CO331 - Network and Web Security (60015)

Week 1

Vulnerabilities

We define **vulnerabilities** as bugs or design flaws in software that can be exploited by attackers to compromise computers. These are taken advantage of by **exploits**, which are pieces of software. If it is unknown to the software vendor, it is referred to as a **zero day** (has not been disclosed to the public domain).

Advisories are used to publicly disclose new vulnerabilities, issued by vendors or security companies. These are important for developers, sysadmins as well as regular users of the software, in order to keep up to date or patch systems.

The **vulnerability reports** often vary in format. Bugs and systems can differ from each other, as well as researchers putting in varying levels of effort. Generally, the key information consists of the affected systems, descriptions, impact, proof of concept code, as well as proposed fixes.

There are a number of approaches for when vulnerabilities are discovered;

- **non-disclosure** keep the vulnerability secret

  This is preferred by vendors who choose not to use resources to fix bugs (based on 'security by obscurity'), or by parties intending to exploit it.

  An issue with hoarding vulnerabilities is the accidental release. For example, *WannaCry* used two exploits hoarded by the NSA - if this was disclosed, many more systems may have been patched.
- **responsible disclosure** affected vendor decides when and what to release

  This approach is preferred by software vendors, motivated by the idea that end users will not develop their own fixes. However this can lead to a long duration between a discovery and fix.
- **full disclosure** make details public

  Eliminates any asymmetric information advantage attackers may have. This method is preferred by security researchers, as well as the open source community. However, it may affect users to attacks.

  The current approach, spearheaded by *Google Project Zero*, is to give a window of time to vendors to fix vulnerabilities before it is publicly disclosed.

Malware

Malicious software can be characterised by infection vector;

- **virus** malicious code copying into existing programs
- **worm** replicates program over network or removable devices
- **trojan / spoofed software** provides (or pretends to) useful service to act legitimate
- **drive-by download** code executed by visiting malicious website

Another way to characterise malware is by purpose (malware often has multiple of these working together);

- **rootkit** strongest, works at OS level (can hide itself)
- **backdoor** allows attackers to connect over network
- **RAT (remote access tool)** remote control
- **botnet** recruit machine into botnet
- **keylogger** logs keystrokes
- **spyware** steals sensitive documents
- **ransomware** blocks access to machine or data until ransom is paid
- **cryptominer** uses system resources to mine cryptocurrency
- **adware** displays advertisements

Malware can exist in several formats;

- injected code added to a legitimate program
- library loaded by a legitimate program
- scripts run by application (such as macros in *Microsoft Office*)
- standalone executable run by the user
- code loaded in volatile memory (fileless malware) - without a file, detection can be difficult

Viruses can propagate in a number of ways, either by the attacker in the case of self-replication, or drive-by downloads, or installed by the user, either through social engineering or compromised certificates (in fake software updates).

A virus can have varying privileges, either from the lowest level (in a rootkit, where it owns the machine), or have user privileges which can do limited damage.

APTs (Advanced Persistent Threats) are used to reach high-value victims. These attacks are specific to the victim, often driven by a human. Decisions are made, depending on the specific configurations, and can involve compromising intermediate systems to reach the victim. Detection is avoided, with the use of rootkits to hide presence, as well as large datasets being exfiltrated over a long period of time. Avoiding detection is important as these attacks are often done over a long period of time, waiting for information to enter the system, as well as retaining access for later use.

On the other hand, **botnets** are generic attacks, which aim to infect as many machines as possible. The idea is to infect many machines (bots) to allow an attacker (botmaster) to control them through a command-and-control server. The botnet can be used for the following;

- **data theft** steal credit card numbers or passwords
- **spam** less likely to be shut down, compared to single server
- **DDoS** flood servers with requests
- **brute-force** similar reasoning to spamming, passwords / credit card credentials
- **network scanning** probing other hosts
- **click fraud** generate advertising revenue from different sources
- **cryptojacking** see above
- **rental** botnets can also be rented out for use by others

Analysis can be performed on captured samples (to aid in detection or removal), obtained from cleaning up an infection or running **honeypots** (by willingly installing malware). Effects on storage, system settings and network traffic are often analysed in a virtual machine sandbox. However, it may be difficult to trigger malicious behaviour (since it may behave differently in a virtual environment).

Detection can be performed by extracting signatures from analysed samples. **Static** signatures are sequences of bytes, typical of malware, and can be detected quite simply and quickly. However, this method is also easy to evade, where samples are artificially made different from each other, with **metamorphic** malware, or by the use of **crypting** services, which encrypt and obfuscate malware until it is no longer detected (FUD).

On the other hand, **dynamic** signatures or behavioural analysis can be performed, where the host is monitored for patterns of actions typically performed by malware (such as reading data then sending data over a network). A way for this to be evaded is for the malware to mix malicious behaviour with legitimate behaviour.

Current defences for malware include standard antivirus software, which scan existing and downloaded files for static signatures, as well as **end-point protection (EPP)**, which monitors the host for dynamic signatures. Browsers also now include blacklists which prevent access to pages known to be hosting phishing sites and malware. Network based protection can also be used.

However, signatures and blacklists are both based on observed malware, therefore attackers have a window of opportunity before detection. As such, prevention is often the best strategy, such as educating humans to avoid direct installs. Software should also be updated and patched in response to disclosures; it is rare that zero-days are used in attacks, as they are difficult to find and expensive.

Threat Modelling

Threat modelling can be used to guide decision making, by considering who the attackers are and their goals. We should also consider what attacks are likely to occur, and what assumptions the system relies on.

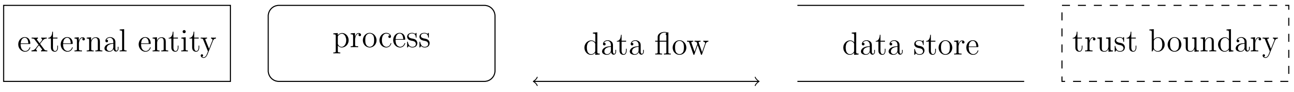
Rather than performing the modelling on the code of the system itself, it's done on the model of the system, thus being free from implementation and deployment details. This allows us to identify better design implementations before the system is built, or can be used to guide the security review of a system after deployment.

There are three key steps;

1. model the system

   This uses consistent visual syntax, to allow for multiple researchers to understand, as well as to build experience. In this course, we focus on **system architecture**, rather than focusing on assets like passwords, credit card numbers, or focusing on attackers.

   Data-flow diagrams (DFD)s are used to depict the flow of information across components. **Trust boundaries** help establish what principal controls what, and attacks tend to cross these boundaries.

   For example, if two processes exist inside the same trust boundary, we generally don't need to be worried about attacks from one process to the other. However, we do need to be concerned about any data flow arrows that cross the boundaries.

2. identify threats (STRIDE / attack trees)

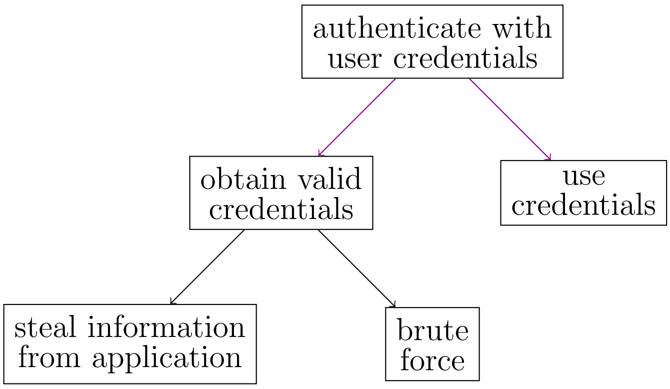
   For STRIDE, we ask what may go wrong in each element of a DFD;

• **s**poofing	pretend to be something else
• **t**ampering	modifying without permission
• **r**epudiation	denying to have performed an action
• **i**nformation disclosure	revealing information without permission

- **d**enial of service	prevent system from providing a timely service
- **e**levation of privilege	achieve more than what is intended

Threats may belong to more than one of these categories, and threats should be document by writing risk-based security tests when possible.

Another approach is to create an attack tree, where the root node represents the goal of the attack, or the target asset. Children are the steps to achieve the goal, and the leaves are concrete attacks; by default, sibling nodes represent **sufficient** steps (only one needs to be satisfied), but special notation is used to represent **necessary steps** (where all need to be satisfied). Note that the course uses lines between the arrows to denote necessary steps, however I will be using matching colours. For example;


authenticate with user credentials
obtain valid credentials
use credentials
steal information from application
brute force

This can also be represented in a textual format, where the root is a bullet point, and the necessary steps are '+', with the sufficient points being '-'.

Attack trees are an alternative to STRIDE, for each element in a DFD, if the goal of an attack tree is relevant, the tree can be traversed to identify possible attacks. Similarly, we can look at previously seen attack trees.

It's important to focus on realistic threats. The threats that should be considered depend on the system being modelled, the budget, and the value of what is being protected.

3. evaluate and address threads (DREAD / META)

The two main approaches for evaluating threats are qualitative (based on insight, experience, and expectations) and quantitative (based on some numerical score). However, quantifying risk is difficult (and realistic parameters are hard to estimate), rare events are also hard to predict (and therefore hard to quantify).

The DREAD methodology is a ranking from 5 to 15, developed by Microsoft;

	rating	high (3)	medium (2)	low (1)
D	damage potential	attacker can subvert full security system, get full trust authorisation, run as administrator, upload content	leaking sensitive information	leaking trivial information
R	reproducibility	attack can be reproduced every time and does not require a timing window	attack can be reproduced but only with a timing window and particular race situation	attack is difficult to reproduce, even with knowledge

E	exploitability	novice programmer could make the attack in a short time	skilled programmer could make the attack	extremely skilled person and in-depth knowledge to exploit every time
A	affected users	all users, default configuration, key customers	some users, non-default configuration	very small percentage of users, obscure feature
D	discoverability	published information explains the attack, vulnerability in most commonly used feature and is noticeable	vulnerability in seldom-used part of product, would take thinking to see malicious use	obscure bug, and users unlikely to work out damage potential

After a thread is addressed, a response should be recommended;

- **m**itigate make threat harder to exploit

  For example, if the threat was password brute-forcing, mitigations could require better passwords or locking accounts after some number of failed attempts.
- **e**liminate remove feature exposed to threat
- **t**ransfer let another party assume the risk

  Continuing with the login scenario, we can use a third party login system. The cost is that the third party has information about customers, and that legal responsibility may still remain (despite technological risk being transferred)
- **a**ccept when other options are impossible or impractical

  If someone was to guess the password on the first try, nothing can prevent it. It's important to keep track that the threat remains active.

Responses should be documented, such as in a project issue tracker.

Week 2

Authentication

The main application of computer passwords are the protection of cryptographic keys or user authentication. Password based authentication is widely used as it is easy to understand, easy to implement, and deploy. Some implementations are as follows;

- **plain-text passwords**

  1. store all credentials in a file (`/etc/passwd` or `/etc/shadow`);

     ```
     alice:foo
       bob:bar
     ```

  2. user gives username and password
  3. check if username is present, if it is; check password matches stored
  4. grant / deny access

  This becomes a valuable target for hackers, as this file alone allows for anyone with the file to impersonate users on the system.

- **encrypted passwords**

  This implementation uses **symmetric encryption**, where the encryption and decryption are done with the same key. The steps are similar to above, however **encrypted** passwords are stored in the file - the remaining steps are the same (except step 3, where we check if the decrypted version of the stored password matches the given password).

  This is more secure than before; where the attack tree now has two children (need to obtain the encrypted file **and** the decryption key).

- **password hashes**

  In contrast to before, this uses a **one-way** hashing function, which should not be reversed. Similar to before, we now store **hashed** passwords in the file. Step 3 now applies the hash function to the presented password, checking that it matches the hash stored in the file. While this is more secure than the previous, it's susceptible to an **offline dictionary attack**, where a large table of candidate passwords and corresponding hashes are built up. A hash in the stolen password file can now be looked up in the **rainbow table**.

- **salted hashes**

  A **salt** is a cryptographically random string, which is combined with the password in the hash. The salted hashes are stored in the file, in the format `username:salt:salted_hashed_password` (where the salt is specific to the user);

  ```
  alice:61C82:2CFAD1C96B8236072823B77EDBF150B1
    bob:8B4D8:7FBA1AFAAB57793255B59A8D596449D3
  ```

  Step 3 now combines the given password with the salt, hashes it, and checks it against the salted and hashed password in the file. The remaining steps are the same.

  It's now impractical to build a rainbow table, as a different dictionary will be needed for each possible salt.

The Linux password file stores passwords in the following format;

`username:password_data:parameters`

Where the `password_data` is stored in the following format;

`$hash_function_id$salt$password`

`hash_function_id`	algorithm
1	`md5`
2a, 2y	`blowfish`
5	`sha256`
6	`sha512`

The problem with passwords is usability; complex passwords are a burden to users. Security questions are also dangerous, as common answers can quite easily be found online via social media. Hints also tend to be chosen such that they easily give away the password. Ideally, we choose a password we can't remember, and don't write it down. However, it's hard for humans to choose and remember good passwords, therefore users tend to use memorable passwords (and users with common interests may use similar passwords).

Because of this, offline dictionary attacks don't need to try every possible passwords; they can start with a dictionary of common words, and then apply rules to generate variants. This can include 'leetspeak', where letters are substituted with similar looking numbers, using a few uppercase letters, and appending common years.

Another issue is password reuse, leading to **online dictionary attacks**. In this situation, attackers submit login combinations to a live authentication system (rather than a stolen password file). Usernames are quite easy to find (as they are public) or can just be email addresses. Previously used passwords are easily to find, where lists of passwords from hacked websites can easily be found.

Defences against this can include limiting the number of attempts per username / IP before blocking access. Another approach is to use CAPTCHAs, preventing simple automation attacks (however it can inconvenience legitimate users). Honeypot accounts can also be made, which are easily cracked. Requests can be blocked from a device attempting to login to one of these accounts.

The best practices to build passwords are as follows;

- filters to select, random looking passwords (force user to use good passwords)
- hash passwords with functions like `PBKDF2` (password based key derivation function) or `bcrypt` (which take long enough to prevent hackers from building rainbow tables)
- don't force users to change passwords often (otherwise users will choose easy passwords)
- don't fail with "user not found"; this allows attackers to find valid users
- block account or requests from same IP after too many attempts
- on a successful long, show information about last login (allows user to report suspicious logins) and notify user if login is from a different machine / location

On the other hand, some practices that could be followed by users (to enhance passwords);

- **password managers**

  Password managers allow users to handle strong passwords for many different websites, as well as avoid phishing sites. However, they are a single point of failure; if the master password is lost, all the other accounts are lost, similarly if a hacker obtains the master password, all passwords are obtained. Online managers are exposed to hackers, whereas offline managers can potentially be unavailable.
- **2FA** (2^{nd} factor authentication)

  2FA prevents attacks based on weak / stolen passwords. However, the main downsides include being locked out of an account without the device, as well as users being given a false sense of security (leading to weaker passwords). It also introduce another device into the user's **Trusted Computing Base**.
- **OAuth or Single Sign On**

  This allows for authentication via a trusted identity provider, such as some social networks, delegating responsibility to a third party. However, this does lead to the cost of giving a third party your user data.

There are also alternatives to passwords entirely, including;

- **hardware tokens**

  Commonly used by banks (creating single use passwords / tokens for logins or transactions). These are expensive and hard to replace.
- **biometric authentication**

  It's impossible to replace if "lost" or revealed (spoofed).
- **RFID tags**

  As a physical object, they are at risk of theft or misplacement. Similarly, due to the nature of RFID, it can be susceptible to proximity based attacks.
- **passwordless authentication**

  A lower value website can be authenticated by the user proving they have access to a certain email. When the user wishes to login, a temporary pin is generated and sent to a given email.

Pentesting

Penetration testing is the process of paying someone to break into a system or organisation, and report the weaknesses (can also include physical security of the building). It's important to scope the pentesting, particularly the goals we are trying to achieve (what's being accessed). Once this is agreed, restrictions on the targets, tools, techniques, and side effects (cannot wipe out an entire database to prove it is vulnerable) need to be discussed.

A pentesting exercise is also defined by the amount of available information;

- **black box** no information, all has to be discovered from a given high-level goal
- **grey box** selected information, e.g; there is an intranet, which contains a database server
- **white box** extensive information about the system, possibly including source code

It's hard to ensure a pentester has tried "hard enough", and one option is to have several teams playing against each other. Certifications (such as *CISSP*) commend higher fees.

PTES (Penetration Testing Execution Standard) are a set of fundamental principles and technical guidelines for pentesting, with the following key steps;

1. pre-engagement interactions sign contract, define scope
2. **intelligence gathering**

   This can be split into two phases;

   - **passive** information gathering

     The aim is to build as much information about the target system without engaging with the target itself. We want to have enough information to build a data-flow diagram of the target, in order to drive the next phase, as well as information about the network structure of the target. However, we don't want to reveal our presence at this phase, and one technique is to prevent any connection to the target (by blocking access through a firewall or proxy).

     One approach is to look for information made public, including possible blog posts from the company which may contain relevant information. From there, we can find any web presence - looking at source code for any links, form fields, as well as references to open source code used (possibly finding bugs or hardcoded credentials) and protocols. While accessing the website should be fine, it's also possible to hide any presence at this point by looking at cached versions of the site only. It's important to note that even publicly accessible data may be protected by law.

     One technique that can be used is **Google Hacking**, which uses search engine operators to locate sites;

     - `ext:pdf` search with specific extensions
     - `site:example.com` search within a given website only
     - `"index.html" inurl:  -html`

       find inside page, but without `html` in the URL (find exposed directories)
     - `allintext:"Powered by phpbb"` locate sites running known vulnerable software

   - **active** information gathering

     We can collect more in-depth information if we are willing to contact our directly, at the risk of being detected (therefore it is better to do it from a different IP than the one used for exploitation). To verify gathered email addresses, emails could be sent to these addresses and checked for any bounces. Network probing can also be done, identifying what subnet addresses are active and what ports accept communications. It's also possible to identify services, by performing **banner grabbing** as some services send identifying information by default, or by reverse-engineering the protocol. It may be sufficient to send random data and to observe the error message.

3. **threat modelling**
4. **vulnerability analysis**

   The target may not be fully patched. From there, we can look in the CVE database for any vulnerabilities in the previously identified components. Automated tools can be used to systematically scan the target (however this generates a large amount of traffic).

   However, if the system is patched, we can attempt to look for new vulnerabilities. If the source code is available, we can use static analysis tools or perform this by hand (which can be very slow). Another approach is to trigger vulnerabilities with educated guesses like SQLi or XSS.

   Another approach is to find credentials, by either looking for default logins or finding password hashes published by hackers.
5. **exploitation**

   In this phase, we actually act on the vulnerabilities identified before. For example; if we have collected credentials, we can attempt to use them to see if they work. We can also run publicly available exploits, either manually (and with our own exploits), or using automated tools such as *Metasploit* (tailored to verified vulnerabilities only, not just everything the tool can do).
6. **post-exploitation**

   If the exploited account isn't an administrator, privilege escalation should be attempted. Typical goals to prove access include;

   - steal data
   - send data back to the hacker
   - maintain access
   - manipulate logs to cover tracks
   - pivot; use host to exploit other targets on LAN
7. reporting

Networks Background

Most of this should be covered in the **CO212** module last year.

LAN Security

We want to clarify the principles of what we consider as legitimate users and attackers on the networks, and the capabilities;

- **participant**

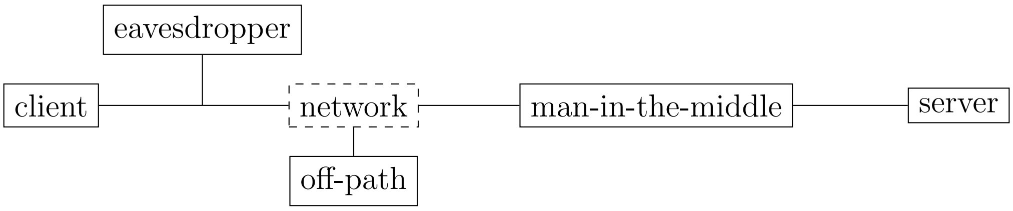
  A participant can send and receive legitimate packets that respect the protocol (for example web browsers and web applications).
- **eavesdropper**

  On the other hand, an eavesdropper can read packets sent to others, and will not / cannot participate. Examples of eavesdroppers include wiretappers and sniffers on a broadcast network.
- **off-path**

  In contrast, an off-path attacker is connected to the same network between the client and the server, however is not in the same local area network (and cannot sniff packets). However, it can participate and create arbitrary packets (which may not abide by the protocols). Examples of this include independent machines connected to the same WiFi.

- **MITM (man in the middle)**

  These are more powerful and completely control the link between one host and the rest of the network. They can participate like a regular participant, but can also read, modify, or delete packets. Examples of this include a proxy, an ISP, a router, or WiFi access point (therefore we should be careful on untrusted networks).



Within the same LAN, devices send messages to each other based on MAC (Media Access Control) addresses. The DHCP (Dynamic Host Configuration Protocol) tells new hosts their IP addresses (and other configuration information). ARP (Address Resolution Protocol) is used to find the MAC of an IP on the same LAN.

As a device typically communicates by asking for data to be sent to a device with a given MAC, LANs typically rely on broadcast medium such as cable (Ethernet) or wireless (WiFi). Conflict resolution requires a minimum packet size, and if this padding data is not properly initialised (either with zeroes or dummy data) - and contains more bytes from the buffer, this may lead to data disclosure. Eavesdroppers hosts can also sniff the network (and we should assume that hosts connected to the network can see whatever we send).

Assume a switch with 3 ports, and devices A, B, and C connected to ports 1, 2, and 3 respectively, where C is the attacker. **MAC flooding** is done in two phases, to force the switch to broadcast traffic;

1. The attacker floods the CAM table with frames with invalid source MACs ($X \to ?$, $Y \to ?$, etc), preventing valid hosts from creating CAM entries.
2. A message $A \to B$ is now flooded out to both B and C, since no CAM entries exist for the valid hosts.

Countermeasures to this include limiting the number of MAC addresses from a single port as well as keeping track of authorised MAC addresses in the system.

Another attack is **ARP poisoning**. By design, MAC is easy to spoof (as a way to deal with conflicting hardware). An attacker can change its MAC address in order to evade access control mechanisms. An off-path attacker spoofing the router can become a MITM. The process for poisoning is as follows;

1. switch needs to find MAC corresponding to an IP
2. attacker spoofs MAC of victim and replies like the victim
3. message is forwarded to both ports that replied (victim and attacker)

Countermeasures include static ARP rules (which can be inconvenient), or to detect spoofed ARP messages (at which point both hosts are kicked off, and an administrator is likely notified).

Week 3

IP Security

The IP (Internet Protocol) is a best effort (may drop / reorder packets) protocol which delivers packets between **source** and **destination** hosts. IP addresses are structured in a hierarchical way and guides routing. However, since these may travel across networks with smaller packet sizes, IP packets can be fragmented. The **D**on't **F**ragment, **M**ore **F**ragments flags indicate the type, the fragment offset field gives the **position** of the fragment in the **original** fragment, and **identification** differentiates fragments for different packets. Different operating systems treat duplicate IP fragments in different ways - this can be used for OS fingerprinting. The TTL (time to live) field is used to discard packets that take too many steps to reach a destination; when the TTL is decremented (at each hop) to zero, the packet is discarded and an ICMP error message is sent to the source.

TTL is used to prevent loops in networks (zombie packets). However, it can also be used by the **Traceroute** algorithm to identify devices on the path to a target. By sending packets with an incrementing TTL (first 1, then 2, and so on), each ICMP error message should come from a host on the path to the destination. This can be used to gather information about a target network (such as firewalls).

The source IP can be easy to spoof (as it is not authenticated). An off-path attacker can send packets with a target IP as the source, leading to the target receiving responses. This is used for attacks such as amplification based DDoS or idle scanning.

The Internet, by design, is a decentralised network of untrusted networks. As such, packets travel through untrusted hosts, hence MITM attackers (such as an ISP) could directly read packets and modify payloads. BGP routing is partly based on trust. As a single AS (autonomous system) cannot track all IP addresses, they must ask each other for a route to reach an IP of a distant AS. They may misbehave (BGP hijacking) by advertising false routes, diverting this traffic, and perform MITM attacks.

There is an ongoing global effort to secure BGP; **MANRS (Mutually Agreed Norms for Routing Security)**, supported by the Internet Society and big players. This specifies best practices for network operators (ISPs), Internet exchange providers (IXPs), content delivery networks, and cloud providers. The **RPKI (Resource Public Key Infrastructure)** is the idea to use public key infrastructure to propagate trust down the address hierarchy. When an AS is looking for a route to reach a system, it will be prevented with BGP advertised routes (as well as certificates, which can only be provided by the owner, or a correct path).

IPsec adds security to IP with two main protocols, in two modes (transport and tunnel, where the former protects the IP payload only, and the latter **also** protects the IP header);

- **authentication header (AH)**

  Preserve packet integrity (recipient will know if tampered with) and protect authentication (recipient will be confident about who the sender is). Packet inspection is allowed, and isn't blocked by firewalls.

- **encapsulating security payload (ESP)**

  This preserves confidentiality of the payload, but may be blocked by security.

ESP tunnel mode is commonly used to implement VPNs. This gives network layer confidentiality, source authentication, data integrity, and replay-attack prevention. In this mode, the protocol is changed from `proto=TCP` in the original IPv4 datagram to `proto=ESP`. After the destination IP address, the original TCP header and payload is replaced with the following;

- SPI (security parameters index)
- sequence header
- IP header
- TCP header + payload
- padding (variable), padding length, and next IP
- (optional) authentication data

The lecture then goes over IPv6 (see **CO212**); we focus mostly on IPv4 in this course.

TCP (more detail in **CO212**) has security issues stemming from an easily accessible state. The sequence numbers are easily predictable, as they are the previous number added with the bytes exchanged;

- a MITM attacker could read the current sequence number and inject new packets (TCP session hijacking)
- an off-path attacker could try to guess the correct sequence number (blind spoofing, read *Off-Path Hacking*)

Typical countermeasures include introducing a time-delay and discarding race-condition packets, or use IDS or protect the payload (e.g. HTTPS).

Port scanning may be a crime (when unauthorised), depending on the legal jurisdiction. The idea is to use the initial steps of the protocol to determine whether a port is open;

- TCP `connect()`

  If the HTTP port is open, sending `SYN + Port 80` to a host will result in a response of `SYN/ACK`. We can then send `ACK` and `RST` to close the connection. However, if a port was closed (say FTP), sending `SYN + Port 21` to a host will result in a response of `RST`.
- TCP idle scan

  To do this, we perform the following steps;

  1. find an idle host (one that is online, but not used actively, such as a printer during the night)
  2. check available IPID on printer by sending `SYN/ACK` to printer, will respond with `RST,IPID=x`
  3. send `SYN,src=<idle host>` to the target (pretend to be target host)
  4. if the port is closed, will reply with `RST` to the printer, or `SYN/ACK` if open
  5. if the port was open (`SYN/ACK`), the idle host will reply to the target with `RST,IPID=x+1`
  6. perform step 2 again; if we get `RST,IPID=x+1`, we know the port was closed on the target, however if we get `RST,IPID=x+2`, we know it is open.

  This can evade some port scanning protection which only monitors connections between internal and external hosts, and not monitoring at an internal level.

UDP is connectionless, which has low overhead and low latency. This can be used for broadcasting or multicasting packets. There is no guarantee that the data reaches the destination, and there is no integrity (optional checksum) - it is up to the application layer to make sense of a UDP stream. See **CO212** for more details.

UDP scans are harder, as we do not expect any acknowledgements (compared to TCP). We can send a generic UDP header with no payload to target ports; if we get a UDP response, the port is open, otherwise if we receive an ICMP error the port is closed (or filtered by a firewall). However, if we timeout without a response, the port may be open (but hosting a service that drops ill-formed packets), or the port may be filtered by a firewall. If we encounter this case, we can probe the port again using UDP packets, but with payloads specific to a protocol (e.g. DNS query). This adds to the difficulty; it is more time consuming (due to a lack of response) and may take multiple attempts to resolve ports. In addition, they are less precise, as some protocols just cannot be probed.

The key threats of TCP/IP are as follows;

- **host and port scanning**

  Used by hackers during active information gathering, with request being hidden within normal network traffic.
- **port sweep**

  Attacker looks for specific service on many machines (likely looking for a vulnerable service). Nowadays, if we encounter a port sweep, it's possible it's a security researcher trying to find the number of accessible webcams (or some sort of insecure IoT device).

- **malicious traffic**

  Normal connection, but may send malicious data that exploits the implementation of the networking stack.
- **(D)DoS**

  Flood target with high volume of network traffic (commonly done with botnets). This either fully takes the target down, degrades performance, or increases costs.

Port knocking is a technique to hide a service form port sanning (either by a system administrator, or an attacker attempting to hide a backdoor);

1. sequential / random scan only finds closed ports
2. client shares a secret with sever, identifying specific ports to probe in a fixed order (e.g. 3,1,2,4) - the last probe is replied to by the server with a random port n, where the service is located
3. client connects to service on port n

Network Defences

Here we take a high level overview of classic network defence systems (mainly firewalls and IDSs (intrusion detection systems)).

The main firewall protects all internet traffic. The internal network is kept separate from Internet-facing services (in the demilitarised zone) such as a web server or FTP server. In the internal network, we may have private databases or sensitive machines; generally data we don't want to share with the outside.

The main goal of a firewall is to enforce security policies on all inbound / outbound traffic for the subnetwork it is trying to protect. A firewall will typically specify what hosts can communicate with what over hosts (and what protocols, or how much data can be exchanged - general properties of traffic). This not only can protect against attacks, but can also control what can be done by hosts. It's common to have a centralised firewall to ensure consistent policies achieving the same goal. Once the general network-wide policies are enforced, there may be other dedicated firewalls for subnets (with modern hosts also having local firewalls). Firewalls can be either;

- dedicated network appliances (with purpose built hardware) *Cisco*, *CheckPoint*
- kernel-level applications (general purpose hosts) `iptables`, `pf`, Windows firewall

They are valuable targets for attackers (if it can be owned, it runs at a privileged position on the victim network / OS).

Policies can be either of the following;

- **packet filters** decision based on individual packet

  This can take into account protocol header fields, source / destination addresses, and ports.
- **stateful filters** take state into account

  Keep track of sessions (for TCP) and check if the packet is part of an established connection. It can also look at timeouts, and the amount of bandwidth used.
- **both** combination, and also support payload inspection

Intrusion detection systems are more powerful than firewalls, and allow for deep packet inspection. This allows for decisions to be made based on the payload (not just headers), and raise an alert (IDS) or drop packets (IPS - intrusion prevention system). They aim to detect or prevent attacks, such as active information gathering (including scans or sweeps), DDoS, worms (as they have a global view of the network), and application layer attacks. The approaches are typically divided into the following;

- **signature based**

  Most common case. They have rules to detect packets that have been observed in the past to be part of / have characteristics of a known attack. Generalising rules (to catch variants) can cause false positives; there is a trade-off between sensitivity and specificity. These are typically human generated. A *ModSecurity* example for detecting XSS is as follows;

  ```
  SecRule ARGS|REQUEST_HEADERS "@rx <script>"
  id:101,msg:'XSS Attack',severity:ERROR,deny,status:404
  ```

  This applies the rule to the arguments or the headers, looking for a regex that contains the script tag, and denies the request with a 404. They are good for matching attacks with known patterns, however they cannot catch unseen attacks. Once a signature is known to be known, it can be easy to bypass (similar to antivirus). They focus on content rather than intent, and are better for stopping automated attacks than manual ones.

  Each rule is simple, however applying the rules to each packet becomes more expensive with more rules and packets. This can be evaded with IP fragmentation. Here we assume there are 10 hops from an attacker to the monitor, and a further 8 hops to the victim (18 hops in total);

  1. fragment a suspicious IP packet in 2 (`ttl=20,seq=6...9: USER` and `ttl=20,seq=10...3: root`)
  2. traceroute to determine distance to IDS and target
  3. send fragment 1 (`ttl=20,seq=6...9: USER`) to reach the target
  4. send a replacement of fragment 2 (`ttl=12,seq=10...3: root`) so that the IDS sees it, but the target does not; note that the TTL is 12, which clears the monitor but expires before the victim
  5. IDS now decides that the communication is safe (it sees `USER|nice`)
  6. send malicious fragment 2 (`ttl=20,seq=10...3: root`), which reaches the target, the target now has `USER|root`
  7. IDS does not interpret the message sent in step 6 as related to the one sent in step 3

- **anomaly detection based**

  This attempts to generalise to attacks that haven't been previously seen. The idea is to learn what normal traffic looks like, and point out statistical anomalies based on features such as the protocol used, packet size, time, order, hosts, etc. It learns a model of benign traffic, with heterogeneous features, either categorical (TCP/UDP/...), or continuous (such as size). It can detect unseen attacks, where there is no existing signature, however it can suffer from false positives (where we have uncommon traffic).

  Simple examples could be packets which are too large, or accesses to high numbered ports which are not typically used (which are typically used as source ports) - these are **point** anomalies, which is one sample being anomalous with respect to others. On the other hand, **contextual** anomalies are only anomalous in specific contexts, but not in others. For example, high bandwidth usage may be an anomaly at night but not during the day. Furthermore, there are **collective** anomalies, which are anomalous with respect to all available samples; for example, a TCP connect scan to the same port of many hosts could be detected as port sweep.

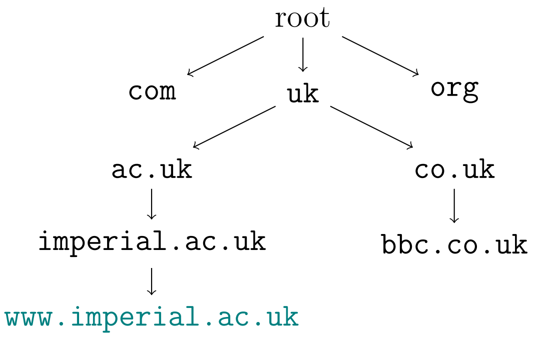
  Models used typically are statistical (non-parametric for histograms or PCA), or parametric (regression), or are classification based (Bayesian networks, neural networks, SMVs, or random forests). Training is commonly semi-supervised, with an unlabelled dataset consisting of only normal events. An unsupervised training mode is unlabelled data possibly containing some anomalies, and a supervised training contains labels for both normal and anomalous events.

- **specification based**

  Logical rules / simple languages dictate whether a packet should be accepted or rejected. However, it's hard to define rules, avoid conflicts, and may be inconsistent.

DNS (Domain Name System)

The DNS allows us to identify hosts via easy to remember, descriptive hostnames, rather than IP addresses. This separates the logical address of a service from the physical address of the host (which allows for hostnames to stay the same when network providers are switched). We need to know the IP address of the target hostname, which can be obtained by a local DNS client / resolver in DNS resolution. Once a name is cached, it will be valid for a limit amount of time. If a name is in the cache, we can use that, otherwise the resolver queries an external primary / recursive DNS server. Typically, they are short queries and responses, fitting into a single UDP packet (512 bytes) - if more data needs to be exchanged, it falls back to TCP. Domain names are organised in a hierarchical manner, DNS is managed by ICANN/IANA, who also run the root DNS servers (note that `com` is a gTLD, and `uk` is a ccTLD (country));



A domain name which can be used to fully access a host is called a fully-qualified domain name (FQDN). The process of DNS resolution, when the local cache doesn't have it stored is as follows;

1. user asks primary DNS server for IP address of `example.com`
2. primary DNS server asks root server for **location of** IP address of `example.com`
3. root server tells primary DNS server to ask `.com` namespace
4. primary DNS server asks `.com` namespace for IP address of `example.com`
5. namespace tells primary DNS server to check primary DNS server of `example.com`
6. primary DNS server asks primary DNS server of `example.com` the IP address of `example.com`
7. primary DNS server of `example.com` tells user's primary DNS server the IP address of `example.com`
8. user's primary DNS server tells user IP address of `example.com`

Common DNS records include;

resource record	description
`A`	mapping of a name to address (IPv4) - performs primary function of DNS; converting names to addresses
`AAAA`	same as `A`, but for IPv6
`NS`	resolver doesn't know, instead replies with a name server which knows (identifies DNS server function as authority of the zone, each DNS server must be represented by a NS record, whether primary, master, or secondary)
`SOA`	identifies which server is the primary one (start of authority); each zone must have an SOA record and only one SOA record can be in a zone
`PTR`	(pointer) does reverse of `A`; provides a mapping from address to name

`CNAME`	creates an alias that points to canonical name (real name) of a host identified by an `A` record - can be used to check cache for the real name
`MX`	identifies a system that will direct email traffic (mail exchange)
`NXDOMAIN`	the name cannot be resolved, it is a non-existent domain (not registered or invalid)

This structure leaves space for a MITM attack, recently done by the Turkish government in March 2014 (to block *Twitter* access) by forcing ISPs to respond to DNS queries for `twitter.com` with the IP of a government website. *DNSpionage* in 2019 had malicious actors compromise DNS resolution of key infrastructure, to spy on emails of certain individuals. Techniques included compromising DNS provider's admin panels (by simple brute force), changing the `A` records of target mail servers. Other cases involved hacking the TLD and changing the NS record to a rogue NS (providing legitimate address normally, but giving a malicious one for `MX` records). This lead to queries for target mail servers coming from victim IPs to a rogue mail server, and legitimate answers for other queries.

Similar to TCP, DNS requests and responses are not authenticated. Attackers can map trusted domain names to malicious IP trivially with MITM (even some ISPs replace `NXDOMAIN` with adverts). Off-path attackers on LAN may be able to inject spoofed DHCP packets and advertise a malicious DNS resolver, or inject spoofed replies to DNS queries (after seeing the query ID). Routers can also be compromised to advertise malicious resolvers (*DNSChanger* malware). DNS cache poisoning is when spoofed responses are being kept by intermediaries, giving a window of time. Name servers can also be hacked.

DNSSEC protects the authenticity and integrity of DNS records with zones that have public / private keys. This forms a chain of trust; starting with the DNS root (the resolvers know public keys of root nodes). The parent nodes then use private keys to sign hashes of the child's public keys, allowing resolvers to check the authenticity of a node's public key. DNS resolution nodes then sign zone data with its private key, allowing resolver check the authenticity of a DNS reply. This may become the standard as more services support it. This however leads to increased load on DNS servers, as well as decreased network performance due to records being longer (and therefore requiring TCP fragmentation).

If a domain does not exist, an `NSEC` record reveals alphabetically-closest neighbours as it proves that the domain does not exist (no further (expensive) queries needed). For example, if we want to resolve `bob.example.com`, the response may be that no records exist between `alice.example.com` and `charlie.example.com`. However, this helps a hacker gather intelligence; we now know that `bob` doesn't exist, and the closest ones are `alice` and `charlie`. NSEC3 mitigates this with salted hashes of domain names (note that the far-right column is sorted by **hash**);

`hash(alice|65BF)` = `F34DDF56`	`4EE23198`
`hash(bob|65BF)` = `7B03235D`	`7B03235D`
`hash(charlie|65BF)` = `4EE23198`	`D14DEA64`
`hash(zoey|65BF)` = `D14DEA64`	`F34DDF56`

Now, we have the following;

1. resolve `bob.example.com`
2. response is that no records exist between `4EE23198` and `D14DEA64`, with a salt of `65BF`
3. this still works as a proof of non-existence, as we can check that `4EE23198 < hash(bob|65BF) < D14DEA64`

The salt works to hinder dictionary attacks, as this can change over time and across zones.

DNS tunnelling has the goal of bypassing a firewall / proxy preventing HTTP communication with the target. For this, we will have the hacker (can be a compromised computer / device the hacker has left connected), the company internal DNS, and the hacker authoritative DNS - which is outside of the firewall;

1. attacker encodes data to be sent in a DNS query for a domain, where the authoritative DNS is controlled by the attacker (for example querying for `x123.attacker.com`)
2. local DNS resolver cannot find it, eventually contacts authoritative server
3. DNS queries (to non black-listed domains) are not filtered by the firewall
4. server replies, encoding data in DNS response
5. firewall forwards response to internal DNS
6. attacker receives and decodes the reply

The simplest version of this exfiltrates data encoded as subdomain-names. An advanced version would be to use a DNS SOCKS proxy to browse any website (slowly) - all encoded via DNS.

Malicious domain registration exploits the value of a particular domain (drive traffic to oneself for ads, or to use trust for malware / phishing);

- **cybersquatting**

  Register a trademarked name to sell for a higher price to the legitimate brand owner.
- **typosquatting**

  Register names that are are a few typos away from a legitimate name, where visitors may visit by mistake. Defensive registrations are now common, for example `goolge.com` redirects to `google.com`.
- **bitsquatting**

  This is less common, as error rates are low (3 bit flips per month on 4GB DRAM) - this can be higher on old hardware, without ECC, or on airplanes. This relies on accidental bitflips in memory or on the wire; `amazon.co.uk` versus `a-azon.co.uk` (where `m` is one bit flip away from `-`).
- **dropcatching**

  Dual of cybersquatting; register domain after it expires to sell back to owners, or to exploit existing trust in the domain.

Malware can also use domain names; if a malicious IP is present in malware, it cannot be replaced easily, whereas a domain can be. Domain generation algorithms can be used to create sequences of candidate names for C&C, each contacted sequentially until one responds (only a few of these need to be registered when needed). Random looking names are easy to generate (and are cheap), but can be quite easy to block by IDS. Dictionary based names cannot be easily blocked as they may be legitimate.

Administrators may lose control of NS pointers, by expired registration, mistyped names, or bitsquatting. In 2017, a security researcher registered a dangling (one was mistyped) NS name for the `.io` zone, allowing for a 25% chance that they could control any `.io` resolution.

TLS (Transport Layer Security)

This is a protocol specification that describes how we can protect the confidentiality and integrity of the data we are sending over protocols such as TCP or UDP. An eavesdropper will see ciphertext they cannot interpret, and if a MITM were to tamper with the data, the receiver would see that it fails an integrity check (and therefore detect it).

The main usage is to protect HTTPS traffic and email. The specification states that TLS should be sent over a reliable medium (normally TCP/IP); DTLS (TLS over UDP) is not widely used, but exists. This only protects the TCP payload data (the IP and port are **not** protected, as they are in the header). The latest version (as of writing) is 1.3, but the most adopted is 1.2. Before TLS, there was SSL (now the word SSL is commonly misused to mean TLS).

A TLS server needs a certificate to state the identity of the principal participating in a TLS exchange, and also declares its public key. This is typically in the **X.509 Public Key Infrastructure Certificate** (specified by IETF RFC 5280), which has the following main attributes;

- **issuer** typically a certificate authority (CA)
- **validity** start and end dates
- **subject** identifies the owner (typically explicit domain name, e.g. `imperial.ac.uk`)
- **subject alternative names** single certificate can cover multiple domain names

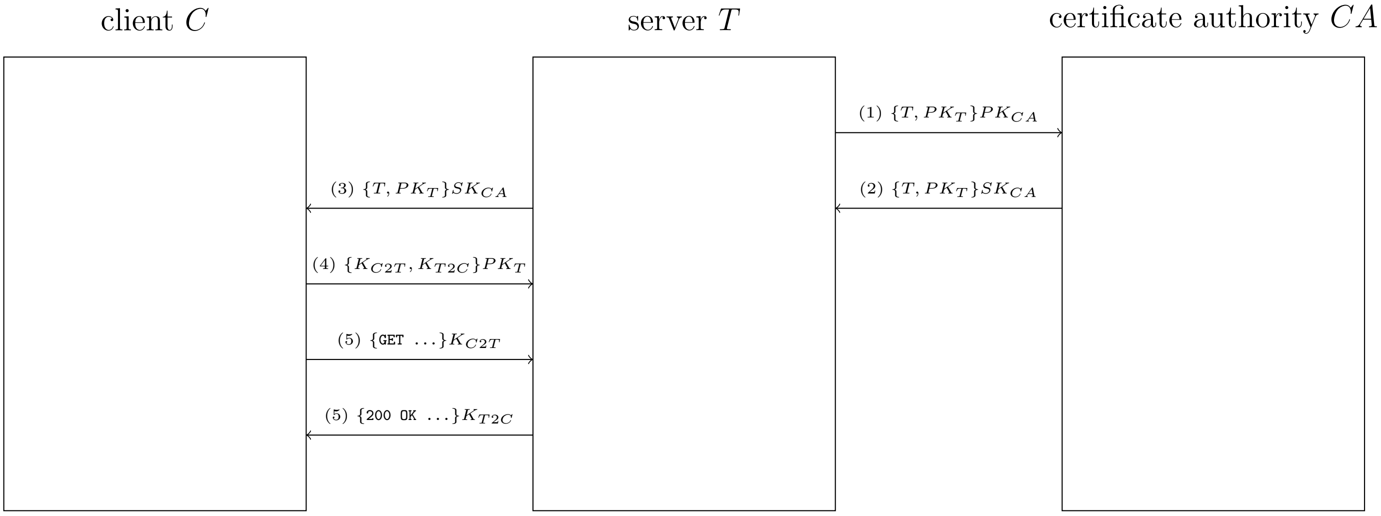
  For example; `*.ic.ac.uk` will match `doc.ic.ac.uk` but not `cate.doc.ic.ac.uk`.
- **subject public key info** contains actual public key
- **certificate signature value** issuer signs the certificate body (ensuring integrity)

Similar to DNSSEC, TLS also uses certificate chains. The TLS client will typically trust a number of widely know CAs. The TLS server may send a certificate chain so the client can verify ownership; for example, `imperial.ac.uk` could be a CA for a subdomain.

We will use the following notation;

- $[SK_A, PK_A]$ is a keypair of cryptographically related keys
  - SK_A is a secret (signing) key of principal A (and kept secret)
  - PK_A is a public (verification) key (can be revealed to public)
- $\{\texttt{msg}\}PK_A$ denotes asymmetric **encryption** using public key PK_A (therefore only A can decrypt this) - this provides confidentiality (protection from eavesdropping)
- $\{\texttt{msg}\}SK_A$ denotes a **signature** using secret key SK_A (only A can generate this) - this provides integrity (every can 'decrypt' this with A's public key)
- $\texttt{decode}(K_1, \{\texttt{msg}\}K_2) = \texttt{msg}$ iff $[K_1, K_2]$ or $[K_2, K_1]$ are valid keypairs
- $\{\texttt{msg}\}K_L$ denotes symmetric (lighter and faster) **encryption** using symmetric key K labelled L

Note that C and T both have PK_{CA}, T has the keypair $[SK_T, PK_T]$, and CA has the keypair $[SK_{CA}, PK_{CA}]$.


client C
server T
certificate authority CA
(1) $\{T, PK_T\}PK_{CA}$
(2) $\{T, PK_T\}SK_{CA}$
(3) $\{T, PK_T\}SK_{CA}$
(4) $\{K_{C2T}, K_{T2C}\}PK_T$
(5) $\{\texttt{GET} \ldots\}K_{C2T}$
(5) $\{\texttt{200 OK} \ldots\}K_{T2C}$

This has the following steps;

1. server needs to obtain certificate from the CA, will send own name and public key (which the client doesn't yet have) to the CA, encrypted with the CA's public key (therefore only the CA can decrypt it)

2. the CA validates the identity of T, and then provides a **signed** message stating that it has been verified (in some way)
3. the server now has its own certificate, when the client tries to send data to the server, it will obtain the signed certificate, which can be validated as the client has the CA's public key
4. the client chooses two symmetric keys to be used for communication and sends it encrypted with the server's public key (therefore only the server can decrypt it)
5. exchanges now happen with these symmetric keys

The TLS 1.2 handshake is built on TCP, after the `ACK`, with the following main messages (none of which are encrypted, other than the session key as before);

- `ClientHello` advertising capabilities of the client (what encryption / compression can be done, and what protocols)
- `ServerHello` server takes the available option and chooses the most suitable one
- `Certificate` server sends chain of TLS certificates (which can be validated by the client)
- `ClientKeyExchange` allows server to compute symmetric session key
- `ChangeCipherSpec` further messages will be encrypted with session key

Since the payload of what is sent over TLS is encrypted, the receiving server cannot really know what website we are trying to access; we have an IP and a port in the header, but we are not revealing what domain name we are connecting to. There is an extension (SNI - server name indication) of TLS which can also communicate what domain name we are connecting to on a specific host, which also helps the server to decide what certificate to show. Clients should check that the certificate name matches SNI. Without SNI, the server would have to provide the same certificate for all hosts on the server (under the same IP), and they may be unrelated to each other (problem of mutual trust between domains, and leads to large certificates, also requires revocation when sites are added or removed). However, with SNI, the client also requests the site, and the server shows a specific certificate.

Trust is required for TLS to work. By default, well known CAs are already trusted by clients. `letsencrypt.org` provides free certificates and automated renewal scripts. Self-signed certificates require the server asking the client to just trust the public key using a different channel (either by installing the key or some configuration parameters). Clients can trust a custom CA as a whole; which is used to enable proxy to inspect TLS traffic in the clear (will be done in labs). A MITM proxy will do the following (note that all communication between the client and the proxy is done with a key X, and all communication between the proxy and server is done with a different key Y);

1. client connects to server, through proxy
2. the proxy forms an SSL connection to the server
3. the server gives a certificate to the proxy, completing the SSL connection
4. the proxy generates a **fake** certificate based on the server's certificate
5. the proxy sends the fake certificate to the client
6. the client sends a HTTP request to the proxy
7. the proxy decrypts content from the client
8. the decrypted content is sent to the server, and a response is retrieved
9. the HTTP response is forwarded to the client

These protections are ineffective when certificates cannot be trusted - compromised CAs can sign spoofed certificates (giving attackers a very powerful attack vector against any TLS connection).

Parameters need to be verified in order to issue a new certificate;

- **extended validation (EV)**

  Done in person / via lawyers / by phone, in a more time consuming way. This proves the identity of the owner / organisation in a stronger (more trustworthy) way. Deprecated now.

- **domain validation (DV)** most common

  Domain owner proves control over domain, when asking for a certificate, mostly internet based. Once a CA generates a random token, the owner does the following;

  - owner places token in DNS record for domain
  - owner servers token at specific URL for domain
  - owner includes token in fresh TLS certificates served from the domain
  - CA emails token to owner, who submits a challenge to CA online

    The email the CA sends to is under that domain, which can be used to prove ownership if the recipient can show the token.

  However, attackers can compromise the DV process to obtain certificates. With IP spoofing, a DNS reply can be faked in order to pretend to own a domain. Similarly, DNS hijacking can be used to serve a token from a controlled host, and obtain a certificate on behalf of that domain. The main mitigation is to use DNSSEC or encrypted emails. These are examples of threat transfer.

Compromised and rogue CAs break the trust in TLS. All created certificates must be reported publicly (Merkle hash trees, key idea of blockchain). Domain owners can monitor logs, detect rogue certificates, and have them removed. Another option is to use DANE, and rely on DNSSEC. When a client queries a domain name, they also obtain valid certificate authorities (that we can trust) from the DNS server. We now trust the DNS operator instead of the chain of certifications. In the TLSA DNS records, we have the following levels of trust;

- 0 (CA specification) well-known public CA that is trusted
- 1 (specific TLS certificate) trust this certificate (if it passes verification)
- 2 (trust anchor assertion) trust this new CA (we do this manually to install MITM proxy)
- 3 (domain-issued certificate) trust self-signed certificate (include information in TLSA record)

TLS can leak information via traffic analysis; BEAST (CVE-2011-3389) compromises TLS 1.0 via RC4 leakage, CRIME (CVE-2012-4929) compromises SPDY via compression ratio. Implementations bugs, such as in OpenSSL, can also cause issues - see HEARTBLEED (CVE-2014-0160) causing data disclosure by buffer overrun (disclosing information from memory, including keys). Formal analysis of the TLS state machine showed that the TLS client can be forced to use a weak ciphersuite (FREAK; CVE-2015-0204).

The change from TLS 1.2 to TLS 1.3 improves efficiency. TLS 1.2 requires 3 round trips to establish a session, and 2 to resume from accidental interruption. TLS 1.3 uses one less in each case (over TCP), with TCP Fast Open - no round trips are needed to resume. This is done by saving state (cookies) when a connection is established - `SYN,Cookie=ck` is sent, and if this is accepted, the first few exchanges are done with the old key, which is then replaced to be more secure.

TLS 1.3 also addresses all recent TLS vulnerabilities, mainly removing weak crypto suites (such as MD5, SHA-1, DES, RC4, and CBC encryption mode). TLS-level compression is no longer allowed, and downgrade attacks can be detected. The handshake is now also encrypted, with the `ClientHello` including the client public key. ENSI (making it possible to encrypt SNI) and an encrypted server certificate prevent MITM learning about the target domain; while this improves privacy, it can cause issues with IDSs; MITM IDS could filter TLS 1.2 traffic based on policies, but with the aforementioned improvements, filtering is much harder.

TLS can also be abused by attackers; it can be used to hide infection and C&C traffic (since the IDS can't check for signatures). DLP (data loss prevention) system are also prevented from checking outbound data for corporate secrets. Data can also be exfiltrated by encoding it as certificates - a compromised host in the network will send data as certificates when an outside client attempts to connect. TLS fingerprinting can be used to detect this, however it is not difficult to spoof the fingerprint (as well as the usual problems of blacklisting).