# Terminology

**Threat.** Set of circumstances that has the potential to cause loss or harm.
**Vulnerability.** Weakness in the system.
**Control.** Mean to counter threats.
**White Hat.** An ethical security hacker.
**Grey Hat.** A computer hacker or computer security expert who may sometimes violate laws or typical ethical standards, but usually does not have the malicious intent typical of a black hat hacker.
**Black Hat.** A hacker who violates computer security for their own personal profit or out of malice.

# CIA

**Confidentiality.** Asset **viewed only** by authorised parties, prevention of unauthorised disclosure of info.
**Integrity.** Asset **modified** only by authorised parties, prevention of unauthorised modification of process or info.
**Availability.** Asset **used by** any authorised party, prevention of unauthorised withholding of info or resources. Trade-off between security and **ease-of-use** (interfere with working patterns users originally familiar with), **performance** (security mechanisms consume more resources.), **cost** (cost of developing).

# Encryption

Required to be **Correct** (for any plaintext $p$ and key $k$, $DEC_k(ENC_k(p)) = p$), **Security** (difficult to get useful information of $k$ and $p$ from $c$.) and **Efficient** (efficient computations of encryption and decryption). **Alice**: originator, **Bob**: receiver, **Eve**: sniff, **Mallory**: sniff and spoof.
**Deterministic.** Always produce same ciphertext when given same plaintext and key.
**Probabilistic.** Produce different ciphetexts even with same key and plaintext.
**Encryption guarentees Confidentiality only!**
**Symmetric-Key Scheme.** Same key for encryption and decryption.
**Public-Key Scheme.** 2 different keys for encryption and decryption. Encryption key is public and decryption key is private.
**Obscurity.** Hide the design of the system in order to achieve security. System must still be secure even if everything about it except secret key becomes known.
**Compression.** Compressing an encrypted file yields little compression gain. Encrypted file is already a random sequence and the compression algo which takes advantage of repeating patterns will not work well on encrypted file.

# Attack Models

**Oracle.** Attacker send queries, oracle output answers.
**Ciphertext-only.** Given a set of ciphertext, may know some **properties** of plaintext
**Known-plaintext.** Given a set of ciphertext and corresponding plaintext
**Chosen-plaintext.** Access to **encryption oracle** (choose and feed any **plaintext** and obtain corresponding ciphertext (all encrypted with same key)). Can be accessed for a reasonable large number of times.
**Chosen-ciphertext.** Access to **decryption oracle**, choose **ciphertext** and output plaintext.
Public key cryptography, encryption oracle always available (public key is known). Padding oracle is weaker form of decryption oracle since it gives out less information.
**Total break.** Attacker wants to find the key.
**Partial break.** Recover plaintext from cipher (not interested in key), determine coarse information about plaintext. If $S_1$ can prevent more attacks than $S_2$, then $S_2$ is more secure w.r.t the attack model.

# Ciphers

**Key Space.** Set of all possible keys.
**Key Space size.** Number of keys.
**Key size.** Size of 1 key in bits.
**Diffusion.** A change in plaintext will affect many parts of ciphertext. This means that information from plaintext is spread over entire ciphertext. Requires attacker to access much of ciphertext to infer encryption algo.
**Confusion.** Attacker shouldn't be able to predict what happen to ciphertext when one character in plaintext/key cahnges. The input undergoes complex transformations during encryption.

# Substituition Cipher

Key is a substituition table that has 1-to-1 mappings between characters. Key space size is 27! and key size at least $log_2(27!) \approx 94$bits to represent 27! keys. See the key as trying to represent all the permutations, so 0 is one permutation, 1 is another and so on. To represent 27! permutations, we need 94bits.
**Known-plaintext.** Given key space, try every key until plaintext matches ciphertext. Possible to obtain parts of the key with known-plaintext. With more information, can fully determine the key.
**Known-ciphertext.** If in english, can try every key until get some english word. Use **frequency analysis** for better performance.
**Monoalphabetic.** Fixed for each letter in alphabet. Given sufficiently long ciphertext, attacker can guess plaintext by mapping frequencies.

# Permutation Cipher

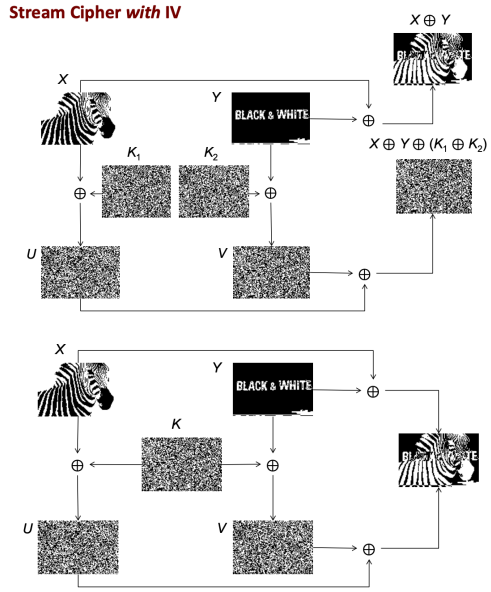
Group plaintext into blocks of $t$ characters. Then apply a permutation to the characters in each block by shuffling the characters. $t$ could be part of the key and kept secret.
**Known-plaintext.** Very easy to deteremine key.
**Known-ciphertext.** Easily broken if plaintext is in english.

# One Time Pad (OTP)

XOR operation - when $A = B$, result is 0, 1 otherwise. $A \oplus B = B \oplus A$.
$A \oplus (B \oplus C) = (A \oplus B) \oplus C$. $A \oplus 0 = A$. $A \oplus A = 0$
Encryption: plaintext $\oplus$ key $\rightarrow$ ciphertext.
Decryption: ciphertext $\oplus$ key $\rightarrow$ plaintext.
Exhaustive search don't work on OTP. OTP does not leak information of the plaintext (except length) even if attacker has arbitrary run time.
**Perfect secrey.** Guarenteed (for any ciphertext $y$ and $x$, chances that attacker correctly predicts $x$ before knowing $y$ and after knowing $y$ are the same: attacker gain no additional information.)

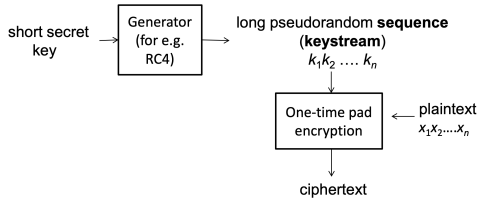
# IV

Initialisation vector or initial value. Helps to "randomise" the generator. Without IV, if same key is used to encrypt 2 different plaintexts, $x, y$, information can be leaked.
$x \oplus y = (x \oplus k) \oplus (y \oplus k) = x \oplus y$. During 2 different encryptions of the **same** plaintext, IVs are likely to be different, which means that ciphertexts would be different. IV makes encryption non-deterministic.




# Stream Cipher & IV

Short secret key passed to generator to generate a key that **matches length of plaintext**. Use OTP to get ciphertext. Generator must be designed so that it gives **cryptographically-secure pseudorandom sequence**, or a **keystream**. Final **ciphertext contains IV** followed by output of OTP encryption. To decrypt: extract IV from cipher and using IV and key, get same pseudorandom sequence and obtain plaintext(using OTP).




# Block Cipher

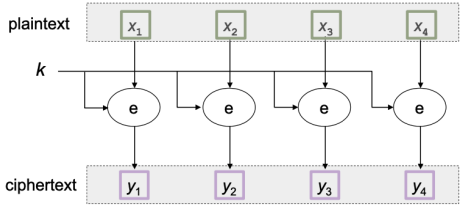
**Block size.** Number of bits of a plaintext.
**Key size/length.** Number of bits of key. The longer the key, the more secure the scheme because of work factor but its slower.

| | Stream | Block |
|---|---|---|
| Advantages | • Speed of transformation<br>• Low error propagation | • High diffusion<br>• Immunity to insertion of symbol |
| Disadvantages | • Low diffusion<br>• Susceptibility to malicious insertions and modifications | • Slowness of encryption<br>• Padding<br>• Error propagation |

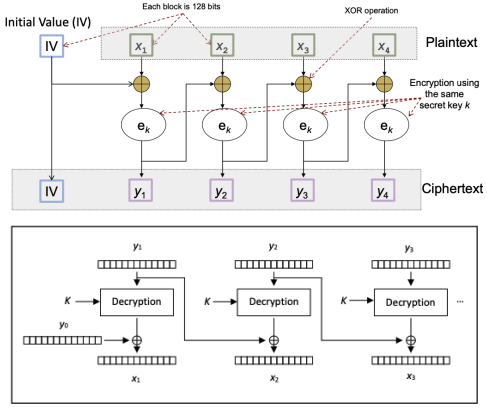
**DES.** Block length of 64bits and key length of 56 fits.
**AES.** Block size of 128 bits, key length of 128,192,256 bits.

# Block Cipher: Modes of Operations

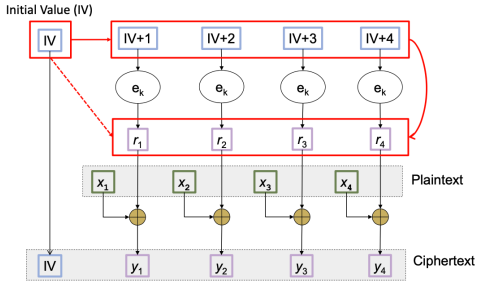
**Electronic Code Block (ECB).** Divide plaintext into blocks, apply block cipher in use to each block with same key.




If attacker finds 2 similar blocks from same ciphertext, can tell that plaintext blocks are the same. To resolve, randomly choose an IV for each block. Since IV appended to ciphertext, final ciphertext size is twice of plaintext.
**Cipher Block Chaining (CBC) on AES.**



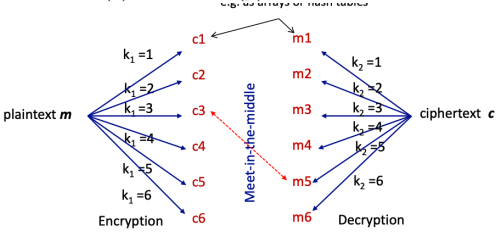

If first ciphertext block is removed, first plaintext block cannot be correctly recovered. But other plaintext blocks still can be recovered. Encryption process cannot run in parallel since there is dependency on the previous rounds. The decryption process can run in parallel since it depends on previous blocks of ciphertexts that are readily available. **Vulnerable to Padding Oracle Attack.**
**Counter (CTR) Mode on AES.**




Turns block cipher into stream cipher - treat each block as a stream.**Vulnerable when counter is sequential.**

# Meet-in-the-Middle Attack

**Double DES (2DES).** Encrypt using DES twice or more using different keys. DES doesnt form a group $E_{k_1}(E_{k_2}(x)) \neq E_{k_3}(x)$ for some $k_3$. 2DES results in key length of $2 \times 56$bits.
**Meet-in-the-middle.** Known-plaintext attack where attacker has plaintext, $p$ and corresponding ciphertext, $c$. Attacker goal is to find the 2 keys. Compute 2 sets, $P, C$, where $C$ contains ciphertexts of $p$ encrypted with all possible keys and $P$ contains plaintexts of $c$ decrypted with all possible keys, find common element in both sets. For **k-bit** keys, number of crypto operations becomes $2^k + 2^k$, using approximately $2^{k+1}$ units of storage space. Use idea that $DEC_{k1}(c) = ENC_{k2}(p)$




# Triple DES (3DES)

Use triple DES using **2 keys** - $E_{k_1}(D_{k_2}(E_{k_3}(x)))$. Keying options includes using 3 independent keys(triple-length keys), 2 independent keys (double-length keys, where $k_3 = k_1$) and 3 identical keys. Encryption options states that the second action must be opposite of the first and last. Choosing $E_{k_1}(D_{k_2}(E_{k_3}(x)))$ with $k_1 = k_2$, its the same as single DES encryption.
**4 DES Operations.** Able to improve using hashing. Instead of k2(k1(m)), store k1(m) in a hash table, then perform k2(hash). This will mean that the number of encryption operations would be dominated by k2. Similarly for decryption. So the total number of operations is about half of the naive.

# Padding Oracle Attack

Attacker has: ciphertext and IV, access to padding oracle, no knowledge of padding oracle secret key. Goal: get plaintext. Padding oracle: knows secret key, accepts a query (ciphertext), outputs YES if padded correctly, NO otherwise. AES is **128 bits**, unused bits padded with some values. Important information to encode, **number of padded bits**.
**Padding using PKCS#7.** - padding bytes are $0X \times X$. If last block is full, then have an extra block of all zeroes.
**AES CBC not secure against padding oracle attack in ciphertext-only**, in particular when done with PKCS#7. Assume attacker knows that its padded with 3 bytes, then plaintext has length of 5 bytes. Attacker wants to find $x_5$, value of the 5th block in the plaintext.
**Algorithm.** Modify $IV_5$ to some special value, $IV_5'$ such that when passed to the oracle, the resulting padding would be correct ($x_5' = 04$). To recover the plaintext value, just perform $04 \oplus IV_5'$. This works because $c$ is kept the same, which means that the intermediate block after decryption remains the same. So $I = IV_5' \oplus x_5' = IV_5 \oplus x_5$, then make $x_5$ the subject to get the plaintext.
**Getting padded bytes.** Modify $IV$ using linear probing. For every byte going left to right, change the $i$th byte of IV and send it to oracle. If oracle says NO, then declare the number of padded bytes as $2^n + 1 - i$.
**Prevention.** Deny access to oracle, change padding standard.

# Attacks on Implementation

When using AES CBC, IV should be unpredicatbale to prevent certain type of attack. Vulnerable to choose sequential IV. Random generation (*java.util.Random* vs *java.security.SecureRandom*), choose the secure one. Don't design own cryptosysmte, or even make slight modification to existing scheme.
**Kerchkhoffs' Principle.** System should be secure even if everything about the system, except secret key is public knowledge.
**Side Channel Attack.** Any attack based on extra information that can be gathered because of the fundamental way a protocol/algorithm is implemented, rather than flaws in the design itself.

# Authentication

**Authentication.** Process of assuring that either communicating entity or origin is one that it claims to be.
**Entity Authentication.** Connection-oriented communication, communicating entity is involved in a connection such as passwords, challenge and response, biometrics.
**Data-origin Authentication.** Connectionless communication, communicating entity is origin of information, implies data integrity, uses MAC or digital signature. **Data-origin authenticity implies data integrity** but not the other way. Authenticity is a strong requirement than integrity. Authenticity-preserving techniques also ensure integrity.
**Password.** Identity dont have to be secret but password should be (only server and authentic user know password, implies that either is authentic). Considered **weak authentication** since its subjected to simple **replay attack**: information sniffed from communication channel can be used to impersonate the user later.
**Strong authentication.** Information sniffed cannot be used to impersonate user.
**Bootstrapping.** Server and user establish common password, server keeps track of file recording identity and password. Password established by user choosing a password and sending through communication channel or just use a default password. Attacker can intercept password during bootstrap (if sent through postal, can just steal mail) or use default password.
**Password-based authentication.** Server authenticate entity, if entity give correct password corresponding to claimed identity, entity is authentic. User tells server username, server ask for password. If user provides correct password then user is authenticated. Can be carried out without interactions, send both username and password together. Server has password file that maps usernames to password.
**Password Reset.** User forget password, current password may be stolen, regular change by policy. Resetting password is tricky since only authorised entity can reset (how to verify?). Anyone who knows old password can change the password. Entity can present additional documents/information to change password.

# Attacks on Password System

**Online guessing.** Attacker directly interact with authentication system.
**Offline guessing.** Attacker obtain password file from system.
**Dictionary attack.** Words from english dictionary, known compromised passwords, etc. Possible to combine the 2.
**Stealing the password.** Shoulder surfing (look-over-the-shoulder attack), sniffing communication channel/device.
**Keylogger.** Capture keystroke and send info to attacker.
**Spoofing.** Fake login screen, have secure attention key that starts the trusted login process when pressed.
**Phishing.** Trick user to send password over network under some false pretense., typically done through emails and websites.
**Spear Phishing.** Targetted phishing.
**Password Caching.** Shared work station could cache keyed-in information, next user can see cache. Clear browser cache and close browser when done.
**Click Fraud.** Occurs in pay-per-click online advertising when a person, automated script or computer program imitates a legit user of a web browser clicking on an ad for the purpose of generating a charge per click without having actual interest in the target of the ad's link.
**Insider Attack.** Malicious system admin who steal password file, system admin account compromised, leading to lost of file.
**Vishing.** Phone calls or leaving voice messages acting as reputable companies to induce revelation of personal information.
**Smishing.** Text messages acting as reputable companies to induce individuals to reveal personal information.

# Preventive Measures

**Phishing.** User education via phishing drill, phishing repository sites, phishing filtering via firewall, mail server, browser.
**Use strong password.** Use automated generator with high entropy. Strong passwords difficult to remember, difficult to enter on mobile.
**Entropy.** Measurement of randomness. If a set $P$ has $N$ passwords and Alice picks one randomly and uniformly then entropy would be $log_2(N)$ bits.

**Entropy per symbol for different symbol sets** $\log_2(N)$

| Symbol set | Symbol count $N$ | Entropy per symbol $H$ |
|---|---|---|
| Arabic numerals (0–9) (e.g. PIN) | 10 | 3.322 bits |
| hexadecimal numerals (0–9, A–F) (e.g. WEP keys) | 16 | 4.000 bits |
| Case insensitive Latin alphabet (a–z or A–Z) | 26 | 4.700 bits |
| Case insensitive alphanumeric (a–z or A–Z, 0–9) | 36 | 5.170 bits |
| Case sensitive Latin alphabet (a–z, A–Z) | 52 | 5.700 bits |
| Case sensitive alphanumeric (a–z, A–Z, 0–9) | 62 | 5.954 bits |
| All ASCII printable characters except space | 94 | 6.555 bits |
| All ASCII printable characters | 95 | 6.570 bits |
| All extended ASCII printable characters | 218 | 7.768 bits |
| Binary (0–255 or 8 bits or 1 byte) | 256 | 8.000 bits |
| Diceware word list | 7776 | 12.925 bits |

For online attacks, at least 29 bits of entropy (RFC 4086). If cryptographic keys generated from password and offline at tacks are possible then the password should have at least 128 bits of entropy (NIST) or 96 bits of entropy (RFC 4086).
**Password Protection.** Limit login attempts (add delays into sessions, add security questions, auto lock account after a few failed), password checker (check for weak passwords when password registered/change), password metering (indicate strength of passwords), password ageing (regularly change passwords), password usage policy (rule set by organisation to ensure users use strong passwords and minimise password loss).

# Protecting Password File

Password file could be leaked. Passwords should be hashed and stored in the password file. During authentication, password entered is hashed and compared with digest stored. Same password hashed into 2 different values for 2 different userid, achieved using **salt** - hash(salt + password).

# Password Reset(Security Questions)

Fallback authentication/self-services password rest. Enchances usability, reduce cost but weakens security. Requirements of answer to security questions: **safe** - answers shouldn't be easy to guess or researched, **memorable**, **Nearly universal**(security questions applicable to wide audience), **Consistent**(answers shouldn't change over time).

# Password Reset (Recovery Email Address)

If user forget password, password reset link sent to recovery email address. Link contains one-time password. Ownerhsip of email proves authenticity. If attacker can steal OTP, can take over account.

# ATM Attacks

To authenticate, present card and PIN. Card contains magnetic strip storing user id, simplifying input of user id. PIN plays role of password.
**ATM Skimmer.** Steal username and password. Consists of a card reader(read magnetic strip), and camera/keypad (record PIN), some means to record and transmit information back to attacker.
**Prevention.** Anti-skimmer device (prevent external card reader attachements), shield keypad, user awareness, new chip-based cards that use encryption.

# Biometrics

Depends on quality of scanner. Still possible to be spoofed.

**Enrollment.** Reference template of user biometric data constructed and stored.

**Verification.** Sample data captured and compared using matching algorithm. Used for **Verification** (1:1 verification if person is claimed person) and **Identification** (1:n comparison to identify persom from many).

| Password | Biometric |
|---|---|
| Can be changed (revoked) | Can't |
| Need to remember | Don't have to |
| *Zero non-matched rate* | *Probability of error* |
| Users can pass the password to another person | Not possible |

**Matching Algorithm.**

**FMR** (false match rate) and **FNMR** (false non-match rate)

$$\text{FMR} = \frac{\text{number of successful false matches } (B)}{\text{number of attempted false matches } (B+D)}$$

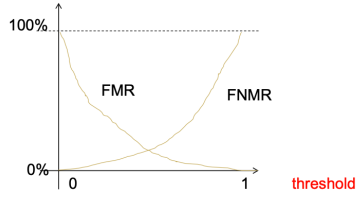
$$\text{FNMR} = \frac{\text{number of rejected genuine matches } (C)}{\text{number of attempted genuine matches } (A+C)}$$

| Actual input \ Matching algorithm's output | accepted/ match | rejected/ non-match |
|---|---|---|
| genuine attempt | A | C |
| false attempt | B | D |

**Failure to Enroll Rate (FER).** Some data not captured for enrollment.

**Failure To Capture Rate (FTC).** Failure to capture during authentication.

Feature points can be similar but may not be the same. Performance depends on quality of scanner. Not possible to tamper, live detection possible too, spoofable.




# MFA

At least 2 different factors. What you know (password, PIN), what you have (ATM card, OTP token), who you are (biometrices).

**OTP Token.** Hardware that generates one time password.

Each token and server share some secret.

**Time-Based.** Based on shared secret and current time interval, password generated, both server and user have common password.

**sequence-based.** : event triggers change of password.

**Password + OTP.** Server issue OTP then user sets password. To authenticate, user press token to get OTP, send OTP + username + password, server verify OTP + password.

**Password + SMS.** User give server phone number + password. To authenticate, server verify password and send OTP via SMS, then user enter OTP. **SMS is not secure!** Consider 2 formats, SMS with OTP and SMS with OTP and transaction details. First one is more secure if adversary gets hold of mobile and previous messages not deleted. Possible to see transaction details, leading to privacy leakage. Second one is more secure if there is malware on the PC that displays a different transaction, then at least can verify.

**Smartcard + Fingerprint.** Server issue smartcard and user enroll fingerprint. User insert smartcard then if authenticated, user present fingerprint. Assumption is that reader and server are secure. Fingerprint template not stored in reader. Reader communicate with server through secure channel.

# Public-Key Cryptography

Alice keeps **private key** secret but tells everyone **public key**. Anyone can encrypt, hence **encryption oracle** always available in PKC. Hence, **plaintext attack** must always be considered in PKC.

**Security Requirement.** Given public key and ciphertext, but not private key, difficult to determine plaintext. Must be difficult to get private key from public key.

**Key Required.** For symmetric-key setting, pairwise secure channels are needed and each pair requires one key, so the total number of keys needed is $\frac{n(n-1)}{2}$. For PKC, **secure broadcase channel** is needed. Each entity publish public and keep private. Total number of private keys is the same as total number of public keys, $n$. Entities don't need to see/know each other before broadcasting the public keys.

**Strengths.** Allow public to encrypt without establishing keys. Also useful for providing authentication.

# RSA

Owner randomly chooses 2 **large primes,** $p, q$ and compute $n = pq$ as public **composite modulus**. $p, q$ are secret. Owner randomly generates an **encryption exponent,** $e$ s.t. $gcd(e, \phi(n)) = 1$, i.e. $e < \phi(n)$ and $e$ is relative prime to $\phi(n)$.

$\phi(n) = (p-1)(q-1)$ is Euler's totient function, which is the number of integers $< n$ that are relatively prime to $n$. Owner then determine **decryption exponent,** $d$ where: $de = 1\ (mod \phi(n))$, i.e. $d = e^{-1}(mod\ \phi(n))$. After getting $d$, owner don't need $p, q$. Owner publish $(n, e)$ as public key and keep $(n, d)$ as private. Attacker cannot derive $d$ from public key because of the **prime factorisation problem**(difficulty of factoring the product of 2 large primes).

**Encryption.** $c = m^e (mod\ n)$.

**Decryption.** $m = c^d (mod\ n)$. $(m^e)^d = m(mod\ n)$, for any positive $m < n$ and any pair of $e, d$.

**Special Property.** Use decryption key to encrypt and encryption key to decrypt.

**Finding random prime.** Randomly pick a number and test whether its prime (this is efficient). Use of **strong primes** to avoid issue.

**Efficiently computing** $d$. Extended euclidean algorithm.

**Easily encrypt & decrypt.** Modular exponentiation operations are able to computed with an efficient algorithm. To make encryption fast, choose a small $e$, but cannot be too small. Usually $e = 65537$. When $e$ is too small, vulnerable under a very special scenario.

**Security of RSA.** Problem of getting RSA private key from public key is as difficult as factorising $n$. But unknown whether getting plaintext from ciphertext and public key (**RSA problem**) is as difficult as factorisation. Some form of IV required so that encryption is randomised.

**Homomorphic Property.** $E(m_1 * m_2)$ can be evaulated by using $E(m_1)$ and $E(m_2)$.

Leads to attacks and information leakage, can be protected by using padding. Use **PKCS#1** adds optimal padding. RSA not necessary more secure than AES, classroom RSA have to be modified to prevent certain attacks because of the homomorphic property. Factorisation can be done by quantum computer, meaning that RSA is broken by quantum computer, but unsure about AES.

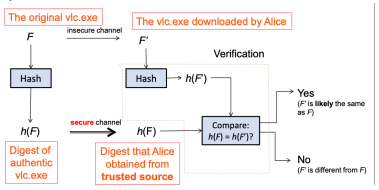
**RSA slower than AES.** When large file encrypted under PKC, improve efficiency by randomly choose an AES key, $k$, encrypt file using AES to produce ciphertext, $C$. Encrypt $k$ using RSA to produce $y$. Final ciphertext consists of $(y, C)$. To decrypt, use private key to get back $k$. Then use $k$ to decrypt $C$.

**ECB with RSA**. Dividing plaintext into pieces and independently encrypting them could leak information.

# Data Integrity (Hash)

**Hash** is a function that takes an arbitrarily long input and output a fixed-size digest. Hash function that takes no key/secret is key-less. Requirements for hash: **efficient** (given m, computationally efficient to compute y = h(m)), **pre-image resistent(one-way)** (given y, computationally infeasible to find m s.t. h(m)=y), **collision resistent** (computationally infeasible to find any pair of distinct messages s.t. both messages have the same digest). Collision resistance implies pre-image resistance. Not pre-image resistance implies not collision resistance. If we have an algorithm to attack pre-image problem, we can product collisions. **Don't use MD5**. Don't hash IV since IV known by attacker.

**Unkeyed hash.** Assumption: **secure channel to send information**. Consider a file, $F$. Alice obtains digest $h(F)$ from secure channel. Then obtain $F'$. Compute the digest of $F'$. If same digest, then likely same file, otherwise different.




Attacker try to find a collision of the hashes. Cryptographic hash functions generate pseudo-random numbers because they become deterministic if the seeds are known. So any form of key or IV with hash functions and known seed is insecure since attacker can just repeat to obtain the values.

# Data Authenticity

**MAC (Message Authentication Code)** is result of using a **keyed-hash** function that takes in an arbitrary long input and **secret key**. MAC can only be generated by someone who knows the key. Security on MAC is for forgery.

**Security Requirement.** After sniffing multiple valid pairs of messages and corresponding MACs, difficult to forge MAC of message not seen before (secure against existential forgery).

**CBC-MAC.** Based on AES, block cipher, operated under CBC.

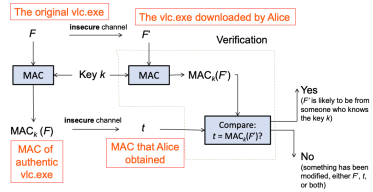
**HMAC.** Based on any iterative cryptographic hash function, hashed-based MAC.

Generally, MAC algorithms (HMAC) is much faster than signature algorithm (DSA). AES-GCM for authenticated encryption.

## Symmetric Key Setting

Use MAC for authenticity. MAC might be modified by attacker but can be detected with high probability.

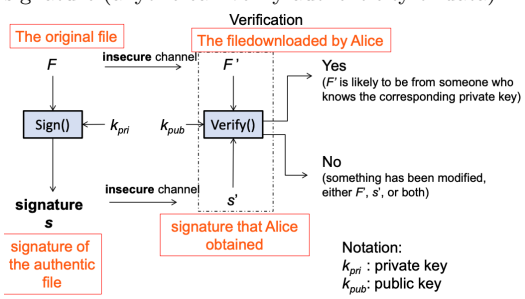
**Security Requirement.** Without knowing $k$, difficult to forge MAC.




Attacker forge valid pair of message/data and MAC. No issue with confidentiality. MAC typically appended to F, then stored as a single file or transmitted together through a communication channel. MAC is the authentication tag. Later an entity that wants to verify authenticity of F can carry out verification process using secret key.

## Public-key Setting

Public-key version of MAC. Owner use private key to generate signature. Only person who know private key can generate. Public can use public key to verify signature (anyone can verify authenticity of data).




**Security Requirement.** Without knowing private key, difficult to forge s.

Signature computed using **private key** and F. Signature appended to F. When "Alice signs file", means that Alice compute signature and append to file. Authenticity of F can be verified by anyone who knows public key. Since private key is used to sign, if the signature match, then its guarenteed to be data-origin authentic. No one except authentic signer can forge signature.

**Non-repudiation.** Assurance that someone cannot deny previous commitments or actions.
Generation of signature: pass file through **unkeyed hash** and then use private key to sign the hash.
Verification of signature: check that signature and hash are the same.
**RSA-based Signature.** Use RSA private key to encrypt hash to get signature then use RSA public key to decrypt signature and compare.

# Birthday Attack on Hash

All hash functions subjected to birthday attack. Make use of birthday paradox - how many people need to be randomly selected so that with some probability, there is a pair with the same birthday. Suppose M messages and each message is tagged with a random value in $\{1, 2, \dots T\}$. If $M > 1.17T^{0.5}$ then with probability more than 0.5, there is a pair of messages tagged to the same value. Suppose digest of hash is $x$ bits, $T = 2^x$. Then attacker wants to find collision. Can just randomly generate $M = 2^{1+x*0.5}$ to satisfy the inequality. Probability that collision occurs is $\approx 1 - e^{\frac{-M^2}{2T}}$. Digest length must be significantly to prevent birthday attack because of the low work factor in breaking the scheme.

# Token vs Session-ID Authentication

**Token.** After user login once, website wont prompt user for password. Browser carry out authentication automatically. Expiry date on token in case get stolen, token can be used by attacker forever. Once expired, forced to re-authenticate.
**Session-ID.** Server keep database of all userid associating each with a session ID. On successful manual login, server generates unique hard to guess session ID and send to user. Then update userid entry with sid and time. Updates to sid required when session expired or when user change password(logged in on multiple devices, if 1 lost at least when password refreshed, all sessions will be logged out). Does not rely on any cryptographic primitive. But need to maintain a database and takes longer to verify.

# Public Key Distribution

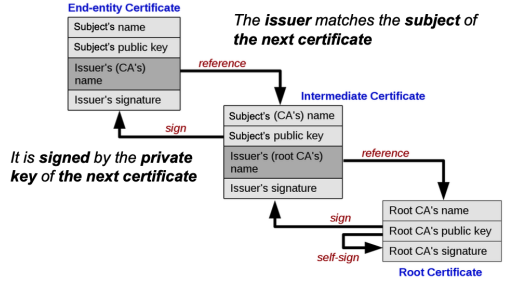
**Public Announcement.** Owner publicly broadcasts public key by email or publish on website. Not standardised and thus no systematic way to verify public key when needed. Eventually trust entity distributing key.
**Publicly-Available Directory.** Search public directory by querying server. Directory contains name and public key associated with it. Anyone can post public key to server and server cannot verify information is correct. Some entity need to be trusted. Server is bottleneck and single point of failure, need to be online to be queried.

# PKI

**CA**(Certificate Authority) issues and signs certificates, has its own public-private key pair. Assume that CA public key has been securely distributed. Acts as directory server but instead issues certs to entities to bind their public keys with them. This overcomes 1) directory server being a bottleneck, 2) verifier needs to have online access to directory server at verification point.
**Certificate.** Digital document that consists of some main items: **identity** of owner, **public key** of owner, **time window** of validity, **signature** of CA. If CA is like directory server, then CA needs to be online for queries and can become bottleneck. Instead, include certificate when sending messages. Then receiver verify signature of CA. Since no one except CA can produce valid signature, authenticity of information in the certificate is as good as coming from CA. CA also responsible to verify that information is correct.
**Self-Signed Certificate.** Certificate that is signed by stated entity's private key. Normally used by root CA or by developers in early stage of software development period when valid certificate of a host is not available.
**Domain Validation.** Purchaser can show the right to adminstratively manage a domain name.
**Organisation Validation.** Purchaser additionally has an organisation that actually exist as a legal entity.
**Extended Validation.** Purchaser can persuade certificate provider of its legal identity, including manual verification checks by a human.

## Types of CA

**Root CA** certificate is self signed.
**Intermediate CA.** Tiers 1,2...**Leaf CA** issues certificates to end entities. Entire structure is tree like.
**Hierachy of Trust.** Trust direction goes from leaf to intermediate then to root.
Alice certificate issued and signed by $CA_1$, and Bob doesnt have public key of $CA_1$. Alice must attach **her and** $CA_1$ certificate. Then Bob can verify the certificates using hierachy of trust, and verify Alice email using her public key. If Alice doesn't attach $CA_1$ certificate, then Bob need to obtain it from other sources. **Note portion** indicates certificate owener can issue certs.
**Certifaction Chain.** List of certificates that starts with an end-entity certificate, followed by one or more CA certs and the last is the self-signed root CA certificate. For each certificate (except the last), issuer matches subject of next certificate in list and signed by private key of next certificate. Last certificate is the trust anchor.




## Certificate Revocation

Certs can be revoked when private key was compromised, issuing CA compromised, entity left organisation, business entity closed. Verified need to check if certificate is still valid even if not expired.
**Certificate Revocation List.** CA periodically signs and publish a revocation list.
**Online Certificate Status Protocol.** Query if certificate is valid.
**Risks of accepting expired certificates** Previous owner still have private key and run a spoofed website with outdated certificate and redirect traffic to the spoofed site and establish at HTTPS connection using private key. Another risk is that attacker can forge different certificate having same signature and run a spoofed website with forged certificate.

# Limitations of PKI

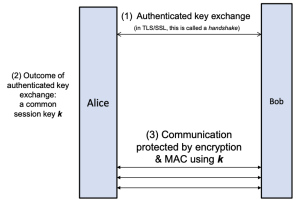
Compromised CA. Abuse by CA (malicious CA), rogue CA can practically forge any certificate.
**Implementation Bugs.** Null-byte injection. Some browsers ignore substrings after null character when displaying addresses but includes when verifying certificate.
**Social Engineering.** With **typosquatting**, where website is a phished website with one character changed to looked very similar to the original one.

# Strong Authentication

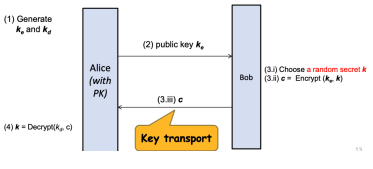
**Symmetric Challenge-Response.** Shared secret key $k$, and both agreed on some encryption scheme. Alice sends Bob message, Bob randomly picks $m$, encrypts it and send the result to Alice. Alice then decrypts the message and send $m$ to Bob. Bob verify that message is indeed $m$. If Eve can obtain all communication, still cant get secret key. **Cannot replay too** since $m$ is random and fresh everytime. Protocol only authenticates Alice (**unilateral authentication**).
**PKC Challenge-Response.** Alice send Bob message, Bob choose random number $r$ and send to Alice. Alice use **private key to sign** $r$ and sends to Bob, signed $r$ and her cert. Bob verify cert, extract public key and then verify signature. **Cannot derive private key** and reply response because $r$ is likely to be different. $r$ refers to the nonce. Assumption is that Mallory **unable** to interrupt the session. If Mallory can, need to introduce **session key,** $k$ established by Bob and Alice. Process of establishing session key is **key exchange**. Subsequent communication must be secured using session key (can be a set of keys). Not feasible to implement a monitoring gateway for all sessions keys shared because it requires a browser change.

# Unauthenticated Key Exchange

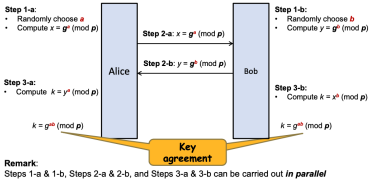
Goal is to find a good key (confidentiality).




**Using PKC.** Alice generates a pair of private/public key. Alice sends public key, $k_p$ to Bob. Bob then randomly choose a secret $s$, encrypt the secret using $k_p$ and send ciphertext $c$ to Alice. Alice use private key to decrypt and obtain k.

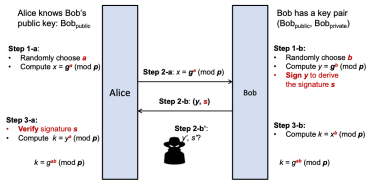



**Diffie-Hellman.** Assume pre-agreed on 2 public parameters generator $g$ and large prime $p$. Difficult to get key just by sniffing, uses Discrete Log Problem. Mallory can be man in the middle and uses her own 2 parameters to communicate with Alice and Bob. Mallory decrypt and re-encrypt messages, able to see and modify messages.



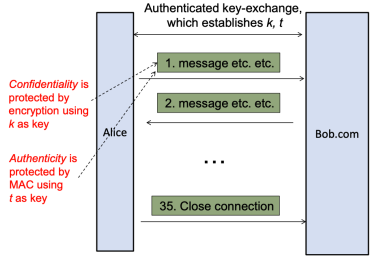

# Authenticated Key Exchange

**Station-to-Station Protocol.** Authenticated key exchange based on DH. Add extra signature to DH.




Above is only unilateral where Alice want to authenticate Bob. To make it mutual, Alice sign her message in 2-a.
**Requirements.** For mutual authentication, Alice and Bob must have a way to know each other public key, for unilateral, just need one.

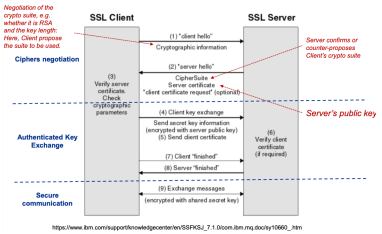
# Secure Channel




Sequence number is to ensure that messages are not reordered, detect duplicates, dropped records.
**Authenticated Key Exchange.** Carry out unilateral authenticated key exchange using Bob private/public key. After authentication, Bob and Alice know 2 randomly selected session keys $k, t$, where $k$ key used to encrypt/decrypt messages, $t$ is for MAC (symmetric).
**Secure Communication.** Communications protected by $k, t, i$. Data of message $m_i$ would be $E_k(i||m_i)||MAC_t(E_k(i||m_i))$ (encrypt-then-MAC)

# HTTPS & TLS/SSL

HTTPS = HTTP + SSL. TLS/SSL sits between transport and application layer.
**TLS.** Involves ciphers negotiation, authenticated key echange, symmetric key based secure communication and renegotiation if needed. TLS handshake is unencrypted. MiTM can see handshake messages but is unable to derive session key and because of that, can only see encrypted messages after that.




User asks for certificate (handshake). Server sends cert (handshake). User verify cert and get public key. User generate session key pair. User encrypt session key pair using public key and send to server. Session key pair will expire after a while and renegotiation is needed. Only the last 2 steps will be executed. Asymmetric encryption used when performing authenticated key exchange. Symmetric encryption for exchange of messages (encrypted using session keys). TLS is a mutual authentication but each stage is unilateral.

**TLS Renegotiation.** Userful if either client/server who are already in a TLS session want to change cipher suite or has current session that is expiring and want to continue. **Benefit** is that current session renegotiated with new handshake and gets resumed with new keys. **Conducted** within the actie TLS session, renegotiation new handshake protected by existing session key (from previous handshake).

**TLS Renegotiation Attack.** Attack allows attacker to **prepend any client message** since server takes client TLS handshake as renegotiation, and believe that attacker initial/preprended data also from client.
Attacker connect to TLS, compelete first handshake and establish session keys. When ready, hijack client connection with server and when client tries to establish connection with server, attacker intercept and proxy client traffic over his encrypted channel. To server seems like renegotiation request and session keys generated by client but redirected to server by attacker (seems like it was from attacker), making server confused over whos keys these are. Attacker gets ack and replay to client. Attacker initial traffic and client subsequent traffic combine. Does not compromise confidentiality but breach integrity. Second handshake has partially encrypted key. Contains session keys encrypted by client using server public key which is further encrypted by attacker current session key.

**Naive Failed Solutions.** From client to server: add handshake number when requesting to connect. Doesn't work since attacker can change the order. Since original client message is in clear to attacker. From server to client: add handshake number as well, face the problem as above.

**Possible Solution.** Send 2 GET commands instead 1 (first is a dummy op). GET command to be process includes part of session-ID or value dervied from session-ID for server verification. Attacker dont know session-ID. But against modularity principle since communication security handled below application layer (shouldnt be done by web developer).

**Transport Layer Possible Solutions.** Server can sign the message using private key. Or apply MAC using secret key previously established.

**Possible App Layer Solutions.** Send 2 GET commands instead of 1, where the first is just a dummy. GET command includes part of cookies for server verification. GET command includes a vlaues that is derived from cookies.

# Authenticated Encryption

Symmetric encryption that returns both ciphertext and authentication tag. Combines cipher and MAC.

**Encryption Process.** $AE(K_{AB}, M) = (C, T)$.

**Decryption Process.** $AD(K_{AB}, C, T) = M$ only if $T$ is valid.

**Encrypt-and-MAC.** Compute cipher and MAC **separately**, sends both over. Usually used in SSH. MAC may not be random and could leak information.

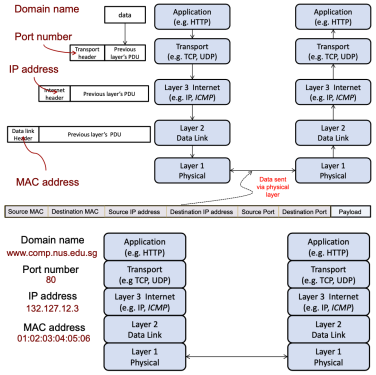
**MAC-then-Encrypt** Compute MAC and append to plaintext then encrypt the entire thing. Used in SSL and TLS(v1.2). Decryption still needed on corrupted message.

**Encrypt-then-MAC** Encrypt first, then pass the ciphertext to be MAC. Then send the entire thing. Used in IPsec. Decryption not performed on corrupted message.

# Network Layers

**Packet Switching.** Deployed where messages are broken into packets and routed via **multiple** switches and routers.

**Network Layering.** Allows peer entities at same layer to conceptually ocmmunicate with each by executing a protocol at that layer. The layer below provides services to entities in layer above (higher level protocols built on top of a virtual connection at lower level).



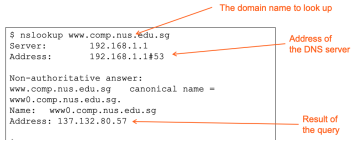

At layer N, messages to be sent is called: layer-N protocol data unit (PDU), encapsulation of upper (N+1) layer PDU. Data may go through multiple hops, intermediate nodes might change header information.

**Listening Port.** Port where process in host is listening. If not listening then its **Closed Port**. Possibility that packet can be lost since this property of **UDP** which is **connectionless-oriented** and **unreliable communication**.

**TCP.** Connection oriented, involving 3-way handshake. TCP is reliable since it has mechanisms to re-transmit, re-order, acknowledge packets so that destination can receive all the sent messages in the correct order. Reliable but not secure, means possible to sniff stuff from the handshake, similar for access point passwords. Intermediate nodes can still read and modify data in the header and payload, vulnerable to **Man-in-the-Middle**. Reliability does not imply security.

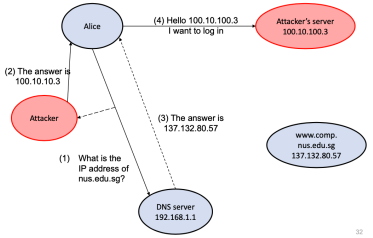
# Domain Name System

Maps domain name to IP. Attacker can target association of domain name with IP. Operates at application layer. DNS server is single point of failure for the network. Denial of service attack can attack DNS server instead.




Look up a locally stored table or query DNS server. Client that initiates is called resolver. If address found then domain name is resolved. Each query is 16-bit Query ID (QID). Response must also contain a QID, must match QID of query. No encryption/MAC involved.

**Attacker at physical layer** (another person on same network). Since WIFI not protected, can sniff and spoof data from/into communication channel but can't modify data sent by Alice. Attacker has own webserver.




Alice asks for address. Mallory sniffs and spoofs reply with same QID. DNS also replies but Mallory closer so Mallory reply reach Alice first. Alice take first reply and connect with attacker's webserver.

# Denial of Service Attacks

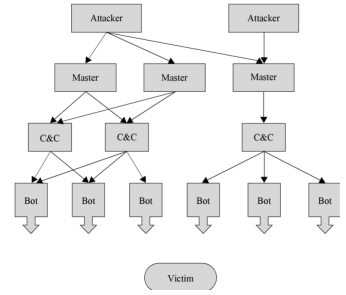
Attack on availability.

| | Stopping Service | Exhausting Resources |
|---|---|---|
| **Local Attack** | • Process killing<br>• Process crashing<br>• System reconfiguring | • Spawning processes<br>• Filling up file system |
| **Remote Attack** | Sending malformed packet attacks | ***Packet flooding*** |

**ICMP/Smurf Flood Attack.** Attacker sends **ICMP PING** request to router, instructing router to broadcase request to all local nodes. Request' source IP address is spoofed with victim's. Router broadcasts **Echo Request**. On receipient, replies to it by sending reply to the source which is victim. Victim overwhelmed with replies and attacker take advantage of amplification effect. Most routers are configured not to broadcast request.

**Botnet.** Bot is a compromised machine. Botnet is collection of connected bots, communicating via covert channels. Command-and-control mechanism, can be control by individual to carry out distributed DOS.

**Covert Channel.** A type of attack that creates a capability to transfer information objects between processes that are not supposed to be allowed to communicate by the computer security policy.
Pharming redirect a website's traffic to a fake site by installing a malicious program on the computer. Pharming can be conducted either by changing the hosts file on a victim's computer or by exploitation of DNS server software.




# Address Resolution Protocol

Maps IP address(logical) to MAC address(physical) using broadcast mechanism. Attacker on local network can target associations.

# Tools

**Wireshark.** Performs capturing at link layer.
**Nmap.** Scans ports by determining which ports are open on hosts in a network.

# Cryptographic Communciation Channel

Security protocol that protects layer k would protect information from that layer and above against an attacker sitting at layer k-1 and below.

**SSL/TLS.** Sits on top of transport layer. Pass data and dest IP to SSL/TLS. Encrypt the data and MAC, then instruct transport layer to send protected data. End-to-end encryption(data encrypted from one end of its journey to the other.) is performed. Receiver end-point decrypts received data at corresponding layer. Link encryption encrypts and decrypts all network traffic at each point until arrival.

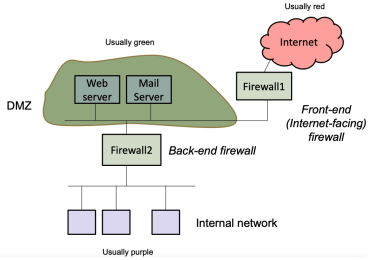
**WPA2.** Provides protection at link (L2, but not all protected) and physical (L1). Attacker can learn MAC address.

**IPsec.** Provides integrity/authenticity protection of IP address but not confidentiality. Protection is at ip (L3). Attacker cannot spoof but can sniff.

# Firewall

Divide internal network into different network segments and deny unnecessary access: principle of least privilege (access information that are necessary for its legit purpose), compartmentalisation (confining information within compartments). Enforces a set of rules by packet filtering typically on transport and network. * is wildcard, table processed top-down, first match determines action taken. Put most specific first and most general last.

**Demilitarise Zone.** is a small sub-network that exposes org's external service to untrusted Internet.



**Types.** Packet filters (filter based on info in packet headers), Stateful inspection (maintain state table and filter based on active connections), Application proxy (understand app logic, relay app-level traffic).

# Network Security Management

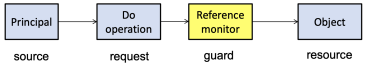
**SOC.** Security Operations Center is a centralised unit in an organisation that monitors theIT systems and deals with security issues.
**SIEM.** Security Information and Event Management is an approach that aims to provide real-time analysis of security alerts generated by network hadware and network applications.

# System Layers

If attacker sneaks into a layer, must not be able to directly manipulate objects/data and processes in more privileged layers.

# Access Control

Gives a way to specify & enforce restriction of operations on objects by principal/subjects.




**Principal/Subjects.** Wants to access object with some operation. Principals are human users and Subjects are entities in the system.
**Reference Monitor.** Controls access.
Accesses can be classified as **Observe**(reading), **Alter**(modify), **Action**(execute). Each object has an owner.
**Discretionary Access Control.** Owner of object decides rights.
**Mandatory Access Control.** System-wide policy decides rights which must be followed by everyone in the system.

## Representation

**Access Control Matrix.** Matrix of principals as rows, and objects as columns. Each entry specifies access rights of principal to object. Table can be very large and difficult to manage.

| principals \ objects | my.c | mysh.sh | sudo | a.txt |
|---|---|---|---|---|
| root | {r,w} | {r,x} | {r,s,o} | {r,w} |
| Alice | {} | {r,x,o} | {r,s} | {r,w,o} |
| Bob | {r,w,o} | {} | {r,s} | {} |

**Access Control List.** Stores access rights to particular object as a list.

| | |
|---|---|
| my.c | → (root, {r,w} ) → (Bob, {r,w,o} ) |
| mysh.sh | → (root, {r,x} ) → (Alice, {r,x,o} ) |
| sudo | → (root, {r,s,o} ) → (Alice, {r,s} ) → (Bob, {r,s} ) |
| a.txt | → (root, {r,w} ) → (Alice, {r,w,o} ) |

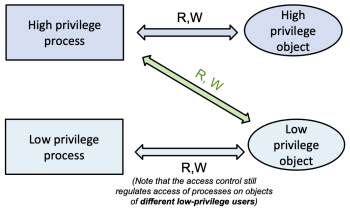
Difficult to get an overview of which objects(files) that a particular subject has access rights to.
**Capabilities.** Subject given a list of capabilities where each capability is access rights to object.

| | |
|---|---|
| root | → (my.c, {r,w} ) → (mysh.sh, {r,x} ) → (sudo, {r,s,o}) → (a.txt, {r,w}) |
| Alice | → (mysh.sh, {r,x,o} )→ (sudo, {r,s}) → (a.txt, {r,w,o}) |
| Bob | → (my.c, {r,w,o}) → (sudo, {r,s}) |

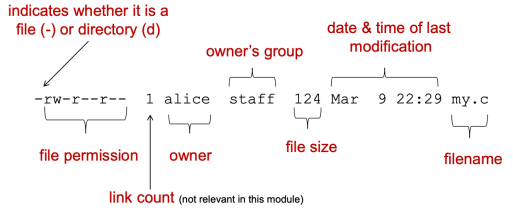
Difficult to get an overview of the subjects who have access rights to a particular object.
**Drawbacks of both.** Size of list can be too large. Instead, group the subjects/objects and define access right on the groups.

# Intermediate Control

Specifies rights for user(owner), group (owner group), others (world). **Non-owner subjects in the same group have same group's access rights**! Groups created by root. Luminus analogy: object created in project groups can only be read by members of group + lecturers.
**Role-based Access Control.** Determined by role of subject. Roles associated with a set of procedures (to carry these out, access rights needs to be granted to certain objects.)
**Least Privilege Principle.** Access rights that are not required to complete the role will not be assigned.
**Protection Rings.** Each object(data) and subject(process) are assigned a number that indicates the ring. Smaller number means more important, more privilege. Subject cannot access an object with smaller ring number. Only can do so if privilege is **escalated**.




Super user and normal user, 2 rings for UNIX.
Principals refer to UIDs and GIDs.
**User.** Unique login name and numeric identifier (UID), can belong to multiple groups. Superuser has UID of 0.
**Group.** Unique group name and numeric identifier (GID).




Permission grouped into 3 triples, that define read(r), write(w, exclude delete), execute(x, s allow execution with permission of file owner) for classes of owner, group and others. - indicates access not granted. Any process invoked by user inherits their rights. **Root can do everything!**

- **Symbolic mode** notation:
  - **Syntax**: [references][operator][modes]
  - *Reference*: u (user), g (group), o (others), a (all)
  - *Operator*: + (add), - (remove), = (set)
  - *Mode*:
    r (read), w (write), x (execute), s (setuid/gid), t (sticky)
  - Examples:
    `chmod g+w shared_dir`
    `chmod ug=rw groupAgreements.txt`
  - What are the file permission bits of `shared_dir` and `groupAgreements.txt`?
    `shared_dir: drwxr-xr-x → drwxrwxr-x`

**Octal Mode.** 3-4 digits, 3 rightmost refer to permissions for user, group and others. If permission granted, then theres a 1 in binary place. Example like 664 means -rw-rw-r-.

**Set-UID.** Process effective UID is **owner** of executable file instead of user running it.
**Set-GID.** Effective GID is **group owner** of executable file.

**Permission-Checking Rules.** . When non-root user wants to access file, check the following in order. If user is owner, permission bits for owner decides access rights. If GID owns file, permission bits for group decides. Otherwise permission bits for other decide.
**Controlled Invocation.** Non-root user requires access to files that only superuser has privilege. Use set of predefined programs with elevate privilege which is provided by high-privilege users. Any user can invoke these.
**Implementation Bugs.** Attacker trick program to perform illegal operations not expected by programmer/designer.
**Process Credentials.** Real UID inherited from user who invokes the process. For processes created by executing a file:

- if permission bit is x, process effective UID = real UID
- if permission bit is s, process effective UID = file owner UID

Effective UID is treated as subject and checked against file permission to determine access.

# Creating Backdoor

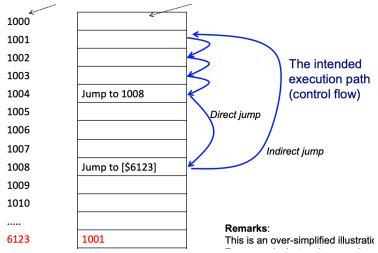
Plant a copied shell, final command should be *chmod* $u+s$, ensuring that the file is world executable. But later on it wont run with root privileges. Reason is that when shell is executed as a set-UID process, immediately change effective UID to real UID (privilege descalation). Instead write code that creates an executable and enable set-UID bit.

# Software Security

Program must be correct, efficient and secure.
**Insecure Implementation.** Results in deviation from programmer intent.
**Unanticpated Input.** Results in accessing sensitve resource, deivate from intended execution path, execute some injected code. Either way, managed to elevate its privilege.

# Computer Architecture BG

**Von Neumann Architecture.** Code and data stored together in memory with no clear distinction.
**Harvard Architecture.** Hardware that separately store code and data.
**Program Counter.** Register that stores address of next instruction. After instruction completed, fetch next instruction from address in PC. Then PC increases by 1 (assuming fixed length).
**Direct Jump.** PC replaced with constant value specified in instruction.
**Indirect Jump.** PC replaced with value fetched from memory/stored in general-purpose register.

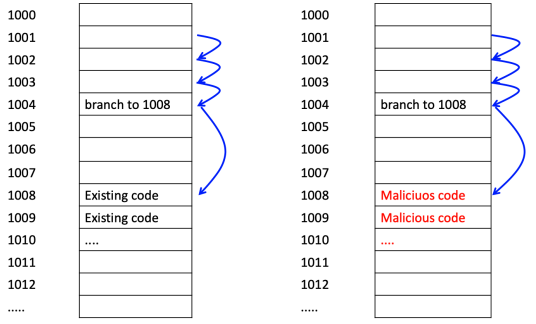



# Stack

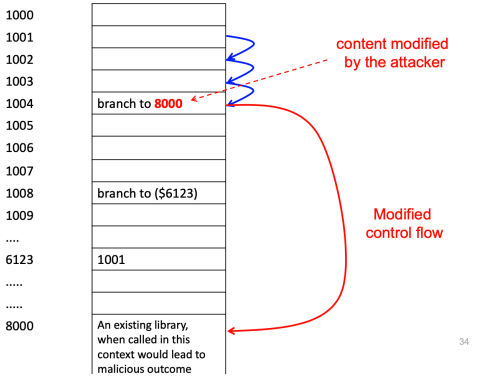
**Call Stack.** Data structure in memory that stores important information of a running process, LIFO. Keeps track of control flow information, parameters passed to functions and local variables.
**Stack Pointer.** Location of top element refered to.
**Stack Frame.** Activation Record that contains local variables, previous frame pointer, return address and parameters in order from top to bottom.

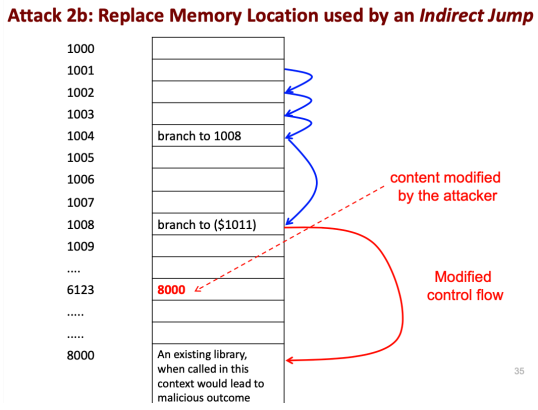
# Process Execution Integrity

Compromising a process memory integrity implies compormising its execution integrity.



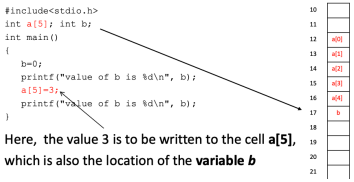

Replace process code
**Attack 2a: Replace Memory Location used by a *Direct Jump***





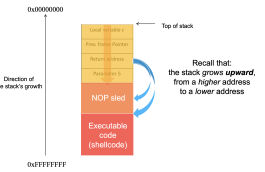

# Attacks on Software

**Integer Arithmetic.** Modulo arithmatic with $2^n$, where n is number of bits. Makes some predicates like $a < a + 1$ not always true(when a = 256, a += 2 then a = 0).




**Buffer Overflow.** Situation where data is written beyond buffer boundary.
**Strcpy().** copy entire string even if length of $s2$ is more than length of $s1$, leading to overflow. **Use strncpy() instead**, copies at most n. But still vulnerable if n not set correctly but strncpy() does not terminate with null naturally, leading to vulnerabilities.

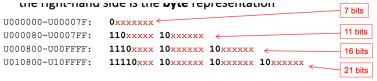
**Stack Smashing.** Special case of buffer overflow that targets a process' call stack. Overflow stack st return address is modified, then execution control flow will be changed. Buffer overflow causes segmentation fault to occur, but everything in the function will still run, does not return.


Targets data in memory that controls process execution (return addresses, function pointers, virtual function table) and important program variables (program specific, credential related, etc).
**NOP Sled.** sequence of no-operation instructions meant to "slide" the CPU instruction execution flow to its final destination whenever the program branches to a memory address anywhere on the slide. Attackers may not be able to have the return address point to the exact location or when the target address that the control flow is looking for is not known precisely, creating a greater area for attackers to strike and still ensure their shellcode will run.

# Data/String & Security

**NULL-Byte Injection.** Strings may be null terminated. Browser verify certifcates based on non-NULL termination and compare name on cert and url based on NULL-termination. Not able to detect null byte in url and able to verify the attacker's cert.
**ASCII Char Encoding.** Encodes 128 characters into 7 bits.
**UTF-8.** Encodes all valid code points in Unicode usning one to four 8-bit bytes. ASCII remains unchanged in UTF-8.
**Variable-length Encoding.** Code points that tend to occur more frequently are encoded with lower numerical values, using fewer bytes.




Byte representation seems unique but it can be extended to longer variants and filled with 0s instead. Lead to inconsistency between character verification and character usage.
**Potential Problems.** Server side program that receives a string from client and then append the prefix directory string and invoke sys call to open and send content. Attacker then uses '../<some file>' and violates the intented file-access containment (original intent is that client can retrieve under some directory only). Server then check validation of input with '../'. If the sys call uses a convention that '%' followed by 2 hex digits indicate a single byte, then by replaceing '../' with its hex representation, it will bypass the check. Hence blacklisting-based filtering could be incomplete due to flexible use of character encoding.
**Blacklist IP address.** Checking done by sections to see if its in blacklist. Possible to exploit by inputting numbers greater than 255 since thats upper limit of 8 bits. Will pass through.
**Prevention.** Always convert input from user to a standard representation immediately. Make use of underlying system access control mechanism.

# Undocumented Access Points

Undocumented access points to inspect certain states. May remain in final production, providing backdoors to attackers.

# Defensive and Preventive Measures

**Safer Function Alternative.** Avoid functions that are known to create problems, such as strcpy(). Some functions can be unsafe depending on its usage. scanf("%s", str) is unsafe since there is no limit to characters that can be stored, but scanf("%20s", str) is safe if the size of the string is greater than 20. Null character added at end of string, the specified length doesnt include this.
**Input Validation using Filtering.** Ensure that input validation/filtering whenever an input is obtained from user, reject on wrong format. Difficult to ensure that filtering is complete.
**White List.** List of items known to be benign and allowed to pass which could be expressed using regex. Some legit inputs may be blocked though. More secure than blacklist but may not be desirable.
**Black List.** List of items known to be bad and to be rejected. Some malicious input may be passed.
**Bounds Checking.** Ensure that assignment is done correctly based on size of array. On assignment, check that index is within bounds then make assignment. If checks fail then process halted. This prevents buffer overflow but at the expense of efficiency.
**Type Safety.** Ensure that arguments an operation get during execution are always correct. Different sized integers are different types. Checking can be done at runtime (dynamic type checking) or compile time (static type checking).
**Manual Checking.** Manually check the program, tedious.
**Taint Analysis.** Automatic checking. Checks whether critical function arguments could potentially be affected by malicious inputs. If so, special checks carried out. Can be static (check code without running/tracing) or dynamic (run with some input).
**Canaries.** Secret values inserted at carefully selected memory locations at runtime. Checks carried out at runtime to ensure values not modified. Help to **detect(not prevent)** overflow. Important to keep values secret otherwise attacker can just write canary value. Can be implemented using compilers.
**Memory Randomisation.** Address space layout randomisation helps to decrease attacker chance. Randomly arrange address space position of key data areas of a process so attacker cannot find location of shellcode (for eg).
**Testing.** Whitebox (tester access app source), blackbox (tester dont access app source), greybox (combination of abv, reverse-engineered binary/exe), fuzzing (send malformed inputs)
**Principle of Least Privilege.** Be conservative in elevating privilege(when writing), dont give users more access rights than necessary and dont activate unnecessary options (when deploying).
**Patching.** Fixes vulnerability, but patch can be useful to attackers since they can inspect patch and derive vulnerability, increasing number of attackers after patch announcement.
**Zero-Day Vulnerability.** Vulnerability in a system that has been disclosed but not yet patched.

**Format String Vulnerability.**

**Initialise Variables.** In C, uninitialised values can take any value which could be sensitive. Drawback is that this takes extra processing time.

# Overview of HTTP

User clicks on URL. HTTP request sent to server with any in-scope cookies. Server constructs and includes a HTML file inside its resposne to browser with set-cookie headers. Browser render HTML file.
**Sub-resource.** Multimedia files, images, CSS, scripts including from external/third-party websites. When parsing a page with sub-resources, browser contacts respective server for each sub-resource. Separate HTTP request for every single file on page.
**HTTP Request.** Contains request line, request headers (things to accept and cookie), empty line and optional message.
**HTTP Response.** Contains status line (status code and reason), response headers (content type, content length, set-cookie that matches request coookie and expiry date), empty line and optional message.
**Client Components.** HTML (content), CSS (presentation), JS (behavior).
**Server Components.** Web server (scripting language), DB server (interaction between web and db via SQL).
**Clickjacking.** Attack that tricks user into clicking a webpage element which is invisible or disguised as another causing users to unwittingly download malware, visit malicious sites, provide credentials/sensitive info, transfer money or purchase stuff online.

# Security Issues and Threat Models

**Browser Operations.** Browser runs with same privilege as user. Multiple servers could provide content, access isolation among sites is required. Rich command set and controls supported togther with plugins.
**Browser Usage.** Stores user sensitive information and secrets (in cookies), user can update content in servers.
**Attackers as End Systems.** Malicious webclient who can access target server and malicious web server that lure victims.
**Forum Poster.** Weakest attacker type, user of existing web app, doesnt register domain or host app content.
**Web Attacker.** Owns valid domain and web server with SSL cert. Entice victim to visit site but cannot intercept/read traffic for other sites.
**MitM.** Attacker has accesss to exchanged network packets at IP layer.
**Passive.** Eve who can just eavesdrop but cant spoof, can act as web attacker.
**Active.** Mallory who can launch active attacks, can be web attacker too. Not generally considered to be capable of presenting valid certs for HTTPS sites not under his control.

# Attacks on TLS/SSL

**Heartbleed Attack.** Improper input validation (due to missing bounds) in implementation leading to buffer over-read. Overread buffer may contain sensitvie information.

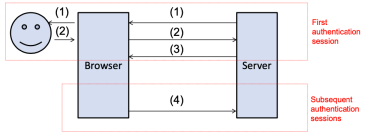
# UI Attacks

**URL.** Consists of scheme, authority, path, query, fragment.
Typically used as part of phishing. Victims may not realise that they are visiting a spoofed site.

**Homograph Attack.** Phishing site with domain name that is only a few pixels different from the actual.
**Boundary Confusion Attack.** Replacing the '/' with characters resembling it.
**Address Bar Spoofing.** Malicious page overlay a spoofed address bar on top of actual address bar.

## Cookies & Same-Origin Policy

**Web-cookies.** Text data set by web server and sent in HTTP response set-cookie header field and consists of a name-value pair. Stored in browser, when user revisit the site, browser sends all in-scope cookies in request's cookie header field.
**Session Cookies.** Deleted after browsing session ends.
**Persistent Cookies.** Expire at specific date/after specific length of time.
**Secure Cookie.** Only transmitted over HTTPS.
**HTTP-only Cookie.** Cannot be transmitted by client-side APIs.
Since HTTP is stateless, need to keep track of web session. Cookie is commonly used to set and indicate a session ID in web auth scheme. Cookie used to remember user information and relevant contentt can be shown to user in subsequent visits.
**Session-ID Auth.** After user authenticated, server send unique unguessable SID. Server keeps track of association between user and SID. In subsequent request, whoever presents SID is accepted as authentic user. SID has expiry date.




Authentication challenge-response initated first. Then server send SID and browser keeps it. Browser present SID with request and server verify.
**Sane-Origin Policy.** Script in webpage can access cookies stored by another webpage if both have same origin (defined by protocol, hostname, port number). Very prone to errors.

## Cross Site Scripting Attacks

Injection attack on web apps where attacker attacks another user by causing latter to run script from an involved web server, avoiding Same Origin Policy. Both attacks exploit trust of involved server.
**Reflected (Non-Persistent) XSS.** When client enter an invalid URL, server response with HTML that also contains the string. If string contained a script, will be executed but wont work if server performs HTML encoding.

- attacker tricker user to click on URL which contain target site and malicious script
- request sent to server
- server constructs response HTML (response contains malicious script)
- browser render and runs script

Browser believes injected script is from server. Since script provided by attacker, has same privilege as web server and run things as though its the webserver.
**Stored (Persistent) XSS.** Malicious script alreayd on web server (forum page where attacker is malicious poster). Script automatically rendered.
**Defenses.** On server-side, filter and remove any malicious script in request while constructing page and in user post before its saved in database.

## Cross Site Request Forgery Attack

Authorisation attack on web apps where attacker issue a forged request to web server on behalf of victim, disrupting integrity of victim user session. Exploits trust of client.
**Clicking.** If user is already authenticated and site accepts authentication-token cookie. Then by clicking on links that make malicious requests, cookie will be sent and operation carried out.
**Without Clicking.** If user visit attacker site, attacker site have multimedia image that makes malicious request. Browser sends request to obtain image and sends cookie together, carrying out operation.
**Anti-CSRF token.** Server attach dynamic token to transaction page. On request, submit token that indicates user transacts from page. Attack cant include token in forged request.

## SQL Injection

Sample query: SELECT * FROM client WHERE uID = '$userinput'. Attacker inputs: idk' OR 1=1 –, changing the entire query to SELECT * FROM client WHERE uID = 'idk' OR 1=1 –'. Since '–' is start of comments, behavior becomes unintended.
**Tautology.** Make conditional statements always evaluate to true and cause SELECT to return all records.
**Piggy-backed.** idk'; DROP TABLE ... –' results in performing the SQL query of dropping tables.
**Prepared Statements.** Lazy execution of queries and specifying what inputs are to be expected.