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.

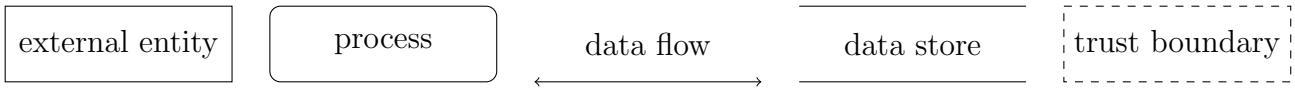
Rather than performing the modelling on the code of the system itself, it's done on the model of the system, thus being free from implementation and deployment details. This allows us to identify better design implementations before the system is built, or can be used to guide the security review of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

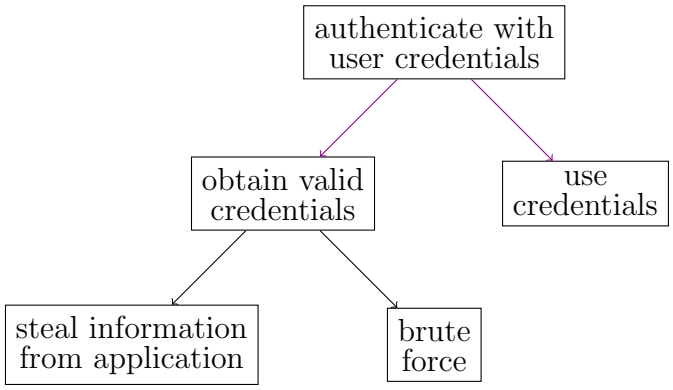
   For STRIDE, we ask what may go wrong in each element of a DFD;

   - **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;



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

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

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

3. evaluate and address threads (DREAD / META)

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

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

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

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

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

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

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

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

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

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

Week 2

Authentication

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

- **plain-text passwords**

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

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

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

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

- **encrypted passwords**

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

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

- **password hashes**

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

- **salted hashes**

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

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

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

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

The Linux password file stores passwords in the following format;

`username:password_data:parameters`

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

`$hash_function_id$salt$password`

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

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

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

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

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

The best practices to build passwords are as follows;

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

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

- **password managers**

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

- **2FA** (2^{nd} factor authentication)

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

- **OAuth or Single Sign On**

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

There are also alternatives to passwords entirely, including;

- **hardware tokens**

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

- **biometric authentication**

  It's impossible to replace if "lost" or revealed (spoofed).

- **RFID tags**

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

- **passwordless authentication**

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

Pentesting

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

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

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

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

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

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

   This can be split into two phases;

   - **passive** information gathering

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

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

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

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

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

   - **active** information gathering

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

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

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

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

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

5. **exploitation**

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

6. **post-exploitation**

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

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

7. reporting

Networks Background

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

LAN Security

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

- **participant**

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

- **eavesdropper**

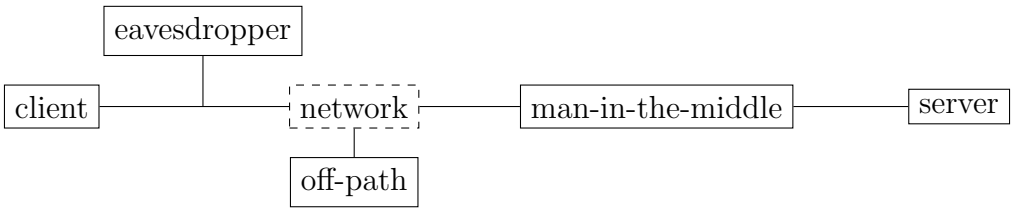
  On the other hand, an eavesdropper can read packets sent to others, and will not / cannot participate. Examples of eavesdroppers include wiretappers and sniffers on a broadcast network.

- **off-path**

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

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

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



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

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

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

1. The attacker floods the CAM table with frames with invalid source MACs ($X \rightarrow ?$, $Y \rightarrow ?$, 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).