Contents

1 Principles of security

1.1 Confidentiality, Integrity and Availability

Classically the goals of information security are defined as:

- **Confidentiality**: information should only be accessible by authorised parties. Tools include **encryption**, **access control**, **authentication** and **physical security**.
- **Integrity**: information is authentic and complete, because only authorised parties can modify it. Integrity can be assured through **backups**, **checksums** (or **hashes/digital signatures**) and **metadata** (who owns or has the right to modify data).
- **Availability**: authorised users can access information and systems when they need to. This requires **protection of infrastructure**, **security policy** to minimise the interference from attacks on legitimate users and **computational redundancies** to cope with hardware or software failure.

1.2 Saltzer and Schroeder principles of security

The eight **design principles** proposed by Saltzer and Schroeder are:

1. **Economy of mechanism**: the design should be as simple and small as possible. Although less feasible in modern commercial systems, it can be important as larger designs can have more errors (or bugs) in them that can become attack vectors, and removing them is more difficult.
2. **Fail-safe defaults**: access decisions should be based on permission rather than exclusion. It is better to deny access to a legitimate users than allow it to a malicious one, particularly considering a legitimate user will report any errors in access control policy (a malicious user would evidently not), allowing for quick and safe rectification.
3. **Complete mediation**: every access to every object must be authorised. This forces a system-wide view of access control, including initialisation, recovery, shutdown and maintenance procedures.
4. **Open design**: the design should not be a secret. Security should depend on good security policy, not the ignorance of attackers. Protection **keys** should be **decoupled from the rest of the design** to facilitate review of the design without compromising security. This is particularly important for systems that are widely distributed.
5. **Separation of privilege**: a protection mechanism that requires **two keys** to unlock is more robust and flexible than one that requires a single key. This is simply because the two keys can be separated physically, and can be handled by separate programs, individuals or organisations. A **single failure** in security policy is **not sufficient to compromise protected information**.
6. **Least privilege**: every program and user should operate with the least set of privileges they require. This minimises damage that can result from error, and minimises the number of potential interactions between privileged programs so that the chance that privileges can be used in unintended ways is minimised. If a misuse occurs, auditing is easier as less users and less programs could have been responsible.
7. **Least common mechanism**: the resources shared or depended upon by more than one user should be minimised. Each shared mechanism represents a potential path of information between users that could result in a breach of confidentiality. As well as this, from a design perspective, serving one user sufficiently with a given set of resources is easier than serving many, as the environment is more controlled and mistakes are easier to handle.
8. **Psychological acceptability**: the human interface must be design for **ease of use**, so that users routinely and automatically apply the protection mechanisms correctly. This will minimise mistakes. The closer protection mechanisms are to what a user sees as necessary, the less likely it is that mistakes will occur. For example if multiple passwords are required, a user would be likely to start writing them down as they might see only a single password as necessary or manageable.

2 Cryptology

2.1 Hashing

2.1.1 Properties and usage

Hashing functions are one way functions that are easy to compute but difficult to invert (it is easy to generate a digest $h = H(m)$ but difficult to obtain m given h). Such functions are useful for authentication as they allow a system to store encrypted passwords. Password hashes are compared for verification rather than raw passwords, and if I server is compromised the attacker gains access to hashes, not passwords. Hashing functions should ideally be:

1. **Preimage resistant**: it should be computationally infeasible to invert the hash. Longer hashes are more resistant to being inverted with **brute force** (providing random input until a matching hash is produced).
2. **Collision resistant**: two inputs should ideally never produce the same hash, or the chance should at least be minimised. This prevents **birthday attacks** where an attacker attempts to find two inputs that produce the same hash, and could for example use the wrong password (that produces the same hash as the right one) to gain access to an account. The **probability of finding a collision** for a b-bit hash function after $2^{\frac{b}{2}}$ attempts is 50%.

In the case of **passwords** it is also a possibility that two users will have the same password. Therefore if one of these users were to gain access to hashes of passwords, they would be able to identify other users with the same password and access their account. To prevent this **password salt** is used. A random string, unique to each user, is concatenated to the password prior to hashing, resulting in a longer **salted password**. This guarantees that each user has a unique password, and helps protect against **dictionary and rainbow attacks** as it increases the length of passwords.

2.1.2 Hash chains and rainbow tables

The optimal way of inverting a hash would be the use of a lookup table containing mappings of all possible (or at least all likely) hashes to their original plaintext equivalents. This would require a **massive amount of memory**. A solution to this is **hash chains**, that are effectively a way of **compressing mappings** between plaintexts and hashes. This is achieved by building **chains** that move between **plaintexts** and **hashes**. A **reduction function** R is used to convert hashes to plaintexts (not invert them, reduction is part of building the chain). A reduction function can be any function that converts a hash to a valid plaintext, for example the first 10 characters of the hash might be taken if you know all the passwords you are trying to crack are 10 characters. Chain i of length j is built for hashing function H as follows:

1. Generate random valid plaintext $m_{i,1}$ and set the 'current message number' c to 1
2. Compute $H(m_{i,c})$ to produce $h_{i,c} \in H$
3. Compute $R(h_{i,c})$ to produce $m_{i,c+1}$
4. Set $c = c + 1$, return to *2* unless c is equal to j, in which case store the start and end of the chain, $m_{i,1}$ and $m_{i,j}$ **OR** $h_{i,j}$ (either would work, original definition suggests the plaintext, the lecture notes suggest the hash)

Given that many chains have been computed the following process can then be used to invert a hash:

1. Take the hash h_x you want to invert, set c to 1
2. Check if h_c is identical to the end of any chain, if it is go to *4*
3. Compute $R(h_c)$ to produce m_{c+1}, compute $H(m_{c+1})$ to produce h_{c+1}, set $c = c + 1$ and return to *2*
4. Get the start of the chain and use H and R to rebuilt it until h_x is reached, storing the plaintext used to produce each hash until the next plaintext is generated so m_x can retrieved once h_x is reached

However **collisions** where two plaintexts map to one hash can become a serious problem in terms of efficiency, as two chains can converge into one whenever the same hash is reached by two different plaintexts, as the remainder of the chain will be identical. This is resolved by the application of **rainbow tables**, where different reduction functions are used at each stage of a chain (R_i applied to h_i for $i \in \{1..j\}$), which means if the same hash is reached at different stages in a chain they are reduced to different plaintexts, preventing convergences due to collision.

2.2 Encryption and decryption

Encryption and decryption algorithms must be **bijective**. For every plaintext (element of set M) there is a unique ciphertext (element of set C) and vice versa.

- $f : M \to C$ is the **encryption transformation**
- $f^{-1} : C \to M$ is the **corresponding decryption transformation**

A **cryptanalyst** could know parts of M, C or both and can perpetrate different kinds of attacks to find encryption key k using either ciphertext, or known pairs of plaintext and ciphertext.

2.3 Symmetric encryption

Stream ciphers are an example of **symmetric encryption**. Each digit in a **plaintext stream** is encrypted with a digit of the **keystream**. Examples of stream ciphers include **Caesar** and **Vigenère** ciphers as well as **one time pad**.

Block ciphers encrypt blocks of text, padding if necessary. Examples include the **Playfair** and **Hill** ciphers.

2.3.1 Vigenère cipher

The Vigenère ciper uses the representation used by most stream ciphers, that 'A' = 0, 'B' = 1 ... 'Z' = 25, allowing for modulo 26 arithmetic. A **running key** is used, in which the complete key is a string repeated to the length of the plaintext. For example the full key for initial key 'run' used to encode string 'hello' is 'runru.' Encoding then takes place using:

$$C = P + K \ mod \ 26$$

The plaintext is encoded one character at a time, using an offset value taken from the keystream (a stream is a string provided 1 character, or other unit, at a time) to produce the ciphertext. The **Caesar cipher** is identical to this, except the **keystream used** is the **alphabet**.

2.3.2 Hill cipher

In the **Hill cipher** (a block cipher), the same alphabetic representation as the Vigenère cipher is used. Messages and represented as column vectors of size n, and the **encryption key** as a $n \times n$ matrix. The **decryption key** is the inverse of the **encryption key**. To encrypt and decrypt the relevant key is multiplied by the relevant text, and the result is taken modulo 26. For example to encrypt 'HI' with the key below:

$$\begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix} \begin{pmatrix} 7 \\ 8 \end{pmatrix} = \begin{pmatrix} 23 \\ 22 \end{pmatrix} mod \ 26$$

So 'HI' with this key encrypts to 'MM.' To decrypt:

$$\begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} -\frac{1}{3} & \frac{2}{3} \\ \frac{2}{3} & -\frac{1}{3} \end{pmatrix} \qquad \begin{pmatrix} -\frac{1}{3} & \frac{2}{3} \\ \frac{2}{3} & -\frac{1}{3} \end{pmatrix} \begin{pmatrix} 23 \\ 22 \end{pmatrix} = \begin{pmatrix} 7 \\ 8 \end{pmatrix} mod \ 26$$

2.4 Playfair Cipher

This is a **block cipher** where the block size is always two. The **key** consists of a **5 x 5 grid**, the letter j in text is replaced with the letter i, and each position on the grid is filled with a unique non-j letter. Repeated letters are separated with x, for example $ee \rightarrow exe$ and if, after this, the number of letters is odd z is appended to the end so as to make the total number of letters even. There are 3 rules for **encryption of a pair**:

1. The letters are on the **same row** on the grid: each letter in the pair is replaced with the letter to its right on the grid. If it is the furthest right on the grid, it is replaced with the furthest left, as rows and columns wrap.
2. The letters are in the **same column** on the grid: each letter in the pair is replaced with the letter beneath it. Again wrapping applies so the bottom letter in a column is replaced with the top.
3. The items are on **different rows and columns**. The two letters form the corners of a rectangle on a grid. Swap each letter with the letter that is in the **horizontally opposite** corner of the rectangle.

Intuitively, for decryption, rules *1* and *2* are inverted. For rows the letter to the left is taken, for columns the letter above. Decryption where *3* applies is the same as encryption, which reverses the initial swap made for encryption.

2.5 Asymmetric encryption

For symmetric encryption, the encryption and decryption keys are the same. For **asymmetric encryption** a **public key** is used by anyone to encrypt a message, that can only then be decrypted with a **private key** held by the individual who generated both keys. Such encryption methods depend largely on **prime numbers** and **modular arithmetic**.

2.5.1 The RSA algorithm

The RSA (Rivest, Shamir, Adleman) algorithm is as follows:

1. Choose two prime numbers p and q, and compute $n = pq$
2. Compute $\varphi(n) = (p-1)(q-1)$
3. Choose e, where $1 < e < \varphi(n)$ and $\varphi(n)$ and e are coprime (greatest common divisor of 1), the easiest way is to choose a prime number and then check it is not a divisor of $\varphi(n)$
4. Compute d where $de = 1 \ mod \ \varphi(n)$ (can be done using the **extended Euclidean algorithm**: resolve d for $e \times d + \varphi(n) \times y = 1$, y is also found by applying the Euclidean algorithm but is unused)

With the **public encryption key** e (e and n are made public) and **private decryption key** d, encryption is achieved using $c = m^e mod \ n$ and decryption using $c = m^e mod \ n$. The algorithm is so good because to obtain d, $\varphi(n)$ and therefore p and q are required, that can only be obtained by factoring n. There is no way of doing this in a computationally feasible timeframe for large numbers (the largest that has been factored is 768 bits, taking 2000 CPU years and factoring time increases exponentially with input size).

RSA has a number of **mathematical vulnerabilities**:

- An attacker only has to search for p and q that are smaller than $\sqrt{n}$, and so if p and q are too small factorising times become computationally feasible.
- If e is small, and the same message is sent with different keys, n for each message (which is public) combined with each encrypted message could be used to find the message (although no keys).
- If d is small ($d < \frac{1}{3}n^{\frac{1}{4}}$) e, n can be used to retrieve it mathematically through Wiener's attack.

- If n is common for all entities, then knowing **any** key pairs (e, d) allows n to be factored because of the mathematical relationship between n, e and d. Once p and q are known, any e can be used to find d.
- If $P = NP$ then RSA is intrinsically flawed, as polynomial complexity factorising of n is possible, the method is just currently unknown.
- If **quantum computing** progresses sufficiently, Shor's algorithm could be used to retrieve the prime factors of n in polynomial time, however the largest number factored in this way to date is 15. **Quantum encryption** provides superior methods of communicating securely.

With the exception of potential theoretical limitations, the best way of overcoming mathematical vulnerabilities is to use a large p, q, e and d (and therefore n), using a different n for each entity that is encrypted. RSA can also have **implementation vulnerabilities**:

- RSA is **deterministic**, the same plaintext will produce the same ciphertext, so if A sends a message, B could try encrypting likely messages until one matched, at which case they would know what was sent. This can be prevented with **random padding**.
- **Timing attacks** exploiting the fact that repeating squares is used for decryption efficiency. d can be read one bit at a time, as each bit is iterated through where d is 1 execution will take longer (would be a very complex attack). Similarly **fault insertion attacks** would rely on manipulating decryption by inserting errors or faults to exploit bugs.
- **Security policy** if d is not kept secret, p or q aren't destroyed, or the plaintext is simply revealed through failures elsewhere.

Protection involves ensuring keys are stored securely, plaintext is destroyed once encrypted, encoding messages prior to RSA encryption and protecting against **side-channel attacks** (analysis of the physical operation of a cryptosystem) with specific APIs or modules.

2.6 Hybrid cryptosystems

Asymmetric cryptosystems are **computationally expensive**, and so quite often they are only used to **establish symmetric keys**. This allows for the efficiency of symmetric systems to be combined with the security of asymmetric systems.

2.6.1 Diffie-Hellman key exchange

Let p be a prime number, and g be a **generator** (primitive root) that both A and B know. A chooses a private integer x, B chooses a private integer y.

1. $A \to B : g^x mod\ p$
2. $B \to A : g^y mod\ p$
3. **Both compute symmetric key** $K = g^{xy} mod\ p$
   - A takes $(g^y)^x\ mod\ p$
   - B takes $(g^x)^y\ mod\ p$
4. $A \to B : \{M\}_K$

The security of Diffie-Hellman depends on the difficult of calculating K given the public keys p and g, and the exchanged numbers $g^x mod\ p$ and $g^y mod\ p$. x and y must be discarded.

If g and p are **publicly known** then Diffie-Hellman is vulnerable to a **man in the middle** attack. An attacker E can intercept the first messages from both A and B and respond with their own private integers for each. This would allow them to intercept messages from both A and B, and modify them before relaying them to the other legitimate party.

3 Authentication

3.1 Problems with protocols

3.1.1 'Who goes there?' protocol

Simple **token authentication**. Used to allow cars (C) into and out of garages (G).

1. $C \rightarrow G : T$
2. $C \rightarrow G : \{T, N\}_{KT}$

A car key sends a token T followed by T and a nonce N encrypted with shared key KT to the garage. If N is fresh (has not been used recently) then the garage opens the door. An eavesdropper can only know T and $\{T, N\}_{KT}$ and each $\{T, N\}_{KT}$ should be different, so a recorded transmission can't be used. However a problem arises from checking if N is fresh, as a significant number of previous N must be checked against, and therefore stored. Alternatively a counter synchronised between car keys and the garage, or a predictable random number generator could be used.

3.1.2 Challenge response protocol

An extension of 'who goes there?' Use in engine immobilisers.

1. $E \rightarrow K : N$
2. $K \rightarrow E : \{T, N\}_{KT}$

The engine transmits a nonce N to the car key. The car key encrypts token T and N with shared key KT. The engine decrypts the message, verifies that T is valid and that N is the same, and if so the engine starts. As the engine (E) generates N there is no need to check if it is fresh, and neither T or KT are transmitted in the clear. However if N is at all predictable an eavesdropper can interrogate the car key with the next N, record $\{T, N\}_{KT}$ and transmit it to the car.

3.1.3 Two factor authentication

Involves a user U, password generator P and server S. Used for smart card readers and other systems utilising two factor authentication.

1. $S \rightarrow U : N$
2. $U \rightarrow P : N, PIN$
3. $P \rightarrow U : \{N, PIN\}_K$
4. $U \rightarrow S : \{N, PIN\}_K$

S provides a random challenge N to U. U enters their secret PIN (shared by S and U) into P, that is also provided with N. P encrypts N and PIN with K (shared by P and S) and transmits it to S. S decrypts the message and verifies N and PIN. An eavesdropper cannot find out PIN ('transmissions' between P and U are direct and can't be intercepted). However, a **man in the middle** attack can be executed. Attacker M intercepts N from S, and relays it to U. M also intercepts $\{N, PIN\}_K$ from U and relays it to S , granting authorisation. M severs the connection between S and U, and in the case of authentication for a bank card, has access to the account of U. From the perspective of S, M appears to be U.

3.2 Authentication using third parties

3.2.1 Intuition

The objective of A and B is to establish a session key that can be used for secure communication. Nobody else should know this session key, and A and B should be protected against man in the middle and replay attacks (use of old session keys). These vulnerabilities exist because of the protocol, not because of issues concerning cryptography.

3.2.2 Needham-Schroeder protocol

A protocol that appeared in 1978, where A and B attempt to establish a session key K_{AB} using trusted third party S.

1. $A \rightarrow S : A, B, N_A$ (N_A prevents replay attacks utilising the message from *2*)
2. $S \rightarrow A : \{N_A, B, K_{AB}, \{K_{AB}, A\}_{K_{BS}}\}_{K_{AS}}$
3. $A \rightarrow B : \{K_{AB}, A\}_{K_{BS}}$
4. $B \rightarrow A : \{N_B\}_{K_{AB}}$
5. $A \rightarrow B : \{N_B - 1\}_{K_{AB}}$

A initiates the protocol, stating to S that she wants to communicate securely with B, and providing her nonce N_A (*1*). S responds with a session key, a certificate for B, the identity of B and N_A so A can determine that a replay is not taking place, encrypted with the key K_{AS} shared by A with S (*2*). A passes the certificate ($\{K_{AB}, A\}_{K_{BS}}$) to B (*3*), who then performs a challenge-response to A to confirm that she is present (*4 and 5*). A **vulnerability** to **replay attacks** exists in this protocol. If an attacker C, who had observed the transmission of $\{K_{AB}, A\}_{K_{BS}}$, were to obtain the key K_{AB} in the future by compromising A, the certificate message $\{K_{AB}, A\}_{K_{BS}}$ could be sent to B who would accept it. B has to assume that K_{AB} is fresh as there is no way of telling if it isn't, and continue the protocol. C would now have established a session using K_{AB} with B who would believe C to be A.

3.2.3 Kerberos

Kerberos fixes the vulnerability of the **Needham-Schroeder** protocol. **Two trusted third parties** are involved: an **authentication server** X and a **ticket granting server** S. A logs onto X using a password. Assuming the password is correct, the server sends a ticket T_A and a session key K_{AS} encrypted with the password. A now attempts to access a resource B controlled by S, and requires K_{AB}, used to authenticate subsequent traffic from A, that has a **lifetime** L. The following listing is from **Security Engineering** by **Ross Anderson**:

1. $A \rightarrow S : A, B$
2. $S \rightarrow A : \{T_S, L, K_{AB}, B, \{T_S, L, K_{AB}, A\}_{K_{BS}}\}_{K_{AS}}$
3. $A \rightarrow B : \{T_S, L, K_{AB}, A\}_{K_{BS}}, \{A, T_A\}_{K_{AB}}$
4. $B \rightarrow A : \{T_A + 1\}_{K_{AB}}$

A asks S for access to B (*1*). S returns a key K_{AB} and a ticket $\{T_S, L, K_{AB}, A\}_{K_{BS}}$ (*2*) that is passed to B (*3*). B can decrypt the ticket using K_{BS} to retrieve K_{AB}, and this also includes part of a challenge-response allowing B to confirm its presence (and successful retrieval of the session key) to A (*4*). The **vulnerability** found in Needham-Schroeder has been **fixed** by using **timestamps** (L) instead of nonces. B is provided with L from S, through A (who doesn't have K_{BS} and so can't modify L) and so can determine if K_{AB} is fresh. The following 'full' listing is from the **lecture notes**:

1. $A \rightarrow X : N_A, A, S$
2. $X \rightarrow A : \{K_{AS}, N_A, \{T_X, L, K_{AS}\}_{K_{XS}}\}_{K_{AX}}$
3. $A \rightarrow S : \{A, B, T_A\}_{K_{AS}}, \{T_X, L, K_{AS}\}_{K_{XS}}$
4. $S \rightarrow A : \{T_S, L, K_{AB}, B, \{T_S, L, K_{AB}, A\}_{K_{BS}}\}_{K_{AS}}$
5. $A \rightarrow B : \{T_S, L, K_{AB}, A\}_{K_{BS}}, \{A, T_A\}_{K_{AB}}$
6. $B \rightarrow A : \{T_A + 1\}_{K_{AB}}$

The most significant vulnerabilities of Kerberos are **synchronisation differences** between the clocks of different machines, either unintentionally or as part of a deliberate attack. Alternatively an attacker could impersonate the authentication server X to retrieve credentials (the password) that would allow for them to impersonate A.

3.3 Managing public keys

The aim is to **bind keys to users**, for example A to K_A. It is difficult to guarantee this. A **certification authority** acts as a **trusted third party** that signs (guarantees) a certificate containing the user identity, valid period for the certificate and public keys for signing and encryption.

A certificate can be described symbolically as:

$$C_A = Sig_{K_S}(T_S, L, A, K_A, V_A)$$

- Sig_{K_S}: indicates the certificate is signed by the certification authority (S)
- T_S: timestamp or start date of the certificate, determined by S
- L: certificate lifetime
- A: identity of the certificate holder
- K_A, V_A: public keys for encryption and signing

GSM is the **Global System for Mobile communications**, the most widespread application of cryptography. **GSM** phones are **not secure** due to problems with both protocol and cryptosystems. The use of a mobile phone should guarantee call security (no eavesdropping) and location security (the user's location should remain anonymous). Temporary IDs (**TMSI**) are used as a user moves from one base station to another, but the subscriber identification (**IMSI**) is transmitted if the base station claims not to understand the TMSI so a fake base station can be used to compromise anonymity.

4 Operating system security

4.1 Methods of access control

Authentication is only part of managing access. The principles of **least privilege** and **separation of privilege** must be applied after successful authentication. An **access control matrix** (**ACM**) defines permissions for each user, with column headings as objects, row headings as users and matrix values as binary values representing the permission level (expressed using r, w and x in written form for read, write and execution rights). Although access control matrices **can** be used to implement protection mechanisms they do not scale well.

Alternatively a more compact structure could define **groups** or **roles** that are assigned permissions. Users are assigned a role, and have the permissions of the role. Such roles can be expressed in hierarchies facilitating inheritance of permissions. Improvements such as this improve **economy of mechanism** and are easier for administrators to use.

Access control lists (**ACLs**) are a way of storing permissions (through distributed representation of ACMs) that are well suited to environments were protection needs to be data-oriented and there are few users. Access rules are distributed into lists for each object, each of which is equivalent to a **single column** of a **access control matrix** (in that permissions for each user are still stored). ACLs are used by both **UNIX** and **Windows** (in a modified form) systems.

An alternative representation of ACMs is **capabilities**, which can be seen as an inverse representation to **ACLs** where permissions are distributed into rows of the ACM. Each permission object contains permissions for a single user, for each object that permissions are required for. Use of this representation is uncommon as changing permissions for a single object can become difficult as capabilities for every user must be modified, rather than just a single ACL for the object.

Note that **Windows** attempts to **combine ACLs and capabilities** to maximise efficiency and performance.

4.2 Bell La Padula model

The **Bell La Padula** (BLP) model introduced the idea of a **reference monitor**, parts of the operating system designed to mediate access control decisions. Access control is mandatory for all objects, and is **role based**. There are multiple levels of classification (top secret, secret, confidential and unclassified), and clearance is required to read them.

The system is focused on **confidentiality** as opposed to integrity or availability. **Each document** is classified at **one security level**, **each user** has clearance **up to one security level** and a document can only be accessed by users or processes with clearance at the same or higher security level. Additional 'need to know' basis restrictions can be imposed on specific documents. **Information flow control** is based on **two properties**:

1. The **simple security property**: no process may read data at a higher level (**no read up** or **NRU**)
2. The ***-property**: no process may write to a lower level (**no write down** or **NWD**)

NWD is the critical innovation of the model, and is targeted at preventing malicious code from being effective. If a malicious *unclassified* user copies across a Trojan, and a *secret* level user sees and executes it, the code will be executed at the *secret* level and will not be able to write to an *unclassified* folder where the malicious user could read it. There are **problems** with the model. A BLP model can be defined that allows a user to request temporary declassification of a file by an administrator, without breaking any BLP assumptions. It has been proven (by John McLean) that BLP rules by themselves, in their explicit original form, are not sufficient.

- A system can be defined without breaking them within which a user can ask a system administrator to temporarily declassify any file, allowing a low level user to view higher level documents without breaking any BLP assumptions. This is addressed by the addition of the **tranquillity property** that prevents **security labels** from changing during system operation.
- Some applications might need to break security policy, for example when a document's classification level needs to be reduced, and the BLP rules doesn't cover this.
- NWD results in documents accumulating at the top security level, or documents must be fragmented based on their content to prevent this.
- The BLP rules do not address the creation and destruction of documents.
- Applications designed to operate within more common security models (such as those found on UNIX and Windows) will not work with BLP.

4.3 Trusted computing

Trusted computing is intended to assure **platform integrity** (guarantee how it will behave) using a **trusted platform module** (**TPM**). A TPM is a **secure cryptoprocessor** mounted on the motherboard. The capabilities of a TPM include:

- Storing passwords, certificates and encryption keys
- **Hardware authentication** (the ability to prove the platform is what it claims to be) and **attestation** (proving the platform is trustworthy and hasn't been breached)
- Monitoring the PC as it boots, with the ability to limit access to highly secure applications if unexpected changes in the configuration have occurred
- Full disk encryption implementations can use the TPM to ensure if a hard drive is installed on another machine it can't be decrypted, as the TPM contains the decryption key

4.4 What goes wrong

4.4.1 Root access through 'setuid'

For a **UNIX** user to change their password, they require root access to write to the */usr/bin/passwd* file. The 'passwd' utility is used, that has a *setuid* bit (flag) set, allowing it to run as root even when executed by an unprivileged user:

- If vulnerable code containing **setuid** is exploited, as the process is running as root, an attacker can **run code as root**, potentially opening a shell for full **root access** to the system.
- If a file is opened by a process such as *setuid* that runs as root and a child process is created, it inherits all file descriptors and could gain root access to system files. This can be fixed by ensuring all files are closed before forking a child process.
- The program could close with files still open.

The *setuid* utility is also vulnerable to **buffer overflow** (**buffer overrun**/**stack smashing**) attacks.

4.4.2 Buffer overflow

Some languages such as C++ do not perform **bounds checking** (checking the size of arguments), so part of a long argument passed to a program by an attacker may be written to memory space outside of that allocated for the data. This could allow for it to be run as code, producing the potential for an attacker to take control of a system. Trailing bytes of the argument (those beyond the intended argument length) usually have a **landing pad** of **no operation** (**NOP**) commands that are ignored during program execution, a **NOP sled**. This means that a program jump to anywhere in the large memory space covered by the 'sled' will result in the malicious code at the end of it being executed in full. These easiest way to prevent this is to **enforce bounds checking**, checking whether arguments are of the correct size and either reporting error or truncating them if they are not.

4.5 Assumptions behind software

1. **Users are trustworthy and will supply correct input**
   - SQL injection and buffer overflow prove this is not the case.
   - Can be fixed by checking input, being wary of SQL sequences and escape characters, initialising all variables, closing all files, suitable bounds checking and using memory protection where possible.
2. **It is OK to have a back door into the software that enables easy access during testing and development**
   - This occurs frequently and functionality can be come dependent on the existence of a back door.
   - This can be fixed by clearly documenting the existence of the back door and ensuring its removal for deployment.
3. **All libraries, third-party resources, and other components will load correctly when an application is compiled or launched**
   - Library code could be used for validation or authentication, resulting in unpredictable behaviour if it is missing.
   - A bad library might have the same name as a good library, but be higher up the search path (for package managers etc).
   - Unhandled exceptions are frequently associated with bugs.
   - These problems can be addressed by testing that components are as expected, that they load properly and through careful exception handling.

4. **Data can be trusted implicitly**
    - Trust may be extended based on identification without authorisation.
    - **Complete mediation** must be enforced to prevent this.
5. **It is OK to use temporary files to store data when they are too large to hold in memory**
    - Cookies are usually stored and transmitted in plaintext, and the same applies for other temporary files.
    - Can be fixed by **never** storing secure data to a temporary file unless it has been suitably encrypted.
6. **The OS will manage process execution in a synchronised manner**
    - **Race conditions** can be used to exploit vulnerabilities.
    - This can be fixed by being aware that an operating system will time-slice processes.

5 Malware

5.1 Viruses

A computer virus is **malicious code** that attaches itself to a host that could be another executable program, the boot sector of a disk or a document that supports macros. Viruses have been more prevalent on Windows machine as it's easier to infect files owned by other users or system files. As with biological viruses, computer viruses create copies of themselves in new hosts.

Virus detection involves searching for the characteristics of known viruses in memory. Viruses can be designed to try and avoid detection through segmenting code, encryption and the presence of junk inbetween code segments. **Polymorphic viruses** us a different encryption key each time they copy, whilst **metamorphic viruses** mutate the junk between code segments.

5.2 Worms

Unlike viruses, worms do **not infect existing executables**. They are **independent processes**, meaning that they they often do not need user action to copy themselves (as a virus would). As with real world disease, the spread of a worm can be modelled based on the quantity of uninfected hosts and the infectiousness.

$$\frac{\delta I}{\delta t} = \beta I(t)[N - I(t)]$$

5.2.1 The Morris worm

The **Morris** or **internet worm** exploited vulnerabilities in: **rsh** (allows for a remote shell to be est), **finger** (provides user information and vulnerable to buffer overflow) and **sendmail**

- **rsh**: allows for a remote shell to be established.
- **finger**: provides user information, vulnerable to buffer overflow due to use of **strcopy** C function.
- **sendmail**: an email router, contained a back door that was exploited to establish a shell.

The worm used **rsh** or **sendmail** to establish a shell running on a target machine that was used to copy across a **grappling hook** C program that was run with command line arguments, including the IP of the source machine. The grappling hook then copied across and installed a worm binary. One installed the binary renamed itself as **sh** (a shell), gathered information about possible targets using standard utilities (such as **netstat**), used a dictionary attack to try and break user passwords and look for machines where they might have a valid login and then tried to establish a shell on target machines using vulnerabilities. Data strings were encrypted, although only using a **Caesar cipher**.

Although there was **no malicious payload** but it was highly effective and damaging. The code was supposed to check for existing infections and terminate itself, but a bug meant **some infections were immortal**, overwhelming the host and eventually clogging up the internet at the time (**Arpanet**) entirely. Great care was taken to camouflage processes and code (hidden as shells and encrypted).

The worm demonstrated the vulnerability of networked computers and the existing software infrastructure. Its creator Robert Morris was prosecuted, and the **Computer Emergency Response Team** at Carnegie Mellon University was set up as a consequence. Internet was, however, largely restored within a couple of days. The fact that it only affected machines derived from **Berkeley UNIX** allowed for quick fixes to be made, and limited the worm's spread.

5.2.2 The Conficker worm

Exploits a vulnerability in **svchost.exe** (container for Windows services) on Windows machines. It spreads using a **buffer overflow** vulnerability in **RPC** (remote procedure call, a method of inter-process communication) and the vulnerable port 445. It can also be spread by network sharing and removable media. It attempts to gain write access with the signed in user, and failing this, cracks user passwords until write permissions are obtained.

The worm has numerous variants and potential uses or symptoms such as users being locked out, remote access, disabling of Windows security measures, or recruitment of the machine to a botnet. It has **upload and download capability** and so can be used to install malware, and prevents removal through concealment and other means. Between 9-15 million Windows machines are believed to have been infected, although patches for the vulnerabilities exploited exist. Microsoft is offering $250,000 for information leading to the arrest of its author.

5.3 Detection and removal

5.3.1 Concepts

Malware can be **detected by its behaviour**. Rootkits modify operating system files, memory and registry entries. Adware, spyware and data harvesters need access to specific methods or operating system utilities that can be monitored.

Writing a program that will detect all malware is impossible, as has been proven by contradiction by Fred Cohen. If a perfect malware detector **is_malware** existed, a malware could be written that only behaved like malware if **is_malware** called on itself returned false. This program could return neither true nor false (as an infinite loop would be produced). The question of whether such a program is malware depends on the **halting problem** and is therefore unsolvable.

5.3.2 Countermeasures and insider attacks

Countermeasures are largely intuitive:

- **Optimise system diversity** - variety in operating systems and software can prevent against exploitation of specific vulnerabilities.
- Good software design and coding practices.
- **Avoid auto execution** of removal media.
- Apply **least privilege** throughout the system.
- Strong passwords, keep software updated, use anti-virus etc.

An **insider attack** is where a software developer intentionally leaves malicious code. A contractor who's contract had ended at Fannie Mae in 2008 left malicious code at the end of a shell script, that was discovered but that could have caused severe damage. Monitoring employee behaviour, version control (through review of modifications), software engineering principles and application of **least privilege** can help prevent this.

6 Network security

6.1 Networking basics

The 5 layer network model (same as **OSI 7** but without application layer divisions):

1. **Application**: higher level protocols associated with application data
2. **Transport**: enables host-to-host communication , usually through TCP or UDP on specific ports
3. **Internet** (or Network): routes packets through a network, each host is individually addressed
4. **Link**: transfer between machines on a local network, usually using Media Access Control (MAC) addresses
5. **Physical**: the hardware that provides the connection such as cable or wireless technology

CIA through networks is considered as follows:

- **Confidentiality**: not ensured by default, encryption must be done explicitly, usually at the application level.
- **Integrity**: is ensured using checksums although this identifies corruption during transmission. No consideration of identity or authentication with IPv4, so again this must be ensured at the application level. This does however facilitate anonymity.
- **Availability**: The ability is robust but availability of paths is difficult to guarantee. Constraints on data flow must be implemented explicitly (again usually at the application level).

6.2 Link layer vulnerabilities

ARP (address resolution protocol) is a **link layer protocol** that resolves network layer (IP) to link layer (MAC) addresses on a local area network (LAN). ARP requests are broadcast on the LAN, and the protocol does **not include authentication**. A attacker's machine can send a forged (**spoofed**) response to an ARP request before a legitimate machine can respond, resulting in the association of the attacker's MAC address with the IP of another machine. This can facilitate **man-in-the middle** attacks where packets can be intercepted, modified or dropped.

This type of attack is called **ARP cache poisoning**. It can be prevented by restricting LAN access to trusted users, certification and cross checking of ARP responses or the employment of **static ARP tables** stored on routers as **read only** (not scalable as changes mean the mapping has to be reset).

6.3 Network layer vulnerabilities

- Datagrams transmitted as **IP packets**, that have a specific structure including source and destination IP addresses. **IP spoofing** forges the IP source address with a different address. This can allow an attacker to probe a network without revealing their identity.
- A **ping flood** is a **DDOS** (distributed denial of service) attack that floods a victim with **ICMP** (internet control message protocol) ping requests: if the target responds both ingoing and outgoing traffic are flooded.
- A **Smurf attack** is a **DDOS** attack in which ICMP packets are sent to a large number of computers across a network, with the source address set to that of the victim's computer. Most computers on the network will respond, flooding the victim's incoming traffic.

Network layer attacks can be defended in a number of ways:

- Blocking packets with source IPs of computers in a LAN from passing border routers from outside the LAN (would prevent a Smurf attack using computers inside of the LAN launched from outside it).
- Pinging a host and comparing the TTLs (time to live) of the ping and a suspicious packet with its IP address. They should be similar and if not, the packet is likely to be spoofed.
- **Authentication headers** to authenticate source addresses.
- Use of **encapsulated security payloads** for encrypting packets to prevent against packet sniffing (other individuals watching network traffic and dissecting packets to retrieve their payloads and other information using tools such as Wireshark).
- Reduce the size of network segments (wireless hubs serving smaller areas) so less traffic can be sniffed from a single location.

6.4 Transport layer vulnerabilities

TCP (transmission control protocol) involves a three-way handshake to establish a connection between two hosts:

1. A sends a SYN (synchronisation) packet with a sequence number x
2. B responds with SYN-ACK (acknowledgement) with sequence number y and acknowledgement number $x + 1$
3. A returns ACK with sequence number $x + 1$ and acknowledgement number $y + 1$

If E can predict x and y the session can be **hijacked** (a **TCP hijacking**). E denies service to A to prevent a response. E sends SYN to B spoofing the IP of A, E waits for B to respond to A and then guesses Y to send back to B. E can now send packets as A to B, but packets from B still go to A. From this E could establish a shell or attempt to instruct B to form a two-way connection. **Complete hijacking** is possible if E can observe network traffic, and use it to predict sequence numbers. To defend against such attacks **encryption** and **authentication** at either the application (TLS/SSL) or network (IPsec) layers is required.

6.5 Application layer vulnerabilities

6.5.1 TLS and HTTPS

By default HTTP requests and responses are delivered without any encryption through TCP on port 80. **TLS** (**Transport Layer Security**) provides security on top of TCP/IP, and was formerly the Secure Sockets Layer (SSL). TLS **encrypts** network traffic and **authenticates** web servers using certificates from **authentication authorities**. On the web TLS is used to provide **HTTPS** (HTTP over TLS) in which all headers, page contents and cookies are encrypted.

To initiate a secure exchange between client A and server B a handshake takes place:

1. $A \rightarrow B : A, A\#, N_A, Cipher\ Preferences$ - A sends their identity, session key (#), nonce and cipher preferences.
2. $B \rightarrow A : B, B\#, N_B, CS, Cipher\ Choice$ - B returns the same information but with a cipher choice **and a certificate** (CS that contains public key K_B).
3. **Certificate exchange**: A compares the certificate from B to a root certificate held in the browser, or one held by a trusted third party. B may also ask A for a certificate at this point.
4. **Key exchange**: A computes a random pre-master key K_0 that is sent to B encrypted with the public key of B, K_B. A and B independently compute master keys K_1 and session keys K_{AB} form information already exchanged.

- $A \rightarrow B : \{K_0\}_{K_B}$
- $K_1 = hash(K_0, N_A, N_B)$
- $K_{AB} = hash(K_1, N_A, N_B)$

5. A sends B 'finished' together with a **message authentication code** of messages so far, encrypted with K_{AB}. If B can decrypt, then A is authenticated to B. Otherwise the session ends with an error.
6. B sends a similar message to A.

Note that this is a **severe simplification** of the full protocol, of which there are numerous variants. **Vulnerabilities** can exist where websites supply forged or expired certificates, or if a browser passes responsibility for checking an invalid certificate to the user who will often just click OK.

6.5.2 XSS (cross site scripting)

XSS or **cross site scripting** enables attackers to inject client-side script (JavaScript, ActiveX etc.) into web pages viewed by other users. It could be used to retrieve cookies or data being displayed to a user to an attacker for example.

There are numerous means of achieving injection. Malicious URLs in emails could redirect to a trusted website vulnerable to XSS in a way that allows for XSS (parameters that can be set to **script** tags for example). Alternatively websites with dynamic content can be injected if data is not managed safely. For example a **script** tag containing malicious code could be provided as input for a profile description, and executed when another user viewed the resulting profile page. This could easily be prevented by validating user input to check for specific searches for **script** tags, or a safer approach could just consider all inputs containing XML tags to be invalid.

6.5.3 SQL injection

Similar to some variants of **XSS**, where a user provides input that is an SQL query or part of an SQL query where one is not expected. This could be used by a malicious user to extend part of a **select** query to also retrieve confidential data about other users from a table for example. In the case of **MySQL** and other common variants this can be prevented through the use of **prepared statements**, which are part of the **MySQL** and some other **SQL implementations**, or the use of **context-specific input checking**.

6.6 Wireless networks

On a wireless network it is almost impossible to prevent silent eavesdropping with tools such as *Wireshark*. Many wireless stations even broadcast the network name and security measure. Access can be restricted to a set of MAC addresses although an attacker can easily reconfigure their machine to have an authorised MAC address.

Encryption such as **WEP** (**wired equivalent privacy**) can help, although WEP is **broken** and has been replaced with **WPA** (Wi-Fi protected access) as well as **WPA2** that enforces **AES** (advanced encryption standard) encryption. WPA uses a **four-way handshake** to establish a connection between a router and a client. It is assumed that AP (access point) and STA (client station) have a shared key PMK (pairwise master key) entered by A into their machine (STA). A PTK (pairwise transient key) must be generated for encryption.

1. $AP \rightarrow STA : N_A$
2. $STA \rightarrow AP : N_S, \{Message\ integrity\ code\ (MIC)\}_{PTK}$
   Note: $PTK = hash(PMK, N_A, N_S, MAC_A, MAC_B)$ where MAC_A and MAC_B are **media access control addresses** and **not message authentication codes**.
3. $AP \rightarrow STA : \{Group\ temporal\ key\ (GTK)\}_{PTK}, \{MIC\}_{PTK}$
4. $STA \rightarrow AP : Acknowledgement$

The main problem with this protocol is the PMK, if E knows this or obtains it by brute force she can obtain PTK. The objective of the protocol is to minimise exposure of the PMK.

7 Forensics

Key points from lectures

- **Chain of custody**: documentation of who has control and custody at all times should be kept, so that a credible case that is has not been tampered with can be made.
- **Seizure**: a specialist task for computer forensics. There is debate as to whether a the power should be left on for running devices, as removing power can prevent shutdown scripts but it also can result in loss of memory. If a device is switched off, it should remain that way.
- **Storage media**: dynamic RAM is volatile, data is lost when power is removed. But hard drives and solid state drives are persistent. Cooling memory (**cold booting**) can slow down memory loss on RAM.
- **Acquiring disk images**: can be done physically and is easier if a disk is removed. A **forensic disk controller** can intercept commands that would modify the contents of a disk. **Taking hashes** of both the original disk and a copy can **prevent** against **allegations of tampering**.
- **Analysis**: could involve simple trawling, or a reconstruction of the timeline of a filesystem.
- **Detecting image manipulation**: **error level analysis** can be used, that detects whether differences between an image and a compressed form of the image are uniform. If not then modification has occurred.