited by the two edges.

Of course, out-of-order segments could still lead to gaps in RX-buffer, causing an unnecessary retransmission. In this case, unnecessary retransmissions are unavoidable when an out-of-order segment reaches the receiver too late, after a long time interval, since the receiver will send a SACK informing the sender of the missing segment, while the very same segment arrives late. However, no matter what, SACK is always convenient, since it reduces a lot the burden to the sender for only missing segments are to be sent.

To optimize out-of-order gaps, **Fast retransmit** algorithm should be adopted (not part of this course).

1.1.10 OPERATING SYSTEM TCP INTERRUPTS AND THE REAL TCP EXECUTION FLOW

Upon packet arrival, a *system interrupt* is sent.

An operating system executes many concurrent activities: application process, other networking processes, manages other system interrupts, and so on.

When a system interrupt is cast, the operating system will try its best to fetch data from the network physical device.

From the TCP point of view, after data comes from the operating system, it will analyze the packet to sort it by *looking at the connection table*, and then will **push in the input queue** the relative packet. The input queue holds a certain amount of segments, stored in the order given by sequence number.

When the TCP protocol decides it's time to process all segments in queue, it deeply analyzes them, picks their payload, grabs their header information, and processes all of them. After processing all segments in input queue, the buffer is emptied.

Basically, TCP can never **immediately** process a segment upon arrival. What it can do best, is to await some time, receive an amount of segments depending on the operating system's load at that moment and many other factors, and then process them altogether. Since many of them are processed at the same time, only a single acknowledgement will be sent back to the sender.

Since there are *many* different sources of randomness, let alone things that are completely out of TCP's control (just think of connection's quality, operating system's load, the other's end capabilities, packet loss, send–receive invocations), there is **no** single correct way for TCP protocol to exchange a given amount of data with another party.

1.2 Flow and Congestion Control

To behave optimally, TCP needs to establish the correct amount of data that should be exchanged in an interval of time.

Whenever TCP decides to transmit, there are three possible scenarios:

1. an empty segment is sent when there is no payload - that is the case, for instance, of acknowledgements or when timer expires;
2. it transmits a single segment if payload data is smaller than MSS - that is for few data;
3. it transmits multiple segments if payload size exceeds MSS - that's the case for huge payloads.

However, not all connections are equally good, and not all endpoints are fast enough to cope with a sender's speed. A sender should not send more segments than how many can be managed by the receiver – *how many* is an upper limit that should be dynamically changed.

The overall idea is the following one: TCP **starts slowly**, then *increases its transfer speed* according to the rate in which acknowledgements are received. Since the interaction is bidirectional and quite complicated, TCP implementation at sender's side tries to adapt to both connection properties and receiver's side properties.

As a general rule, the number of in-flight bytes is *always lower than an upper bound that is dynamically updated*, with an upper bound that is chosen according to proper criteria. This means that TCP can automatically *adapt* to the peculiar characteristics of the connection and receiver's speed.

In order to adapt the two different aspects of a TCP connection - the *connection quality* and the *receiver's speed* - two bounds should be adopted: the **Flow Control** and the **Congestion Control**. The TCP layer will always send less data than the minimum between the two upper bounds.

The first algorithm constructs a bound in such a way that the capacity of the receiver is always respected. Let an extremely fast computer send data faster than a receiving, slow, computer. Slower computer has not enough speed to collect all data that has been sent – the flow control algorithm lets the sender speed adapt to the receiver speed by means of a *send window*.

The second algorithm, the congestion control, constructs a state variable called *congestion window*, whose goal is to *model the maximum current throughput of the network*.

The number of in-flight bytes *must be slower than both send window and congestion window multiplied by MSS*, so that

$$\text{in-flight} \leq \min(\text{sndWin}, \text{congWin} \cdot \text{MSS}).$$

The mental model should be the following one: if the transmission buffer is full of data, a number of in-flight bytes equal to the minumum of both quantities should be sent, otherwise just send `snd.User - snd.Ack` bytes (those still to send).

1.2.1 FLOW CONTROL

Buffers cannot increase indefinitely: boundaries must be set in order to assure system stability. In Linux kernel, buffer sizes are set upon compilation and are then fixed; different operating systems may show other behaviors. In case data cannot be pushed into the TX-buffer anymore because it is full, application

should be suspended. When the RX-buffer is full, packets have to be discarded since they cannot be collected. Flow control will act to prevent this situation by letting the sender know how much free space is available to the RX-buffer.

The goal of flow control is to keep a fast transmitter from overrunning a slower receiver.

Since application sends data much faster than ACK arrive, the TX-buffer could end up being filled up by incoming segments. Suppose at receiver's side the application invokes `receive()` much slower than incoming packages speed. This way, the RX-buffer could fill up as well. To solve this issues, as application invokes `send()` when TX-buffer is full, it is put to sleep (blocked) until TCP gets proper ACK and advances `snd.Ack` (when it has more free space).

TCP sender keeps track of free space in other end's RX-buffer by looking at a variable in header (WindowSize) that informs it of how much free space is available, so that it can actually predict how much free space there is. When it guesses that receiver's RX-buffer could have no space left, it stops transmission and suspends the application. If the maximum window corresponds to the size of a single segment, the protocol is called **stop-and-wait** (that is - send a single segment, then wait for ACK). Larger windows enable pipelining of multiple segments in a row, allowing far more efficient usage of the connection.

Flow control algorithm adopts the concept of **sliding window**. Each segment header has a `WindowSize` field in the header that contains *how many free bytes are in the RX-buffer*. From sender's point of view, `snd.winSize` contains the *number of free bytes in RX-buffer of the other side*.

The value is initially set to the receiver's entire buffer size, and it is updated dynamically upon sending a segment. If a segment `S` carries `S.Ack` that is greater or equal to `snd.Ack`, then the variable `snd.winSize` is updated and set to *S.windowSize*. The `snd.WinSize` value ranges from 0 to the buffer size at the receiver's end.

Ultimately, the number of in-flight bytes must never exceed the number of bytes in `snd.winSize` that are available in the RX-buffer. This means that `snd.Next - snd.Ack` should **never** be greater than `snd.WinSize`.

The end goal of TCP is to reach a number of in-flight bytes that is as close as possible to the `snd.winSize` number of bytes, in order to optimize the connection efficiently.

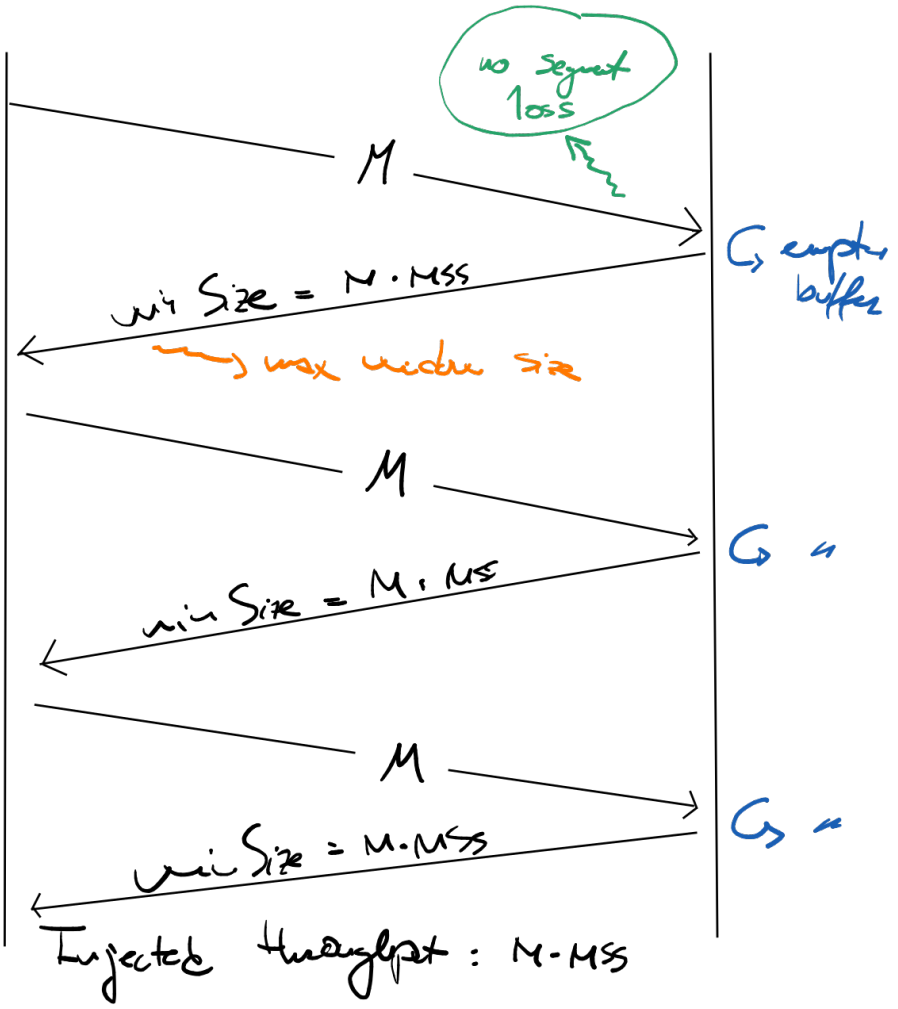


Figure 1.6: Ideal throughput condition, in which the injected throughput is equal to M · MSS at each time T. In scheme, every timing is considered constant, and the total time T is the sum of the time in which all the segments arrive at the receiver's end, the receiver's processing time and the acknowledgement's time to arrive at the original sender's side.

1.2.2 Performance estimation

From flow control, let's determine the maximum T-IN-TCP (input throughput) that can be obtained. TCP is injecting some throughput into the network, and the amount of it is controlled by the flow control algorithm.

Suppose the receiver has a window size of $M \cdot MSS$, and the application requesting data invokes `receive()` on a large buffer, such that the buffer is emptied as soon as there is new data.

The best case one can get by flow control is $M \cdot MSS$ every RTT – this is the case where the receiver's application empties the RX-buffer the moment the segments arrive. Suppose in fact $M \cdot MSS$ bytes are sent at each round, and no packet loss occurs: in that case, the receiver would immediately and successfully grab all segments, sending the corresponding acknowledgment answer.

In the case the application empties *half* the buffer at periodic intervals, the throughput would be the half as well, $M/2 * MSS$ every RTT. After the first round, only half of the segments would be sent, since the window size is indeed the half of the total one.

If the entire buffer is emptied at once, but with a delay, or not aligned with the receipt of segments, the throughput would be $M * MSS$ every RTT + Δ (more time than plain RTT). Combining all cases, one could get less than $M*MSS$ data every *longer* than RTT, depending on the parameters and the circumstances of the network.

For the best case, it's important to recall that some important assumptions have been made:

- all intervals have same time duration T, as described in Figure 1.6;
- all the segments arrive together, or at least not after a time which is irrelevant or already counted in T.;
- acknowledgement arrives immediately, and not after a huge delay;
- receiver is perfect in managing segments, and provides an acknowledgment segment for the full data.

In every case, a greater RX-buffer will result in higher throughput, since there is a proportionality between throughput and buffer size. Viceversa, greater RTT will result in less throughput – LANs will have greater throughput than WANs since they possess a lesser Round–Trip Time. However, in this formula the *quality* of the network is not taken in account: segments can be loss, especially when too many of them are injected through the internet. For this reason,

this much-optimistic model cannot be applied in cases where the throughput is high (read – when RX-buffer is insanely huge). Every time a retransmission is needed, *detection and recovery* takes place, and throughput becomes much smaller than the ideal case. This phenomenon is much heavier as the RTT increases, since retransmissions are harder to assess; Moreover, it is far from being predictable.

Another thing to consider is *maximum network speed*. Injected throughput could be *even greater* than maximum network speed, in that case many segments would be loss, resulting in a sub-optimal efficiency[4].

Establishing size of RX-buffer

Size of RX-buffer should have the following characteristic: the size should be *a multiple of MSS* – this is done to avoid leaving unused space at the end of the buffer, since in the lucky case exactly a multiple of MSS will be delivered. Non-multiple sizes would make no sense.

However, the RX-buffer size is a *trade-off* between greater ideal throughput and lower throughput due to excessive retransmissions, and should be chosen accordingly. Unexpectedly, $64KB$ are chosen; the value is OS-dependent. The size is managed as if it were a multiple of MSS – when almost full, pretend it is full and ignore last portion of buffer.

Flow control	Ethernet	Internet
MSS	$1460B$	$536B$
RX-buffer	$64KB$	$64KB$
number of segments in-flight	44	122

Table 1.2: Quantities involved in flow control, on Ethernet and Internet.

1.2.3 Sustainable throughput

A sustainable throughput is indeed **not infinite**. After a certain injected throughput `T_MAX`, the network bottlenecks and starts losing segments, exhibiting a behavior that will lead to less performance. Thus, an optimal throughput should be determined by and largely depends on *connection quality*. Injected segments are delivered at same speed only if the injected throughput is lower than the maximum possible throughput, that is `T_MAX`.

[4] This is solved by Congestion control mechanism.

An excessive throughput **will** result in the following consequences:

- packets exit **slower** than they entered;
- network's max sustainable throughput T_MAX will *collapse* – this fenomenon is called **network congestion**;
- network **will need some time to remove its congestion**, with even worse behavior in the case the unsustainable throughput is maintained for a longer time.

Intuitively, a network collapse happens because networks possess different maximum speeds, a quality that cannot be appreciated by TCP, let alone for networks *different* than the one in which the device is sending data.

It is then up to the TCP layer to detect issues during path, and slow down segments injection. Routers commonly have a mechanism for packet queuing, which can be filled up, resulting in packet discarding. Such discarded packets will need a retransmission, leading in even lower performance. New injected segments and packets will increase the time for which the queue is dissolved.

1.2.4 CONGESTION CONTROL

Congestion Control algorithm takes into account the quality of the network, allowing the sender to adapt to it instead of making it collapse.

Suppose the bound of flow control is so large that congestion control is predominant (which means snd.congWin*MSS < snd.WinSize). A congestion control algorithm could try do infer the maximum throughput by looking at the RTT and calculating MSS/RTT < T_MAX. However, this is unfeasible, since it requires too much rounds and tentatives to be inferred.

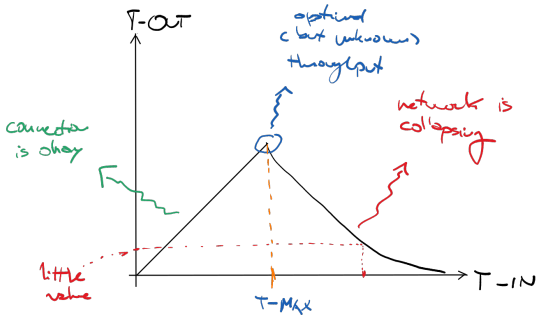


Figure 1.7: Network collapse after injecting a too-high throughput. Notice how injecting an even greater throughput will result in a *lower* throughput.

A second issue is that the quantity T_MAX *varies during time*: it depends on nominal throughput of networks and routers, and on **competition** of different injected throughputs in the network. Basically, the competition in the network is *unpredictable* and *time-varying*, allowing no fixed-value algorithm to work.

Actual algorithm

Congestion control algorithm has some important components,

1. the *slow start*;
2. the *congestion avoidance*;
3. the *fast retransmit*;
4. the *fast recovery*;

each one with several variants and many parameters to be properly set. We will see *Tahoe* algorithm, but the most used one is the *Reno* algorithm.

The core idea of the TCP Congestion Control is to increase the so called **congestion window**, a window size-tracking variable, every time an acknowledgement is received, while decreasing it a lot after a retransmission timeout had expired, starting again from a lower speed.

Initially, the congestion window is set to a very low value: that is called *the slow start algorithm*. The slow start algorithm takes place if the congestion window value is below a certain threshold, called *SlowStartThreshold*; under the slow start algorithm, the size of the congestion window increases very, very quickly. After the congestion window size is above the threshold, the *congestion control algorithm* takes place, and makes the window increase very slowly.

1. The state variable *CongestionWindow(t)* keeps track of a 'virtual window' that is the actual quantity of data that can be injected through the network, and is initially set to a very small value (*slow start principle*). Its value will increase whenever Acknowledgements are received in time – the network is assumed to be not congested. Upon retransmission (timeout expiration) the state variable is decreased a lot, and the network is assumed to be congested.
2. Another state variable, called *SlowStartThreshold*, is a threshold under which *CongestionWindow(t)* increases very fast, while above it the same state variable increases with a slower rate. Initial values of *snd.congWin* and *snd.ssThresh* are, respectively, 3 MSS and $64KB$.

3. Whenever an acknowledgement is received in time, the congestion window is increased. If the congestion window is lower than the threshold, increase fast with **slow start algorithm**; otherwise, increase slowly with **congestion avoidance** algorithm. The first will cause a fast growth in size of the window, the latter will instead produce a slower growth.

4. Upon retransmission (timeout expiration), the congestion window is reset to MSS. Differently, the threshold is set to the maximum value between 2 MSS and `(snd.Next - snd.Ack)/2` – that is, the maximum value between the double of MSS and the half of the currently in-flight bytes. The assumption upon retransmission is that the network is congested: this behavior is inefficient in the case the network is not congested. Since *many* concurrent causes may determine a packet loss – for instance, small RTO, network loss (e.g. WiFi) – one can end up with a congestion control algorithm forcing a slower pace of the network even though there is no actual congestion.

5. The congestion window is increased according to the algorithm running at a given moment:

   Slow start algorithm Congestion window is increased by +MSS whenever **any** in-time ACK is received. This is usually unaccurate, since the congestion window is usually *doubled* after all segments have been received;

   Congestion avoidance algorithm Congestion window is increased by +MSS whenever **the last** in-time ACK is received, for all in-flight data.

The two algorithms increase the congestion window at *much* different speed. Slow start algorithm, despite its name, will *double* the congestion window each time an acknowledgement for all segments is received. Congestion avoidance, instead, will increase the congestion window *linearly*, for which the window is increased by 1 (single MSS) each time an acknowledgement for all segments is received. The slow start will provoke an exponential growth, while the congestion avoidance will provoke a linear growth.

Congestion control and maximum throughput of the network

What can happen on start?

Let's assume all segments are lost, and retransmission occurs after RTT (that means, RTO is a good approximation of RTT). Let's also assume slow start produces an exponential growth. Initial value of *ssThresh_start* is $64KB$. Starting

with threshold value below `T_MAX`, slow start will yield an exponential growth; after some time, congestion avoidance will enter, a segment loss will occur and the congestion window will be reset to half the *ssThresh_start* original value.

Initial threshold above `T_MAX` will trigger a segment loss, and a reset of the threshold (without congestion avoidance algorithm, that has not enough time to act). After the first segment loss, threshold will be halved by setting it at half of the in-flight bytes, and system will act as it started below the `T_MAX` throughput. The "exponential, linear, drop" pattern will occur *forever*, and the three phases (slow start, congestion avoidance, loss detection) will happen one after the other.

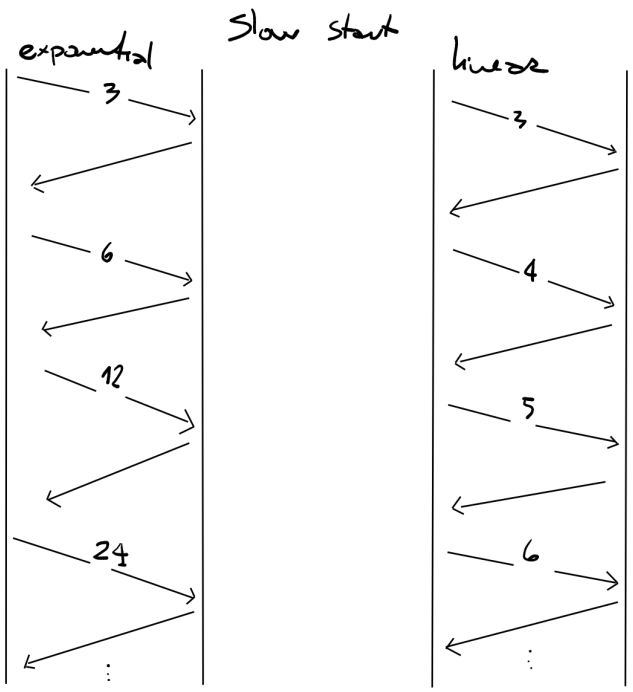


Figure 1.8: The two kinds of slow start evolutions. At left, exponential, at right, linear.

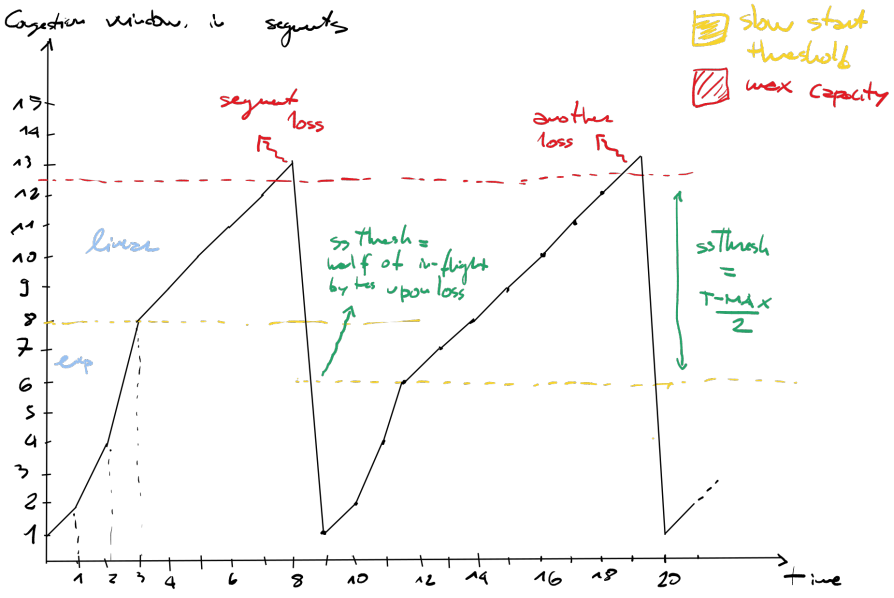


Figure 1.9: Congestion control and its dynamics. The pattern indefinitely repeats.

With same assumptions as above, but with slow start provoking a **linear** kind of growth, the pattern will be *completely linear*, since no exponential growth occurs. Basically, only the starting phase of each cycle will be different. The end result in both patterns, however, is that the average `T_IN` value will be **much** smaller than `T_MAX`: the TCP congestion control cannot expoit internetwork capacity efficiently. Injected throughput will always be lower than the maximum possible: the consequence, is that the average throughput is lower than what the network could sustain, however, this is very important in order to avoid network collapse and congestion.

There is a good reason for that: it is almost impossible to predict how the network will behave. A network could host many devices, each one having different demands, producing many kinds of traffic, each one with its own *volume*. The end result will be an unpredictable mess of network traffic. The only way to adapt to this is to adopt an algorithm using a heuristic behavior, that tries to adapt to the network as well as avoiding congestion the most.

Fast retransmit in a nutshell

Suppose K segements are transmitted in a burst, and a single segment among them appears to be lost. Resetting the congestion window to its minimum value is an inefficient move; since only a single segment of the row has been lost, one could argue there is no need to drastically decrease the congestion window.

Another strategy could be *waiting for a timeout before resetting the congestion window*: the solution is inefficient too, since waiting before retransmitting could lead to long breaks (how much to wait? how to estimate a correct time interval?).

The **Fast retransmit** algorithm addresses precisely this kind of issue: the solution is to retransmit *immediately only* what appears to be the missing segment, and decrease the window *but not* to its minimum value (that is, the maximum value between 2MSS and half of the current in-flight bytes).

Avoiding collapse of the network and the concept of fairness

The congestion control slows down the network in order to assure both *collapse avoidance* and *fairness*.

Intuitively, one would like to reach the maximum possible value for the congestion window and the maximum throughput sustainable; however, both are **unknown**. We can detect detect its value only after having injected an exces-

sive throughput, by detecing segment loss. However, whenever a segment loss occurs, the network is already congested.

An aggressive slow down occurs at each detection of segment loss, in order to prevent further collapsing of the network. In fact, suppose many machines are injecting a throughput which is closer to the maximum possible: in that case, if the network is congested, *it will require more time to return to its normal state*, because all machines are insisting injecting throughputs close to unsustainable ones, and removing congestion **will** require larger time. Basically, this is a heuristic way to avoid congestion on a network where multiple machines are using transmission resources altogether.

In the end, congestion control strives to ensure **fairness** with respect to the other devices. Fairness is the tendency of each competing TCP connection on a router to consume the same fraction of the router capacity. Each connection, on average, will try to consume an equal fraction of router's capacity. However, today's standard impose **node-level fairness**: this means that each node should have the same amount of network capacity (thousands of TCP connections can be opened at the same time from a single node). Note that UDP connections *do not* implement any flow or congestion control, and transmit at full-speed: they are said to be 'selfish'.

Basically, the end goal of fairness is to grant anyone the same access of the network, at the expense of your own networking speed (one could argue, however, that if anyone tries to push as much traffic as possible in the network, no one could be able to use the network, not even the same people that are pushing all that massive traffic).

Estimating the average throughput

Some assumptions should be made in order to compute average throughput, given Congestion Control algorithm is running properly:

- slow start has *linear growth*. The congestion window starts from 1 and goes up to $k + 1$;
- T_MAX is constant and in between $K \cdot MSS$ and$(K + 1) \cdot MSS$;
- *all* segments are lost at once (no wild retransmissions);
- all segments are received up to k-th round; segments in round $k+1$-th are all lost at once;

The *average throughput* that is injected is the average number of bytes in each pattern, divided by $(K + 1) \cdot RTT$. The number of bytes in each pattern is equal

to MSS * number of segments in each pattern - since we are assuming a linear growth, the segments are increasing linearly up to $k+1$, that means the number of segments belonging to a pattern is $\frac{k \cdot (k+1)}{2}$; hence, the *average throughput formula* is

$$A_t = \frac{MSS \cdot \frac{K(K+1)}{2}}{(K+1) \cdot RTT} = \frac{MSS \cdot K/2}{RTT}.$$

Thus, the injected throughput is the same one would obtain by having a *constant window* of $K/2 \cdot MSS$ size – on average and under ideal conditions, one can only get **half** of the maximum network throughput.

Slow start in exponential fashion makes this calculation quite more complex, but we will genuinely assume a linear growth in all our calculations, to make them simpler.

Idle connections

Another important scenario that has not been disclosed yet is about *not transmitting data after many RTTs*, the case of *idle connections*. The basic idea is to reset the values of congestion window and slow start threshold after a timeout expired (RTO).

Without a reset, one would use the last values that were determined by the congestion control algorithm. In order to avoid bad performance, the sender **resets both congestion window and slow start threshold** after a timeout. Basically, by resetting the connection state regarding previously achieved speed, the approach tries to be very pessimistic and conservative, in order to avoid any kind of congestion in the network.

1.2.5 Interaction between flow control and congestion control

The interaction between the two TCP control systems is not straightforward. After connection opening, the bound imposed by congestion control is *much* tighter than the flow control one (2–4 MSS instead of 64KB).

Hence, at the beginning the network *will follow congestion control*'s dictated bounds: data transfer begins slowly.

After connection opening (steady-state), there are two possible cases:

- `T-MAX > T-FLOW-CONTROL` – when the maximum network throughput is greater than the maximum flow control throughput: in this case, the bottleneck is the **receiver's capacity**. The steady state throughput will be $\frac{M \cdot MSS}{RTT}$, while its connection opening transient throughput will be half of that value, that is $\frac{M \cdot MSS}{2RTT}$. What happens here is that there is a steady-state with a fixed value due to the flow control algorithm, while before reaching it and during the opening phase the speed increased following congestion control (slow start or congestion avoidance depending on how the slow start threshold has been initially selected);
- `T-MAX < T-FLOW-CONTROL` – when the maximum network throughput is lower than the maximum flow control throughput: in this other case, the bottleneck is the **internetwork**, and congestion window will go up and down, never reaching the value corresponding to the receiver's buffer size. The steady state throughput will be $\frac{K \cdot MSS}{2MSS}$. Basically, in this case the flow control never occurs, both receivers will never reach their full speed and the real culprit would be the internetwork.

Since the steady state during congestion control is variable and corresponds to half of the actual internetwork speed, reaching a flow control steady state is much more desirable, especially when the flow control window size is set slightly below the maximum throughput. However, the exact `T_MAX` value is unknown and varies overtime, therefore it is impossible to reach such an ideal scenario.

Congestion control variants

Many variants of Congestion control have been developed. In particolar, there are many families for different use-cases:

congestion collapse this class attempts to increase the average T-IN upon congestion collapse in a standard, wired internetwork connection;

wireless this class strives to optimize correct non-wired environments, with algorithms tailored to environments with high network loss;

high-speed this class is tailored to sustain very-high bandwidth connections (e.g. backbones), with *high bandwidth-delay product*.

The latter is the case of *backbones*, with very-high bandwidth-delay product – those are all the networks that, if TCP is left as default, will lead to several minutes of connection start and gigabytes of data exchanged for the sole goal

of bringing the connection to full-speed! In order to optimize them, several variants of TCP have been developed.

As an example, suppose a sustainable throughput $T_{MAX} = K \cdot MSS/RTT$, and suppose the length of the first initialization period be $t_{init} = K \cdot RTT$. Solving in K leads to

$$t_{init} = \frac{T_{MAX} \cdot RTT^2}{MSS}.$$

By looking at the formula, one recognizes that the initial time has a squared dependency over round-trip time, a direct proportionality over T_{MAX} and is inversely proportional to MSS. In all connections where both RTT and T_{MAX} are large or extremely large, the initialization time could become relevant as well as the sheer quantity of the data that is needed (more than 3 minutes of transient, more than 100GB transmitted). For this reason, variants of TCP for backbones are mandatory.

The following Table 1.3 sums up the values for some kinds of connections, in the case of congestion control only.

	10Mb/s 10ms RTT	**10Mb/s 2ms RTT**	**1Gb/s 10ms RTT**	**10Gb/s 10ms RTT**
MSS	536B	536B	536B	536B
Maximum congWin	23	5	2332	23321
Congestion control throughput (Mb/s)	5	5	500	5000
Transient	0.2s	~0.0s	23.3s	3.9m
Transmitted data	145.8 KB	5.8 KB	1475.6 MB	145.76 GB

Table 1.3: Warm up time for different kinds of connections, using the standard TCP protocol with a standard congestion control. Notice the sheer number of seconds required to warm up a 10-gigabit network and the unsustainable quantity of transferred data when adopting the standard TCP protocol.

ESTIMATING THE DATA EXCHANGE TIME

Let T_MAX be known and let M the flow control window size be known as well; B bytes should be sent. To estimate the time of the entire data exchange, a solution could be to first estimate the maximum number of bytes in the transient

phase, that is

$$N = MSS \cdot [(\frac{M \cdot (M+1)}{2}) - 3];$$

if the obtained quantity $N > B$, then the entire data exchange occurs under congestion control. Otherwise, a portion $B - N$ of the bytes would fall under one of the two possible steady states, depending on the relationship between the maximum throughput of the network T_MAX, and the window size $M \cdot MSS$.

Notably, throughput in congestion control cannot be assumed to be exactly $\frac{T\_MAX}{2}$ — in fact, that is the average value over *multiple* periods, possibly many of them. In the case of a small value B to delivery, one could not have enough periods to assume the average throughput is the half of the maximum throughput of the network.

Knowing T_FC in place of M, one can easily find a solution by the same means as above. However, one should take in account that in order to understand whether the steady state lies in congestion control or in flow control, one has to confront the throughputs so that only in the case of $T\_MAX < T\_FC$ the steady state occurs under congestion control.

1.2.6 RELIABILITY OF TCP

TCP is a **reliable** protocol. The reliability in TCP only means that exchanged data are received, in order, and with no duplicates.

A common misconception about TCP is that the `send()` system call provides both delivery and reliability. Unfortunately, the `send()` call only guarantees that the data has been handed to the underlying operating system, without being able to tell whether the data has been successfully delivered or not. In order to do so and achieve reliability, the application must ask for data by invoking `receive()` — the `receive()` call will check whether there is a segment with a correct ACK or not.

Application layer invoking `send()` does not know how many bytes are received by the other side of the connection. The `send()` call may complete with no error, but no byte could have been transmitted – there is no way to know how many packages are actually delivered to the other party.

Suppose the `send()` invocation ends with an error message. Upon an error taken during the $n + 1$-th send invocation, there is no way to know if previous messages were actually delivered or not. Despite TCP being reliable, there is **no guarantee** that messages up to n-th were delivered and transmitted, since

the error message for `send()` call simply means there have been a failure in the TCP protocol, and data cannot be handed over to the operating system.

The correct reasoning is that all bytes *up to* k are correctly delivered, but one cannot know the exact value of k.

Regarding bytes delivered to the application level at the other end, bytes were received by TCP layer up to k but probably $k - k'$ bytes *are still to be delivered to the application*. This means only a portion of them, k', have been delivered. Both k and k' are *unknown*.

Hence, there is no exact possible reasoning on TCP when it comes to a `send()` invocation, since application should call `receive()` instead to be sure all the data up to k has ben received.

Only after *receive* call, with all ACK correctly received, one can be sure that all previous data have been successfully delivered.

Upon `receive()` or `send()` error, one *cannot tell anything about the previous `send()` invocations up to that `receive()` call*, since there is no clue whether the packet has been received correctly or not.

Any of these scenarios might be the reason for a `send()` failure:

- *not received*: packet never received by the partner;
- *not delivered*: packet arrived, but partner crashed before reply;
- *delivered*: packet delivered, but not reaching, connection is broken.

All these scenarios must be handled differently and correctly. Response to failure should always be a correct failure handling and proper recovery from damage, no matter its kind. Typical examples of error handling are:

1. first, take an error;
2. repeat the message;
3. wait some time;
4. send request again;
5. do until receiving a response.

However, this approach is often wrong. In fact, the party is repeating the same request over and over, possibly producing an effect multiple times.

When dealing with financial transactions, same requests can be performed multiple times, and lead to catastrofic errors and wrong operations. Only read operations can be handled this way effectively.

1.3 Opening a TCP connection

1.3.1 Sequence numbers initialization

Opening a TCP connection involves the decision of the **starting sequence numbers**, whose value is not necessarily set to 0.

Upon connection opening, since they are not set to zero, each of the two sides should agree to their initial sequence numbers. The ideal starting point is that every `snd` pointer should be set to the same value, as well as the other party's `rcv` buffer.

A first idea could be to simply send initial segments in which `snd.Next` are specified in header. Special segments could carry a flag that denotes a connection request (from the client) and `OK` (from the server). Both parties will set their initial `rcv` pointers accordingly to the value read in the header from the other party.

In reality, this first implementation suffers from many issues:

- delayed duplicates from client may be received after a long time – the server might uncorrectly believe another connection request is coming from the client (since there are port numbers, however, the new connection will be opened only in case the previous one has been closed);
- delayed duplicates from server may be received after a long time – this way, an opening with wrong sequence numbers may occur. In fact, suppose a server initial `snd.Next` number is equal to 63 and sent via a delayed segment. Now, if the server has sent another segment (which has not been delayed) with, let's say, `snd.Next` equal to 91, the client would believe the server has chosen 91 as the starting sequence number. This is not correct, indeed.

The solution to the above problems, in which delayed packets may be undistinguishable from not delayed ones, is the "*cross your fingers*" approach. Both machines *assume* that a predefined, *maximum segment lifetime* exist, after which segments cannot exist.

The lifetime in question is named **Maximum Segment Lifetime (MSL)**, and it is arbitrarily set to $2min$ ($120s$). After such MSL time passed, the machine simply assumes that the missing segment will never reach it. This is, in reality, wrong: IP do not have concept of *time*, and packets are discarded only based off their number of *hops*. Since the assumption does not match reality, the algorithm execution does not provide any guarantees — it does, however, guarantee that no wrong duplicates may possibly reach the other endpoint.

THE THREE-WAY HANDSHAKE PROTOCOL

The sequence numbers are generated by the **ISN-generator** (initial sequence number generator), a 32-bit counter increased every $4\mu s$, even when the connection is switched off. Overflow will eventually happen every 4 to 5 hours. The generator is used to generate an initial sequence number from its current value; a sequence number can be used again after at least 4 hours, after the ISN-generator will overflow.

Upon connection opening, the initial sequence number will be obtained by querying the ISN-generator.

The three-way handshake protocol (Figure 1.10 will begin with sequence number initialization, whose value has been obtained from the ISN-generator (suppose it's x), and send a `SYN(seq=x)`[5] packet to the server. The latter will answer with a package such as `SYN(SEQ=y, ACK=x+1)`, signaling both server's initial sequence number and that it has been acknowledged bytes up to x, and it is ready to receive the next (actually the first) byte.

The client will then respond with a second message, containing data and having sequence and acknowledgement numbers `(SEQ=x+1, ACK=y+1)`, showing that it is ready to receive the first byte after y as well; now, connection has been established and both parties will assume the connection is successfully opened.

The client side will be sure that no delayed duplicates are coming, otherwise x would have come from a connection which has been opened more than 4 hours ago! The same reasoning happens from the server side: any delayed duplicate should have been traveling for more than 4 hours, which is almost impossible to happen.

This protocol even protects in the case *old duplicates* are delivered to a server: in that case, the server will open a connection since the request looks legiti-

[5] SYN bit is a flag in TCP header that is always zero, except in the first and second segments, and it is used only for requesting connections.

mate. However, when the client receives the packet from the server – for something they did not request at all – it will respond with a REJECT packet[6]. The server will finally answer with a REJECT(ACK=y+1) to signal it has correctly acknowledged the request to close the connection.

A very unlucky case is where the server receives two delayed duplicates, one coming to open a connection and the other one just after the server sent its ACK; however, there is little to no chance that the ACK values are correct (and, in that case, it would have traveled for more than 4 hours, much more than MSL value of 2 minutes). In fact, in those cases, rogue packets from a client would contain CR(seq = x) and DATA(seq=x+1, ack=z), with the server expecting an acknowledged sequence number y. The same goes if the server receives a REJECT (it cannot be a duplicate). The mechanism is shown in Figure 1.11.

As we have already said, RST flags are used to reset a connection, or close an unwanted one. RST segments should be sent in two cases,

- unexpected rcv-seqnum in 3-way handshake;
- received segment for closed connection;

and in those cases, the RST segment is sent with the sequence number that is expected by the peer.

In case the RST packet is legit (that is, when it carries the sequence number the machine expects), then **close** the connection and **ignore** rcv-segment.

1.3.2 Interface between 3-way handshake and application layer

To open a connection, application layer just requests TCP to open a connection. The TCP layer will handle the connection opening with the appropriate system calls, with the protocol as described above. The system calls in sequence are

- socket;
- bind;
- listen(num, ...);
- accept.

System call *listen* has a *num* in argument that tells *how many connections can be queued*, that is, the number of incoming SYN segments that should be queued

[6] RST is a header flag as well as SYN, and it is used only to reject connections.

before opening connections. Until `listen` call has been invoked, no connection can be opened and the server will respond with `RST`. Up to *num* `SYN` segments can be accepted at the same time *before* invocation of accept and delivery of `SYN-ACK` packages. For instance, if `num=5`, the server will wait for 5 connection requests before actually answering to the client and opening the connections.

Closing a connection is done with the `FIN` flag. It is handled the same way as `SYN` is, and where one party wants to close connection, sets `FIN` flag to 1 and the other party will respond with a `FIN` flag to 1 as well.

This protocol can be abused to block a server with little to no effort. Attacks whose goal is to prevent the target working are called **Denial of Service** attack. An example of DoS attack is the **SYN-flood** attack.

The idea is to keep sending `SYN` segments so that server queue *is always full*: the effect will be that legitimate `SYN` requests from legitimate clients will be discarded. Some variants even hide the `IP-src` by modifying it, in order to hide attack origin. This way, it is impossible for a defender to prevent the SYN-flood by blacklisting IP addresses. Moreover, `SYN-ACK`s responses will be delivered to (fake) IP-src address.

SYN-flood attacks are very hard to detect and very cheap to operate. Indeed, randomly forging `IP-src` addresses makes almost impossible to filter them out with a firewall. SYN-flood attacks are also cheap: for 128 opened connections every 3 minutes, only $128 \cdot 40 = 5120B$ every $180s = 228bps$ bytes are needed to the attacker to forge necessary packets. To forge 1280 connections, $1280 \cdot 40 = 51200B$ every $3m = 6826bps$. A small cost of sending a packet causes the listener to force a listening to a connection, a **much** higher cost.

By designing a new protocol, one would likely avoid spending resources upon the first `SYN`, by operating *statelessly* until the initiation can demonstrate its legitimacy. However, a new protocol is almost impossible to design, since existing network architectures cannot simply be replaced.

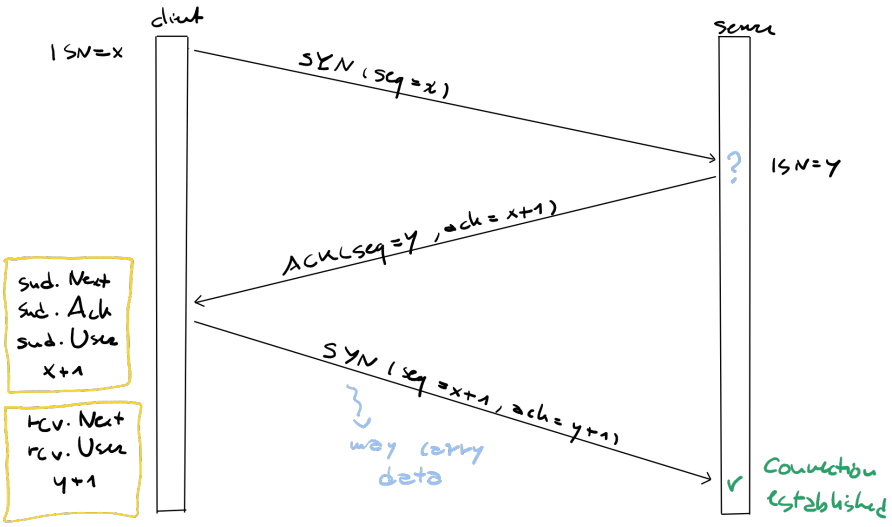


Figure 1.10: The three-way handshake.

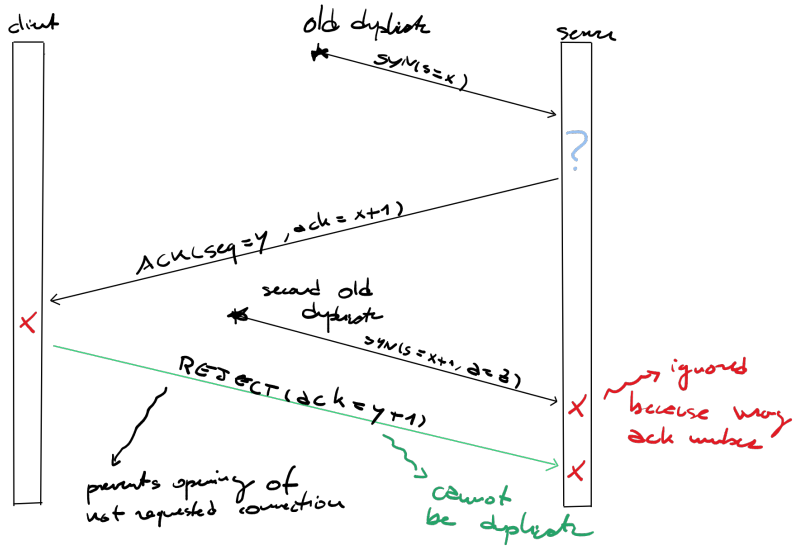


Figure 1.11: Possible errors in TCP handshake. The TCP protocol and its sequence number system is designed to avoid unsolicited connection opening, preventing the server to waste resources with no reason.

CHAPTER 2

THREAT MODEL

2.1 UNDERSTANDING THREAT MODEL

TCP offers **no authentication mechanism**. It means that if, for example, the address has been maliciously altered in DNS, a connection can be opened to the attacker's service rather than to the legitimate one and the client would not notice the difference. Network attacker may be in control of the router, too – he might be capable of switching DNS requests and respond with fake responses.

TCP offers **no integrity mechanism** either. A network attacker might change what the machine sends and receives – a communication may be altered by an entity that acts in between the two legitimate parties. Fortunately, an improved version of TCP, **TLS (Transport Layer Security)**, is available. It works by acting on top of TCP, and uses *cryptography* to provide integrity to protect the message and offers a strong authentication mechanism.

However, TSL can still be compromised: Suppose in fact the client has been infected by a *malware*[1] — then the attacker can control the client's behavior from remote, for example modifying web pages, opening connections, attempting privilege escalation[2] on the machine. In this case, TLS does not guarantee secrecy, integrity, authentication.

An crucial question arises: *is then TLS vulnerable and therefore pointless?*

[1] A *malware* is a malicious software, secretly installed by the attacker and against the user's consent, to perform ill-intentioned behavior.

[2] *Privilege escalation* is a technique in which an attacker that has gained unprivileged access to the machine tries to obtain super-privileged access by means of exploits, vulnerabilities and by other techniques. For instance, an attacker could try to obtain *root access* and root privileges to perform anything he wants on the machine.

Basically, it all depends on **threat model**. A threat model is the set of actions that *we assume the attacker can execute*. One makes an *hypothesis* on the set of attacks that are possible, and then it reasons accordingly. In fact, reasoning about security of a system without a well-posed threat model makes no sense at all, because an attacker that doesn't have a predefined set of attacks and can do anything is an attacker it is impossible to defend against. That said, a *real attacker* may not be confined in a single, limited, threat model, because it both may be modeled unaccurately and it may be unpredictable.

An important remark is that all protections in software are designed with a specified threat model in mind. They do not work outside their threat model which they are based on, and for this reason software should be chosen accordingly. Both protections and possible kinds of attackers are important to be established before acting, in order to minimize expenses, maximize defense capabilities, and increase the chance an attacker is successfully repelled.

Important questions when dealing with attackers are the following ones,

What's the *threat model* for that specific attack technique?

What's the *least pessimism* required in order to execute an attack?

What's the *best software* in terms of *cost, defensive capabilities* and *chance to be attacked* that should be used to defend to a given attack?

Typical threat models involve the following scenarios, in increasing pessimism:

- **Network attacker** – the attacker can only operate in the network. He is able to observe and communicate. DoS (Denial of Service) and MITM (Man in the Middle) attacks are usually what a network attacker can do;
- **Compromised endpoint** – the attacker compromised a node in the network with a malware. That node is fully compromised and should be considered not trusted;
- **Physical access** – the attacker obtained physical access to an endpoint. Regardless of the fact a machine is already infected or not, physical access grants much more privileges than the above threat models, for instance it can compromise much more easily an endpoint or can install devices that are physically capable of acting as man in the middle;

- **Evil maid** – a *variant* of both *physical access* and *insider* threat models in which an person that has frequent physical access to a machine is instructed to perform operations on the device. An evil maid has no access on supply chain or organization internals, though;
- **Insider/Supply chain** – this very dangerous attacker has access to portion of the *software supply chain*, for instance it can manipulate software libraries the main software depends on, infrastructure on which the software development occurs, and so on. In the case of the insider, it also has access to the internal communications and is able to actively spy the organization he is into.

The TLS (and by extension, the HTTPS protocol) grants *secrecy*, *integrity* and *authentication* only for a network attacker–based threat model. Outside that model, TLS and HTTPS offer no guarantee of protection.

2.1.1 CHOOSING THE DEFENSES

Threat models are also useful to define from which kinds of attacks one should defend, from which ones one cannot defend and should act accordingly, and from which ones one simply has to "cross his fingers" (do nothing).

The first category of attacks should be treated with proper software that offers protection against a well–defined threat model. Outside that model, the software should be considered useless and the system that it protects is compromised and not anymore trusted. This category of attacks is rather frequent, doesn't cost too much, and poses immediate danger to the activity of the organization.

The second category, whose best defense is *damage compensation*, is the category of attacks that are not very frequent and whose relative defensive mechanisms cost too much to be economically viable and sustainable for the defending organization. In these cases the organization simply prepares for damage mitigation, with strategies such as *data backups*, *backups of systems* and by other means.

The third category is both rare and costly. These kinds of attacks are very dangerous, but occur extremely rarely. It is therefore rational to simply allow this kind of risk to happen, without taking any countermeasure in account.

An crucial remark is that for *choosing the defenses*, it is not important **how** an attacker has managed to operate an attack, and it suffices to know the specific threat model and attack set an attacker is able to conduct. For *assessing likeli-*

hood of a threat model, in order to decide the actions to take, it suffices to know how an attacker has been able to achieve a certain attack capability.

In fact, the details on how an attack works and how it is only necessary to find and determine the cause of a given vulnerability or exploit.

The first step of a defender should be to determine a threat model by looking at how simple is for an attacker to achieve a certain specific attack. The next step is to determine the best defenses *given* a threat model.

2.1.2 THE NETWORK ATTACKER THREAT MODEL

The *network attacker* threat model is implicitly assumed in this book. A network attacker can only operate in the network, by communicating, observing and altering existing traffic. An attacker of this kind cannot act on the endpoints.

The implicit assumptions are that everything on the endpoints is perfectly implemented, operated and maintained — this means that there are no vulnerabilities, no exploits, and no obvious mistakes by the system administrator. Software is perfectly maintained as well. No mistakes are allowed, no remote code execution, no vulnerability, no mistakes from the users, in cryptography, protocols and communication, and operating systems.

These kind of assumptions are important to confine the attacker to the network threat model; otherwise, an attacker could be able to infect an endpoint, upgrading the threat model to the *compromised endpoint* threat model.

The key defensive tool against a network attacker is **cryptography**. Cryptography allows both *secrecy* and *integrity*, and by means of a structured trust system it allows *authentication* as well. All of these protocols make use of cryptography:

- TLS and HTTPS;
- Kerberos;
- Windows Active Directory;
- OAuth (SPID);
- SSH;
- IPSec;
- Legally binding digital signature;

- WhatsApp, Signal, Telegram;
- and so on.

Examples of possible attacks a network attacker is capable of are listed below,

- **DNS Spoofing** — a DNS attack in which an attacker modifies the IP address (and the ID in header) contained in the response to a DNS query, in order to hijack a client to an attacker controlled server, and act as man in the middle, or showing a perfect copy of another website. Windows systems are particularly vulnerable to this kind of attack, since by default they prefer any IPv6 server and happily substitute the preconfigured IPv4 server upon a malicious packet containing new instructions arrive;
- **ARP Spoofing** — fake responses crafted by an attacker, whose goal is to make a device change its default gateway to an attacker's controlled device, so that it can act as man in the middle. Open WiFi is vulnerable to this kind of attacks;
- **BGP Spoofing** — highly-skilled attack in which an attacker manages to alter the routing of the internet, so that it can direct traffic to different places. Border Gateway Protocol messages are not authenticated; Autonomous Systems can claim ownership of any IP address;
- **Dishonest system administrators** can act as man in the middle;
- **Judicial authorities** or **intelligence agencies** possess huge available resources, and can force organizations, internet nodes and system administrator to act as man in the middle.

2.2 ATTACKING TCP CONNECTIONS

A legitimate threat model for a TCP connection is the network attacker threat model. A network attacker can observe, communicate and alter traffic by acting as man in the middle in all communications between two endpoints.

The first model of the network attacker that strives to control a TCP connection is the **on-path attacker**. The on-path attacker is able to read integrally the communication between the two endpoints, and can act as man in the middle.

The objective of the first kind of attack we will encounter is the *client impersonation* in a TCP connection between a client and a server. The server will see the IP of the client at the other end of the communication, whilst packets are actually sent and received by the attacker.

There are three possible ways to perform such an attack:

1. **Opening a connection** and impersonating the client's IP address;
2. **Injecting some data** in an already opened connection;
3. **Closing a connection** already running.

In the second method, the attacker only injects *part* of the communication, and represents a less pessimistic approach, in which the attack does not manipulate the communication and can only inject data. The third method is the more optimistic one, with the attacker only capable of closing an existing connection in order to perform some kind of Denial of Service.

All three attacks are trivially feasible, with an attacker able to act as man in the middle. In the first case, the attacker is able to intercept an opening connection, and starts handshake with the server, opening a connection.

An on–path network attacker is able to intercept all segments. In particular, such attacker is interested in obtaining sequence numbers, so that it can successfully replicate the connection state, inject data, and manipulate correctly the connection by means of forged segments with correct sequence number.

Data injection by an attacker cannot happen at an arbitrarily high rate. Attacker must forge correctly numbered segments, so that both the client and the server could not notice the manipulation. This in the past was a tremendous issue, since there were big problems with remote shells from clients authenticated by IP — today it poses no longer a problem, since authentication occurs almost always by encrypted protocols such as *SSH (Secure Shell)*. Injecting data is easier at TCP level, with carefully crafted headers and proper payload; backwards, data injection at application layer is much harder, since an attacker must know application protocols and expected data at time t. This, however, is feasible: the *National Security Agency* has done it many times in the past, in a highly targeted way, with web browsers driven to web servers that inject malware, by means of injecting HTTP redirect in HTTP responses.

Attacker can also close a running connection. All he has to do is to send a single `RST` segment with correct sequence number to be accepted by the server, and the closing procedure will start. Closing a connection is really feasible: a sequence number can be easily inferred by looking at the incoming sequence numbers, and to increase the likelihood of picking the correct sequence number, *many* forged segments are sent with sequence numbers that are likely to be the correct ones.

The second kind of TCP attacker is the **off-path attacker**. An off-path attacker is less powerful than the on-path attacker, since it is not in the path and thus cannot read the correct sequence numbers.

An off-path attacker is **not** able either to open a connection or to inject data. However, despite the deep limitations in what he can do, he is still able to *communicate* to each endpoint and this suffices to be able to close a connection and perform other kinds of Denial of Service attacks.

The attacker's objective is to determine the client information regarding *sequence number* and *port number* at time t_x: the two informations suffice to close a connection easily. The procedure is as follows:

1. to begin with, the *correct sequence number* should be estimated by means of stochastic techniques;
2. the *client's port number* is determined as well, by means of stochastic techniques or by looking at the default numbers;
3. for each pair of *possible candidates*, send a `RST` segment to the server.

To determine a possible candidate, the idea is to first get the current initial sequence number from the client, then estimate the amount of data exchanged by the TCP connection, so that by adding the estimated amount one could obtain the initial sequence number at the time and choose the candidate in the most promising range. The initial number is obtained by a fraudulent connection opening with the client, so that the initial sequence number can be obtained; if the *slope* of the initial sequence number generator is known at the time interval `t` (that is, if one knows `ISN@C(t)` and how `ISN@C` behaves he can estimate `ISN@C(topen)`), the attacker can make a guess on the *starting* sequence number the client employed. After that, estimation is based on the application protocol and the total number of bytes sent can be guessed. A *range of likelihood* for sequence number is determined, and finally as many segments as possible are sent to the server, hoping for the best.

The power of this method lies in the sheer computational power of the attacker. If the attacker is able to produce a huge number of sequence number and port number candidates to forge enough `RST` segments in a sufficiently tight time interval, the server could end up accepting the fraudulent close request, determining the closing of the connection. This results in a powerful method for Denial of Service, for any kind of existing TCP connection.

Historically, in 1985 people started realizing what *could* be done against TCP initial sequence numbers; there were no proof of concepts, however. The first

real attacks were performed in 1995, and pushed the widespread adoption of pseudo–random number generators that should be addressed by the TCP algorithm in order to determine its initial sequence number. Basically, that made apparently impossible for an attacker to determine the correct sequence numbers. That said, in 2001 attackers were still able to determine it by means of *statistical techniques*, so people realized that the widely adopted algorithm were suffering from vulnerabilities and design failures. The number of attempts for succeeding was much smaller than the amount that was believed to be necessary for the attack to work; clever bruteforce was feasible. In 2001, a new algorithm was built in order to make the initial sequence number choice unpredictable. Unironically, in 2004 attackers were already able to produce efficient Denial of Service attack, since the required number of attempts was sufficiently small to be feasible.

The lesson learned is that even if *today* realistic threat models assume that a certain attack is not possible, *after some years* someone realizes that the assumption is wrong; this is due to the fact that resources, knowledge and techniques vary over time.

The same fate struck the WPA2 protocol; designed in 2004, a serious weakness uncovered in 2017 allowed an attacker to read, inject and manipulate data in the encrypted protocol. Since the weakness was in the standard itself, any correct implementation of WPA2 was likely affected.

In 2014, off–path data injection became possible as well. On HTTP, it is now possible to insert *scripts* and *HTML segments* in a communication, just by sending packets to an endpoint.

2.3 The Man–On–The–Side attack

A **Man–On–The–Side** attack is an attack in which an "observer" network attacker attempts to hijack the web browser of the target to a website controlled by the attacker, with the goal of injecting exploits on the victim's browser.

To do so, an attacker first intercepts the HTTP request, and *immediately sends back* an HTTP redirect, with the sequence number expected by the victim's browser. The latter performs the redirection to the attacker's controlled website, with an HTML document forcing the download of many exploits.

The legitimate response from the server will arrive later, thus appearing *duplicate* and it will be discarded by the victim's browser.

This kind of attack is very powerful if performed by intelligence agencies, usually with the help of Internet Service Providers.

Man on the side attacks are fully prevented by HTTPS protocol.

Chapter 3

Network Address (Port) Translation

In networking, some IP addresses are *reserved*. Among all the possible addresses, there are 4 ranges that cannot be assigned to any "public" entity:

1. the `10.0.0.0/8`, 2^{24} addresses in total;
2. the `172.16.0.0/12`, 2^{20} addresses in total;
3. the `192.168.0.0/16`, 2^{16} addresses in total;
4. the `224.0.0.0/4`, 2^{28} addresses, reserved for *multicast*.

Addresses belonging to the first 3 ranges are said to be **private** IP addresses, in contrast to the **public** IP addresses. The first addresses belong to the internal network of a single organization, while the second addresses must be purchased from an ISP, or publicly purchased from some governative organization, and they cannot be assigned autonomously. The number of devices is much greater than the number of available public IP addresses. Private network numbers are then necessary in order to manage the network of organization. A private IP address is unique only in the boundaries of a single organization.

As a general networking rule, every frontier router must discard incoming or outgoing packets with private IP addresses. This means that each and any device possessing a private IP address cannot communicate directly with any IP address outside an organization, and it has IP connectivity only with nodes internal to the organization. Since nowadays it is very important to be able to communicate externally (many applications do require direct internet access),

a different communication mechanism for devices with private IP addresses should exists.

To achieve the goal, there are two different technologies with similar functionalities, but different implementation. The first is the **NAT**, Network Address Translation. The network address translation is no longer used. The second one is called **NAPT**, Network Address Port Translation, and it is usually shortened with *NAT*. NAPT is what's used today on gateway routers.

3.1 How NAT works

The basic idea of **NAT** is to *modify the header of a packet* so that the private addresses are replaced with a public address, and the port is changed accordingly. Suppose a device with private IP address `192.168.0.1` is trying to connect to `4.26.18.144`, a public IP address, on port 777. The packet will travel to the default gateway, our NAT device. The NAT will then inspect the packet, look for address and port information in header, and *modify them* so that the source IP address now is the public IP address of the router (let's say, `80.68.17.190`), and the port number is changed with an on–the–fly generated port number that is currently available (suppose 18923 instead of 777). Basically, a NAT devices alters IP address and port information of a TCP and UDP packet and maps them into the public address of the router *plus* a port number which corresponds to the device that is requesting the connection. NAT module will also modify the *header checksum*, reconstructing it in order to match the new packet that has been built (otherwise, all modified packets would be discarded within the next hop).

All a NAT module should do is to remember for some time that the device packet has been mapped to a specific public IP address (all devices will be mapped to the same one) and port number. Each time a packet from the inside travels through a NAT, a new device—port number mapping is produced and stored in a table, the **NAT Mapping Table**. The mapping table stores all the mappings across time. After a time interval, a mapping is discarded.

Upon new connection from inside to outside, NAT will look at the mapping for translating `IP-I, port-I` to `port-E`. If there is no existing entry, choose unused `port-E` and allocate the entry in the mapping table. After that, modify all TCP/IP headers according to the table.

When a packet from the outside comes to the NAT module's external interface, NAT device will inspect both the packet header and its own mapping table: if

a correct mapping exist, packets with the correct public IP address as recipient will be delivered to the correct recipient *inside* the organization; which device, it depends on the mapping between the port number and the internal device. Basically, incoming data are translated back again by means of the *same* association in mapping table. Such translation at both sides will result in packets correctly delivered to their destinations.

A very important aspect of the NAT translation is the way that a NAT module *protects the network*. In fact, **any** packet, request or access from the outside for which an existing association does not exist, **will be discarded**. This is a crucial facet of a paramount importance: one of the most important defense mechanisms is to avoid access from the outside; by using a NAT module, one is able to immediately obtain network insulation from the outside, for free. For this reason, NAT is said to be *"a native firewall"*. All of this does not require change or configuration on internal nodes, they only need to set up a proper gateway — address translation happens to be **transparent** to the clients.

NAT modules can be configured with **immutable entries** as well. Immutable entries are entries that should be configured first, so that they never expire and they are chosen accordingly. Immutable entries make sense when one has to run an internal server that should be accessible from outside; it is not uncommon to generate immutable entries so that, for instance, `IP-I1, 80` is bounded to port 80 as well as `IP-I1, 443` is bounded to port 443. Usually one wants to choose and dedicate known port numbers so that their services work.

The number of external connected clients is irrelevant. The NAT module, in fact, only maintains the state of the *internal* nodes, and not about the external endpoints; this means that as many nodes as one wants can be connected inside a single NAT device.

The drawbacks of this approach are that no more than *one* internal server for any given protocol (on the standard port) is allowed, otherwise there would be collisions in the mapping table, that are multiple devices mapped to a same external port number.

3.1.1 Entry releases in NAT

Since port numbers are limited in number (16 bits only, from which one has to ignore known port numbers) and precious, NAT discards old and unnecessary entries in the table. To do so, it observes TCP segments that **close connections**,

in particular looking for `FIN` flags. Upon observing the last segment, an entry could be released.

Moreover, old entries are discarded as well:

- if an entry is **unused** during time interval t_u, the entry is removed — TCP connections will crash as a result (and UDP connections may not work as well);
- if an entry is still in use, but allocated **timer** t_a **expired**, the entry is removed. The TCP connection will likely crash as well as the case before. Most NAT modules also implement this feature.

3.2 Multiple NAT modules on the path

Multple NAT modules on a single path are the norm. What usually happens is that two devices of two distinct organizations want to communicate — at each organization's boundary lies a NAT module.

The mechanics are completely similar to the previous ones. What happens is that *each NAT module hides internal IP address from the other's end*, performing transparent translation as nothing occurred.

Let organization 1 have a client that wants to connect to organization 2's server. Organization 2 has correctly set up an immutable entry for its own network service, so that it is accessible from outside. Now, consider a packet from the client to the server and its response. The following table summarizes the exchange:

3.3 Multilevel NAT

NAT modules can be structured in a **multileveled** way. In particular, external IP addresses **need not to be public**, and might be private addresses as well. This means that one can set up a NAT module that translates between private IP addresses in a same organization, provided a further NAT module exists and translates the private IP address into a public one.

More generally what a NAT does is translating *internal* IP addresses into *external* IP addresses; numerical values do not matter. As a matter of fact, packets could find further NAT modules on the path that translates its addresses multiple times, from private to private ones, private to public ones, and public

	src IP	src port	dst IP	dst port
Inside Org 1	IP-I1	port-I1	IP-E2	port-E2
Internet	IP-E1	port-E1	IP-E2	port-E2
Inside Org 2	IP-E1	port-E1	IP-I2	port-I2
Inside Org 2	IP-I2	port-I2	IP-E1	port-E1
Internet	IP-E2	port-E2	IP-E1	port-E1
Inside Org 1	IP-E2	port-E2	IP-I1	port-I1

Table 3.1: Message exchange between a client and a server in the case of multiple NAT modules on the path.

to private ones, before arriving at destination. It usually happens that an ISP owns some public addresses, but only gives you access to the internet through some NAT modules. Basically, the home router's external IP address is usually a private IP address, and the device is visible from the real IP address only through a series of NAT modules. The same goes for **hotspots** on smartphones; since the device has a private address, there is a further NAT that allows the device to access the internet.

Most networks are behind many multilevel NAT architectures before even reaching the internet. In order to expose a server by configuring NAT modules only, **all** NAT modules should be within the same organization and be configured properly. Since most of the path is usually out of the organization's control, it is practically unfeasible to expose a server to the outside. What should be done is buying a **public** and **static** IP address, so that the server will be accessible regardless.

3.3.1 NAT and protocols without port number

There are tons of protocols that are port number agnostic. For instance, the **ICMP** protocol does not adopt ports. Since there are no specified port numbers, a NAT module could not be able to handle such packets.

In reality, a NAT modules handles ad hoc all ICMP packets, by creating a *temporary association* that it is not based on port numbers (suppose a `ping` message exits the network — the NAT module creates a temporary association so that the response could be delivered to the requesting client).

If the client requesting a `ping` is outside the network, the NAT module will respond to ICMP request autonomously.

3.3.2 NAT–unfriendly protocols

Some protocols are said to be **NAT–unfriendly**. NAT–unfriendly protocols are those for which *server addresses are specified in application data*. For all those protocols, if there is a NAT module along the path, they may not work properly.

For instance, consider **FTP** protocol. FTP uses two ports, port 20 for communication and port 21 for actual data transfer. Suppose now a client uses command port to communicate that one wants to use a *different* port than 21 for data transfer — the server responds, let's say, with port 10000. Now, the NAT module that actually transforms the packets **will be unaware** and simply keep checking for packets at port 20 — even worse, port 10000 may already be in use! Thus, FTP protocol is said to be NAT-unfriendly due to the incompatibility between how port numbers are exchanged and how NAT actually works.

Despite this drawbacks, today's NAT modules are able to analyze and inspect application payload, so that port exchanges in FTP will be predictable and assigned properly, by understanding the case in which there is a change of data port from 20 to another one.

Another NAT-unfriendly protocol is the **HTTP** protocol. Embedded and hard-coded urls in HTML documents may ask for non standard ports — in that case, NAT module should be configured for accepting traffic to that port, from the outside (while there are no issues for an internal client). Even though it is unfriendly in practice, HTTP protocol is considered to be a NAT-friendly protocol.

Chapter 4

Virtual Private Networks

A **Virtual Private Network** is a technology that uses the concept of the **tunnel** to provide connectivity between two endpoints, be them public or private IP addresses. A tunnel is a *virtual* network.

Suppose a VPN connects the two gateways of two organizations: **all IP traffic** between the two organizations is *routed on the tunnel*. Basically, it happens that from the *logical* point of view the two organizations are connected by a virtual network. Each device at the tunnel end will be equipped with two network interfaces; each network interface has its own IP address, with its subnet mask and its default gateway. On a VPN, the default gateways are the other end's IP address, so that the packets going outside will always reach the other organization.

Tunnel does not provide routing: organizations are expected to set it up appropriately. Suppose a client of the first organization sends a packet somewhere. If the destination IP address lies in the network address ranges of the second organization, then the routing should be configured so that the packet will travel through the tunnel to reach the other organizations; otherwise, the packet should be delivered through the internetwork. The same goes viceversa.

The tunnel offers a functionality of packet **encapsulation**. Packet encapsulation means that the original packet is enclosed within the *frame* of the virtual network. What happens is that the packet, travelling on the tunnel, is first encapsulated into a proper frame, in which it travels through the tunnel. After the packet has reached the other end of the tunnel, the packet will be extracted by removing the frame capsule and it will be treated as if it were never part of a virtual tunnel.

A tunnel may be established between **any** pair of nodes, be them *at the border of the organization* or be them *internal nodes*. The traffic actually routed on the tunnel will depend on the routing tables specified by the two endpoints. Since the virtual network will "bypass" NAT modules, integration between remote internetworks that have private IP addresses will be possible even without using traditional address translation.

The tunnel is an **abstraction**, only valid from a logical point of view. In particular, the tunnel functionality is provided by a regular TCP or UDP connection between two endpoints, through the internet. This means that the frame is encapsulated in an external packet — a packet that could actually travel through the internetwork.

VPNs might have many implementations. One of them is by means of a **Point-to-Point Protocol** tunnel. The original packet that goes through the tunnel will be first encapsulated within the PPP frame of the virtual network, and then delivered through the tunnel, by means of encapsulation in a legit packet. The packet is then delivered via the internet, inspected by the endpoint, extracted, read the frame, and removed the encapsulation to extract the final payload, the packet that belongs to another IP address inside the organization.

Basically, what happens is that if a packet should be sent to a network number that must be routed on the tunnel, it must first be encapsulated in a PPP frame, then the frame will be put inside an IP packet with the destination IP address and the packet will be finally sent over the internetwork.

Let `IP-X` and `IP-Y` be the two gateways of Organization X and Organization Y. The two organizations have formed a VPN tunnel between their two gateways. Suppose a `IP-s` client belonging to Organization X wants to send a packet to the `IP-d` client in Organization Y. First, routing should be set up so that the packet to `IP-d` should not be sent through the internet, but to the tunnel. Second, the packet `IP-s, IP-d` will be encapsulated into a PPP frame to be delivered in the tunnel. Third, the tunnel does not exist in reality: this means that the actual tunnel is provided by a communication between the two gateways of the two organizations. For this reason, the gateway `IP-X` will manage to send the packet and thus the PPP frame will be encapsulated inside an `IP-X, IP-Y` packet, that will travel through the internet.

```
IP-X, IP-Y | PPP Frame header | IP-S, IP-D | payload
```

Another way to produce a tunnel is by means of a **TCP tunnel**. The TCP tunnel encapsulates all packets by means of a TCP header, having port numbers specified. The payload of the IP packet `IP-X, IP-Y` is a TCP segment. Yet

another way is the **UDP tunnel** way, in which the payload of the IP packet that travels through the internet is now a UDP packet.

Irrespective of implementation, an IP packet of the form `IP-X, IP-Y` will have as payload the tunnel protocol, whose content depend on the adopted tunnel protocol.

4.1 SECURITY GUARANTEES OF VIRTUAL PRIVATE NETWORKS

Traffic inside a VPN has some important security guarantees, so that it is practically a good means of achieving a secure communication between two endpoints. In practice, IP traffic in virtual network has *secrecy*, *integrity* and *mutual authentication* properties; for this reason the traffic is **isolated** even though the tunnel is implemented on a *shared* medium.

VPN also offer protection against **replay attacks**. For all the above reasons a VPN guarantees much more security from a network attacker, all of this by **reducing the attack surface** a lot:

1. due to **secrecy**, an attacker cannot tell what's in the traffic, since there is no way for him to inspect the data;

2. due to **integrity**, there is no hope for an attacker to manipulate the data inside the tunnel;

3. due to **mutual authentication**, there is little to no way for an attacker to impersonate one of the endpoints;

4. for all the above, the **replay protection** is guaranteed.

An attacker can only detect the existence of VPN traffic, possibly performing some DoS attacks (only if MITM). A network attacker can conceptually be in any network — in practice, however, it is much simpler to sniff and manipulate traffic **when closer** to the organization boundaries. In other words, an attacker will try to shorten the distance between himself and the organization to attack. When closer, focusing on traffic of interest becomes easier. Bigger risks are posed on **open** Wi-Fi networks and personal Wi-Fi in **promiscuous** environments (for instance, at a restaurant, at the airport, at the school.

4.2 Three use cases for a VPN

4.2.1 Scenario I — connecting remote organizations

The first useful scenario for a VPN is **to connect remote organizations with internet connectivity**. In this first scenario, two VPN endpoints are configured by the two organizations so that they created a virtual tunnel between them. The IP traffic of all clients across the two organization will never travel on the internet without being first encapsulated in the tunnel. The VPN connects all traffic between two organizations, without logically passing through the internet.

A *specific* case of this first scenario is when a *remote worker* connects to the organization's intranetwork by means of the VPN tunnel. In this case, all the traffic is encrypted in transit and an attacker cannot possibly read, manipulate and communicate data. All the IP traffic with organization is on VPN; a remote worker connects to the organization, and then all its internet traffic is filtered by the organization's proxy.

Basically, this scenario boils down to **securing** integration between **remote locations** that have **internet** connectivity. All devices are under the same administrative domain.

4.2.2 Scenario II — using a VPN provider services

The second, useful, scenario is the one in which personal devices are connected to a VPN that is **managed by a VPN provider**. A VPN provider *sells* a VPN service to users, so that they can connect to any of the servers in the subscription. What happens is that *all* the internet connectivity of the device is routed inside the VPN tunnel, to the VPN provider's servers: the traffic generated by the device will resemble a traffic that **originated by the VPN provider's IP address**. Some degrees of anonimity are guaranteed by the vary fact that **many** users are connected to the same server, and **all** appear as if they were a single one.

A network attacker can no longer intercept traffic between the device and the VPN provider; still, traffic is not more safe when the packets come out of the tunnel. The VPN provider, in fact, highly reduces the set of the possible attacks (the *attack surface*).

A VPN provider usually offers multiple locations for its VPN servers. This is done for many reasons: for instance, a user may want to choose a server close to his location; another one might be interested in unlocking content which is

not available in their own country; another one could simply want to switch to a far away location in order to obfuscate its traffic even more.

A paramount remark is that the VPN provider acts as a **MITM for all traffic**. Choosing a VPN provider means that it must be trustworthy — let alone the fact that they represent a highly–valuable target for high–skill criminal groups, intelligence agencies, and many other potential group that could **target** a VPN provider specifically.

This second scenario represents the use case in which one wants to **secure** an internet connection **irrespectively** of the **physical location**, also hiding its traffic (possible anonymization benefits) behind the IP address of the VPN provider of choice.

Possible ways to employ a commercial VPN services exist depending on the provided software. There could be either **full–device** or **per–app** VPNs, which in turn may be **OS–integrated** or **installed as third–party clients** (for instance, *OpenVPN* is both a third–party software, on Windows, or integrated with OS, on Linux). VPN activation can either be triggered—automatic or manual.

4.2.3 SCENARIO III — ACCESSING RISKY DEVICES

Real organizations possess many devices that both play a *crucial* role for the organization and *cannot be replaced* for costs and time reasons. Those devices use old application protocols with *weak* or *unsecure* defenses, with *several critical vulnerabilities*, not patched for the same above reasons (or, possible end-of-life software, or vendor not willing to patch, or, or…).

The network attacker threat model **is just a model**: what happens in reality is that an attacker **might realistically be in internal networks**. Usually a (former) network attacker has acquired access to at least one of the devices, due to one of the following reasons:

1. *too many networks* and *too many networking devices* to properly defend;
2. difficulty or impossibility to prevent *physical access*;
3. some user can have compromised his *credentials*;
4. the *bring your own device* (BYOD) policy.

The threat model must be upgraded, including the new scenario in which a network attacker has managed to get access to an intranet.

For this reason, one of the most systematic and powerful defenses is to **restructure networks and routing** so that the set of **risky devices** can **only** be accessed

through a VPN. This is a remarkable defense mechanism, since there is no need to modify risky devices while the guaranteed protection is much higher (they are no longer exposed to the network).

This mechanism is only usable *within* organizations. Also remember that this is as secure as the VPN credentials are — if an attacker manages to steal the VPN credentials, the game is over.

4.3 OPENVPN

Among all possible implementations, we will consider **OpenVPN**.

The scenario is as follows:

- *one* remote client (although OpenVPN supports multiple clients at once);
- an OpenVPN server that is running;
- both endpoints are configured properly.

OpenVPN is an *open source* application program, composed of a *server* and of a *client*, that creates and uses *UDP tunnels* for TLS payload (thus, carrying encrypted payloads), with the option of enabling TCP tunnels.

An `IP-CV` (on real address `IP-C`) wants to send a packet to `IP-D` inside the organization. What happens is that the packet `IP-CV, IP-D` is routed inside the tunnel as an *encrypted payload*. The packet traveling in the tunnel has the form

```
IP-C, IP-S | UDP-H | OPENVPN HEADER | TLS(IP-CV, IP-D | payload)
```

and sent into the tunnel. Basically, a UDP packet carries a OpenVPN header, whose payload is the TLS encrypted version of the encapsulated original packet. The UDP packet is sent through the internet via an ordinary packet `IP-C, IP-S`, so that it can reach the VPN server. The latter will inspect the packet and open it, freeing the payload inside the organization's network.

Although the OpenVPN protocol has not been specified in any Request For Comments, it is very used and reliable; it can manage multiple independent tunnels, and TLS is implemented with *OpenSSL* library.

An important remark is that application layer, TCP layer and IP layer **need not to know anything** about the VPN; the only difference will be that the application layer communicates through `IP-CV` (virtual) instead of `IP-C`.

4.3.1 VPN establishment

In order to establish a connection with a running OpenVPN server, four steps are necessary:

1. first, one has to obtain the IP address `IP-S` of the server. In order to do that, the client *must already know* the server's address;
2. then, a TCP connection to `IP-S` should be opened;
3. *user authentication* takes place: for instance, password on HTTPS might be a possibility;
4. an **UDP** tunnel must be created:
   (a) obtain `IP-CV`, subnet mask, and `IP-GW-V` (both virtual private addresses);
   (b) obtain IP routing table configuration (which of the network numbers shall be routed in the tunnel);
   (c) configure local applications (for instance, which applications should use `IP-CV`, some of them or all of them?).

Establishing a VPN connection means the tunnel is then ready to work. A remote worker connecting to an internal server (`www.x.com` at IP `IP-WS`) will do that through the VPN. To find `IP-WS` a DNS query to the organization's `IP-NS` should be performed in the tunnel:

```
IP-CV, IP-S | UDP-H | OpenVPN-H | TLS("www.x.com A ?")
```

Routers, however, should be configured so that the response to the above request is routed to the tunnels as intended.

All an attacker can understand is that *"client `IP-C` is using OpenVPN with `IP-S`"*.

The very same approach can be used to understand what happens on a OpenVPN tunnel in the scenario II (with a VPN provider). All the traffic is routed through the VPN, and can only reach the client back by means of the very same tunnel. After having left the tunnel, the packet sees its encapsulation removed and it passes through a configured NAT module which translates the packet with the correct addresses:

```
IP-CV, IP-X | port-s, port-d | payload
IP-R, IP-X | port-sE, port-d
```

The NAT module is mandatory to distinguish between the various clients. In fact, it is impossible either for a private address to travel on the internet or for a VPN to distinguish between the various clients without a NAT.

The very same path will be traveled back by all the packets going to the client: through the NAT, then to the OpenVPN server, then into the tunnel and finally to the client. Again, all an attacker can understand is that *"client `IP-C` is using OpenVPN with `IP-S`"*.

The last question is *does OpenVPN work across NAT*?

In order to answer that, simply remind that the OpenVPN is based on UDP or TCP, therefore **it works just like any other protocol**: NAT modules will modify the "external capsule", leaving the encrypted payload intact. For this reason, no matter how many NAT modules — if the path client–server is practicable, the VPN will work.

Client–side, there is no need to configure NAT modules, while on server side static port mapping is required for functioning. The default port numbers, identical on both sides, are `UDP/1194-1194`. Various differences arise from other configurations; for instance, different port numbers could be set, TCP might be used. Another configuration is the TCP + random client + 443 server + no OpenVPN header, which actually *looks like HTTPS traffic*. This is a kind of configuration that takes shape of allowed traffic, and is particularly useful in all scenarios in which a firewall might be blocking OpenVPN traffic.

4.4 OTHER IMPLEMENTATIONS

Other VPN implementations are **L2TP** and **PPTP** which are, respectively, based on **IPsec tunnel** and **PPP tunnel** technology. The latter, developed by Microsoft, should never ever be used, since it is considered unsafe.

L2TP implementations exist both as application programs and natively supported be operating system. The basic idea of *IPsec* is that the IP packet in payload is *directly contained* in another, encrypted, IP packet — this way, IPsec is in a way "leaner" than OpenVPN, which in turn requires a TCP or UDP packet to travel to destination.

The key practical problems of IPsec–based VPNs is that it should be configured properly both in the client and in the organization's server and machines (especially in NAT and in firewall).

Within the tunnel, there are strong security guarantees such as in all examples above — the attacker only knows the IP addresses of tunnel endpoints, the VPN protocol, and possibly the *volume* of the traffic (this last knowledge is only enforceable by almost–omniscent adversaries, such as intelligence agencies, big CDN providers and some powerful organizations). An attacker does not know the content of the encrypted payload (the packet that lies inside), and can only DoS, if on–path.

4.4.1 SOME PRACTICAL RISKS

VPNs are indeed a powerful and systematic way to defend against *network attackers*. Some risks do exist, though. For instance, **before tunnel establishment** there may be no strong security guarantee. A network attacker, when connecting from an unsafe location, might be able to intercept the traffic and act as MITM. An attacker that manages to attack the DNS interaction could be able to impersonate the VPN server, steal credentials and read and manipulate all traffic.

The real defense against the above issue is to perform **certificate pinning** (that is, installing the VPN server's certificate *on the client*), hardcode the IP address of the VPN server directly in VPN client, and **Strict Transport Security**.

Another practical risks involve **phishing** attacks, in which an attacker sends fraudulent links to fraudulent VPN services. In those cases, the user should remain vigilant and not fall prey of those baits.

Also, VPN software might suffer from **vulnerabilities**. Vulnerabilities in VPN software are a real, practical problem that can be used by an attacker to surpass the VPN protections. VPN software can be vulnerable on the server side too, with attackers capable of executing exploits, neutralizing VPN security guarantees and entering the organization. In this case, the defensive tools itself allowed access to an attacker.

Chapter 5

HTTP Overview

Chapter 6

Web Security

The widely used **HTTPS** protocol means **HTTP over TLS**. Based on HTTP, it uses TLS to provide warranties of *secrecy*, *integrity* and *server authentication* against a network attacker that can observe, communicate, and act as MITM. HTTPS is adopted by more than 80% of the websites, and it is almost–mandatory for applications as well. HTTPS is a pillar of the web security against network attackers. A remarkable aspect to consider is that the HTTPS protocol is **useless** against a *Man In the Browser* attacker — that way, the protocol doesn't guarantee anything; in fact HTTPS protects only the communication against a network attacker.

HTTPS protects against **sidejacking** attacks. Sidejacking attacks are attacks in which a victim authenticated against a server W on HTTP sees its authentication cookies stolen by an attacker that is able to observe — the cookie grant the attacker the possibility to impersonate the victim on server W.

Extremely simplifying, TLS is implemented in this way:

1. first, a client asks for a `DNS-NAME` to a DNS server;
2. obtained the IP address, it contacts the server, and *asks for the certificate of relative DNS-NAME*;
3. the server replies with its own public certificate.
4. the client can validate this certificate by looking at the digital signature provided by a trusted authority;
5. the client sends a *session key* encrypted with the public key of the server;
6. only the server is able to decrypt it — communication holds with this key.

A first *intrinsic* vulnerability of HTTPS is the special case of "sidejacking" in which the victim *visits an attacker–controlled URL while logged on W*. That way, an attacker could impersonate the victim *on real website* by stealing authentication cookies.

The attack works as follows. An authenticated HTTPS user is pushed to visit an attacker–controlled URL, in which a remote resource with an `http` URL is called. The victim's browser will stil send **cookies in clear** with HTTP, de facto making possible to intercept them and use them. What happens is basically that an attacker's website can force the victim to fetch a HTTP resource on W, in order to steal the cookie in clear by observing the victim generated traffic.

The solution to this kind of vulnerability lies in **secure cookies**. Secure cookies are a special kind of cookies that **can never be sent on HTTP** (basically HTTPS–only cookies). Not implementing secure cookies greatly lowers the defenses.

6.1 HTTPS AND PROXIES

A **proxy** is a server that filters out HTTP traffic generated by a client. Usually, a client first connects to the proxy and *requests* a certain web resources. The proxy will then fetch it in place of the client, examining it, and then delivering it to the client. Numerous kinds of filterings can be applied depending on scenario. A proxy basically *forwards* HTTP messages of a client, while two TCP connections are opened – one from client's browser to the proxy, and the other one from the proxy to the requested remote service.

Then there's the case of the HTTPS proxy. An HTTPS proxy **cannot** read HTTPS traffic — that means that the proxy can only **forward** traffic by creating the two TCP streams at its ends; not understanding anything that passes through it.

A proxy cannot understand HTTPS traffic: to do so, it would be able to decrypt the traffic between the browser and the website, but that would be impossible:

- the browser will not connect to a web service if it cannot authenticate it — trusted certificates do not allow the proxy to give its own in place of the website's one;
- a proxy cannot read the traffic if the browser validated it and the key exchange occurred in private, that is when the proxy forwarded the valid server certificate to the client's browser.

Basically, if the proxy forwards the server certificate it cannot read the traffic due to HTTPS security guarantees — on contrary, if it blocks the certificate and sends its own, the browser will never accept it and will not keep the connection.

6.2 POSSIBLE ATTACKS DESPITE HTTPS

A network attacker could only observe, steal non–secure cookies and perform a Denial of Service attack. However, it can also **perform other kinds of attack** that will, basically, *circumvent* all the infallible HTTPS bells n' whistles.

In fact, two kinds of well–crafted attacks can occur:

1. the **malicious HTTP redirection** to the only apparently correct HTTPS website;
2. the **Rogue Certificate for target website Attack**; victim will witness a perfect HTTPS copy of the **exact** URL he wanted to visit;

The first kind of attack has two variants. The first variant is when an attacker incercepts the HTTP traffic and *redirects* the victim to its own server. The attacker then is able to act as MITM, with a complete copy of the real website allowing full manipulation. The victim will see HTTP in their URL.

The second variant of the HTTP redirection is when an attacker redirects to a HTTPS domain owned by him, whose URL is very similar to the legit one and whose content is a copy of the correct one. The user is unable to spot differences of that malicious copy, never realizing that the domain is different (unless tech–savy).

6.3 REDIRECTING A HTTP CONNECTION

Most HTTPS websites can also be reached with HTTP. This isn't a worrying fact per se; what happens is that usually a *redirection* occurs, and the browser is gently redirected to the HTTPS version of the page.

In the cases where a redirection has not been implemented yet, an attacker will be able to see, control and manipulate a connection. Suppose that happens on `esse3.units.it`; two kinds of attack can occur this way:

- in the first scenario, an attacker registers a **similar** URL, such as `esse3--units.it`, obtains a legitimate certificate for that domain, and hijacks the existing HTTP connection so that the browser won't redirect to the legit domain, but to the attacker controlled domain. In this case the

browser will display a valid HTTPS connection, but with another domain — the attacker will act as MITM, in the sense that it will "copy" the content of the legit website;

- in the second scenario, an attacker doesn't register a new URL, but simply hijacks the connection, acting as MITM. What happens here is that the browser redirects to an attacker controlled domain *in HTTP* and the attacker acts as MITM by copying the content of the original website. These are said to be **SSLStrip** attacks, a form of "downgrade" attack in which the SSL version is downgraded (or SSL is removed) in order to prevent some security measures and act as MITM.

These attacks possess very huge potential impact. An attacker will be able to *record everything*, *manipulate everything*, and *modify requests and responses*. The attack is very cheap: only a *proxy* with some plugins or coding, plus the ability to forward requests to real website are all that's needed to perform this attack.

A general lesson on security is that even though the website implements HTTPS, a **victim with a risky internet connection** and a **victim that does not verify the URL bar** is damned — attackers will focus on the *weakest* point in security.

What to do to defend to redirection attacks, then? Removing the support for port 80 is not enough — a user is simply required to *follow* a HTTP URL: the connection will happen despite the real server would never have accepted it.

The first solution is to *always look at the landing URL*; a solution which isn't always fine, it's hard to apply systematically and it's hard to be applied by everyone too.

The second solution, a technical one, is to enable the option *HTTPS by default* in the browser. If in the past it connected to a website with HTTPS, the browser will remember this and automatically connect with HTTPS even though the user follows a HTTP URL.

The third solution is to enable the HTTP header **Strict Transport Security**.

Strict Transport Security (HSTS) is a *policy* adopted by a server and a browser, and exchanged via the HTTP header in which a *max-age of HSTS Policy* is specified, with a maximum period of 1 year. The HSTS Policy states that whenever a new HTTP or HTTPS connection should be established to the web service, **it must be done on HTTPS**.

Whenever a new connection is performed, the policy is refreshed and the time interval starts again from zero.

Another important policy is the **Content Security Policy**. The CSP's `upgrade-insecure-requests` instructs a user agent to treat all of a site's insecure URLs, served over HTTP, as though they have been replaced with secure versions, served over HTTPS. The policy is valid only *for links in the page transported by that response*, and *only for that specific response* — it has no persistent effect overtime. Differently to HSTS, it doesn't last overtime and it happens only when a page is transported by that response (it depends on the referrer). HSTS works even across different websites.

With HSTS the *vulnerability window* is **dramatically reduces** — a browser is vulnerable only if it has never contacted the server or if the policy had expired before any new contact with the server. A proper value of `max-age` should then be properly set.

A final remark is the **HSTS Preloading** practice. Some modern browsers are **preloaded** with a list of common-HSTS policies. Those websites are virtually immune to SSLStrip — however, this is not scalable.

6.3.1 How to become a MITM for HTTP and HTTPS connections

There are many ready to use tools to become MITM. The tools implement some attacks that are capable of forcing a victim to connect to an attacker-controlled IP address, on port 80.

The first method is by means of **DNS Spoofing**. DNS Spoofing is a particular kind of attack in which an attacker spoofs a DNS-Response, substituting the legit IP address with an attacker-controlled one; for instance, the DNS request `name a ?` can be spoofed to `name A IP-A`, so that the victim will connect to that IP address. To be able to spoof you should observe the UDP traffic on port 53, and *respond before the name server*. The attack is easy, since the protocol is UDP and not TCP - however, client port number and query ID should be copied as well, while the checksum should be changed accordingly.

A method for performing DNS Spoofing is the **Evil Twin** method, in which an attacker impersonates a Wireless Access Point with the *same name* as the target WiFi network. The Evil Twin will also provide DHCP and DNS servers - the victim connects over the strongest (in signal) WiFi network, and obtains the IP configuration from the attacker: the DNS server will now be *the one on the Evil Twin*. Hence, DNS Spoofing can happen. Evil Twins work only against victims that do not have a proper WiFi configuration. The "proper" WiFi configuration

means that the network should authenticate to the client as well, by means of a shared secret.

A second method for DNS Spoofing is the **DHCP-offer-IPv6** attack method on Windows machines. Windows machines allow an external entity to send a DNS configuration to it; since Windows privileges IPv6 over IPv4, the new configuration will be accepted.

A third method for DNS Spoofing is the **DNS cache poisoning**. DNS cache poisoning is the act of entering false information into a DNS cache, so that DNS queries return an incorrect response and users are directed to the wrong websites. A DNS resolver will save responses to IP address queries for a certain amount of time. In this way, the resolver can respond to future queries much more quickly, without needing to communicate with the many servers involved in the typical DNS resolution process. DNS resolvers save responses in their cache for as long as the designated time to live (TTL) associated with that IP address allows them to. DNS cache poisoning is performed by sending to a DNS resolver two requests:

1. the first one, a request for a certain target domain name;
2. the second one, the (fake) response for the target domain.

Instead of using TCP, which requires both communicating parties to perform a 'handshake' to initiate communication and verify the identity of the devices, DNS requests and responses use UDP, or the User Datagram Protocol. With UDP, there is no guarantee that a connection is open, that the recipient is ready to receive, or that the sender is who they say they are. UDP is vulnerable to forging for this reason – an attacker can send a message via UDP and pretend it's a response from a legitimate server by forging the header data.

If a DNS resolver receives a forged response, it accepts and caches the data uncritically because there is no way to verify if the information is accurate and comes from a legitimate source. DNS was created in the early days of the Internet, when the only parties connected to it were universities and research centers. There was no reason to expect that anyone would try to spread fake DNS information.

Despite these major points of vulnerability in the DNS caching process, DNS poisoning attacks are not easy. Because the DNS resolver does actually query the authoritative nameserver, attackers have only a few milliseconds to send the fake reply before the real reply from the authoritative nameserver arrives.

There are also many, many other ways to achieve the condition of MITM (corrupt sys–admins, vulnerability in networking hardware, Intellicence Agencies, misconfigurations, and so on…). Moreover, it could suffice to be MITM only for *certain* messages in order to perform an attack. When reasoning on security protocols, an attacker should be assumed to possess unlimited storage capabilities, unlimited process capabilities and be able to observe, modify and drop any message at any time.

6.3.2 HANDLING HTTP — A SERVER'S POINT OF VIEW

Servers must choose among various behaviors for a key practical problem: *how to respond to HTTP requests?*

The first approach is to respond with HTTP content as well — this is *wrong*, because if a content can be served over HTTPS, it should be served over that. The second approach is to respond with a *redirection* to HTTPS. This is the most pragmatic and practical approach; not really secure as the communication could still be intercepted. A HSTS policy could be served as well. The third — and most secure — approach is to *not respond* (and do not even listen).

The second approach is the most practical one, and almost the most secure one – HSTS makes the client vulnerable to **only** the first request. Subsequent requests — provided the user didn't change browser, clean the HSTS policies along with browsing data, and visited the website again in a year – will be protected by HTTPS. Despite that, HSTS is used by only 23.5% of all the websites; a clear **misalignment of incentives** since all the costs are on the server, but the risks are on the client.

FURTHER EFFECTS AND HSTS ATTACKS

An attacker could intercept a HTTP request, and voluntarily respond with inserted HSTS policies to cause a Denial of Service. Secondarily, an attacker could intercept any HTTP request and removing the HSTS policy.

6.4 UPGRADING YOUR WEBSITE TO HTTPS

Suppose you have a large and complex website served over HTTP and you want to migrate to HTTPS. In theory it is simple: just replace any absolute URL having `http` string with `https` string, and configure the web server so that every HTTP request is redirected to HTTPS.

In practice, this is very hard and complex. Big web services are usually a *collection* of different websites, produced and maintained by different actors, teams, usually with different needs and objectives. What looks easy is rather an obnoxious task of replacing all URLs and configuring the service properly.

Moreover, what about content produced by JavaScript code?

The most useful pratical solution consists in two steps:

1. configure the web server for HTTPS redirection;
2. enable HTTP Strict Transport Security.

It won't matter if some pages are still reachable from HTTP links — the browser will issue only HTTPS requests. There is no need then to inspect and modify any content, with just an initial vulnerability window.

A potential problem still persists. If any resource cannot be served over HTTPS, it will **never be fetched by any browser**. It is really hard then to assure that the HTTPS redirection won't affect the working of many important pages — a long testing phase is required.

A curious way to solve this is to enable for a certain amount of time the **Content Security Policy Report Only** Policy. In header, `Content-Security--Policy-Report-Only: URL-pattern URL-contact` will be delivered to a browser. The policy stats that should a browser encounter a site not compliant with the `URL-pattern` (for instance, not working on HTTPS) still fetch that URL, but send a note at `URL-Contact`. The `URL-contact` is a web server prepared to receive HTTP requests carrying JSON data, while `URL--pattern` could be `https`.

This way, a service could start logging CSPRO notes to build a map of the HTTPS broken webpages. This way, the website configuration could be fixed.

6.5 THE CHAIN OF TRUST

A **certificate** is a text with a proper format in which an *issuer* Certification Authority states that a *subject* domain has a public key `KPUB`, along with details such as hash algorithm of digital signature, valid from, valid through, parameters, alternate name and so on.

The key concept behind certificates is the concept of **certificate chain**. A certificate chain is the tree relationship between a certificate and another, in the case where the first one is adopted to issue the second one. The *root certificate*

is a certificate that is *self-signed*[1] by a Certification Authority, an entity who has been assigned the task of issuing certificates, and is at the root of the dependency tree of certificates. A root certificate is assumed to be true by the web browsers by means of a trusted certificates list, that it is said to be the *TrustSet*.

Suppose a Certification Authority `CA1` issues a certificate for `Server-Name`. The following table shows the situation,

Name	KPUB	Subject	Issuer	Signature
`CERT-1`	`KCA1`	`CA1`	`CA1`	Self-signed
`CERT-2`	`KS`	`Server-Name`	`CA1`	Verified by `CA1`

Table 6.1: First example of small certificate chain. The *root* certificate is `CERT-1`, while `CERT-2` has been derived from and verified by the first one.

The above certificate chain is then `CERT-1`, `CERT-2`, with the `CERT-1` that is said to be a **trust anchor** for the certificate `CERT-2`.

Real website almost always are based on certificate chains composed of more than 2 certificates. Usually, everything but the trust anchor is transmitted over the internet.

Two kinds of certificates exist — the first one we already encountered is the **normal certificate**. Normal certificates describe the associations between a subject (usually a domain) and their public key, and offer no guarantees about assertions by the subject. The second one is the **delegation certificate**, which grants powerful rights such as the fact that now the subject *can act as a Certification Authority*. This means that everyone possessing a delegation certificate will be trusted as a Certification Authority by browsers and all devices trusting the entire chain trust.

Possessors of delegation certificates are also able to create new delegation certificates as well — potentially in an endless chain. As a rule, any certificate must be signed by delegation certificates. An example of chain of trust could be the following one,

6.6 ROGUE CERTIFICATES

A scary category of attacks can be performed the moment an attacker collects a **rogue certificate**. Rogue certificates are delegation certificates which are not

[1] Remember: anyone can generate a self-signed certificate for **any** subject.

Name	KPUB	Subject	Issuer	Signature	Type
1	KCA	DigiCert	DigiCert	Self–signed	Delegation
2	KT	TERENA SSL CA3	DigiCert	Signed by DigiCert	Delegation
3	KU	www2.units.it	TERENA	Signed by TERENA	Normal

Table 6.2: A certificate chain. All certificates must be signed by a delegation certificate, except the first one, the *trust anchor*, which is self–signed.

issued to a legitimate Certification Authority, but to the attacker's organization. Attackers have been able to deceipt the certification authority, allowing to bypass the normal checks and get (or steal) a delegation certificate.

If an attacker managed to obtain a rogue certificate, then *complete copy and full manipulation, along with correct HTTPS link to exact URL* will be possible. This is because by issuing rogue normal certificates signed with a rogue delegation certificate the browser will trust them, even for malicious or attacker–controlled domains. Moreover, this issue does not involve just browsers, but can affect applications, mobile and IoT devices, and so on.

The attack works by sending a *different* chain, but considered trusted by the victim. The browser will not suspect anything, and the user too (in order to spot this, a user should be aware of the correct chain of trust by inspecting the certificate and remembering the correct chain). A MITM attacker can now interact with the user by means of a *perfect* copy on correct HTTPS URL — thus, he can observe and modify all the traffic.

In recent years, due to structural flaws in the HTTPS certificate system, certificates and issuing CAs have proven vulnerable to compromise and manipulation. Attacker operates on the Certification Authority, by exploiting *procedural errors* or by means of *cyber attacks* on it. Moreover, an Intelligence Agency can require a delegation certificate from a CA on some countries.

An attacker could also operate *on victim*, by means of injecting a rogue certificate CA–R either in KeySet or in TrustSet. After that, a MITM attack can be easily performed, as the attacker is only required to generate an on–the–fly keypair and certificate C for a target website, and serve it to the victim. The victim will consider it trusted — since C has been generated with CA–R.

Operating on victim is easier for two reasons,

1. the website need not to be selected in advance, as an attacker can generate C for any web service;
2. the victim will be less powerful than a Certification Authority;

despite that, a successfull attack to a CA has its own advantage, for even though a website should be selected in advance, attacks on **any** client with that installed trust set can be performed.

A third category of entities that can act as MITM as if they possess a delegation certificates are **companies that possess some SSL Interception Proxies** with the capabilities of delegation certificates. Companies producing such Interception Proxies own delegation certificates, so all their proxies can use them to incercept HTTPS traffic for certain websites.

6.6.1 THE CERTIFICATE TRANSPARENCY

The **Certificate Transparency** refers to the fact that *Logging Authorities* maintain publicly available logs of certificates issued by the major CAs. Every CA must log certificates in at least 2 public logs — every subject should periodically monitor those logs.

If an attacker manages to obtain a certificate for a certain domain–subject, then the subject has the opportunity to know it, since everything is recorded in public logs.

Certificate Transparency only allows detection, it doesn't act as a prevention tool (damage is likely already done) and addresses only "real" Certification Authorities — a MITM that generates certificates on–the–fly won't be detected this way.

Moreover, *certificate revocation* is complex and takes time; subjects have additional tasks to monitor frequently those logs, although a third (trusted?) party could be delegated to do such task. And finally, *log integrity* should be guaranteed somehow.

*"A defense mechanism need not to be **perfect** — its necessary property is that its usage will cost **more to the attacker** than to the defender."*

6.6.2 MECHANISMS TO PREVENT ROGUE CERTIFICATES

The first technique to actually prevent rogue certificates attacks is the **certificate pinning**. Certificate pinning is the practice to *pre–install* the *full* chain of trust directly in the client's software. The client will also be instructed to **not**

trust any other chain for a specific service name. Ultimately, any MITM with a different chain that is valid at the client will not be able to impersonate the service name.

The major conceptual change is that certificates should be transmitted over an *additional, trusted channel*, independent of the first one. Certificate pinning is impractical for browsers (except *Google Chrome* with pinning for *Google* services) as there are simply too many services that should have their certificate pinned; regardless, the practice is very powerful for *PC and smartphone applications*, as they only need to interact with a *specific* subset of domains which can be easily pinned.

All certificate should *expire* after a certain date, as prudent engineering practices enforce. Expired certificates are said to be **revoked**. Suppose certifiate pinning has been employed for a specific application, and the service has been switching to a newer certificate (the older one had been expired); how can clients be managed?

An idea would be to make sure that **all** copies of clients are updated before the expire date — those not updated will remain disconnected. Basically, the last certificate is transmitted over HTTPS by means of the old certificate.

Moreover, suppose the private key of a certificate has been compromised: a subject should be prepared for changing the keypair together with the certificate *at any moment*. A naiive solution would be to use the same update mechanism (with the main certificate) to deliver a new one to clients – but this is a mistake, since attackers have compromised the keypair. What has to be done is to deliver updates through a *different* keypair and certificate, with software update authentication made with different keypairs. Software updates should be *digitally signed* with a public key that was meant to be used *for updates only*. As the main certificate revocation occurs, a new software update delivering the new certificate should be handed over by means of this secondary, update, channel. Usually, this channel is performed by a **software update organization** (Microsoft, Google, Apple, Linux distributions) that is able to manage this crucial channel. The client software should know the public key of the update channel in order to fetch updates, and must accept only if signed with such key (by means of certificate pinning).

Problems don't end up here, as the software update organization should be able to handle two more issues:

1. how can the software update organization possibly authenticate the update request from a certain subject?

2. how can the software update organization revoke its update public key safely?

Basically, certificate pinning is a very powerful weapon against rogue certificates, although it has its own drawbacks. For instance, a subject should implement it carefully in order to not leave clients permanently disconnected, and prepare for certificate revocation at any moment before expiration. Revocation should also be handled by means of a different — also powered by certificate pinning — software update channel.

6.7 BRIEF SUMMARY OF HTTPS ATTACKS

Attacks can be summarized into two, main, categories:

- the first one being a **user following an HTTP link**, with a MITM attacker providing a complete copy of the real site, either with an inserted *HTTP link* (SSLStrip) or with an inserted HTTPS with *slightly different URL*. This category of attacks can be mitigated by either *verifying landing URL*, or *enabling HTTPS by default*, or *enabling Strict-Transport-Security*;
- the second one being the **MITM with Rogue Certificates**, with an attacker having managed to either *obtain a delegation certificate* or to *inject a rogue certificate in the KeySet or TrustSet of the victim*, with the user that will see *HTTPS link with exact URL*. These attacks are detected by means of *Certificate Transparency* and are mitigated by *Certificate Pinning* and the building of a proper software update mechanism.

The aforementioned techniques work less with a network attacker that can only observe and communicate (man-on-the-side), as it can only inject fake responses for redirection in place of providing an exact copy of the web service with malicious modifications. As the HTTPS provides security guarantees against a network attacker that is not able to perform attacks on organizations or clients, **HTTPS should be adopted widely, as soon as possible**. The key attack point, now and in the future, will be that between organizations and browsers. Moreover, there could be *vulnerabilities* in TLS protocol, TLS implementation or Web Server HTTPS implementation.

Chapter 7

Passwords

Passwords are used both in clients and in servers for authentication. In principle, they should be stored somewhere. In practice, they all reside in the so-called **Authentication Databases** (`AuthDBs`). Authentication Databases are present in both clients and servers, and collect all informations regarding logins (usernames, passwords) on a machine. Authentication Databases can also manage credentials across the entire organization, as happens with the Active Directory domain controller for Windows machines.

The core concept of *authentication* is that each registered user must prove to the machine that *he knows a certain secret* — for instance, a password. Typically, such proof knowledge can be either transferred locally (everything happens on the same machine) or remotely (through the network or the internetwork), either in the same or in different organizations. Location of the AuthDB depends on the application; be it a web site, a file server, a mail server, a printer, an API server and so on. AuthDBs located on servers typically store credentials for more than one user. Since an AuthDB contains sensitive information, a very concrete risk is that *it is stolen by an attacker*; an event that occurs very frequently. A stolen AuthDB has the potential to give up to the attackers the credentials of all the included users.

A best practice for avoiding this scenario is to **store non-reversible functions** of the password `pwd`, and not `pwd` in plain text. Authentication for a user works by first letting him send `user, pwd`, and then computing `F(pwd)` and checking the presence of `user, F(pwd)` in the AuthDB. Since only the user knows the password, the secret is safe against the stealing of credentials from the AuthDB.

Any attacker that wants to collect real secrets for the users is now forced to apply `F()` to a *list of candidates passwords* and see if any of the results matches any entry in the stolen AuthDB. This practice is called **offline guessing**. Offline guessing is called in such way to distinguish it from **online guessing**, an attack practice that will be covered later in this chapter.

The non-reversible function largely depends on operating systems: for instance, on Windows it is collected in a `SAM` file, while on Linux secrets are collected on `/etc/shadow` file. Android uses analogous mechanisms as well. When served over the network (usually by means of an authentication protocol over TCP), it depends on the server implementation. Not infrequently, passwords are stored "in the clear", in plain text as many servers have their default settings to store cleartext passwords. Across organizations, users would like to use the same credentials everywhere; hence, keeping all AuthDB synchronized is close to impossible. For this reason, the **Single Sign-On** technique is adopted, as we will see in a later moment.

7.1 PASSWORD ATTACKS

Password attacks pose a huge threat to organizations, so that password-based authentication is no longer enough, and **two-factor authentication** should be used in its place.

Passwords can have a huge *value*, that it can be:

- *direct*, let's think of banking or e-mail services;
- they might allow *credential stuffing*, with many users using the same password across many services — this amplifies the value of a credential;
- they might let *organization multistep attacks*, with the so-called "lateral movement phase" and even adoption of spear phishing;
- they might be *sold on the black market*;
- and so on.

Password attacks are usually **not targeted** — passwords of any user across an organization are enough. Password attacks are indeed dangerous threats that could affect virtually anyone, no matter the technical skills of the person in exam.

The thread model involves many steps further than the classic network attacker threat model. In particulare, to steal passwords an attacker is required (and supposedly able) to:

- install *malware* on devices;
- perform *phishing* attacks or campaings;
- perform *spear phishing* attacks;
- achieve *physical proximity* so that a password could be collected (Insider threat model);
- of course, *steal the AuthDB* of an organization;
- and if the authentication protocol has no security-integrity-confidentiality guarantees, a network attacker that can observe is all what's needed to obtain the credentials.

Let's focus on the threat posed by *stealing an organization's AuthDB*. An attacker has now free and unlimited access to the AuthDB, but he should be able to determine passwords from applying `F()` to *a set of candidate passwords* and perform a search in the database. No limit on speed (except hardware) and number of attempts should be considered. This is said to be **offline guessing**.

Differently, a network attacker that is able to communicate will be also able to perform **online guessing**, an attack that is carried out against a "dumb" server with no detection capabilities. Perhaps a server is not configured or able to detect multiple (possibly in the order of thousands) of login attemps; hence, online guessing attacks are feasible. The online guessing attack consists in trying to use each password in a candidate set *against the real server*, hoping to find the correct one. There is a *tight limit* on speed and number of attempts. These attacks still succeed, because some passwords are **very** weak, or are left to their — publicly known – defaults. Online guessing is "equivalent" to the offline guessing, only against the users involved.

An important remark is on **password strenght**. Password strenght becomes relevant **only** if the attacker is performing guessing attacks (offline or online guessing); otherwise, with malware, phishing and Insiders the password choice becomes irrelevant as long as it is stolen with other means other than guessing.

As a matter of fact, a harder password is only *harder to guess*, not *harder to steal*.

7.1.1 HOW GUESSING ATTACKS ARE PERFORMED

OFFLINE GUESSINGS

Offline guessing attacks involve two steps:

1. the first, a number of **likely passwords** are attempted. This step is performed with the help of *dictionaries* of common passwords, predictable patterns, passwords from previous breaches, default passwords and word lists and the so-called *mangling rules*, the set of common "likely alterations" of words in passwords (append or prepend a special character, replace o with 0, and so on);
2. the second, **all permutations** of symbols belonging to a certain alphabet (for instance, English alphabet with the addition of commonly used non-word characters), up to a *certain length*. Basically, a *systematic exploration on string space* is performed.

The last step is *very less likely* to yield correct passwords (exhaustive search of solutions), therefore is *much more expensive* and less cost-efficient for an attacker to perform. Hence, **an attacker is a lot more likely to stick with the first step only**, never executing the second step; or executing it with a low length (for instance, strings of 6 or 7 chars).

As a rule of thumb, a password is more important not to be in any dictionary or past breach than "complex", since the first attempts will likely be performed on breached or commonly-used passwords.

Mangling rules play little to no role on making a password stronger. Automatic tools can produce very quickly a huge range of mangling rules for any common password, making mangling attempts useless.

AuthDB stealing is one of the most important risks across organizations and across users. Authentication databases records `< username, C >` are implemented with 4 "general" approaches, from worst to best:

1. *in clear* — the worst of the worst, consists in storing the password in plain text along with the username. An attacker would immediately obtain passwords for all users;
2. *encrypted* — a little better, but still useless. Storing `< user. Ek(pwd) >` is uneffective the moment `Ek` is compromised (usually along with an attack). Both the previous storing mechanisms are not to be used at all;
3. *hashed* — the bare minimum requirement, passwords are stored after having hashed them by a unidirectional function. The server should not

know the password, and only store `< user, H(pwd) >`. When server receives `pwd`, it immediately computes `F(pwd)` and checks against the database entry;

4. *hashed and salted* — the best way to store credentials, it involves the previous step *plus* the addition of a "salt" to decrease the likelihood of an attacker to possess a pre-computed table of hashes to speed up password guessing. By altering the credentials in an unpredictable way, salting renders table attacks with pre-computed tables unfeasible, as the memory required for computing every possible combination of salts on passwords would be too much. Such attacks are called **lookup table attacks**.

An attacker cannot directly obtain `x` from `H(x)`. Attacker can only perform offline guessing with a candidate password set, with aforementioned techniques. Offline guessing attacks are generally performed with specific *cracking tools*, possibly with *cracking hardware*.

The method generally works as follows:

- for each candidate password, compute the hash;
- for each user and password record in the Authentication Database, check if the computed hash matches any hash present in the table.

Since the overall process is very costly, an attacker will rarely try to find all permutations in symbol set, and will rather focus on the likely passwords. If an attacker wanted to excute an exhausting search on symbol set, *any 8 characters password will be easily broken in a few days*. Passwords should *at least* be 10 chars long. Every 8 chars password can be bruteforced in a single day as of 2019.

HASH FUNCTION REQUIREMENTS

A *hash function* has basically three requirements:

- `H(x)` **must not provide any information** on `x`;
- `H(x)` is of **constant length** (for instance, 128, 256 bit);
- `H(x)` is **computationally heavy**.

Ultimately, it should be only crackable by bruteforce attacks, and they should be hard to perform.

The state of the art regarding hash functions are **PBKDF2** and **bcrypt**, and they have computational weight parameters that are proper quantities (iterations

for PBKDF2, rounds for bcrypt). *NTLM*, a very weak hash function used in Windows is millions of times weaker than the above two.

Hash function algorithms, and cryptographic algorithms in general are *public* for scrutiny by cryptography experts. Generally, the rule is *"do not roll your own crypto"*.

LOOKUP TABLE ATTACKS

Lookup table attacks are offline guessing attacks in which *the dictionary is known in advance*. Basically, hashes are pre-computed so that a *simple search* can be performed, and are valid for any AuthDB that uses the corresponding hash function. Lookup tables require a lot of storage, but are available on the internet in cracking tools.

The protection against lookup tables is **salting**. Salting consists in picking the input of the hash function — that is the password — and adding a *random number or symbol* to it, at any position. The new "password" will then be the password itself *plus* the salt; its corresponding hash will be stored in the AuthDB together with the salt that should be added to compute the hash. A given password at different users is associated with different random numbers. Basically, the lookup table **cannot** be pre-computed anymore, since the random numbers would make the task practically unfeasible.

All the server is required to do is to first retrieve the salt from the AuthDB corresponding to a user, then concatenate the password with the salt and computing the hash function. A further check is performed to confirm the match. An attacker, instead, cannot compute in advance a lookup table, since it is unfeasible.

Moreover, salting prevents *password frequency analysis*. Password frequency analysis is a technique that involves an attacker looking for frequent passwords by simply looking at the AuthDB — many users have the same passwords, probably in a dictionary. With salting, hashes are unpredictable, thus frequency analysis of passwords is not possible anymore.

On Windows, the non-reversible function produces hashes that are not salted, with a lightweight hash function fast to compute. Even worse, Windows authentication protocols require the knowledge of `H(pwd)` (a knowledge that it is sufficient to have to impersonate a user — the hash is even stored in the machine's memory; basically, if AuthDB is stolen, no guessing is required to impersonate anyone).

On Android or Linux the non-reversible function makes use of salts, and it is very heavyweight. On Linux, the file in which it is stored is the `/etc/shadow` file.

A very general and powerful approach in security is the **defense in depth** approach. It consists in multiple, **independent** layers of defense, which should result in a *higher increased cost for the attack than for the defender*.

ONLINE GUESSING

Now let's talk of *online guessing* attacks. Online guessing attacks are different, since a very low number of attempts can be performed — either because the network introduces some lag (an invalicable limit) or because there could be a chance to be detected by the organization to which the attack is performed.

Online guessing attacks are usually executed by means of *botnets* when outside organization, or by a set of clients or machines when inside an organization. Usually, online guessing attacks are not cheap — for this reason, an online guessing attack will focus on common passwords, predictable patterns, default passwords and passwords from previous breaches **only**. If an access has not been found, then the most rational behavior would be to change service or webapp.

The first technique is the **credential stuffing**. For each username, try a few thousand passwords — basically, for every user multiple passwords are tried in a row. Since this can easily be detected by a defender (its log will contain thousands of failed login attempts on single users), another technique is preferred: the **credential spraying**. Credential spraying is different in the sense that the two for loops are now reversed: for each password, try it on many usernames. Defender's log will now present a less-obvious pattern of login attempts. Typically, there are up to 50 passwords sprayed across usernames, resulting in up to 20000 attempts per hour, each following a hard-to-predict pattern (2019 data — around 0.5% of all accounts get compromised each month, with nearly all hacked accounts on *legacy protocols*).

As a general remark, the realistic threat model for organizations is to **assume breach** inside the organization, with possibily many credentials that have been compromised.

Against online guessing the most important defense is *detection*; relatively simple with stuffing, more complex with spraying; depends on guessing rate and traffic usually occurring across an organization.

The possible action against online guessing attacks are the following ones:

1. *automatic username lockout* — a wrong practice, since it will then be trivial to perform a Denial of Service attack on a target;
2. *blacklist IP addresses* involved in guessing attacks — a weak action, since it's easily circumvented, with a lot of potential false positives and users damage more than attackers;
3. *alert toward targeted usernames* — only feasible for stuffing;
4. enforcing **time throttling** depending on failed attempts, with a threshold--based system having a progressive increase of response delay — a really effective technique that prevents most of guessing attacks. Despite being really useful, this mechanism is rarely implemented for servers.

NETWORK ATTACKERS AND PASSWORD GUESSING

A network attacker observing traffic in a network is completely equivalent to an offline guesser, against the observed users with hashed and salted credentials. This means that one should assume a network attacker to be able to perform *offline guessing attacks* against observed targets, by analyzing and collecting observed traffic.

Let an Authentication Protocol traffic travel over an encrypted channel (examples of this are BASIC/FORM over HTTPS, SMTP/POP over TLS, MSCHAPv2 over TLS, and so on); guessing should not be feasible due to encryption — however, it might still be able to guess a easy-enough credentials, since modern cracking tools handle most authentication protocols. The possible number of hash per second depends on hardware, with GPU clusters and with weak hash functions such as NTLM we are in the order of millions guesses per second.

7.1.2 HOW TO CHOOSE A PASSWORD

The key objective when choosing a password is that **it must resist guessing attacks**. The first and most important requirement for any password is **not to be in any dictionary** — not a common or predictable pattern, not a word, not a mangled common word. Second, **passwords should be unique for each service** — an extremely important remark, since a compromised credential could lead to compromising even more services. The password, ultimately, **must be sufficiently long and complex**.

To put it briefly, a password should be **at least 14 characters long**, with special symbols that matter not particularly. This last requirement is the most

fundamental one, since all attacks will use dictionaries and perhaps only some of them will also perform an exhaustive search on permutations space. Moreover, passwords used on multiple services pose a serious threat to security for aforementioned reasons.

A practical and very effective way to generate passwords is to *concatenate two or three uncorrelated long words, possibly in different languages*, an easy-to--remember technique which makes computations too high in cost to be sustained.

The best practice is still the use of a **password manager**, which is able to generate, store and apply credentials directly on devices.

Additional requirements such as *change password* or similar are even harmful, since forcing password expiry carries no real benefit, while the user is pushed to cheat on the password generation mechanism by choosing weaker passwords. Also *complexity requirements* do not bring any benefit against guessing attacks, only placing extra burden to uers, many of whom will seek for predictable patterns that are easy for them to manage and to remember. The only requirement should be in password length.

THE "SECURITY VS USABILITY TRADE-OFF"

Generally speaking, the security will bring down usability, and viceversa, in an unavoidable trade-off. Pushing users against the security side will **inevitably** affect usability. When forced "to the ropes" to maintain very high security levels, users will find out ways to manage their security (in this case, their passwords) so that the usability increases and reaches acceptable levels for them — de facto, loweing the security. A good sweet-spot should be achieved by the security manager, by producing security policies that take the user behavior into account.

That said, system administrators **can** carry the security burden, and thus they **must** change their passwords frequently.

Chapter 8

Multi–Factor Authentication

Multi-Factor Authentication is an *additional identity proof* that should be given alongside with the main credentials, username and password. Multi-Factor Authentication is indeed very effective against realistic threat models — enabling 2FA is a very high priority defensive investment, because it literally saves you from credentials stealing and *significantly increases the cost for attackers*. MFA is optional for common web services, mandatory for banking services.

The scenario is the one in which a user *authenticates* with username and password to a service by means of an authentication protocol, either in the same or across different organizations. Basically, if the authentication mechanism involves **a second factor**, even if an attacker knows both username and passwords he will not be able to access the user profile; the attacker will be neutralized "almost completely" (attacks on 2FA cost too much to be sustainable, unless directly targeted user);

The second factor could either be a *smartphone*, with (in increasing order of security) an OPT SMS, an OTP authenticator app, push notifications, or a *security key* having USB, NFC or Bluetooth capabilities. The most secure option is the security key one, but all 2FA methods are generally enormously more secure than password alone.

Also *smartcards* and *security tokens* (generators of one-time-passwords) are available, which are similar to a security key, but less flexible.

All *roaming authenticators* will connect to the device which is connecting to a service — the roaming authenticator then will also connect with the remove service by itself, with a complex protocol. All protocols are standard protocols.

8.1 2FA IN PRACTICE

Two-Factor Authentication might be enabled by linking the second factor to the service during an authenticated session, a process that generally is necessary *once*. The 2FA-login should be performed this way: the user browser or device authenticates with the server, and the latter asks for a *challenge* — a knowledge or a token that can only be retrieved by the second factor. The second factor *confirms* the browser login, either on the same channel or on different ones (that's the case of smartphone-2FA).

Second factor is *not always* required: some services are able to identify a previously logged-in browser and skip the second factor requirement, as the browser is considered a *trusted device* by the service.

8.1.1 ONE-TIME-PASSWORD FUNCTIONALITY

The OTP works like this: a browser authenticates itself to a service, and then the latter asks for a further *challenge*. After a waiting period, the user will read the OTP code, insert it in the browser (as asked by the service), and it will send it to the service — if the OTP code is the one that is expected, the authentication will be a success.

Usually, OTP codes are 6-digits long with uniform distribution, valid for **a single** login attempt, and known *only* to the smartphone of the user. They can be sent by *SMS* if the service chooses to send the OTP to the user smartphone (to check whether or not he is the real owner of the service) or be determined by an **OTP Authenticator** (that is an app installed on the smartphone of the user). Each installation generates a unique sequence OTP(t), and the service knows OTP(t) for that installation.

The user, only once and at registration, has to enable 2FA, choose and install the appropriate authentication application and link his account at service with the application on the smartphone. At every login, the user has to provide username, password, witness the "insert OTP" field, open the authenticator app, read the proper OTP for the service he is connecting to, and provide the OTP.

Usually a service, if it supports an authenticator app, it supports many of them through standard protocols.

Remember: there is *no direct communication* between the OTP authenticator and the server; the code is provided through the browser. Usually the protocol involved in transmission of a second factor is the `FIDO2` protocol.

8.1.2 OTP: AUTHAPP IMPLEMENTATION, TIME-BASED OTP

The first requirement of an AuthApp implementation is the *linking*. A private key `K1` is *generated by service* and *securely sent to AuthApp* upon activation — the so-called linking process. The Authentication Application is usually linked to *several* services, unless the application is service-related. For instance, an Authentication Application will contain the pairs `U`, `K` related to various services, which private keys were generated by the service and securely sent to the Authentication App.

One of the many ways in which this happens is by means of the following procedure:

1. user's browser successfully authenticates to a service;
2. the service sends a link `URL-U-tmp`, usually by means of showing a QR code that should be captured by the smartphone on which the AuthApp is running;
3. the smartphone follows the link and opens a secure TLS connection with `URL-U-tmp`, a proof that the owner of the device is the same person that surfs through that browser;
4. at this very moment, the service generates `K1` and privately sends it to the smartphone by means of the TLS encrypted channel;
5. both smartphone and service will know shared secret `K1`.

The mechanism with which those endpoints agree on the OTP is the **Time--Based OTP**. Time-Based OTP comes from the idea that the OTP should be *time-dependent* in a predictable way if one knows secret `K1`, and unpredictable if one doesn't know the secret. Basically, the formula is as follows,

$$OTP(t) = Enc_K(t - t_0),$$

where t is the current time, t_0 is the conventional zero time (same for both parties) and K is the shared secret (the private key) that both parties (and only them) own. Basically, the result is the encryption of $t - t_0$, a result that after

a normalization yields a 6-digits number. The encryption function is usually a HMAC function.

OTP is generated at both parties independently. This means that the AuthApp shows a code that the user should enter in the browser as prompted by the service, and the service will then check the OTP based on the very same algorithm that generated it on the smartphone application, with the same shared secret K.

Possible issues of this mechanisms are the following ones:

- clocks cannot be perfectly synchronized;
- messages do have latency;

hence, the above formula cannot reliably work! The solution is to apply a sort of *module operation*, in which

$$OTP(t) = Enc_K((t - t_0)\%DX),$$

with $DX = 30s$ — this means that the OTP changes once every $30s$ (and not at each and every second), and there is a $30s$ time window to insert the OTP code before it expires.

The algorithm is actually much more complex than it looks.

The TOTP is a powerful solution against any attacker that only knows *username* and *password*. To become a threat to TOTPs defense mechanisms, an attacker should also become a MITM. The main issue is that **OTPs are fishable**: an attacker could simply act as MITM between the service and the user and the service. If he also knows username and password, then he is able to intercept the OTP message that the user sends to the service, de facto allowing to steal it and to connect to the service in user's behalf. Automated and configurable tools exist for this sole purpose.

Another powerful threat model against which the OTPs are not effective is the **vishing** (Voice phishing) attacker. An attacker practicing *vishing* talks to the user and convinces him–her to be the legit website organization. If the attacker is prepared, determined and possesses enough information on the victim, then the attacker can be successful.

Despite the above limitations, *OTP makes phishing much more costly to attackers*.

8.1.3 SMS-based OTP vs AuthApp-based OTP

SMS-based OTP involves the service sending a SMS to the smartphone, containing the OTP the service expects.

However, this doesn't guarantee security against those threats:

- *malware* on smartphone (can read, forward, delete SMS);
- *SIM swap* (a fraudulent SIM change — phone number fraudulently taken by another SIM);
- *SMS routing attacks* involving weaknesses of *SS7 phone protocol* and resulting in the SMS sent to the attacker.

Differently, the AuthApp OTPs do not suffer from the previous issues, except from malwares that are still able to determine the OTP despite the AuthApp prevents to perform screenshots.

Moreover, in SMS-based mechanisms the service needs to know the user's phone number, while on AuthApp-based techniques the service needs not know the user's phone number.

Regardless of the adopted MFA method, the enhanced authentication provides a huge advantage against attackers as **the cost of the attacks increases dramatically**.

8.1.4 Security Keys

Security keys are indeed the most powerful 2FA method. Widely adopted security keys are the *Google Titan* and the *Yubico Security Key*.

Upon login, the security key must communicate with the Browser, usually by means of *near communication means* such as USB, Bluetooth and NFC. The security key flashes to notify the user that a login is in progress, and expects the user to press a button on the security key. A browser can never login to a service as long as the security key is far from it.

During a login, the common procedure for browsers still occurs, with the username and passwords credentials sent to the service. After that, the service immediately **asks to sign a challenge**. The challenge can only be successfully signed by the private key `Kpriv-X` stored *inside* the security key. The key signs the challenge and the browser sends it back to the service, which verifies the signature. The very same procedure can occur for a smartphone browser. The protocols are,

- `WebAuthn` for the browser with the service;
- `CTAP` for the browser with the security key (for instance, via Bluetooth, NFC or USB);
- `FIDO2` for the overall protocol.

Linking a security key involves creating records in the security key of the format `DNS NAME, username, prv key, pub key`, with a single security key associated with *many* services. **Only** the security key knows the private keys of the services. An authenticated browser session (with username and password) to service `S` exists. The security key generates the pair `prv key, pub key` to be used only with `S`; the browser then sends the public key `pub key` to `S` by means of the already existing authenticated and secure channel. The service will associate `pub key` with the corresponding username.

Since the keypair is completely generated inside the security key, **full decoupling** from real entities is guaranteed.

Login occurs in the way that had been presented before. The challenge is sent along with *username* and *public key* (so that the key can actually verify the correspondence), and the challenge is *signed with the private key stored in the security key*. The service verifies the challenge has been correctly encrypted by using the public key.

Some more things should be provided before the security key works. For instance, the browser **must be authenticated with HTTPS and with real domain**, otherwise the key will refuse to encrypt the challenge. The model is broken only in the case of false certificate (and if the attacker already knows the public key — that is an attacker that already existed at the beginning). Thus, the security key protects against *real-time phishing* (DNS name mismatch, phishing does not work as the user is completely bypassed).

Other security guarantees provided by a security key are the following ones,

- if the attacker has physical access to the security key, the *cloning* procedure must be very difficult — in particular, extracting set of service names must be as unfeasible as possible;
- a service must be able to know who the Manufacturer of a security key is, and the product ID that the key holds — this is because an attacker might be the Manufacturer itself, so that the trust to certain security key may be *revoked*;

- the Set of Services might collude to link respective user identities — the Service cannot *identify* which specific security key the user is using for login process.

8.1.5 SMARTCARDS

Smartcards are based on the very same principle of security keys, with a similar defensive strength, but they all *require a dedicated device* to interact with them (a smartcard reader) and the set of user identities is *static*, defined at creation time. Smartcards are usually adopted to legally bind identity proof in some countries, or in certain intra-organization settings.

The unsolved threat models are the *attacker that owns a valid certificate for a service* and the *attacker that has malware on the Browser's device*.

8.1.6 PUSH NOTIFICATIONS

Smartphones can be used as security keys. In particular, if the service supports it, a smartphone can be used to receive a push notification, informing the user of a tentative login. The user might accept or not. The smartphone should be able to communicate with the service — no smartphone near the user, no login from that user. Push notifications work very similarly to security key, in the sense that `DNS NAME`, `username`, `prv key`, `pub key` are stored inside the device. The implementation is *almost identical* to that of the security keys, with the only difference that now the user should press a button in the smartphone to allow the login.

As the smartphone receives the push notification, a "side" HTTPS authentication to the service is performed. The service sends a challenge to be signed, along with username and public key stored in the smartphone, so that it can be checked. The smartphone signs the challenge, to be verified by the service. The only "relevant" differences between this mechanism and the security key technique is that in this case a push notification is sent, and an independent HTTPS connection is established.

Push notification system is still *vulnerabile to real-time phishing*. An attacker could act as MITM, and as the attacker logs in into the service, the user is asked to press the button — the attacker will login at user's behalf. Push notifications describe which device or browser is logging in, often in a not easy to understand way for an end user. Basically, this system does not bypass the user and therefore it doesn't solve phishing and vishing.

8.2 The concept of trusted devices

From certain devices, a special handling of 2FA is managed by a service. For example, a service might consider **trusted** some devices, not asking them for 2FA again. The terminology of what a "trusted device" is not uniform, and many, many different scenarios exist. For instance, there are Google Services for Android phones or Chromebooks, Microsoft Services for Windows devices, `FIDO2` standards for any service and device in general.

The typical scenario is the one in which a user owns several devices, and the user has activated 2FA on service `S`. As the user performs a 2FA-login from a device, the service declares that device *trusted*: second factor is *no longer required for login*, and the user will only need the classic "username and password" from that device. The service might decide autonomously to require a second factor again, for instance if the trusted device is not trusted anymore. That may occur in the cases where the device connects from an anomalous geographic location, or when it has been declared trusted a too long time ago (or, with other heuristics and security policies). Obviously, the device should not public or shared!

A trusted devices stores a *cryptographic token* which is agreed by it and the service. Basically, this is very similar to app linking for push notifications.

8.2.1 Passwordless login

Passwordless login is possible in the case that *a trusted device* has a "built-in authenticator" — a fingerprint reader, a face recognition mechanism — from which the device can perform authentication based on *biometrics*. In that case, the device "guarantees" for the user: the user **authenticates with the trusted device**, then the device authenticates with the service, allowing user to access `S`. This is the common scenario with banking apps. The user *must have told the service to trust the local authenticator* for this to work.

It works as follows:

1. the service asks the trusted device to authenticate the user with the built-in authenticator;
2. the trusted device asks the user to authenticate itself with biometrics;
3. if the authentcation is a success, password is not required and the login proceeds — otherwise, the authentication will require knowledge of password.

8.2.2 THE LOSS OF A SECOND FACTOR

The first solution to the **loss of the second factor** is to *login from a trusted device*. An old second factor can be revoked, and a new one can be defined.

But what if no trusted device is left?

Two-Factor Authentication always allow *exceptional login* with **alternate second factors**. Recovery information should be provided to the service — for instance, a "downgrade" to `OTP --- SMS` or `OTP --- email` might be allowed to avoid account lockout. This is, however, **not** password recovery as they are in addition to password login.

Still, the unsolved threat model is the one in which the attack knows both password **and credentials for exceptional login**. This might be exploited in the cases where a single password is used across various services – especially if the e-mail password is the same one as the password of the service that allows `OTP --- email`. In all cases, a service never allows changing 2FA recovery information without a second factor. It allows protecting against attackers that know only the password.

The most secure option *by far* for reclaim ownership of your account when the second factor is lost are **recovery codes**. Recovery codes are special codes that are generated once and can be used as a special second factor. They can be stored as secrets in password managers: they prefer security against usability in the security–usability trade-off!

Chapter 9

Authentication

Authentication is the process in which an entity authenticates against another, either in the same or in different organizations. Credentials are stored in AuthDBs, and the sets of users and passwords are completely independent of each other.

Usually, authentication is performed following an *authentication protocol* over an *encrypted channel*. Some examples of this are the *BASIC–FORM over HTTPS*, *SMTP–POP over TLS* and *MSCHAPv2 over TLS*. Many protocols do exist — involving file servers, printers, and many other machines.

Authentication provides some *guarantees* to a certain entity, from another entity. For instance, in **client authentication** the server does not provide any proof of its identity; what happens, instead, is that the client provides a proof of its identity by means of a protocol, that could be POP, BASIC–FORM, FTP and for *Active Directory* the Windows **NTLM** protocol. On contrary, in **server authentication** the server itself should authenticate to the client (that's the case of TLS and web services).

A powerful form of authentication is the **mutual authentication**, adopted by *Kerberos*, in which both client and server possess the means to authenticate to each other.

Authentication can occur in three different aspects:

1. by means of something you *know* (a password, a private key);
2. by means of somethin you *have* (a smartphone SIM, ATM card, smartcard);

3. by means of something you *are* (fingerprint, face).

The first will be considered in this chapter (although the entirety of all the three above had been inspected in the previous chapter — since the two-factor authentication involves all of the above three, with the latter only used for local authentication).

The threat model of the authentication is a Network Attacker that can observe, communicate and act as a MITM. Its attack objective is to obtain guessing material from the execution of an authentication protocol, and then performing offline guessing with cracking tool afterwards, on captured material. The overall difficulty is equivalent to an *hashed and salted* database cracking. The end goal is the **impersonation** of one of the two parties in **one execution** of authentication protocol.

9.1 NTLM

The **NTLMv2** (**NT LAN Manager v2**) protocol is a Windows Challenge–Response authentication protocol used on networks that include systems running the Windows operating system, and on stand-alone systems; it considered largely insecure, and should be replaced with *Kerberos* protocol instances.

The NTLM server contains an *AuthDB* possessing the credentials of any user, in the form of `username, H(pwd-user)`. Each application protocol for Windows networks is integrated with NTLM in *very-specific ways*; usually, information at the beginning of a (usually TCP) connection is related to the NTLM protocol, but no further rules can be generally inferred without be more specific.

The NTLM works as in Figure 9.1, and is composed of the following messages:

1. negotiation;
2. server challenge;
3. challenge response, client challenge, username.

The above chain of messages can be found in almost every protocol in which NTLM is included.

A first remark of this NTLM protocol is that **the server does not prove its knowledge of the `H(pwd-user)`** — in practice, the client authentication is completely *unilateral* as the server can actually be impersonated by anyone.

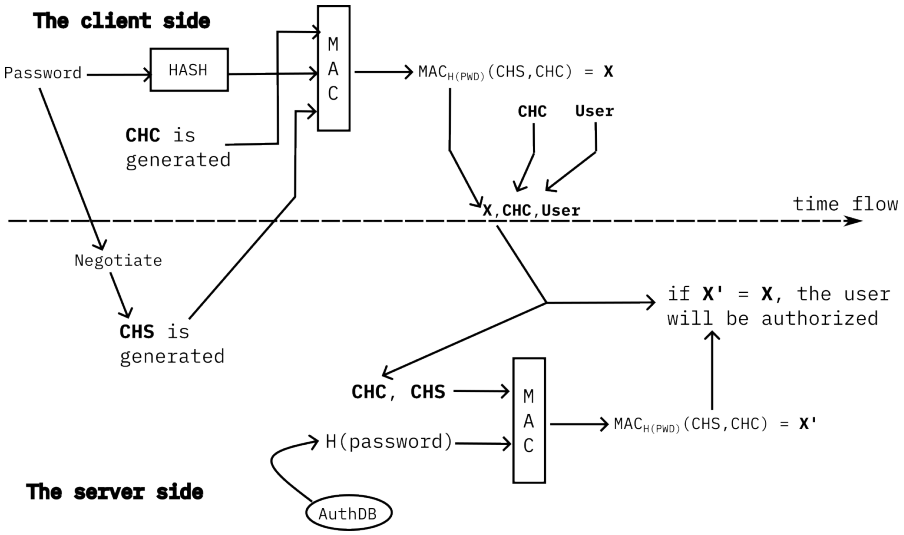


Figure 9.1: The NTLM scheme. A login is first *negotiated* by sending username, after which the server sends a **server challenge** `CHS`. Another challenge, the **client challenge** `CHC` is determined and concatenated to the `CHS` to form a string. The string is then transformed by a MAC() algorithm by means of the key determined with the hash of the user's password. MAC-digest along with username and `CHC` are sent back to the server, that is able to check the integrity of the original message by using the MAC again with the string and with the hash of the password (present in the AuthDB). The **only** security control is in this last phase, where the two strings are checked to be equal.

With the established threat model, an attacker can only execute guessing attacks, as the string `X` cannot be replayed in future executions (`CHS` changes at every execution). The `H(pwd-user)` is **unknown** to the attacker, that only knows the MAC result (`X`), `CHC` and `CHS`. Equivalently to salted database, an attacker cannot compute anything in advance — this means that he must employ a cracking tool and perform an exaurient search in the password space (or, more cleverly, adopt a candidate password set as seen in previous chapters).

Unluckily, the encryption algorithm used in NTLM is very *fast* to compute, all while capturing an NTLM execution is very easy. The consequences are that guessing attacks are actually possible, and basically *unavoidable* in Windows environments. The number of realistic guesses is very high. Windows password have a huge value for attacks, as most organizations implement **Single Sign On (SSO)**. Thus, possessing strong passwords should be a priority in large organizations.

However, on NTLM that is almost *useless*: the client only **demonstrates the knowledge of the hash, that is `H(pwd-user)`**. Basically, **knowledge of the**

hash of a password suffices to impersonate the user, a weakness that renders NTLM very dangerous to use. What happens is that on Windows environments hash knowledge is enough to impersonate an user, no matter how strong the password is: offline guessing can be done with the goal of obtaining `H(pwd-user)` rather than `pwd-user` itself. Moreover, softwares capable of extracting the hashes from a Windows machine exist: when a client is compromised, the user can immediately be impersonated on the network.

The hash of the password is nowadays computed by a function that is called **NetNTLM Hash**: the computation hardness of the function is a lot better than standard NTLM, however still very weak when compared to blowfish or bcrypt (53.9k times weaker).

9.1.1 NTLM AND SECURITY GUARANTEES: AUTHENTICATION IN PRACTICE

NTLM does not provide **any security guarantee after authentication**. Differently from HTTPS (which provides secrecy, integrity and server authentication), there are no guarantees after a client authenticated to a server. The drawback of this is that the communication occurs without encryption or any security guarantee that might be expected from an authentication protocol (that is, no shared secret key `K` is established). Of course, a network attacker can inject application data in open connections, alter the traffic, and observe everything — something that cannot occur in the cases of an authentication procedure that guarantees something more. Weak protocols in this sense are HTTP BASIC or FORM, POP, and **NTLM**. Strong protocols are HTTPS, Personal or Enterprise Wi-Fi, any protocol over SSH (secure shell).

MITM attacks are trivial against NTLM. A MITM attacker can simply watch and collect all material, performing offline guessing on the collected material. More simply, he can perform a **relay attack** in which just after he impersonated the server and the client in a NTLM authentication procedure, he impersonates the client and further communicates with the server. Of course, the attacker should be able to impersonate different protocols at once, so that the attack can work on any service. This all requires the exact timing, and it only works on a single authentication. Basically, **NTLM is weak to relay attacks**.

NTLM can be made much more secure. In fact, NTLM can be upgraded to **NTLM with Signing & Sealing**, in which the server must *prove knowledge of the hash of the password*, negotiate a secret `K` with the client, and after the authentication secrecity, integrity (with the option of server authentication) are

guaranteed. A MITM cannot execute relay attacks (since he does not know K), however the security mechanism is not enabled by default.

9.2 KERBEROS

Kerberos is a computer-network authentication protocol that works on the basis of tickets to allow nodes communicating over a non-secure network to prove their identity to one another in a secure manner. Its designers aimed it primarily towards a client-server model, and it provides mutual authentication—both the user and the server verify each other's identity. Kerberos protocol messages are protected against both eavesdropping and replay attacks.

Kerberos is built on symmetric-key cryptography and requires a trusted third party; optionally it might use public-key cryptography during certain phases of authentication. Kerberos uses *UDP port 88* by default.

Kerberos was initially designed in 1978 by **Needham** and **Schroeder**. In 1981 it had its first attacks, and some fix occurred – in 1987, the first release called *Project Athena* (MIT), and in 1988 the first conference took place. Only in *Windows 2000* Microsoft systems started implementing it for real. History tells that for most crucial protocols a huge amount of time and money should be spent on research, and the results can be perceived only after some years.

9.2.1 PRIVATE KEY CRYPTOGRAPHY

In **private key cryptography**, along with secrecy (encryption) every message exchange contains a *Message Authentication Code* to provide **integrity**. MACs are appended to each message and allow checking for the integrity of the message. If a receiver realises that the digest of the message and the MAC don't match, then the message has been subject of errors or it might have been tampered. For simplicity, MACs will be omitted from now on.

Private key cryptography assures secrecy, mutual authentication and integrity. It resembles **digital signature**, since only the real part is able to use that secure channel (as it owns the specific secret that allow the communication to take place).

9.2.2 A SIMPLE VERSION OF KERBEROS

The first simplified version of Kerberos has the following properties:

1. mutual authentication;

2. key establishment;
3. secrecy, integrity, mutual athentication;
4. uses private key cryptography;
5. enforces short key lifetimes, with frequent rotation.

The architecture is as follows:

- every user `U` has a **secret key** `KU`;
- every server `S` has a **secret key** `KS`;
- each secret key had been derived by a **password**, so that
  - `KU = H(U, pwd-U);`
  - `KS = H(S, pwd-S);`
- an **Authentication Server** AS knows all secrets, as it possesses an AuthDB containing the pairs `username, K-username`;
- the username is used as a **salt**;
- every node has a good, local clock.

Kerberos is particularly suitable for **local** environments, as an AS should exist for all users and servers. This is not scalable, as private key cryptography can't be managed on a large scale.

The first phase of any login is the **interactive logon**. An interactive logon sees a user *locally logging in* a machine (a workstation, a physical terminal of a server). After the local login, the workstation immediately authenticates to the AS — if the operation results successfully, the interactive logon completes correctly, and the user is correctly authenticated (he knows `pwd-U`. The user can use all the services provided on the local workstation.

The next phase is the **network logon**. A network logon involves a user's workstation `U` authenticating to **any** server `S` (he is authorized to perform the login on). The already-authenticated workstation authenticates on user's behalf, and the end result is the user `U` able to use services from server `S`.

In this simplified model, the user workstation `U` interacts with the authentication server AS by sending a message, containing:

- `U`;
- `S` to which a connection should be performed;

- `Eku(t1)`, with `t1` local clock and `ku` **private key** known only by the client;

If `t1` is sufficiently close to the clock of `AS` and the encryption key is the correct one, then the `AS` will respond with a message[1], encrypted with `ku`, containing the **session key** `Kus` and the clock `t1`. Since `ku` had been used, the message is authentic, and since `t1` is recent, the message is also fresh. The session key is a special key constructed by the `AS` whose purpose is to allow interaction between `U` and `S`.

Since the message is encrypted with `ku`, how can the server possibly get `Kus`? The answer is by means of **service tickets** — service tickets are messages containing `Kus` encrypted with the key `ks` belonging to the server. Service tickets are appended to the response message to the client. Service ticket is then sent to the server `S`. A service ticket contains `Kus`, can be read only by the server `S` since it is encrypted with `ks`, and can be only issued by `AS` (since no other entities will know `ks`).

A service ticket is of the form `Eks(Kus, U, S, IP-U, Tas, LIFE, ...)` – it contains `Kus`, but also informations regarding the workstation (user, server, IP of workstation, time of `AS`, lifetime, and many other informations) so that the ticket **can only be used by the legitimate workstation, to submit requests to the legitimate server**. The lifetime is provided (usually in minutes) to prevent *replay attacks*.

During a request to `S`, the workstation `U` must authenticate as well, with an **Authenticator** message encrypted with the session key `Kus`. In order to do so, a message `Ekus(U, IP-U, TU1)` is appended so that the server can properly read informations on workstation (and comparing them to common network inspection). If the authentication succeeded, `S` will send back to the user `Ekus(TU1)`, so that he can check `S` identity and freshness of communication. Basically, both authenticator and ticket must agree on user's name, IP address and timestamp for this to work. The overall scheme is shown in Figure 9.2.

Network attackers in this path cannot succeed. First of all, communication between `U` and `AS` works perfectly, as the entire process runs on encrypted communications (by means of secrets known only by the two parties); secondly, communication between the `AS` and `S` guarantees secrecy, mutual authentication and integrity. The user becomes trusted as long as it cannot tamper `Kus`, because it doesn't know `ks`. An attacker cannot forge messages that should be

[1] Up to version 4, `AS` would respond to messages `U,S` without `Eku(t1)`, thus providing unnecessary material for offline guessing since an attacker could simply ask for thousands of tickets.

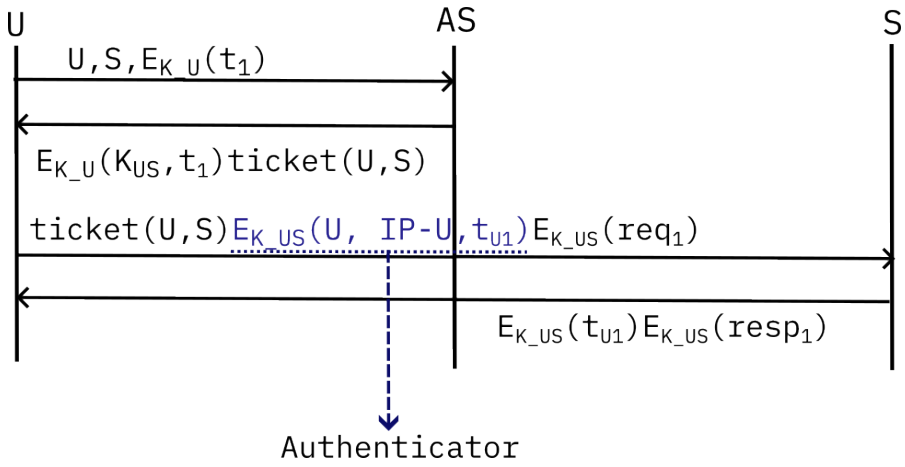


Figure 9.2: A simplified overview of the Kerberos protocol. Letter U denotes 'user', letter S denotes 'service', while AS stands for *Authentication Service*. In blue, the *Authenticator* component of the message is highlighted.

encrypted with Kus, cannot impersonate another user (as U would be different). The only attack left is Denial of Service.

Among a single session (until the ticket expires), all the Kerberos-related messages are always the same, as the ticket can be reused again. The only different things will be timestamps and requests–responses between U and S. Since timestamps will mutate, the Authenticator message will mutate as well.

Kerberos can be seen as a *protocol* for making sure that an **Authentication Token**, authentic and intact across untrusted channels, is presented by its owner and to the legitimate service. With these expedients, Kerberos is able to provide secrecy, integrity and mutual authentication, at the cost of a lesser usability, greater costs and greater administration difficulty.

9.2.3 KERBEROS AND SECURITY PROPERTIES IN DETAIL

IMPROVING THE SIMPLIFIED KERBEROS MODEL TO WITHSTAND REPLAY ATTACKS

A network attacker cannot read, modify or forge messages; however, he can **replay** them:

- replaying *responses* will result in a detection, as the responses are always matched with the corresponding timestamp;
- replaying *requests* will be undetected with the above techniques only.

However, requests to AS pose no problem, since the attacker will receive many copies of the same response (containing the service ticket), but he will never be able to decrypt them — differently, an attacker may want to replay some request to a server S, so that **multiple executions** of a same command can be provoked.

Since the client will insert the local clock in every sent message, the defense is to *compare the clock contained in the message with the local clock*, and then **discard** every message received with `msg.clock - rcv.clock` greater than a certain threshold (before of after the `rcv.clock`). In other words, **given a tolerance** δ every message whose clock is different from the clock of the receiver will be discarded. An attacker could still perform a quick replay and send a lot of requests during the threshold — this is now possible, however, since **the server will store all received message with clock within δ, and discard all copies of the same message**, thus preventing massive spam of replay attacks. This technique is also adopted in many modern protocols, such as *OAuth* and *SPID*.

Kerberos can still be vulnerable to guessing attacks. Guessing can, of course, only be made against Ku and Ks as both are derived from password (plus a salt), and not against Kus.

THE THREE DIFFERENT VERSIONS OF KERBEROS

Kerberos comes in three versions:

1. the **very strong** configuration, providing *private* messages with guarantees of secrecy, integrity and mutual authentication;
2. the **strong** configuration, providing only *safe* messages (integrity and mutual authentication);
3. the **so-so** configuration, in which after the first interaction the connection **is the same as TCP**, therefore it is vulnerable to relay attacks just as default NTLM configuration. Sadly, this is the default configuration when setting up a Kerberos instance.

The first and strongest configuration is the one we have already seen, and it is depicted in Figure 9.3.

The second configuration has **no secrecy in requests**, only providing mutual authentication and integrity by means of *Message Authentication Codes* appended in each message (Figure 9.4).

The third, the weakest of all, provides mutual authentication **only** in the first interaction of the protocol (Figure 9.5).

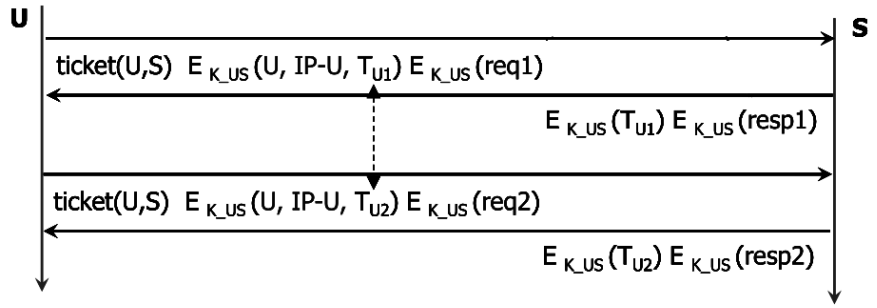


Figure 9.3: The "very strong" Kerberos configuration.

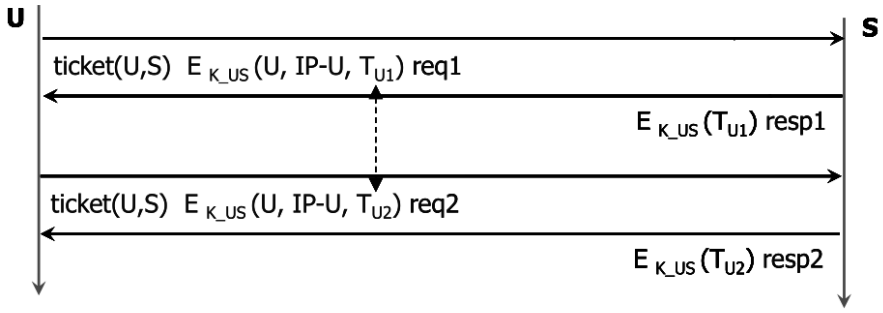


Figure 9.4: The "strong" Kerberos configuration.

TICKET LIFETIME

During a **Kerberos Session** there exists a single ticket. After the ticket expires, a new Kerberos Session should be created and a new ticket should be granted, all by contacting the AS again. The Kerberos Session and the ticket lifetime is *completely independent* of the TCP connection lifetime — a single TCP connection can end up by using *multiple* service tickets, and a single service ticket can be used on *multiple* TCP connections.

The main reason to adopt a limited lifetime is the *Crypto Prudent Engineering Practice*, which states that **every key must be changed every now and then**, a fundamental practice for every practical application in cryptography. There could be many reasons for this, for instance the cryptographic algorithm could have weaknesses exploitable by collecting a lot of traffic, or the key might be found *somehow*, so all the traffic would be decrypted, forever. Thus, the session key that the service ticket grants must be changed every now and then.

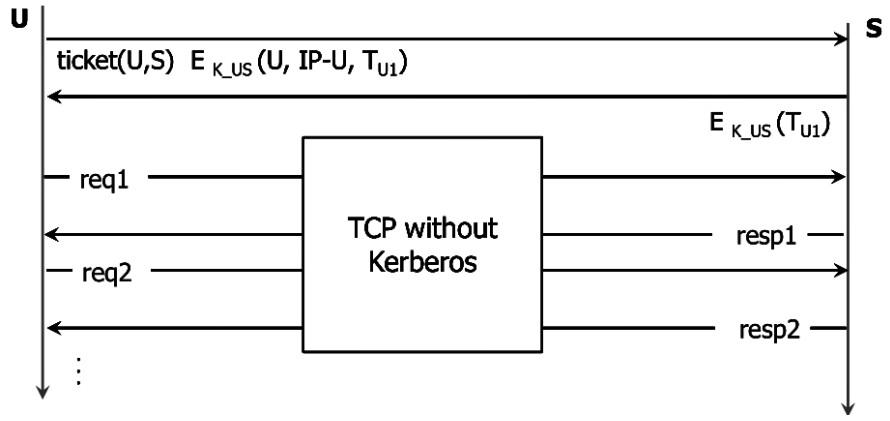


Figure 9.5: The "so-so" Kerberos configuration.

The problem with ticket expiring is that it protects a strong key by increasing the risk on a **weaker** key (Ku), by providing more opportunities for observing traffic (dictionary attack), and more traffic for cryptanalysis. The issue is solved by means of KDC (next section).

9.2.4 KEY DISTRIBUTION CENTER

In order to understand how the real Kerberos works, further modifications should be done.

The **Key Distribution Center** is a server belonging to a Kerberos instance whose goal is to **provide tickets**. A real Kerberos works with two, distinct services (can be run on the same machine — in fact, it runs on port 88 on the same machine): one that **initiates authentication**, and the other one that **provides tickets**. The first one is the AS that had already been depicted in the previous paragraphs, while the KDC role will be explained now.

Suppose a user wants a ticket to communicate with a server S. The user sends a request to the **Ticket Granting Service** (TGS). The AS grants the user a **Ticket Granting Ticket** (TGT), that is a ticket with which the user workstation will be able to communicate with the TGS. After that, the user presents its TGT to the TGS, along with a request for service S. The TGS will respond with the **service ticket** for communucating with service S. To summarise, a first ticket — the *Ticket Granting Ticket* — should be provided by the Ticket Granting Service to request other tickets. Subsequent tickets — the *Service Tickets* — are the same ones we illustrated in the previous sections. Next Figure 9.6 illustrate the whole process in detail.

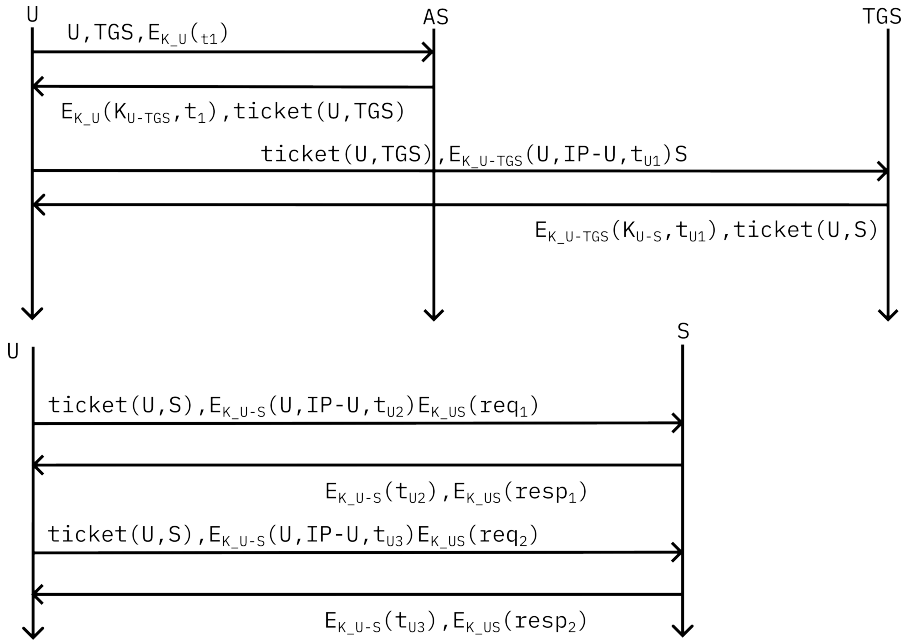


Figure 9.6: The real Kerberos configuration.

Hence, the process can be summarized in 4, distinct steps (two phases):

1. the user U performs an **interactive logon** on the workstation. A concatenation of username and password is passwd through a hash function, which produces Ku;
2. the workstation *proves its knowledge of Ku* with the Authentication Server AS — the latter grants the user's workstation a **ticket granting ticket**, along with session key Ku-tgs. The *interactive logon* phase ends here;
3. the user performs a **network logon** to a service S. The workstation presents the TGT to the TGS, also proving the knowledge of Ku-tgs; the TGS gives it the *session ticket* for the communication between U and S, along with Ku-s;
4. the user completes the network logon by presenting the ticket to S, proving knowledge of Ku-s.

A Ticket Granting Ticket lasts *hours*, and can be used to request new service tickets until it expires. During a single day, some of them might be requested by a workstation whose TGTs expired. A service ticket, instead, lasts *minutes*. This means that *the secret Ku is used only once in hours*, protecting it from the collection of guessing material, while the strong secret (the session key) is rotated multiple times, possibly hundreds of times each day. Basically, the high volume traffic is only performed with strong secrets, protecting the weaker user key.

Some weak points of real Kerberos exist. For instance,

- collecting service tickets is very easy. Service tickets are encrypted with Ks, hence collecting guessing material to attack server password is a realistic threat;
- if weak or default passwords are used this is dangerous;
- even more dangerous in the case of *overprivilege*.

hence, an attacker that has obtained credentials of a single user (very realistic) can then obtain ST(S) for **any** server S by simply asking TGS for tickets (for this reason, servers passwords must be managed **very** carefully); this attack is known as **Kerberoasting**. There are many automated tools for this data collection, and there is no need to observe the traffic (only by communicating and possessing credentials of a user). When the attacker can only observe (and does not possess credentials) the attack is feasible, but not easy. In fact, he must perform the equivalent of an offline guessing attack, with salted pass-

words. This is still feasible as `Kerberos 5` protocol allows computing 13200 times more hashes than `bcrypt` or `blowfish`.

9.3 ACCESS MANAGEMENT IN KERBEROS

Kerberos allows fine-tuning the **permissions** related to **Access Management**. In particular, access management is performed in a *centralized fashion*, with **access right specified on the ticket**. Basically, when a ticket is forged by a `TGS`, the access rights for a user are defined in the ticket (the server trusts them blindly) since the access rights are set up in the KDC. User requesting an operation which is out of his rights will result in a "not authorized" message from the server. Each resource has an Access Control List described in terms of groups of usernames – the ticket specifies which groups the username belongs to.

Kerberos can only work effectively towards a *single organization*. In fact:

- a server in the second organization must be able to verify the authenticity and the integrity of tokens **issued by the first organization**;
- the protocol is not NAT-friendly;
- the access right policies should be mapped between organizations (hard to do due to privacy);
- each organization should trust the other. Perhaps, this is the most crucial aspect.

For the above reasons, Kerberos is practically used only between different parts of a same organization — and not across the internetwork. Protocols that reliably work across the internet are **SAML**, which supports authentication and authorization, often used for authentication only, as in the case of SPID — and **OAuth**, which supports authentication and authorization as well.

Chapter 10

Identity and Access Management

Large organizations involve hundreds of servers, each one storing files and databases, thousands of workstations (either private or shared) and thousands of users (belonging to tens of partially-overlapping groups). Some servers might even be accessible from the outside (POP, HTTP, for example). Users and groups are said to be **identities**, while everything else falls in the category of **resources**. What an identity `U` can do, is to request **operation** `O` on resource `R`.

Every software that manages a resource authenticats the identity that requests an operation, and verifies whether that identity has the access rights (the authorization) for such operation. That should be done in a centralized fashion, with a single `AuthDB` managin all identities and related permissions.

The **Identity and Access Management** are the *procedures* and *technologies* for management of individual *identities*, their *authentication*, *authorization*, *roles* and ultimately their *access rights* **within** or **across** system and enterprise boundaries.

Almost all large organizations use **Windows Active Directory**, along with **LDAP**.

10.1 Identity and Access Management within an Organization

IAM within an organizations is realized thanks to the **Directory Service**. A Directory Service is a *centralized* repository containing all *identities* and their

credentials, all *resources* and all *access rights* associations between identities and resources.

A process of *Single Sign On* can be performed by users, with each resource executing authentication by interacting with `DS`. Access rights are stored on `DS` — therefore, authorization can be queried to the Directory Service.

10.1.1 Windows Active Directory

On **Windows Active Directory**, a single entity — the **Domain Controller** — runs the three single most important services:

- the *Kerberos AS*;
- the *Kerberos TGS*;
- the *Directory Service*;

thus, it runs both Kerberos and the Directory Serivce and should definitely be safeguarded from attackers. Obtaining access to this server usually leads to a complete catastrophe.

Active Directory works wonderfully over Kerberos. The process is the following one:

1. `U` presents credentials to a workstation and performs the first phase of interactive logon;
2. the workstation obtains `TGT(U)` from `DC`;
3. the workstation *verifies* whether `TGT(U)` specifies that *the user `U` can logon on the workstation*;
4. if yes, the workstation stores `TGT(U)` in its memory;
5. the network logon phase starts. The workstation presents `TGT(U)` to the Domain Controller, which in return grants `ST(U,S)`;
6. the workstation presents `ST(U,S)` to server `S`;
7. the server `S` verifies in `ST(U,S)` *which operations* the user `U` can execute on `S`;
8. the user has correct privileges, both workstation and server `S` store `ST(U,S)` in their memory;
9. the server executes `S`.

A third way to logon is the **Remote Logon**. A remote logon occurs when a user wants to connect to a different workstation (not the same one) — it works like in the case of a server:

1. the workstation presents `TGT(U)` to `DC` and obtains `ST(U,WKS')`;
2. the workstation presents `ST(U,WKS')` to `WKS'`;
3. the remote workstation `WKS'` *verifies* in `ST(U,WKS')` which operations *the user `U` can execute on `WKS'`*.
4. if all correct, the workstation and the remote workstation will store `ST(U,WKS')` on their memory.

A fourth way of logon is the **Network Logon with Delegation**. In this scenario, an user `U` authenticates to `S`, and the latter accesses `S'` **on behalf of `U`**. The typical case is when the `S` is a web server, and `S'` is a database server. In this case,

1. the ticket `ST(U,S)` must be a **delegation ticket**, having a special flag denoting such peculiarity;
2. the server `S` presents `ST(U,S)` to `DC`, obtaining `ST(U,S')` — with this ticket, the server `S'` obtains the user's identity;
3. both servers keep `ST(U,S')` in memory.

An important remark is that *everything can be done with NTLM as well*, for compatibiluty reasons. The important aspect is that **credentials such as hashes of user passwords are stored in memory**, always present in workstations — a network attacker can grab oggline guessing material for `H(pwd-U)`, a secret that it allows impersonating the user `U`.

10.1.2 LDAP

Lightweight Directory Access Protocol is a protocol designed for Linux (or different than Windows) machines to interact with an exiting Active Directory configuration.

Some services or web applications run on Linux. In order to interact with users of an existing Active Directory configuration, the Linux machine **asks Directory Service by means of the dedicated LDAP application protocol** if the credentials are valid and what are the proper access rights of an user (logged in with an application-specific authentication protocol, let's say HTTPS, FTP, POP and so on). Basically, the Linux machines authenticates to the `DC` with an *LDAP Authentication*, for instance with *TLS+username-password* (**LDAPS**)

carried on a special LDAP message. Certain LDAP operations require **LDAP authentication** of the client application. After that, the Linux machine is successfully authenticated to the DC, and can then ask for credentials validity or access rights.

LDAP has a double meaning:

1. a **standard** for describing entities of IT interest, such as *users*, *resources* and *access rights*;
2. a **protocol** for querying a directory service (the server that stores those descriptions).

Part II

Cyberattacks

Chapter 11

An introduction to Vulnerabilities

Vulnerabilities are a **mistake in software** that can be directly used to *gain access* to a system or to a network.

The list of known vulnerabilities (CVE, *Common Vulnerabilities and Exposures*) can be found at `https://cve.mitre.org/`.

Vulnerabilities allow a wide range of attacks to succeed, with impacts ranging from better to worse:

1. information disclosure;
2. privilege escalation;
3. code execution of existing operations, or arbitrary code execution.

and are also categorized according to various methods of execution (not exhaustive list):

- physical access needed;
- can be exploited by a malicious running program;
- remote execution with human interaction;
- remote execution without human interaction.

A more general definition is the following one:

"A vulnerability is a flaw or weakness in a computer system, in its security procedures or internal controls, which could be exploited to violate the system security policy. (NIST SP 800-28 Version 2)"

Typical examples of vulnerabilities are:

- over-privileged user accounts;
- oversized administrator groups;
- bad practice in workstation administration;
- bad practice in password management;
- excessively loose firewall policies;
- and so on…

In a nutshell, vulnerabilities are software or configuration–procedural **mistakes** that allow an attacker to violate the security policies in effect.

Vulnerabilities are a *growing trend* (we are not in the range of 20000-per-year), and they are only the publicly known vulnerabilities. There are, however, the so-called **zero-day** vulnerabilities which are known to the attacker only, and unknown to both software manufacturer and defender.

Every kind of software can be affected by vulnerabilities. Vulnerabilities are an intrinsic feature of software — not a sporadic, occasional phenomenon. In practice, those categories can be affected:

- end user software;
- network devices;
- security software;
- software implementing security protocols;
- Industrial Control Systems (ICS);
- Medical Devices;
- IoT devices (Cars, Ships, home devices);
- and so on…

11.1 The Common Vulnerability Enumeration procedure

The **Common Vulnerability Enumeration** is a procedure for assigning a *unique ID* to every known vulnerability. The first step is the *submission* of the vulnerability details to a certain consortium (for instance, `https://www.cve.org`; *analysis* will follow by

- assignment of a new ID for the vulnerability, in format `CVE-YYYY-NNNN...`;
- merging of the vulnerability with an existing one, if already existing.

CVE is the standard de facto nowadays, and all the organizations and documents will refer to a certain vulnerability by the same ID, its CVE. This is of a huge importance in order to avoid any confusion.

Along with the CVE, the **impact** (or **risk evaluation**) of a vulnerability is provided, so that a defender can understand the danger of the vulnerability in question. Moreover, **reason** can be provided as well (does the attacker induce a crash? does the attacker induce an endless loop? does the device has a too small amount of resources? can the attacker consume all resources easily?)

Parallel to CVEs, there exist **Common Weakness Enumeration** (CWE). CWEs are a *list of types of mistakes*, constructed *once and for all* (periodically revisited). Some of these mistakes in the list include

- weak encryption;
- improper validation of syntactic correctness in input;
- improper certificate validation;
- improper access control;
- improper...

Each CVE might be associated with one or more **CWE**; this is very useful in order to grasp the underlying reasons quickly.

Chapter 12

Attacks

12.1 The reasons behind attacks

Attacks are most frequently done to **obtain money**. The first and foremost reason will always be to make money as easiest as possible. Some attackers, however, might be motivated enough to performing an attack to **steal informations** or **distrupting the operations** of a target organization.

Attacks are a **professional activity**, with criminal groups of tens of people, hierarchically structured, and teams that are able to update malwares (for instance, the Conti Group updated their malwares every 4 hours, in order to avoid detection from *Windows Defender*, so that signature is different). To obtain money, many creative ways do exist:

- steal and use *banking credentials*;
- steal and sell *credentials*;
- steal and sell *long term cookies*;
- employ *Remote Access Trojans* to infect a device, and sell or rent it;
- and so on.

Usually, the victim is **not** aware of what happened, except in the case of **ransomwares** in which the victim is asked to pay an amount of money in order to unlock the files on his computer that had been encrypted by the attacker. Another kind of malware *steals* data instead of encrypting it, asking ransom for not making it public.

With all the above attacks, *the attack cost is very low*, while the *potential Return on Investment (ROI) is very high)*: this means that a lot of potential attackers are incentivized to perform attacks to obtain money. On top of this, anonymous payments worldwide allowed attackers to ask for ransoms anonimously, while the importance of data for organizations has grown exponentially in the last years. Ultimately, this is a huge societal problem.

Three distinct target categories might exist:

- organizations — servers and data;
- Industrial Control Systems — sensors and actuators;
- single individuals;

each one possessing their own attackers and their own crucial aspects.

The term Organizations covers a whole range of entities, such as Hospitals, administration of manufacturing companies, universities, and any other entity that possesses servers and owns data. An organization might or might not possess an ICS infrastructure — thus, with organization we mean the administration, logistics, payroll, sales–purchasing, warehouse, email–web (and so on) part, while with ICS one explicitly suggests the sensors and actuators infrastructure in a manufacturing or control process.

Organizations are often under attack of criminal groups. Criminal groups usually possess high skill and a great amount of resources, that very often exceeds those of the target organizations. Crime groups will typically attack organizations in order to get huge amounts of money with the least possible effort — a rational behavior.

A lot of money can be obtained by attacking

- single organizations — asking for huge ransoms;
- many individuals — asking for small ransoms;

while attacks on ICSs are frequently justified by other motivations (for instance, disruption and information disclosure). In fact, attacks to ICS rarely grant money. Very often, attacks to single individuals are designed to affect a very broad range of devices, to maximize the possible targets. Each individual will likely yield a small amount of money, but the very wide set of victims will provide a large amount of money, enough to justify the efforts. Ultimately, attacking a category of devices is far simpler than attacking powerful organizations.

12.2 ATTACKING AN ORGANIZATION

Attacking an organizations usually is a many-steps process: several phases follow one after the other, each of several steps, that can last from minutes to months. To describe attack phases, several *models* have been built; the most famous two are the **Kill chain** model, the first widely used, and the **MITRE ATT&CK** model, the one that represents today's standard.

The MITRE ATT&CK framework is used to describe an attack, and is composed of 13 possible phases, called *Tactics* — some tactics are optional for an attacker, so that an attack can succeed without seeing all of the tactics involved. Tactics are also not necessarily executed in sequence.

Each tactic comprises several *Techniques*, that are the ways for executing each step of a tactic.

The MITRE ATT&CK framework involves three *Matrices* in which all the tactics and techniques are exposed[1].

The role of vulnerabilities is relevant in all tactics that involves penetrating security breaches; that are *Initial Access*, *Execution*, *Lateral Movement* and *Privilege Escalation*.

12.2.1 INITIAL ACCESS

The **Initial Access** tactic is the one that is almost always observed: an attacker overcomes the first defenses of an organization. Several techniques exist:

Drive-by Compromise A user visits a website over the normal course of browsing, and a *vulnerability* is exploited

Exploit Public-Facing Application A *vulnerability exploitation* is performed against an internet-facing machine (be it a server or a workstation)

Phishing *Malicious attachments* or links in e-mails lead to a vulnerability exploitation

Valid Accounts *Compromised credentials* are used to enter the organization

Other 5 techniques in MITRE ATT&CK framework

A remark is that the techniques either exploit an existing vulnerability or get to deceipt a real human.

[1] https://attack.mitre.org/matrices/enterprise/

An important cornerstone of initial access is **phishing**. Phishing is a still widely used and very effective attack technique, which is of two forms,

- **non-targeted**, when adversaries conduct malware spam campaigns or other non-targeted phishing attempts, with even low success rates leading to huge gains (cost-effective);
- **targeted**, in the case of **spearphishing**: a specific individual, company or industry is object of targeting by the adversary. Defending against spearphishing is **very difficult**, and it usually follows the phase of *Reconnaissance*.

12.2.2 EXECUTION

Execution techniques result in an *adversary-controlled* code, running within the organizations. Twelve techniques exist for this. Attacker's code is usually arbitrary code, in the sense that the device is now completely controlled by the attacker and can respond at will.

12.2.3 PERSISTENCE

Persistence should be granted across *system restarts*, *changed credentials* and *interruptions* that could cut off the attacker's access. Nineteen techniques exist. Persistence is necessary in order to keep the infected machines as such — attacks might require weeks of efforts, so an attacker should carefully design its exploits so that they will also grant persistence across interuptions or configuration changes.

12.2.4 ESTABLISHING COMMAND & CONTROL

Command & Control are a set of techniques that adversaries may use to *communicate with systems under their control*, within a victim network. Adversaries commonly attempt to *mimick normal, expected traffic to avoid detection* — usually by means of existing protocols, but new "unseen" protocols might be created by an attacker. The attacker will also *obfuscate his location* to avoid detection from the defenders or other security organizations.

An example of C&C structure is the **DNS Tunneling** technique, in which an attacker controlled device encodes a *request message* to attacker in a DNS query, for instance `dfg99872gh.innocent.com A?` — whose CNAME response will be let's say `hhjsd67.innocent.com`, encoding a response message from

the attacker. This way, a very hard to detect channel can be established. Sixteen more techniques exist.

12.2.5 DISCOVERY

Discovery are a set of techniques in which an attacker *gains knowledge* about the internal environment of an organizations and decides how to act accordingly. The goal of discovery is to learn of networks, hosts, devices, applications, but also users, groups, access rights. Twentynine techniques exist.

A simple and lightweight program for discovery is **`nmap`** (Network Mapper), an open source tool for *network exploration* and *security auditing*. The tool `nmap` is capable to scan either large networks or single hosts — it uses raw IP packets in novel ways, to determine:

- what *hosts* are available on the network;
- the *services* that are running;
- the *Operating Systems* and their version the hosts are running;
- the kind of *firewalls* in use;
- and many more.

Usually, `nmap` is quite noisy, but with certain flags it can be configured to be hardly detectable. Of course, many more programs exist for this purpose.

12.2.6 LATERAL MOVEMENT

Lateral movement is the phase in which an attacker, after having obtained access to a single point internal to the organization, performs attacks and operation in which tries to access new devices, usually in the same privilege sphere that the already attacked device lies; applying then Execution and Persistence on other hosts. Lateral movement, indeed, is much easier when the attacker had obtained the privileges of an administrator. There are 9 techniques for lateral movement in MITRE ATT&CK framework, under the tactic *Lateral Movement*.

Usually, the lateral movement phase comes after the attacker gained foothold during the first phase and after having guaranteed **persistence** with related techniques.

Lateral movement abuse is based on expansion of *credentials* and *access rights*, executed with *credentials misuse*, *vulnerability exploitation* and *Access Rights*

abuse. Usually, a crucial aspect allowing all of this is **misconfiguration**, with overprivilege or inadequate account auditing. An attacker might be able to devise a technique to circumvent access right policies, by exploiting bad configurations from careless system administrators.

With lateral movement, an attacker can access various kind of data — which and with which access rights will depend on the available credentials. Of course, if the Domain Controller is accessed, the attacker will have the upper hand on the defenders, basically owning the data across the entire section of the organization pertaining to the Domain Controller.

The main reasons for success of lateral movement are the following ones:

- target has some device with *default* credentials;
- target has some device with *easy to guess* credentials;
- target has some device accessible by *stuffing or spraying* credentials;
- credentials used locally by the attacker have some *access rights* on the target — suppose the attacker managed to get his hands on a privileged machine: the remote logon, for instance, will grant him high access rights;
- an unpatched *vulnerability* can be exploited;
- a careless system administrator did a poor job regarding access rights, misconfiguring them;
- and so on...

For instance, the tool `CrackMapExec` is able to *identify* all machines in the provided IP range, *attempt credentials* on those machines and ultimately *extract password hashes* from all machines, where local admin.

Lateral movement might happend with automated tools as well. The automated tool will attempt logon on more and more machines, making *credential set* progressively larger and larger, until some credential allows executing operations for exfiltration-impact. This can be done either with Kerberoasting or NetNTLM Collection.

12.2.7 EXFILTRATION

The end goal of **Exfiltration** is to **steal data** from the target machines — transferring it over the already set up C&C channel (or an alternate channel), usually employing compression, encryption and paying attention to size limits. There are 9 techniques to do so.

For instance, `HTran` is a tool for *proxying TCP connections* – installed on unsuspecting machines with prior attacks, it opens a TCP connection to a proxy, while the latter opens a connection to the attacker. Of course, many instances of `HTran` could be running between the attacker and the organization. It makes harder for the defender to guess where the attacker is. `HTran` usually hides its traffic on the majority of HTTPS users in the organizations, thus with the traffic commonly leaving the organizations without raising suspects.

Exfiltration effort might be split between many devices that are infected, in order to both relieve the load across devices and to make harder to detect the illicit traffic.

12.2.8 RECONNAISSANCE

In this phase, the adversary is trying to gather information they can use to plan future operations towards a target, before attack.

Reconnaissance consists of techniques that involve adversaries actively or passively gathering information that can be used to support targeting. Such information may include details of the victim organization, infrastructure, or staff/personnel. This information can be leveraged by the adversary to aid in other phases of the adversary lifecycle, such as using gathered information to plan and execute Initial Access, to scope and prioritize post-compromise objectives, or to drive and lead further Reconnaissance efforts. This is done *before* attacking.

There are 10 techniques.

12.2.9 RESOURCE DEVELOPMENT

The adversary is trying to establish resources they can use to support operations towards a target, before the attack occurs.

Resource Development consists of techniques that involve adversaries creating, purchasing, or compromising/stealing resources that can be used to support targeting. Such resources include infrastructure, accounts, or capabilities. These resources can be leveraged by the adversary to aid in other phases of the adversary lifecycle, such as using purchased domains to support Command and Control, email accounts for phishing as a part of Initial Access, or stealing code signing certificates to help with Defense Evasion. This is done *prior* to executing the attack, and consists in 7 different techniques.

12.2.10 Impact

The adversary is trying to manipulate, interrupt, or destroy your systems and data. This is done *in place* of *Exfiltration* — for an attacker, it might suffice to interrupt data services at organization (disruption).

Impact consists of techniques that adversaries use to disrupt availability or compromise integrity by manipulating business and operational processes. Techniques used for impact can include destroying or tampering with data. In some cases, business processes can look fine, but may have been altered to benefit the adversaries' goals. These techniques might be used by adversaries to follow through on their end goal or to provide cover for a confidentiality breach.

Usually it comprises *ransomwares*, *web defacement*, *disk wiping* and many more possible damages. Moreover, an attacker could modify design data, emails content, or other relevant stuff at the organization.

12.2.11 Organizing defense

At this point, it should be clear that organizing the defense should address many of the tactics above. Insisting on **complete prevention of Initial Access is usually meaningless**, because the perimeter is too large to be properly controlled. The defense must consist of three pillars:

- prevention;
- detection;
- and mitigation–recovery;

with the last two **much more important** than the first one.

A maturity test for organizations is to ask CTO to provide an inventory of network-interfacing systems, software with versions, who is in charge of things, and all the necessary details. Whether or not this report is timely–provided and enough detailed tells a lot about an organization's defense capabilities.

12.2.12 Human-operated attacks vs automated attacks

Human-operated attacks are usually very effective, but they cost very much, since a human is supervisioning all the steps of the attack; they can be tailored to the specific environment of the organization. On contrary, *automated* tools are executed by a program and thus not very effective; they cannot be tailored to the specific environment, and the investment can be amortized over many

targets. Automated tools occur more frequently than human-operated attacks, and are much less dangerous due to their lack of adaptiveness.

However, in some circumstances automated attacks are **very effective**, resulting in a fast execution within the organization, and high damage. If organizations share some vulnerability, automated tools are capable of **propagating quickly** across different organizations (Petya, notPetya, Wannacry).

12.3 PRIVILEGE ESCALATION

Privilege escalation attacks are indeed very dangerous, as they allow an attacker to reach the state of maximum privileges over a machine, possibly allowing arbitrary code execution. Locally, they mean game over.

On Windows, local users are defined in a SAM file, in a local machine, while domain users are defined in Active Directory (for machines domain-joined). Any process running as administrator can read and write the entire disk-memory of the PC (hence, it can do whatever it wants).

The system process `lsass.exe` (**Local Security Authority Server Service**) keeps a LSASS table of `process_name, H(pwd-U)`, so that each process is labeled with the (ticket) hash of the user.

Of course, in Windows a process running as administrator that knows `H(pwd-U)` can assume the identity of user `U` without knowing the password! It is sufficient to write at the correct location in LSASS. An administrator can obtain `H(pwd-U)` of every local user, thus impersonating anyone in the local workstation. A terrible side effect is that the process with admin rights can obtain `H(pwd-U)` of **every domain user executing a process on that machine** (for instance, with a remote logon for troubleshooting-administration). Therefore, the admin process can impersonate every identity that operates on that PC, even a Domain Controller administrator.

For an attacker, an effective strategy is to become administrator of a machine. In order to do so, *privilege escalation* techniques must be adopted, such as vulnerability exploitment or local access rights misconfiguration.

If this succeeds, **Pass-the-hash** technique can be used. The admin process reads LSASS periodically with an automatic tool, collects credentials of any domain user `U-D` executing a process on that machine, and attempting logins everywhere as `U-D`. If the login succeeds, the attacker might also attack with that machine. Of course, it all depends on the rights of `U-D`: if `U-D` belongs to an

administrator group, or even worse it belongs to *domain administrator* group, then the range of attacks that can be performed is order of magnitudes greater.

As a rule of thumb, **never log on low-tier machines with higher-tier credentials**!

Some variations to the formula have been developed:

- **Pass-the-hash**, attacks based on reading and writing password hashes from and to LSASS;
- **Pass-the-ticket**, reading and writing tickets from and to LSASS;
- **Silver Ticket**, a forged server authentication ticket `TGT(U,S)`, where `U` is the administrator of `S`;
- **Golden Ticket**, a forged `TGT(U,S)` where `U` is the **administrator of the domain controller** `S`.

12.3.1 Access Rights abuse

A different kind of *privilege escalation* is the **Access Rights abuse**. During access rights abuse, an attacker exploits vulnerabilities in how access rights are defined so that logon on many machines and credentials discovery is faster.

For instance, let's say that `U-X` and `U-Y` belong to a certain *credentials set*, and that

- `U-X` has access right *administrator of machine group A*;
- `U-Y` has access right *add a PC to machine group A*.

Suppose an attacker is capable to operate as `U-X` and as `U-Y`, he is capable of inserting a lot of machines in group A (as `U-Y`) and start monitoring LSASS in all of those machines (as `U-X`); when a server-domain administrator executes a process in any of those machines, the attacker will get its credentials.

Another example of this is the `Backup Operators` group: a group that can read data from all servers, by default, including Domain Controllers. This is usually done in order to perform backup maintenance and administration.

Suppose `U-X` has the access right to *add a user to group `Backup Operators`*. Now, an attacker can insert `U-Y` **administrator** in `Backup Operators`, stealing data on all servers (even Domain Controllers).

These abuses involve *no password* and *no software vulnerability*. Instead, they only rely on **overprivilege** of groups and accounts, leading to **undesired chains** of access right; straight to the Domain Controller full ownership. The above

are usually (but not always) the result of inadequate auditing of accounts and asset management.

12.3.2 THE PRINCIPLE OF THE LEAST PRIVILEGE

The **Principle of the Least Privilege** states that

> *"Every program and every user of the system should operate using the least set of privileges necessary to complete the job."*
>
> Saltzer and Schroeder 1975

and is a very deep principle, comprising any running system, no matter its goals, its requirements.

Bloodhound is a security software which can easily identify highly complex attack paths, that would otherwise be impossible to quickly grasp. It reveals hidden and unintended relationships within an AD environment, by means of queries to LDAP and Domain Controllers.

Chapter 13

Categorizing the attacks

13.1 Target categories

There are three attack categories. The first one is to *organizations*, the second one is to *Industrial Control Systems*, and the third one is against *single individuals*.

13.1.1 Attacking Organizations

Attacking an **organization** is a very mainstream and remunerative way for an attacker to obtain money. Since there are many similarities between organizations, a *given* set of skills, tools and knowledge might be reused or almost completely reused across many, different organizations.

Attacks on organizations involve (all or some of) the steps we have previously encountered when discussing MITRE ATT&CK framework.

13.1.2 Attacking Industrial Control Systems

Industrial Control Systems can be put under attack by a motivated attacker, with the goal of *disruption of systems*.

ICSs are usually built with the **Air Gap** principle in mind: the IT part is connected to the internet, while the ICS part is **fully disconnected** from the IT part and from the internet. Theoretically, no attacks are possibile, however quite often it happens that there are some laptops or systems to the ICS, or ICS is permanently accessible from the IT for remote control and monitoring: this could expose the ICS to attacks.

Attacks on Industrial Control Systems are much less likely to occur than attacks to organizations. Industrial Control Systems attacks are typically *targeted*, *techniques cannot be used again* and are *harder to get money from*. Differently, attacking an organization does not typically require tergeted efforts and techniques, and ultimately there are consolidated means to obtain money from them. Basically, ICSs have very few similarities between them, thus attacks cost a lot and have a low return in money. Very few rational attackers would compromise an Industrial Control System in order to obtain money.

Some important remarks should be done, though. Attacks to Industrial Control System are usually motivated by *strategic* and *intelligence* reasons. The so–called *state-backed groups* are fundamentally hired by state-level adversaries whose interests are different from the common criminal group. A country or an organization could be interested in *data stealing* for tactical or techonological reasons. Another motivation is *disruption*, in which the functioning of a strategic industry is prevented by means of a cyberattack. In all cases, a huge amount of resources is available to attackers, while interests are not related to money.

Typically, an attack to Industrial Control System begins with many months of extensive *Reconnaissance*, with a very-carefully crafted **spear phishing** attack which is quite impossible to detect and to prevent. The later lateral movement phase is made easier due to the extensive data collection on the specific target.

Disruptions happens at the "digital level", however they can affect physical world as well, such as when an attacker disables critical sensors, deletes all data available, prevents working of automation systems, or provokes volunitarily damages to the facilities through altering machine–controlled mechanisms. State–sponsored disruption can create very high damages across target, such as in the case of targeted fuel pipelines, nuclear power plants, and military facilities.

13.1.3 ATTACKING SINGLE INDIVIDUALS

Attacker is usually interested in affecting and debilitating a single device, owned by a large group of people. For this reason, attacks on **single individuals** follow a rather simple pattern when compared to the previous kind of targets: there are the initial phases of *initial access*, *execution* and *persistence*, along with *Command & Control*, however there immediately comes the **impact** phase. No *discovery* and no *lateral movement* are usually adopted, much frequently for a reason of costs.

Impact comes through adoption of *ransomwares*, *exfiltration*, and *disruption of operations*. Especially the first one is employed to directly ask for money to the individual.

In fact, regarding attacks related to a single individual the most frequent motivation is **money**. Single individuals category of attacks is **never targeted** — the attacker employs techinques crafted for a category of devices, and puts into effect the attack to as many targets as possible. Each target can yield little money with respect to attacking a single enterprise, however the sheer number of affected users can quickly sum up to huge amounts of money.

Attacks to individuals are usually **automated**. It is much simpler to create a tool that is effective in attacking a single individual than creating a tool that is effective in attacking a large organization. Since there is no need for lateral movement, discovery phase, and it is much easier to just adopt a ransomware, it is much quicker to build automated tools for compromising a device, encrypt all the data and ask for money. The only way for an attacker to make those attacks *economically viable* is to use automation as the fundamental technique, for each target will yield only a small economic gain.

13.2 ATTACK CATEGORIES

An ideal set of attacks represents all possible categorization of attacks. Partitioning this set can be done in many ways — one of them is due to *Steve Bellovin*, who categorized attacks in 4 classes:

- **targeted** attacks involving a **low-skilled** attacker;
- **targeted** attacks involving a **high-skilled** attacker;
- **not targeted** attacks involving a **low-skilled** attacker;
- **not targeted** attacks with a **high-skilled** attacker.

13.2.1 TARGETED ATTACKS

Targeted attacks are all those attacks in which one *selects* a target, *collects information* on the target, and only then *executes* the attack.

Targeted attacks are very hard to automate (is possible) and may be lengthy or costly, depending on the target in question. To justify the huge investment, expected gain should be high enough.

For these reasons, targeted attacks possess low likelyhood, and usually single individuals have no reason to think they should be targeted by this kind of threat. Targeted attacks are relevant to organizations, ICS, nations, and many other kinds of large organizations.

13.2.2 NOT TARGETED ATTACKS

Not targeted attacks are attacks in which there is no unique target: one *selects* a target, *collects information*, then *executes attack*; however, *if the attack becomes too difficult, then target is quickly changed*. Since money is the most frequent motivation, a cost–effective target should be chosen. Any attacker will simply look for the low–hanging fruit: *who pays is irrelevant*, as long as it pays a proper amount of money.

Since not targeted attacks yield money and quickly, they are very frequently adopted.

Not targeted attacks can be inflicted to individual users as well. In those cases, attacker will construct a tool that can execute an *automated* attack, choose a large set of *almost random targets*, and attempt to use the tool on as many targets as possible. This kind of attack is very cost–effective: one initial investment could quickly and effectively lead to many gain opportunities, with even a very small expected gain per–garget that could justify the time and money spent for crafting the attack. Users are chosen according to vulnerabilities in their operating system, applications, websites of choice.

Not targeted attacks are usually feasible against single users, a lot less against organizations (automated attacks have less effect on organizations). Attacking organization is more costly, but yield much more money per–target.

Now, we have partitioned the attack set into two categories, the first one of less–frequent, targeted attacks, and the second one of more–frequent, non–targeted attacks.

The other relevant way to categorize the attack is according to the **skills** of the attacker and their **amount of resources**.

13.2.3 HIGH–SKILLED ATTACKS

Skilled people with lot of time and resources will produce attacks belonging to this category. High–skilled attackers are capable of attacking a *broad range* of

systems, employing a lot of *manual* and *stealthy* operations required to target highly–valuable people and organizations.

13.2.4 LOW–SKILLED ATTACKS

Low–skilled attackers with little to none resources will craft automated tools with one–shot usage. They are tailored to *specific* and *common* systems with vulnerabilities, and they usually affect only a very specific set of systems.

13.2.5 THE THREAT MATRIX

The **Threat Matrix** arranges all possible 4 categories in a matrix:

1. the **opportunistic** (*non–targeted*, *high–skill*) attacks. Opportunistic category usually targets organizations and looks for *money*. Attacks in this category are rather frequent. An opportunistic attacker will simply switch the target the moment he realizes it is too hard to compromise the security of an organization, and will look for better opportunities (the low–hanging fruits). The most representative attackers belonging to this category are *crime groups*;
2. the **Advanced Persistent Threat (APT)** (*targeted*, *high–skill*) attacks. Attacks belonging to this category are performed by *state–sponsored groups*, *national intelligence agencies* and are backed by a lot of resources. The targeted attack is usually motivated by *information stealing* and *disruption*, with less frequent attacks. Persistence of these attacks lies on the fact that the attack is very interested in compromising the target, not stopping after the first unsuccessfull attempts;
3. the **low–skilled**, **non–targeted** attacks. Those attacks are motivated by money, and are always automated. The low investment involved means that a lot of potential attackers can exist and employ the necessary skills to perform these attacks;
4. the **low–skilled**, **targeted** attacks. This category is not very interesting due to the low skill and few resources involved, and no attacker with rational behavior exists in this category.

From a defender's point of view, **low–skill attacks** are very frequent and practically unavoidable: for this reason, these kind of attacks **must be addressed** by basic security hygiene, which usually suffices to defend against these attacks. Those attacks are uneffective against system updates, basic controls, a careful enough user.

By far, the most dangerous category is the **high–skill**, **non–targeted** attacks. These attacks are relatively frequent, and it is *very* unlikely that the defender's resources exceed those of the attacker — this means that the attacker will *almost always* surpass the capabilities of a defender, especially because the defender's focus is not defense. Moreover, costs are **highly asymmetrical**: the attacker may concentrate his efforts on a few points in a few moments, while any defender must defend *everything* and *always*. For this reason, even with comparable resources, the attacker is almost certain to win. An attacker that believes the target is too difficult will simply switch to another one until it finds a low–hanging fruit.

As an example of this asymmetry, just consider an enterprise having hundreds of personal computers or notebooks, end–of–life web frameworks, network printers, webcams, heating and cooling systems…While an attacker simply has a few computers, network resources, probably a botnet at their disposal (much cheaper)! It is evident that the attacker almost always has the advantage over the defender.

The most effective and most rational behavior for a defender towards opportunistic attacks is to *encourage the attacker to change target*. Penetration should be *expensive*, defense must (at least) *appear* good, and many techniques such as **defense in depth** can be employed to implement *multiple independent layers*. Another technique is to prevent workstations to communicate each other, making lateral movement close to impossible, convincing the attacker to switch to another, easier, target.

As a golden rule for defensive choice, *for every dollar spent in a defensive mechanism, an attacker is forced to spend much more than a dollar*. The cost of the tool should force the attacker to spend much more to overcome the tool. Examples of defensive mechanisms belonging to this category are *HTTPS*, *HTTP Strict Transport Security*, properly configured *endpoint firewalls*, and many other.

In the last attack category, **Advanced Persistent Threat**, the attacker usually has more resources than the defender. In this case, the most rational behavior for a defender is to just "cross the fingers" and hope that an APT attack will never occur. A strong focus on opportunistic category of attacks should instead be enforced, and most if not all the security budget should be spent addressing them.

Chapter 14

Attack tools

14.1 Botnets

A **bot** is a device with a *stealthy* and *remotely–controlled* malware running on it. A very large set of bots is said to be a **botnet**, whose components are collectively controlled by a single entity, called the **botnet master**. Usually, a botnet master is a criminal group.

To survive, a botnet needs a dedicated Command & Control network. A botnet is extremely relevant in practice — as of today there are several botnets around the world, resulting in many problems.

A botnet is usually created by implanting malware by *automated* and *not–targeted* attacks that are very cheap to operate, against a lot of devices with the end goal of making them bots. Single individual devices (smartphones, home routers, PCs) can be involved, or devices within an organization (PCs, workstations, printers, routers). Some of the largest botnets nowadays are solely composed of consumer home routers.

Infection of internet–facing devices may occur *quickly*, *easily* and in an *automated* way. The *IoT devices* are a significant component of today's botnets, due to the easiness of compromising them. Vulnerable devices can be remotely exploited by means of an entirely automated technique. For instance, criminal group behind *Mirai* (2016) had performed attacks on SSH ports of random IP addresses with a dictionary of 62 credentials. This was sufficient to double the number of bots every 76 hours, with a peak of 600000 bots! The botnet was made of home routers, webcams, DVR displays, and similar devices.

The issue with unsecure devices is relevant when costs and incentives are analyzed:

- the owner of an infected IoT device does not possess enough knowledge and skill to understand that the device has been infected, and he is almost certainly unable to remove the malware. IoT owners has little to no incentive in fixing the device, which is probably working regardless of the malware;
- IoT manufacturers, as well, have little to no incentive to fix vulnerabilities. Gain margins are very tight, developing secure software is very costly and patch development is even more costly. Moreover, there is no liability of such incidents.

14.1.1 MAKING MONEY WITH BOTNETS

Botnets usually exist for *money* reasons. In ordet to obtain money, one could employ various techniques, for instance stolen and used credentials and banking credentials, stolen and sold long term cookies, devices infected by the so called *Remote Access Trojans*, who fall under the category of remotely controllable malware. Therefore, making money out of a botnet is rather cheap, easy and quite manageable. A huge amount of attacks involve, for instance, large video and entertainment brands — in fact, access to verified accounts is rather wanted in the black market; this is done by credentials stuffing. Credentials stuffing tentatives are increasingly growing.

A botnet could be used to **test credentials** of a data leak. For instance, bots can be configured to try to login with some credentials, in order to sell working ones to a higher price in the black market.

A bot master could also **rent** his botnet. Each bot is equipped with a *downloader*: the bot renter can install whatever software he prefers to apply his own set of attacks and mischievous behavior. The unitary cost of a single bot is *tens of cents*, with thousands of them that can sum up to a hundred of bucks (the lesson here is that botnet are very, very affordable).

Another kind of way to make money is to adopt the **clickfraud** method. Let a published of a website enroll some advertising, from an Ad Network of choice. Advertisers pay the Ad Network to show advertisement on some websites. Overall, the money will end up in the publisher's website for each click on advertisements. The publisher website is controlled and managed by the attacker. Bots are then instructed to go to the attacker's website, and to click on the advertisements, in order to increase the ad–click counter for the attacker, thus providing

him fraudulent money. Ad networks should be able to detect whether a bot or a human is clicking on an ad; this is, however, very difficult and even harder for small advertisement networks.

From the economic point of view, the owner of a bot does not experience a significant loss or increased energy consumption. His device will most likely work regardless of the presence of a malware (it's in the attacker's interest that the device survives as much as possible in an infected state), and he will perhaps lack enough knowledge to uncover the malicious traffic. On contrary, the advertiser suffers a lot of financial loss. The Ad Network does not suffer from anything relevant (it gets paid anyway), just a small reputation loss. This asymmetry in costs makes repelling this kind of fraudulent activity near to impossible.

Each rented bot is paid with a few clicks — 100 bots will roughly cost 10$, while a single click will generate from 0.1$ to 1$. If a bot is instructed to click 10 times a day, a single day should suffice to repay that bot. This means that there is a huge potential for gain, with 100000$ dollars *a day* with a botnet of 10000 bots, no taxes involved.

A botnet involved in clickfraud was called *ZeroAccess* (2014). According to the estimates, ZeroAccess likely generated a million of fraudulent clicks per day — which results in, approximately, 100000$ per day. ZeroAccess was partly dismantled by a Microsoft and Europol takedown, by taking down Command & Control structure. With *hours*, there had been a new clickfraud modules, by updating the botnet with a new Command & Control. For no reason, the day after the botnet decided to stop the activity. Three months after, the botnet resumed the activity.

Another way of making money is by **running services** on a botnet. For instance, services involve usage of distributed network traffic, such as Denial of Service and spam sending. For instance, *Mirai* botnet was able to lend 100000 bots, every hour, with 10 minutes interval, for 2 weeks at only 7500$. Mirai was made of digital cameras and DVR players.

14.1.2 THE COMMAND & CONTROL INFRASTRUCTURE FOR BOTNETS

In the case of botnets, a Command & Control architecture is necessary to control thousands of bots. A simplifying assumption is that each bot contacts bot master directly. Each bots will contact the bot master, and he will give back

instructions. A key problem, however, is *how to locate* the bot master, since bot master's location could be obfuscated.

From the defender's point of view, a system administrator can analyze and possibly block some network traffic at the organization's boundary. With a lot of effort, the organization could be able to detect and identify some C&C traffic, with increasing difficulty as the attacker's skill are higher.

A very skilled and resourceful defender could *reverse engineer* C&C and bot code. It is then important to share findings with the defender community – this is extremely importante in practice, and constitutes what it is said to be the *Threat intelligence*, not part of this course. For instance, if FBI manages to understand that a pattern of traffic is related to a specific botnet, the FBI could share its knowledge with enterprises such as Microsoft, Apple, Symantec, and so on.

From the attacker's point of view, loss of *some* bots is tolerable; for instance, bots that happen to be blocked, detected, or replaced with not–infected devices. However, it is not acceptable to lose the *entire* botnet. For this reason, Command & Control traffic should be bullet–proof, secure, and authenticated, since defenders might attempt a botnet takeover (or other attackers, of course). The guarantees for the C&C protocol are those of *authentication*, *integrity* and *secrecy*.

Historically, there has been an evolution in the techniques and the resources necessary to fighting botnets.

In **Generation 0**, each bot contacted the master at a *predefined `IP-BO` address*. IP address was hardwired in the code; as long as defenders detects the nature of `IP-BO`, for the attacker it was game over: every organization could simply blacklist `IP-BO`.

The **Generation 1** used a technique called *IP fast–flux*. Each bot contacted a *predefined `N-BO` domain name*. The attacker *frequently modifies* the IP address by changing the DNS record `N-BO A IP-X`, so for a defender blocking by IP address was no longer possible. Still, as long as a defender detects the nature of `N-BO`, every organization could still blacklist `N-BO`. Moreover, legal actions against Registrar that manages `N-BO` could dismantle the botnet completely — hence, *questionable Registrars* are adopted by attackers.

In **Generation 2**, bots contained a **predefined algorithm**, the Domain Generating Algorithm, that generated a *different domain name `N(day)` everyday*. This meant that everyday bots contacted a different name — rules blocking names

and IP addresses no longer worked, with firewalls that should have been updated everyday. On any different day, a new Registrar might be involved.

With a lot of effort, defenders might *reverse engineer the bot code*. However, this is a very hard task — just realizing which names are generated by a bot at the boundary of an organization is astonishingly complex. After the domain chain generated by the algorithm has been discovered, defenders could purchase a domain that will be generated in the future, taking control of the botnet.

The attacker could detect that some DGA domains are no longer free: defenders are acting. In order to keep control of the botnet, he will *develop a new DGA* and distribute it with a software update to the bots.

DGA Improved is a technology invented in 2011. The DGA generates *tens of thousands* different names everyday; each bot will contact all those names, and if the attacker possesses a single domain name the bots will be commanded by it. As long a single domain responds, it's ok with authentication, and the command & control chain will persist.

Of course, even if a defender understands how this algorithms works, it is economically unsustainable to prevent the attacker adopting *a single* of those domains.

Today, attackers own many different servers that are organized at multiple levels in multiple groups. Mirai possessed 484 different servers with 33 independent clusters that were observed. Moreover, bots can have a so–called *peer–to–peer structure*, with all bots act as peers; just sending a command to a single bot will result in command propagation through the entire botnet.

Only very high profile and equipped defender organization can filter or dismantle botnets, with a process that usually involves lot of time, efforts, and collaboration, usually on side channels (for instance, payment of domains). Due to the incredible costs involved, defense is feasible *only* againt the most important threats.

A system administrator at an organization should forget about the possibility of discovering or dismantling botnes (since these operations could only be done by very specialized organizations). Bots should instead be detected and isolated. In most cases, an organization has some sort of *big firewall* at the boundary, which is able to inspect the application traffic as well as the IP and TCP level. Firewall rules should be updated everyday by purchasing a proper license. In order to be able to do so, however, the organization should subscribe and join the Threat Intelligence program.

14.2 VULNERABILITIES

Vulnerabilities are *mistakes* in software, that can be used by an attacker to take advantage and control over a system.

Vulnerabilities can affect most if not all systems, and are especially dangerous for network–connected devices.

Generally speaking, **bugs** are errors or flaws that cause the software to both produce an *incorrect* result, or to behave in *unintended ways*. Differently, vulnerabilities are a *subset of bugs*; in fact, they are bugs that allow violation of some **security property**.

14.2.1 EXPLOITS

An **exploit** is a tool for exploiting a vulnerability; that is, something capable of driving the vulnerable software to its execution path that it yields the mistake, unintended results, to violate some security property. An exploit can either be a piece of code, a chunk of data, or a sequence of commands.

Usually exploits should be carefully crafted. For instance, an exploit could be a Word document, whose content is crafted in a peculiar way so that it exploits a mistake present in the software. The user is simply required to open the document to be infected. Sometimes, the file opening has no visible effect – in this case, the gravity of the vulnerability is even higher.

When a vulnerability is discovered, the immediate reaction should be to *release a patch* to fix it. Patches are downloaded by the user or the user's software automatically, so that the addressed exploit technique will no longer work.

In the case of the Word document, a specific exploit involved inserting an URL in a document, whose content was a VBSCRIPT script. Word automatically downloaded the code and run it — attacker could simply insert any code and the end result was to download and run a malware in the user's machine. HTML document containing the script should have been saved with `.rtf` extension, and be served by the remote web server with `Content-Type: application/-hta`.

The next step is to create a deceiving Word file (whose appearance is legitimate), containing a Word Object with link to the malicious URL. Finally, modify Word file with a binary editor, and insert a string `objupdate\` into a specific portion of the document.

In the end, an exploit is *an input not handled correctly* by the vulnerable software. A vulnerability is first discovered, and *then* an exploit should be prepared. Both steps are difficult, and they often are a *full–time job*. Security researchers first find vulnerabilities, and then usually build a so–called **Proof of Concept (PoC)**.

Since writing exploits is very difficult, not all vulnerabilities are exploited. For some of them, it could be even almost impossible to write an exploit, so from the point of view of the cost–effectiveness an attacker may be discouraged to exploit such a harsh vulnerability.

14.2.2 EXPLOIT INJECTION

To be useful, an exploit should be **injected** in a vulnerable system. Exploit injection is the third step after vulnerability discovery and exploit creation, and it may be result in varying levels of execution difficulty.

Exploit injection occurs with many possible ways; there are two different ways to categorize injection techniques, the **user action required** category, and the **no user action required** category.

Examples for the first category are *sending a malicious file*, *sending a malicious link* that should be opened with a specific, vulnerable software, *putting a file in a USB pen* and so on.

All the above injections involved the user clicking or performing a specific action (which is more often triggered by deceipt). Differently, the second category of injection occurs **remotely** and with no user action required. Those are the vulnerabilities in remote servers that handle specific commands or messages. The category is said to be **Remote Code Execution**. The gravity of RCE increases as much as there is no need for authentication, if the injection happens easily or not, and if it allows information disclosure only or privilege escalation.

Another way to categorize an injection is the **distance** in which it can be used. **Local** injections are all that can be done *only* by a program *already* running on the victim device. Differently, **remote** exploit injections can be done *remotely*.

A very dangerous category of vulnerabilities are the so–called **Wormable vulnerabilities**, that is a vulnerability with these properties:

- remote code execution;
- unauthenticated attacker;

- no user action required.

and an exploit for this vulnerability can *propagate itself automatically*. A possible, simple exploit could be to attempt to connect to *all* IP addresses and inject automatically a copy of itself on every vulnerable IP address found. Experience shows that within a few minutes all vulnerable systems worldwide reachable from patient zero will be infected. To prevent massive infection, tight firewall rules and NAT are important.

THE SECURITY—USABILITY TRADE-OFF REGARDING EXPLOITS

No input whatsoever assures maximal security; however, this will not guarantee any productivity. On the other hand, minimizing security means accepting every possible input; that said, an attacker has the largest possible set of attacks at his disposal. Productivity is the maximum one, provided you are not hit by an attack.

Avery input is a potential threat, and it might be an exploit. The risk level should be decided by reasonning on the strictly necessary inputs, and by removing the unnecessary ones. The set of possible inputs *should be minimized*.

14.2.3 MANAGING IOT AND ICS DEVICES

Some categories of devices that are increasingly connected to the network are:

- Industrial Control Systems;
- Electricity meters;
- Insuline pumps;
- sex toys;
- foam mattresses;
- webcams;
- thermostats;
- washing machines and dishwashers;
- and so on.

The usual reasoning on those devices is that *as long as an authentication password is required and correctly managed, the device is protected*. The threat model that this reasoning implicitly defines is that the attacker can only communicate with the device, this is however not realistic.

An attacker can more than often execute arbitrary commands on the device (due to a vulnerability). For instance, `shellshock` familty of `bash` vulnerabilities affected vulnerable Linux instances in late 2014 with a crafted command (`User-Agent: () {:;}; command`), with no authentication required.

Attackers can also use correct credentials by vulnerabilities that expose them. Some malformed requests, for instance, could make the device respond with their configuration information, including credentials.

The correct reasoning is that attacks will fail **only** if the device has no vulnerability. A vulnerable device may be exploited by anyone that:

- is *aware* of the vulnerability;
- is *able* to exploit it;
- is *motivated enough* to spend resources on the attack.

> "Just because a device can connect to a network does not mean that it has to be connected or that that network has to be the Internet. If a connection is really required, differentiate between IoT devices that need to be connected to the Internet and those that do not."

Thus, the best practices regarding IoT devices are to **minimize network exposure** for all control system devices and or systems, ensuring that they are not accessible from the internet, and to locate control system networks and remote devices **behind firewalls**, isolating them from the business networks. When remote access is required, **use only secure methods** such as VPNs; however, recognize that VPNs may have vulnerabilities and should be updated to the most current version available. VPNs are also as secure as the connected device — if a device runs a malware, the game is over.

Isolation and the concealing of those devices is the bare minimum security practice nowadays.

14.2.4 OTHER VULNERABILITY DEFINITIONS

More generally, a vulnerability is not only a software problem, and does not only involve the possibility of running exploits and the initial access to the software by an attacker.

Vulnerabilities might exist in other forms. For instance, that is the case when *user and root accounts have hardcoded passwords* that cannot be disabled, and that they are the same for all devices in the world; the same goes for same hardcoded RSA-keys in all devices.

Secrets embedded in devices are clearly vulnerabilities that are not captured by the original definition, since they are more a design flaw than a mistake in software coding. A good definition is the following one,

> *a **vulnerability** is a flaw or a weakness in a system's design, implementation or operation and management that could be exploited to violate the system's **security policy**.*

This definition involves *software mistakes*, *hardcoded secrets* such as keys or passwords, *over–privilege by default*, and so on.

As general rules,

- never ever design systems that embed the *same secret in all their instances*;
- never ever design systems that embed a secret that *cannot be modified*.
- never ever embed hardcoded passwords, keys or tokens on code.

14.2.5 ASSESSING RISKS

The risk of a given vulnerability depends on *several* factors; the **impact** of the vulnerability (does it allow only to disclose informations? Or it allows privilege escalation and code execution?), the **difficulty of exploitation** (whether or not an exploit is hard or easy to craft) and **difficulty of injection** (local or remote? Needs user intervention?).

Taking in account all these factors is difficult. A simpler way for assessing risk is to assign a **score** that quantifies the risk of a vulnerability. Those possible metrics are called **vulnerability metrics**, several standards for attemptint to quantify all the aforementioned features by means of a single number, the score.

The most widely used standard is the *CVSS (Common Vulnerabilities Scoring System)*, and it takes 22 categorical features as input and assigns certain features in a slightly subjective way (for instance, easiness of exploitation is a subjective feature that is assigned differently than remote versus local). The output is a discretized score in range 1 to 10:

- *None*, 0.0;
- *Low*, from 0.1 to 3.9;
- *Medium*, from 4.0 to 6.9;
- *High*, from 7.0 to 8.9;
- *Critical* from 9.0 to 10.0.

It is useful to remember that score is not an intrinsic property of a vulnerability, but just one of the *many, different ways* of distilling many features into a single number. Every vulnerability must be analyzed based on the specific environment, and the score is just one of the possible factors to consider. Low–risk vulnerabilities in systems that are vital to an organization can be more dangerous than high–risk vulnerabilities affecting systems that are not important and almost unreachable.

14.2.6 THE MOST SECURE SOFTWARE

"Which software is the most secure?"

Unfortunately, there is no way to quantify how secure a software is.

Let's take the *number of known vulnerabilities*. Do they depend on the software or on how many people are looking for them? Are these people all technically skilled? Do they spend a lot of effort in doing this kind of activity? These features can hardly be taken in account.

Suppose now to know the number of known vulnerabilities, the number of patched ones, the number of *zero days* vulnerabilities and the time required to patch, averaged. Is it possible to combine them? How so?

First, let's question whether the estimate at a certain time of the zero days is reliable. Some zero days may not be already discovered, some others may be even actively exploited. Hence, the zero days estimate is even less correct when making predictions in the future.

Moreover, the number of known vulnerabilities at a certain time is **not** a good predictor of the number of vulnerabilities that will be discovered in the future, nor of the number of zero days existing currently and in the future.

More reasons come from the exploits assessment. Some software may have many vulnerabilities that are hard to exploit, and some other software may possess very few vulnerabilities, but very easy to exploit.

That said, there is a unique way to determine whether a software is more or less secure. A **key indicator** of the security of a software is *how the vendor **reacts** to a vulnerability*, a factor much more important than the vulnerability. A good

vendor acknowledges the vulnerability, releases a patch as soon as possible, and possesses a bug bounty program.

Another characteristic of a good vendor is the *uniformity* of updates and support time for devices.

14.2.7 THE VULNERABILITY LIFECYCLE

The **vulnerability lifecycle** is the entire period of time between *discovery* of a vulnerability and its *fix* on software across the world.

The **ideal** case of vulnerability lifecycle starts with a *discovery* event by a security researcher, in which a *private disclosure* to the software vendor immediately follows. The software vendor will then work its best to provide a *patch*; the patch is released together with a *public disclosure* of the vulnerability. After some time, the patch is adopted: the *patch application* occurs, and the vulnerability lifecycle ends with that.

To summarise,

1. the first phase is the vulnerability *discovery*. Discovery usually occurs through the work of a security researcher;
2. the second phase is the *private disclosure*. Security researcher notifies the software developer, soliciting to take action and work for a fix;
3. the third phase is the *public disclosure*. A vulnerability becomes public information, possibly just after a *patch* had been made available;
4. the fourth and last phase is the *patch application* phase. System administrators are prompted to update their systems in order to include the released patch.

In this model, the maximal risk starts at public disclosure, where *everyone* knows that a specific vulnerability exists. The problem with this is that *before* applying a patch the instance **is vulnerable**: the risk is that an attacker starts attempting to explot the vulnerability before people starts applying patches (for instance, the `log4j` fiasco). Applying patches should be rather quick and should involve all the systems affected by the vulnerability.

14.2.8 A BUMP INTO REALITY

The ideal case does not always match reality, though. One of the previous steps might not occur:

1. **researches** *may not notify* the software vendor — they could rather sell their information to criminal organizations (or be part of them...), looking for a greater amount of money. There is a huge *illegal* market of software vulnerabilities, especially when the vendor is reluctant to act. *Zero day vulnerabilities* are those known to one or more organizations, while not known to the software vendor. Zero day vulnerabilities possess high value until discovered and fixed. There are even legal zero day buyers–sellers (*Zerodium*). Zero days are very frequent, with 207 estimated from 2004 to 2016, 50% unknown and with average life expectancy of approximately 7 years (only 25% survive less than a year and a half). As of 5 April 2022, *Google Project Zero* estimates are of 211 zero days on the wild. Zero days are used very infrequently — before using a zero day exploits, the attacker tries with other techniques. Zero days are highly effective, but they cost very much, and they could be discovered by defenders at each usage. Minimizing the risk of zero day discovery is crucial for intelligence agencies. Moreover, the same organization that owns a zero day will be vulnerable to that specific vulnerability as well;

2. **software vendor** *may not develop* a patch — they could bot be interested; Also, *patch development is costly*. It may also be very difficult, and in most cases there is *little to no contractual obligation* for patch development. Depending on the specific vulnerability, fixing a vulnerability might be rather expensive, difficult, and time–consuming. Further, a software could be in its *end of life* cycle, under which no support is provided (therefore, all copies of that software will be vulnerable *forever*).

3. **public disclosure** may occur *without an available patch* — someone leaks the information for some reason. Another issue with software vendors is that they could *downplay relevance* and do not act. Researcher might be tempted to keep a vulnerability secret, as the advantage is that he will not lose time and won't fear legal actions. The public pressure on vendors is the only thing that can possibly work, so the only way is to *disclose the vulnerability*, along with a proof of concept exploit for maximal public pressure. The most ethical approach dictates that, before disclosing, a *reasonable deadline* (grace time) should be given to the software vendor for acting — that is called *Responsible Disclosure*;

4. **system administrators** *might not apply the provided patch* — they could be lazy or unskilled. Moreover, applying a patch *takes time*, requires usually *service downtime* (think of Linux kernel patches) and *compatibility problems*. Indeed, a system administrator may not have enough time to

upgrade; or he may be even *unaware* of specific vulnerability, unaware of the importance of patching or even not aware of which systems should be managed. For these reasons, system administrators are not unfrequently the weakest ring of the vulnerability lifecycle chain.

Unfortunately, the population of security experts is incentivized in discovery vulnerabilities, keep them hidden and diffuse knowledge only in restricted circles, either for money advantages or for idealistic reasons. Users will not even know about the vulnerability that has been actively exploited.

The general takeaway of this reasoning is that vulnerabilities *affect every kind of software*. For this reason, vulnerability patching is **crucial**. In many cases, though, the most important defense mechanism doesn't exist or cannot be applied.

ASSET MANAGEMENT

An **asset management** is an accurate description of systems that are connected to the network. The description involves software, versions, known vulnerabilities, and whether devices are exposed to the internet or not.

The asset management is a *basic* requirement; often not fulfilled adequately, especially since it's much harder than it seems. As a maturity test for an organization, ask a CTO to provide the asset management — accurate answer should come in less than 30 minutes.

14.2.9 OTHER IMPORTANT ISSUES

Since there are many vulnerabilities cannot be patched, what could we do? The answer depends on our role in an organization.

Moreover, who pays the cost of a security incident? Is the vendor liable for a vulnerability in its product? Is a developer obliged to develop patches, and for how long?

Some good practices, depending on who you are, can be summarized as follows:

- **Non–IT Engineers** that include software in their products should consider themselves software developers, and the related company as a software company, thus managing software security accordingly;
- **IT Administrators** should periodically check their devices for known vulnerabilities. In particular, there exist tools — for instance, *Nessus* — which are capable of discovering and assessing vulnerabilities, along with

an asset management. Every organization will find a lot of vulnerabilities in its software and services: a proper *structured process* for managing vulnerabilities is required, since patching all or even most of them is simply a not realistic approach. Some vulnerabilities might be ignored, while others could be fixed or marked as "reconsider later";

14.2.10 REASONS WHY VULNERABILITIES EXIST

Vulnerabilities arise from various, fundamental, problems in software and in security. In particular, the following reasons are intrinsic to software only, but *not* intrinsic to other kinds of technologies:

- in software, more security implies *more costs*. Each added cost has no impact on usability and it is rarely perceivable by users (improved security may even result in decreased usability!);
- software market does not *incentivize security*. Adequate incentives might improve the scenario, although the fundamental problems will remain. There will be a framework change sooner or later in the regulations;
- there is little to no liability involved when an incident occurs.

It does not end here: more reason follows directly from the *intrinsic nature* of the software, as a quite peculiar engineering product, which can manifest a very different behavior from any other engineering crafted product.

The first issue regards the impossibility of producing a convincing amount of tests to effectively tackle security. Typically, tests should be run before releasing a product. In order to release an *engineering artifact*, among the "set of all possible inputs", the *test set* can only be a small set of points in the region of all possible inputs. For each input belonging to the test set, the engineer's duty is to verify that the outcome is correct. Now, let's consider an input that *does not belong* to the test set.

Testing a software requirement is rather easy: it is sufficient to check that the software has some properties. Testing the requirement means tries executing the task — if the task is performed successfully the software is ready, else refactor or redesign. This kind of requirements of what the software **can** do is said to be a *positive requirement*.

In security, this doesn't work. The requirement is usually that "a certain action **cannot** be done", a *negative requirement*. Testing such requirement means identifying **all** the possible ways for attempting an unauthorized action, and check them all. This is radically more difficult, since a security developer should

check all possible ways for executing such action, and test them all. Testing against negative requirements is very difficult; moreover, it is impossible to really identify all the possible ways.

Normal users inject inputs that are related to the *expected usage*. Adversaries do not behave nicely: they actively search for inputs that *violate* security guarantees. They do so by looking for inputs that are both *totally unexpected* and *wrong*, usually with the help of automated tools.

An **hostile** environment is an environment in which inputs may be extreme and perhaps uncommon — in software, an **adversarial** environment is an enviroment even more dangerous in which inputs that are actively selected to provoke undesired behavior; "the worst possible input at the worst possible time". Basically, software is subjected to *actively* attempts to exploiting vulnerabilities.

In addition, the difference between any non software system and any software system is that *for non software systems it is very unlikely that the behavior of the system will be different from the test set*, provided the test set is chosen properly. This means that totally unexpected and undesired behavior is unlikely to happen. On contrary, *software is* **not continuous**, *the behavior could be radically different from inputs belonging to test set*. A successful test for an input can tell nothing about the system's response to a similar, but distinct, input.

Software also possess a lot of the so–called **internal state**. Injecting an input at different instants will produce different outcomes, since the output depends on many *past* inputs. In practive, one may consider a test set that is extremely small when compared to the huge subset of all the possible input sequences, given their dependencies on past inputs as well.

The practical consequence is that every software artifact is release with an unknown number of *bugs*, inputs that provoke some undesired behavior, and whose behavior might include the violation of security properties: a bug can be a vulnerability.

The possible, practical solutions to this issue are both analyzing how the system will be used, and including in test set as many *realistic input sequences* as possible. Basically, one tries hard to understand how the system will be used.

The other fundamental problem is related to cybersecurity.

In a nutshell:

- non–IT engineers must cope with **hostile** environments;
- IT engineers must cope with **adversarial** environments.

PROBLEMS LEFT IN TESTING SOFTWARE

A software artifact must face inputs that are both *not in the test set* and *actively searched* by adversaries. Since security in software is a negative requirement, making sure that the artifact does not exhibit undesired behavior is extremely hard. Basically, there is *no proof* that a software artifact will not behave unproperly, since software is not continuous, testing is very costly and any piece of software is drowning in an adversarial environment.

What a software developer that is involved in security can do is to aim for some **degree of confidence**, also called **assurance**. Basically, the software should behave according to positive and negative requirements, up to a certain degree of confidence. Many complementary techniques allow to achieve a certain assurance, for instance development process (for instance, two teams working at a same project), programming languages and methodology, testing methodology, operating system technology and so on.

As a matter of fact, assurance is *not a binary notion* (not a proof of whatsoever) and *cannot be quantified or measured*. To put it brutally, *software is a cross–the–fingers technology*.

14.3 MALWARE

Malware is a software that *deliberately* fulfills the *harmful* intent of an attacker. Usually, malware acts in a *stealthy* way, unless it is designed to manifest its presence, for instance in the case of *ransomware*.

Malwares are typically classified in multiple classes, depending on the classification adopted (trojan, bootloader, rootkit, and so on).

Malwares might be installed during any phase:

1. manufacturing;
2. installation;
3. configuration;
4. use, with both physical and non–physical access means.

The most intuitive way is during use of the software, and by means of remote access. Depending on the specific context, though, the defender shall consider one or more phases.

Regarding remote malware installation (with physical access is more trivial), MITRE attack framework has three phases that correspond to malware installation: *initial access*, *execution* and *persistence*. Basically, there are many ways to execute malware, but the most important ones are by means of **software vulnerabilities** and **user execution**; the latter is when an adversary may rely upon specific actions by a user, that it is subjected to **social engineering** to get them execute malicious code by performing an action.

Social engineering is a very common technique: rather simple, cheap and has a high probability of success. Since social engineering is so common, effective and widely adopted, defense from malware cannot be completely technical. Technology should limit opportunities and impact of vulnerabilities, but in the end of the day *users need to be aware and educated* to prevent social engineering manipulations.

14.3.1 ANTIVIRUS SOFTWARE

Antivirus software has a role in detecting and preventing *exploit injection* and *execution of malicious code*, and is usually adopted in Windows environments.

Antivirus attempts to detect and prevent exploit injection. An antivirus might be able to prevent exploits in the lapse of time between public disclosure of a vulnerability and its patching through a software update. Even though a vulnerability will never be patched, an antivirus in principle may be able to protect the system forever. Antiviruses can only detect what is *already known* by the antivirus, by means of signature detection. Some products possess *anomaly detection*, for software and behaviors that could potentially harm the system. This system, though, suffers from a lot of false positives, experts analysis is needed and it is almost never used for antivirus. Anomaly detection must also refer to already known attack patterns.

However, **polymorphic malwares** do exist, and they make antivirus a lot more useless due to impossibility of comparing signatures.

Antivirus software works by detecting *expoint injections*. What happens is that a vulnerability v is first discovered;

1. figure out how v works;

2. develop detection rules for **all** the possible exploits[1] of v; repeat as necessary overtime;
3. integrate the new collected signatures in the antivirus;
4. release the new signature set;
5. antivirus are ready to be updated (**signature download**).

The signature development and download phases are very delicate — they *take time*, and they leave the antivirus unable to protect the device to new threats. This means that as soon as a new vulnerability is discovered, a *time delay* is required before the antivirus starts protecting against the very same one.

WannaCry was able to propagate very quickly in 30 minutes only: a too thin time interval for any antivirus product to possibly react.

Further problems for antiviruses: signatures for *zero-day* attacks do not exist, since they cannot possibly be detected (otherwise, they wouldn't be *zero-day*s). Moreover, signatures cannot be developed for **all** vulnerabilities: there are just too many of them, and antivirus vendor must prioritize the most common and dangerous ones. Even another issue is that the signature must detect exploits, not vulnerabilities. Still, given any exploit, an attacker might be able to obtain *nearly-infinite* other exploits that **look different** but are **funcionally equivalent**. Since writing accurate signatures is rather hard, criminals may achieve antivirus evasion this way.

This all boils down to four factors:

1. signature development *takes time*;
2. anomalies provoke too many false positives to employ *anomaly detection*;
3. signatures cannot be developed for *all the possible attacks*;
4. attackers can *"morph"* their malware to evade antivirus detection, without limits in morphing capabilities.

Antiviruses hardly protect from the attacks of *today*, but they protect from the attacks of *yesterday*. Moreover, they prevent only the most frequent and most dangerous attacks.

[1] This is crucial — there may be *many* different exploits for a single vulnerability, each one having possibly a very distinct behavior. This is not easy, especially in exploits carried by let's say Word documents (an antivirus should be able to detect a very general and broad range of all possible documents that can carry the exploit).

VIRUSTOTAL WEB SERVICE

VirusTotal web service aggregates many antivirus products and online scan engines to check for viruses, or to verify against any false positives. The service is free and can be used without purchasing a license.

14.3.2 SUPPLY CHAIN COMPROMISE

Malware can also be installed during *manufacturing*, *installation* and *configuration* phases. This represents a *huge* problem; no longer a theoretical possibility, it will become more and more relevant.

One of the techniques of MITRE attack framework Initial Access phase is the **Supply Chain Compromise** techniques: adversaries might be able to manipulate *products* or *product delivery mechanisms* prior to receipt by the final people. This means that an attacker is able to *introduce a malware directly in the supply chain*, without people possibly know. That can be done by manipulation notebooks, firmware, software and so on.

This represent a strong technique to *target specific important people*. Installing malware before a device, a software, or a system arrives to the targeted individual can be done by *updates manipulation*, *preinstallation on device* or by altering *distribution channels*. From a defender's point of view, this is a true nightmare, since *no one can be trusted anymore*.

SolarWinds software for network security monitoring has been compromised in late 2020 — adopted by over 18000 large and security–conscious organizations, the compromised software has seen *malicious updates* installed and distributed. Many more examples are available on the web (and on the last pack of slides).

Supply chain attacks are marvellous for an attacker that wants money, since *a single attack can result in many, many organizations infected*, all the organizations that use the compromised software or device. The average open source project depends on 180 libraries on average — there are 180 different points in which a malware could be inserted. This usually occurs in any public–facing repository, as software that is submitted is generally not inspected by any entity or group.

Ultimately, supply chain compromise may take place in **any** of the following ways,

- manipulation of *developer tools*;

- manipulation of *developer environment*;
- manipulation of *source code repositories*;
- manipulation of source code in *open–source dependencies*;
- manipulation of *software updates and distribution mechanisms*;
- compromised or infected *system images* (for instance, USB devices);
- replacement of legitimate software with malicious modified versions;
- sales of *couterfeit products* to legitimate distributors;
- *shipment interdiction* (remember NSA tampering with routers shipped to foreign customers);
- and so on.

From the defender's point of view, this represent a real nightmare, since even the **perimeter** of defense is unknown! Questions arise whether hardware and software vendors and manufacturers might be compromised — moreover, internal applications could still be compromised (who built our website? on which platform does it run? with which libraries?) as well as software development tools and libraries in use.

What can be possibly done in practice?

The only defense is like any other malware — scan downloads for malicious signatures, test software before production deployment for any suspicious activity, perform physical inspection of hardware to look for potential tampering.

An interesting approach is the **zero–trust** approach. The idea is that whenever you buy a machine, the machine should authenticate itself, with integrity checks on hardware–level, operating system–level and software–level.

Today's best practice regarding supply chain compromises is not really a defense — it is something related to understanding *who your providers are*, and *decide who you put your trust on*.

Another important approach is *reproducible builds*. Reproducible builds use verification of distributed sources and binaries through has check or other integrity checking means. The state–of–the–art is the authentication and in-

tegrity of compiler, compiler options, libraries involved, source code and so on.

Chapter 15

Some important case studies and related considerations

- Vulnerabilities affect most software in everyday usage — for this very reason, Microsoft releases a patch every month fixing hundreds of bugs and vulnerabilities.
- Remotely controlling a malware on a ICS—SCADA devices with custom-made tools had been performed by APT actors, by developing tools to *scan for* and *compromise* affected devices. Several vulnerabilities could be exploited by remotely-controlled (with custom protocol) infected devices. Those Schneider and Omron devices were exploited or compromised by *bruteforcing credentials*, online password guessing + dictionary + (very few) defaults. Installed malware could open a remote virtual console, so that basically arbitrary code could be executed. Objective were focused on *disruption*, *sabotage* and similar.
- To solve above issues, *suggested mitigations* **didn't include patching**: ICS devices often do not have patches. There are 14 items in the list — most of them really hard to perform (a nightmare for security workers). One of the listed items is to change passwords into *device-unique strong ones*.
- Other important security mitigations: enforce *principle of the least privilege*; *isolate ICS—SCADA devices from the network* and use *strong perimeter controls*; *limit ICS—SCADA connectivity* to what's necessary only;

ensure applications are *only installed when necessary* for operation; *update operating system*; use *multi-factor authentication*; use *strong passwords*, ensure *antivirus* and security software is *up to date*.

- Computer attacks no longer affect only data — they also affect *physical world* and *public safety*.
- The key way to understand how an organization works is to **focus on the incentive structure** of your environment.
- One should invest *a lot* in **detection** and **monitoring**, instead of *prevention*.
- Install *independent*, cyber-physical *safety* systems. These are systems that physically prevent dangerous conditions from occurring if the control system is compromised by a threat actor.
- In *SolarWinds* fiasco, only **FireEye** society was able to detect the attack.
- Microsoft President told that *not even in intelligence agencies, federals employed basic cyber hygiene and security best practices*. Basic countermeasures are considered strong enough to make sure that attacks have limited success (as it would have happened in case of *SolarWinds*).
- **asset management** is crucial for correct **patching** of all systems.